DART

Release 10.6.0

Data Assimilation Research Section

Dec 12, 2022

GETTING STARTED

1 Ensemble Data Assimilation **3**

2 Organization of the documentation **5**

3 Manhattan Release **7**

4 Quick-start **9**
4.1 Quick-start for developers . . . 10

5 Citing DART **11**

6 References **13**
6.1 System requirements . . . 13
6.2 Fortran90 compiler . . . 14
6.3 Locating netCDF library . . . 14
6.4 Downloading DART . . . 15
6.5 Compiling DART . . . 15
6.6 Verifying installation . . . 17
6.7 Introduction to ensemble data assimilation . . . 22
6.8 The Lorenz 63 model and its relevance to data assimilation . . . 23
6.9 Data assimilation in DART using the Lorenz 63 model . . . 25
6.10 What is DART? . . . 35
6.11 The benefits of using DART . . . 36
6.12 A brief history of DART . . . 37
6.13 High-level data assimilation workflows in DART . . . 39
6.14 DART's design philosophy . . . 46
6.15 Important capabilities of DART . . . 47
6.16 Working with collaborators on porting new models . . . 51
6.17 DART build system . . . 56
6.18 Assimilation in a complex model . . . 60
6.19 Message Passing Interface . . . 63
6.20 Filters . . . 66
6.21 Inflation . . . 66
6.22 Required model_mod routines . . . 69
6.23 Suggestions for a "simple" model . . . 71
6.24 Suggestions for a "complex" model . . . 72
6.25 How to test your model_mod routines . . . 74
6.26 Controlling which files are output by filter . . . 74
6.27 Advice for models with multiple vertical coordinate options . . . 77
6.28 Data management in DART . . . 79
6.29 Programs included with DART . . . 80

6.30 Adding your observations to DART . . . 83
6.31 How DART supports different types of observations: the preprocess program . . . 84
6.32 How DART stores observations: observation sequence (obs_seq) files . . . 86
6.33 Detailed structure of an obs_seq file . . . 87
6.34 Creating an obs_seq file of synthetic observations . . . 91
6.35 Creating an obs_seq file from real observations . . . 94
6.36 Available observation converter programs . . . 98
6.37 Manipulating obs_seq files with the obs_sequence_tool . . . 99
6.38 The difference between observation TYPE and QUANTITY . . . 100
6.39 Adding support for a new observation TYPE . . . 100
6.40 Introduction to DART's support for RTTOV . . . 100
6.41 DART Observations . . . 102
6.42 Converter programs . . . 107
6.43 AIRS and AMSU . . . 109
6.44 Program convert_airs_L2 . . . 111
6.45 Program convert_amsu_L1 . . . 114
6.46 Aviso+/CMEMS Observations . . . 119
6.47 PROGRAM level4_to_obs . . . 121
6.48 CHAMP . . . 125
6.49 PROGRAM cice_to_obs . . . 125
6.50 CONAGUA . . . 126
6.51 PROGRAM COSMOS_to_obs . . . 126
6.52 PROGRAM COSMOS_development . . . 132
6.53 PROGRAM dwl_to_obs . . . 137
6.54 GMI Brightness Temperatures . . . 138
6.55 NOAA GOES-R Series Advanced Baseline Imager (ABI) Level 1b Radiances . . . 140
6.56 GPSPW . . . 141
6.57 GSI2DART . . . 141
6.58 GTSPP Observations . . . 144
6.59 MADIS Data Ingest System . . . 146
6.60 PROGRAM MIDAS_to_obs . . . 148
6.61 DART observations and MODIS products. . . . 150
6.62 PROGRAM MOD15A2_to_obs . . . 150
6.63 PROGRAM MOD15A2_to_obs . . . 156
6.64 MPD . . . 156
6.65 PROGRAMS LPRM_L3_to_obs.f90 AMSR_E_L2_to_obs.f90 . . . 156
6.66 PROGRAM prepbufr . . . 160
6.67 PROGRAM create_real_obs . . . 166
6.68 PROGRAM SMAP_L2_to_obs . . . 170
6.69 ROMS observations to DART observation sequences . . . 173
6.70 PROGRAM SIF_to_obs_netcdf . . . 178
6.71 SSEC Data Center . . . 181
6.72 PROGRAM sst_to_obs, oi_sst_to_obs . . . 182
6.73 SSUSI F16 EDR-DSK format to observation sequence converters . . . 186
6.74 WOD Observations . . . 188
6.75 GND GPS VTEC . . . 192
6.76 GPS Observations . . . 192
6.77 Oklahoma Mesonet MDF Data . . . 203
6.78 VIIRS/AQUA Satellite Ocean Color . . . 204
6.79 QuikSCAT SeaWinds Data . . . 206
6.80 Even Sphere . . . 209
6.81 Obs Error . . . 214
6.82 Radar observations . . . 215
6.83 PROGRAM snow_to_obs . . . 216

6.84 PROGRAM text_to_obs . . . 217
6.85 Total Precipitable Water Observations . . . 219
6.86 PROGRAM tc_to_obs . . . 222
6.87 Tropical Cyclone ATCF File to DART Converter . . . 222
6.88 PROGRAM littler_tf_dart . . . 224
6.89 PROGRAM rad_3dvar_to_dart . . . 228
6.90 3DVAR/4DVAR Observation Converters . . . 229
6.91 Checking your initial assimilation . . . 230
6.92 Computing filter increments . . . 230
6.93 Computing filter increments using a complex model . . . 231
6.94 DART missing data value . . . 233
6.95 DART quality control field . . . 234
6.96 Examining the obs_seq.final file . . . 235
6.97 MATLAB® observation space diagnostics . . . 235
6.98 DART Tutorial . . . 250
6.99 Conditional probability and Bayes' theorem . . . 251
6.100 DART_LAB Tutorial . . . 255
6.101 CLM5-DART Tutorial . . . 256
6.102 WRF/DART Tutorial Materials for the Manhattan Release. . . . 282
6.103 Supported Models . . . 294
6.104 9-variable . . . 298
6.105 AM2 . . . 299
6.106 bgrid_solo . . . 300
6.107 Atmospheric Models in CESM . . . 303
6.108 The CAM-FV DART Interface . . . 306
6.109 Community Earth System Model . . . 316
6.110 CICE . . . 320
6.111 CLM . . . 322
6.112 PROGRAM clm_to_dart . . . 336
6.113 PROGRAM dart_to_clm . . . 341
6.114 CM1 . . . 345
6.115 COAMPS Nest . . . 354
6.116 COAMPS . . . 354
6.117 ECHAM . . . 362
6.118 FESOM . . . 362
6.119 GITM . . . 365
6.120 PROGRAM netcdf_to_gitm_blocks . . . 370
6.121 gitm_blocks_to_netcdf`` . . . 373
6.122 Ikeda . . . 376
6.123 LMDZ . . . 383
6.124 Lorenz 05 . . . 383
6.125 Lorenz 63 . . . 385
6.126 Lorenz 84 . . . 387
6.127 Lorenz 96 . . . 388
6.128 Lorenz 96 2-scale . . . 390
6.129 Lorenz 96 Tracer Advection . . . 391
6.130 Forced Lorenz 96 . . . 394
6.131 MITgcm_ocean . . . 396
6.132 MPAS_ATM . . . 398
6.133 PROGRAM mpas_dart_obs_preprocess . . . 408
6.134 MPAS OCN . . . 411
6.135 PROGRAM model_to_dart for MPAS OCN . . . 428
6.136 NCOMMAS . . . 431
6.137 NOAH, NOAH-MP . . . 445

6.138 null_model . . . 450
6.139 PBL_1D . . . 452
6.140 pe2lyr . . . 452
6.141 POP . . . 459
6.142 MODULE dart_pop_mod (POP) . . . 466
6.143 ROMS . . . 473
6.144 ROSE . . . 479
6.145 Simple advection . . . 480
6.146 SQG . . . 482
6.147 TIEGCM . . . 489
6.148 WRF-Hydro . . . 492
6.149 WRF . . . 495
6.150 PROGRAM replace_wrf_fields . . . 500
6.151 PROGRAM wrf_dart_obs_preprocess . . . 502
6.152 MODULE model_mod . . . 505
6.153 MODULE model_mod . . . 517
6.154 Contributors' guide . . . 526
6.155 Requesting features and reporting bugs . . . 527
6.156 Mailing list . . . 527
6.157 DART Manhattan Differences from Lanai Release Notes . . . 527
6.158 Forward Operators . . . 538
6.159 Approaches for Common Situations . . . 538
6.160 Parallelism Implementation Details . . . 539
6.161 Other Parallelism Options . . . 540
6.162 Netcdf Inflation Files . . . 540
6.163 State Stucture . . . 541
6.164 Filter async modes . . . 543
6.165 Distributed State . . . 543
6.166 MODULE location_mod (channel) . . . 545
6.167 MODULE location_mod . . . 559
6.168 MODULE (1D) location_mod . . . 560
6.169 MODULE location_mod (threed_cartesian) . . . 572
6.170 MODULE location_mod (threed_sphere) . . . 586
6.171 program obs_seq_verify . . . 606
6.172 PROGRAM wakeup_filter . . . 614
6.173 PROGRAM compare_states . . . 615
6.174 PROGRAM gen_sampling_err_table . . . 617
6.175 PROGRAM perturb_single_instance . . . 619
6.176 system simulation programs . . . 620
6.177 PROGRAM compute_error . . . 621
6.178 PROGRAM preprocess . . . 623
6.179 PROGRAM obs_impact_tool . . . 626
6.180 program create_fixed_network_seq . . . 629
6.181 program obs_loop . . . 630
6.182 program perfect_model_obs . . . 632
6.183 program obs_selection . . . 635
6.184 program obs_sequence_tool . . . 638
6.185 PROGRAM integrate_model . . . 649
6.186 PROGRAM obs_diag (for 1D observations) . . . 650
6.187 PROGRAM obs_diag (for observations that use the threed_cartesian location module) . . . 658
6.188 PROGRAM obs_diag (for observations that use the threed_sphere location module) . . . 674
6.189 PROGRAM fill_inflation_restart . . . 691
6.190 program obs_seq_coverage . . . 694
6.191 PROGRAM advance_time . . . 705

6.192 program model_mod_check . . . 706
6.193 PROGRAM closest_member_tool . . . 713
6.194 PROGRAM filter . . . 716
6.195 program obs_keep_a_few . . . 721
6.196 program create_obs_sequence . . . 723
6.197 PROGRAM obs_seq_to_netcdf . . . 724
6.198 program obs_common_subset . . . 736
6.199 MODULE ensemble_manager_mod . . . 739
6.200 MODULE random_seq_mod . . . 762
6.201 MODULE mpi_utilities_mod . . . 768
6.202 MODULE time_manager_mod . . . 779
6.203 MODULE utilities_mod . . . 791
6.204 MODULE types_mod . . . 804
6.205 MODULE schedule_mod . . . 808
6.206 MODULE obs_kind_mod . . . 811
6.207 MODULE DEFAULT_obs_kind_mod . . . 819
6.208 MODULE obs_sequence_mod . . . 820
6.209 MODULE smoother_mod . . . 840
6.210 MODULE assim_model_mod . . . 849
6.211 MODULE assim_tools_mod . . . 863
6.212 MODULE cov_cutoff_mod . . . 873
6.213 MODULE obs_model_mod . . . 875
6.214 MODULE reg_factor . . . 878
6.215 MODULE adaptive_inflate_mod . . . 881
6.216 MODULE quality_control_mod . . . 888
6.217 MODULE filter_mod . . . 893
6.218 MODULE location_mod . . . 903
6.219 forward operator test README . . . 904
6.220 PROGRAM PrecisionCheck . . . 906
6.221 MODULE obs_def_gps_mod . . . 910
6.222 MODULE obs_def_dew_point_mod . . . 915
6.223 MODULE obs_def_ocean_mod . . . 916
6.224 MODULE obs_def_1d_state_mod . . . 920
6.225 MODULE obs_def_radar_mod . . . 928
6.226 MODULE DEFAULT_obs_def_mod . . . 941
6.227 MODULE obs_def_mod . . . 942
6.228 MODULE obs_def_rttov_mod . . . 955
6.229 Multi-Component CESM+DART Setup . . . 963
6.230 PROGRAM create_ocean_obs . . . 967
6.231 PROGRAM dart_to_ncommas . . . 970
6.232 PROGRAM ncommas_to_dart . . . 973
6.233 Copyright . . . 975
6.234 Changelog . . . 975
6.235 404 Error . . . 996

The Data Assimilation Research Testbed (DART) is an open-source, freely available community facility for ensemble data assimilation (DA).[1] DART is developed and maintained by the Data Assimilation Research Section (DAReS) at the National Center for Atmospheric Research (NCAR).

[1] Anderson, J. L., T. Hoar, K. Raeder, H. Liu, N. Collins, R. Torn and A. Arellano, 2009 The Data Assimilation Research Testbed: A Community Facility. *Bulletin of the American Meteorological Society*, **90**, 1283-1296, doi:10.1175/2009BAMS2618.1

CHAPTER

ONE

ENSEMBLE DATA ASSIMILATION

Ensemble DA is a technique for combining observations with numerical models to estimate the state of a physical system.

It enables modelers, observational scientists, and geophysicists to:

- Generate initial conditions for forecasts.
- Create a retrospective estimate of the state of a system, a practice known as producing a *reanalysis*.
- Assess the relative value of specific observations on forecast skill, a practice known as conducting an *observing system experiment (OSE)*.
- Estimate the value of hypothetical observations in order to inform the design of an observing system, a practice known as conducting an *observing system simulation experiment (OSSE)*.
- Determine a model's systematic bias in estimating the state of a system, a practice known as diagnosing *model error*.

The DART software environment makes it easy to explore a variety of data assimilation methods and observations with different numerical models. It provides powerful, flexible DA tools that are easy to use and customize to support efficient and reliable DA applications. While DART is primarily oriented for DA research, it has also been used in operational settings.

DART includes:

- A comprehensive tutorial introducing the concepts of ensemble DA.
- Extensive documentation of its source code.
- Interfaces to a variety of models and observation sets that can be used to introduce new users or graduate students to ensemble DA.

DART is also designed to facilitate the combination of assimilation algorithms, models, and real or synthetic observations to allow increased understanding of all three. It provides a framework for developing, testing, and distributing advances in ensemble DA to a broad community of users by removing the implementation-specific peculiarities of one-off DA systems.

These tools are intended for use by the full range of geosciencies community: beginners and experts; students and teachers; national centers and university research labs.

CHAPTER

TWO

ORGANIZATION OF THE DOCUMENTATION

Because of DART's extensive scope, this documentation is detailed and carefully organized, enabling you to easily find the information you need. If you have any questions or suggestions for improvements, please contact DAReS staff by emailing dart@ucar.edu.

The documentation is partitioned into three parts:

- a user guide that explains how to install DART and perform data assimilation
- source code documentation that provides a detailed description of the programs and modules in the repository
- a comprehensive description of data assimilation theory

CHAPTER

THREE

MANHATTAN RELEASE

The current version of DART is the Manhattan release. Email dart@ucar.edu for advice if you are interested in a model which has not been converted from the previous Lanai release.

CHAPTER

FOUR

QUICK-START

DART is available through GitHub. To download the latest version of DART, use:

```
git clone https://github.com/NCAR/DART.git
```

Go into the `build_templates` directory and copy over the closest `mkmf.template._compiler.system_` file into `mkmf.template`.

Edit it to set the NETCDF directory location if not in `/usr/local` or comment it out and set $NETCDF in your environment. *This NetCDF library must have been compiled with the same compiler that you use to compile DART and must include the F90 interfaces.*

Go into `models/lorenz_63/work` and run *quickbuild.sh nompi*.

```
$ cd models/lorenz_63/work
$ ./quickbuild.sh nompi
```

If it compiles, run this series of commands to do a very basic test:

```
$ ./perfect_model_obs
$ ./filter
```

If that runs and you have Matlab installed on your system add `DART/diagnostics/matlab` to your matlab search path and run the `plot_total_err` diagnostic script while in the `models/lorenz_63/work` directory. If the output plots and looks reasonable (error level stays around 2 and doesn't grow unbounded) you have successfully installed DART and completed your first assimilation with it.

If you are planning to run one of the larger models and want to use the Lorenz 63 model as a test, run `./quickbuild.sh`. It will build filter and any other MPI-capable executables with MPI.

Important: The `mpif90` command you use must have been built with the same version of the compiler as you are using.

If any of these steps fail or you don't know how to do them, go to the DART project web page listed above for very detailed instructions that should get you over any bumps in the process.

4.1 Quick-start for developers

To create a fork of DART for your own development you will need a GitHub account.

1. fork the NCAR/DART repo on GitHub
2. clone your (new) fork to your machine - this will set up a remote named 'origin'.

```
git clone https://github.com/USERNAME/DART.git
```

where *USERNAME* is your GitHub username.

3. create a remote to point back to the NCAR/DART repo. Convention dictates that this remote should be called 'upstream'

```
git remote add upstream https://github.com/NCAR/DART.git
```

Use 'upstream' to keep your fork up to date with NCAR/DART. GitHub has documentation on working with forks.

4. Download one of the tar files (listed below) of 'large' files so you can test your DART installation.
5. If you want to contribute your work back to the DART community, create a feature branch with your work, then issue a pull request to propose changes to NCAR/DART.

There are several large files that are needed to run some of the tests and examples but are not included in order to keep the repository as small as possible. If you are interested in running *bgrid_solo*, *cam-fv*, or testing the *NCEP/prep_bufr* observation converter, you will need these files. These files are available at:

Release	Size	Filename
"Manhattan"	189M	Manhattan_large_files.tar.gz
"wrf-chem.r13172"	141M	wrf-chem.r13172_large_files.tar.gz
"Lanai"	158M	Lanai_large_files.tar.gz
"Kodiak"	158M	Kodiak_large_files.tar.gz
"Jamaica"	32M	Jamaica_large_files.tar.gz
"Hawaii"	32M	Hawaii_large_files.tar.gz

Download the appropriate tar file and untar it into your DART repository. Ignore any warnings about `tar: Ignoring unknown extended header keyword`.

CHAPTER

FIVE

CITING DART

Cite DART using the following text:

> The Data Assimilation Research Testbed (Version X.Y.Z) [Software]. (2019). Boulder, Colorado: UCAR/NCAR/CISL/DAReS. http://doi.org/10.5065/D6WQ0202

Update the DART version and year as appropriate.

CHAPTER

SIX

REFERENCES

6.1 System requirements

The DART software is intended to compile and run on many different Unix/Linux operating systems with little to no change. At this point we have no plans to port DART to Windows machines, although Windows 10 users may be interested in the free Windows Subsystem For Linux which allows developers to "run a GNU/Linux environment – including most command-line tools, utilities, and applications – directly on Windows, unmodified, without the overhead of a virtual machine" (see https://docs.microsoft.com/en-us/windows/wsl/about for more details)

Note: We have tried to make the DART code as portable as possible, but we do not have access to all compilers on all platforms, so unfortunately we cannot guarantee that the code will work correctly on your particular system.

We are genuinely interested in your experience building the system, so we welcome you to send us an email with your experiences to dart@ucar.edu.

We will endeavor to incorporate your suggestions into future versions of this guide.

Minimally, you will need:

1. a Fortran90 compiler,
2. the netCDF libraries built with the F90 interface,
3. *perl* (just about any version),
4. an environment that understands *csh*, *tcsh*, *sh*, and *ksh*
5. the long-lived Unix build tool *make*
6. and up to 1 Gb of disk space for the DART distribution.

History has shown that it is a very good idea to remove the stack and heap limits in your run-time environment with the following terminal commands:

```
> limit stacksize unlimited
> limit datasize unlimited
```

Additionally, the following tools have proven to be *nice* (but are not required to run DART):

1. ncview: a great visual browser for netCDF files.
2. the netCDF Operators (NCO): tools to perform operations on netCDF files like concatenating, slicing, and dicing
3. Some sort of MPI environment. In other words, DART does not come with *MPICH*, *LAM-MPI*, or *OpenMPI*, but many users of DART rely on these MPI distributions to run DART in a distributed-memory parallel setting. In order to use MPI with DART, please refer to the DART MPI introduction.

4. If you want to use the DART diagnostic scripts, you will need a basic MATLAB® installation. No additional toolboxes are required, and no third-party toolboxes are required.

6.2 Fortran90 compiler

The DART software is written in standard Fortran 90, with no compiler-specific extensions. It has been compiled and run with several versions of each of the following:

- GNU Fortran Compiler (known as "gfortran") (free)
- Intel Fortran Compiler for Linux and OSX
- IBM XL Fortran Compiler
- Portland Group Fortran Compiler
- Lahey Fortran Compiler
- NAG Fortran compiler
- PathScale Fortran compiler

Since recompiling the code is a necessity to experiment with different models, there are no DART binaries to distribute.

If you are unfamiliar with Fortran and/or wonder why we would choose this language, see the *Why Fortran?* discussion for more information.

6.3 Locating netCDF library

DART uses the netCDF self-describing data format for storing the results of assimilation experiments. These files have the extension *.nc* and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 netCDF interface which is available through the `netcdf.mod` and `typesizes.mod` modules and the `libnetcdf` library. Depending on the version, the `libnetcdff` library is also often required.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules).

Warning: You must build netCDF with the same compiler (including version) you plan to use for compiling DART. In practice this means that even if you have a netCDF distribution on your system, you may need to recompile netCDF in a separate location to match the compiler you will use for DART. The library and instructions for building the library or installing from a package manager may be found at the netCDF home page.

Important: The normal location for the netCDF Fortran modules and libraries would be in the `include` and `lib` subdirectories of the netCDF installation. However, different compilers or package managers sometimes place the modules and/or libraries into non-standard locations. It is required that both modules and the libraries be present.

Note: The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed later. Depending on the version of netCDF and the build options selected, the Fortran interface routines may be in a separate library named `libnetcdff.a` (note the two F's). In this case both libraries are required to build executables.

6.4 Downloading DART

The DART source code is distributed on the GitHub repository NCAR/DART with the documentation served through *readthedocs*.

Go to https://github.com/NCAR/DART and clone the repository or get the ZIP file according to your preference. See the github help page on cloning for more information on how to clone a repository. Take note of the directory you installed into, which is referred to as `DART` throughout this documentation.

To checkout the latest release of DART:

```
git clone https://github.com/NCAR/DART.git
```

If you have forked the DART repository, replace `NCAR` with your Github username.

Note: If you are interested in contributing to DART, see the *Contributors' guide* for more information. In short, you will need to be familiar with the GitHub workflow.

Unzip or clone the distribution in your desired directory, which we refer to as `DART` in this document. Compiling the code in this tree (as is usually the case) may require a large amount of additional disk space (up to the 1 Gb required for DART), so be aware of any disk quota restrictions before continuing.

6.4.1 Organization of the repository

The top level DART source code tree contains the following directories and files:

Directory	Purpose
`assimilation_code/`	assimilation tools and programs
`build_templates/`	Configuration files for installation
`developer_tests/`	regression testing
`diagnostics/`	routines to diagnose assimilation performance
`guide/`	General documentation and DART_LAB tutorials
`models/`	the interface routines for the models
`observations/`	routines for converting observations and forward operators
`theory/`	pedagogical material discussing data assimilation theory
Files	**Purpose**
`CHANGELOG.rst`	Brief summary of recent changes
`copyright.rst`	terms of use and copyright information
`README.rst`	Basic Information about DART

6.5 Compiling DART

To build DART executables you will need to

1. Create an mkmf.template with appropriate compiler and library flags.
2. Choose which model you want to use with DART, and cd into that work directory. For example, if you want to build DART for the lorenz_63 model, `cd DART/models/lorenz_63/work`
3. Build the DART executables with `./quickbuild.sh`

For more detail on how quickbuild.sh works see *DART build system*.

6.5.1 Customizing the mkmf.template file

mkmf.templates for different compilers/architectures can be found in the `DART/build_templates` directory and have names with extensions that identify the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. **Our intent is that you copy one that is similar to your system into** `DART/build_templates/mkmf.template` **and customize it.**

For the discussion that follows, knowledge of the contents of one of these templates (e.g. `DART/build_templates/mkmf.template.intel.linux`) is needed. Note that only the relevant lines of the file are shown here. The first portion of the file is a large comment block that provides valuable advice on how to customize the *mkmf* template file if needed.

```
MPIFC = mpif90
MPILD = mpif90
FC = ifort
LD = ifort
NETCDF = /usr/local
INCS = -I$(NETCDF)/include
LIBS = -L$(NETCDF)/lib -lnetcdf -lnetcdff
FFLAGS = -O2 $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

FC	the Fortran compiler
LD	the name of the loader; typically, the same as the Fortran compiler
MPIFC	the MPI Fortran compiler; see the *DART MPI introduction* for more info
MPILD	the MPI loader; see the *DART MPI introduction* for more info
NETCDF	the location of your root netCDF installation, which is assumed to contain netcdf.mod and typesizes.mod in the include subdirectory. Note that the value of the NETCDF variable will be used by the "INCS" and "LIBS" variables.
INCS	the includes passed to the compiler during compilation. Note you may need to change this if your netCDF includes netcdf.mod and typesizes.mod are not in the standard location under the include subdirectory of NETCDF.
LIBS	the libraries passed to "FC" (or "MPIFC") during compilation. Note you may need to change this if the netCDF libraries libnetcdf and libnetcdff are not in the standard location under the "lib" subdirectory of NETCDF.
FFLAGS	the Fortran flags passed to "FC" (or "MPIFC") during compilation. There are often flags used for optimized code versus debugging code. See your particular compiler's documentation for more information.
LDFLAGS	the linker flags passed to LD during compilation. See your particular linker's documentation for more information.

6.5.2 Building the Lorenz_63 DART project

In order to get started with DART, here we use the Lorenz 63 model, which is a simple ODE model with only three variables. DART supports models with many orders of magnitude more variables than three, but if you can compile and run the DART code for any ONE of the models, you should be able to compile and run DART for ANY of the models. For time-dependent filtering known as **cycling**, where observations are iteratively assimilated at multiple time steps, DART requires the ability to move the model state forward in time. For low-order models, this may be possible with a Fortran function call, but for higher-order models, this is typically done outside of DART's execution control. However, the assimilation itself is conducted the same way for **all** models. For this reason, here we focus solely on the Lorenz 63 model. If so desired, see *The Lorenz 63 model: what is it and why should we care?* for more information on this simple yet surprisingly relevant model. See *A high-level workflow of DA in DART* for further information regarding the DART workflow if you prefer to do so before building the code.

There are seven separate, stand-alone programs that are typically necessary for the end-to-end execution of a DART experiment; see below or the *What is DART?* section for more information on these programs and their interactions.

Note: some higher-order models have many more than seven programs; for example, the Weather Research and Forecasting (WRF) model, which is run operationally around the world to predict regional weather, has 28 separate programs. Nonetheless, each of these programs are built the same way.

To build all DART programs for Lorenz_63:

```
cd DART/models/lorenz_63/work
./quickbuild.sh
```

The DART executables are built in the `work` directory. If the build is successful, you will see the following seven programs in your lorenz_63 work directory.

Program	Purpose
preprocess	creates custom source code for just the observations of interest
create_obs_sequence	specify a (set) of observation characteristics taken by a particular (set of) instruments
create_fixed_network_seq	specify the temporal attributes of the observation sets
perfect_model_obs	spinup and generate "true state" for synthetic observation experiments
filter	perform data assimilation analysis
obs_diag	creates observation-space diagnostic files in netCDF format to support visualization and quantification.
obs_sequence_tool	manipulates observation sequence files. This tool is not generally required (particularly for low-order models) but can be used to combine observation sequences or convert from ASCII to binary or vice-versa. Since this is a rather specialized routine, we will not cover its use further in this document.

Note: The most common problem is that the netCDF libraries and/or include files were not found in the specified location(s). The second most common problem is that the netCDF libraries were built with a different compiler than the one used for DART. Find (or compile) a compatible netCDF library, edit the `DART/build_templates/mkmf.template` to point to the correct locations of the includes and library files, then run `./quickbuild.sh` again.

6.6 Verifying installation

Note: These verification steps require MATLAB®. UCAR Member Institutions have access to institutional licenses for MATLAB, thus we have created verification tools using it.

The Lorenz model is notoriously sensitive to very small changes; in fact, the story of Lorenz discovering this sensitivity is a classic in the annals of the study of chaos, which in turn was instrumental in the development of data assimilation as a field of study. See *The Lorenz 63 model: what is it and why should we care?* or *What is data assimilation?* for more information.

This sensitivity is of practical interest for verifying these results. The initial conditions files and observation sequences are provided in ASCII, which is portable across systems, but there may be some machine-specific round-off error in

the conversion from ASCII to machine binary. As Lorenz 63 is such a nonlinear model, extremely small differences in the initial conditions may eventually result in noticeably different model trajectories. Even different compiler flags may cause tiny differences that ultimately result in large differences. Your results should start out looking VERY SIMILAR and may diverge with time.

The simplest way to determine if the installation is successful is to run some of the functions available in `DART/diagnostics/matlab/`. If you have yet to download DART, see *Downloading DART* for instructions on how to do so. Usually, we launch MATLAB from the `DART/models/lorenz_63/work` directory. First, open your terminal and navigate to the directory where you downloaded DART. Then, execute the following command in your terminal to navigate to the `DART/models/lorenz_63/work` directory.

```
$ cd DART/models/lorenz_63/work
```

Note: The lines of code that are marked with a `$` symbol are commands that you will execute in your terminal. To execute a command, type it in exactly as seen, unless specified otherwise, and press enter.

In the case of this Lorenz model, we know the "true" (by definition) state of the model that is consistent with the observations, which is generated by running the `perfect_model_obs` program. We therefore must first run `perfect_model_obs` before launching MATLAB and running our diagnostic scripts. To do so, execute the following three commands in the terminal.

```
$ ./quickbuild.sh
$ ./perfect_model_obs
$ ./filter
```

You are now ready to launch MATLAB. To do so, execute the command `MATLABROOT/bin/matlab -nodesktop`, and replace `MATLABROOT` with the directory that MATLAB is installed in on your computer. By default, MATLAB is installed in the following locations:

```
Windows (64-bit):
C:\Program Files\MATLAB\R20XXx (64-bit MATLAB)
C:\Program Files (x86)\MATLAB\R20XXx (32-bit MATLAB)

Windows (32-bit):
C:\Program Files\MATLAB\R20XXx

Linux:
/usr/local/MATLAB/R20XXx

Mac:
/Applications/MATLAB_R20XXx.app
```

`R20XXx` must also be replaced with the version of MATLAB that you have installed. For example, the full command for a Mac user having installed the latest version of MATLAB would be `/Applications/MATLAB_R2022a.app/bin/matlab -nodesktop`.

If the corresponding MATLABROOT listed above does not work, you can identify the correct directory by opening the MATLAB application and executing the command `matlabroot`. This will display the directory where MATLAB is installed. It will have the correct version already incorporated into the MATLABROOT.

Upon successfully launching MATLAB, the MATLAB header will be outputted to your terminal as it is below.

```
$  MATLABROOT/bin/matlab -nodesktop
```

(continues on next page)

(continued from previous page)

```
                                < M A T L A B (R) >
                      Copyright 1984-2022 The MathWorks, Inc.
                    R2022a Update 1 (9.12.0.1927505) 64-bit (maci64)
                                      April 6, 2022
```

Now that you have launched MATLAB, you must now use the MATLAB *addpath* command to make the `DART/diagnostics/matlab/` functions available for execution in any working directory.

```
$ addpath ../../../diagnostics/matlab
```

You are now ready to run the diagnostics. The following MATLAB scripts (plot_total_err and plot_ens_time_series,) compare the ensemble members with the truth and can calculate the error in the assimilation. Running `plot_total_err` and `plot_ens_time_series` will produce the graphics as shown below:

Note: When prompted to input file names while running the diagnostics, simply press enter to select the default.

```
$ plot_total_err
Input name of ensemble trajectory file:
<cr> for preassim.nc

Comparing true_state.nc and
          preassim.nc

$ plot_ens_time_series
Input name of ensemble trajectory file:
<cr> for preassim.nc

Comparing true_state.nc and
          preassim.nc
Using Variable state IDs 1  2  3

pinfo =

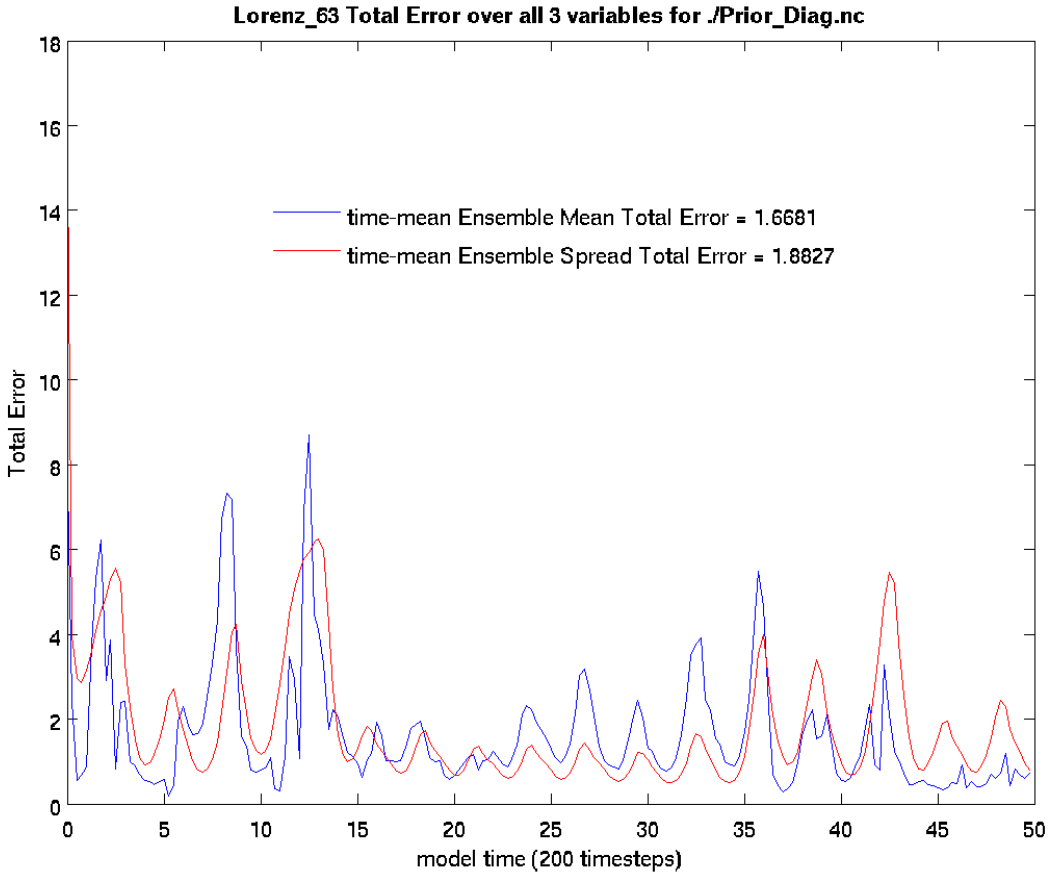
  struct with fields:

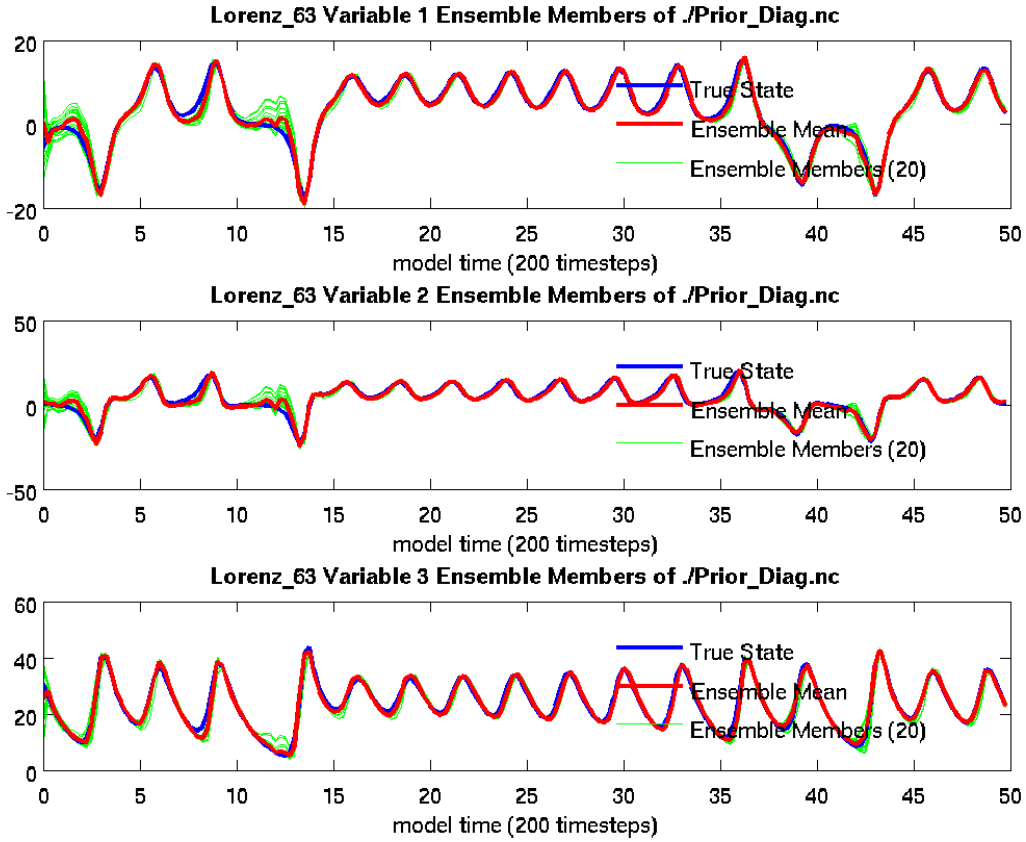
                model: 'Lorenz_63'
              def_var: 'state'
       num_state_vars: 1
           num_copies: 20
      num_ens_members: 20
     ensemble_indices: [1 2 3 ... 18 19 20]
        min_state_var: 1
        max_state_var: 3
       def_state_vars: [1 2 3]
                fname: 'preassim.nc'
           truth_file: 'true_state.nc'
           diagn_file: 'preassim.nc'
           truth_time: [1 200]
           diagn_time: [1 200]
                 vars: {'state'}
```

(continues on next page)

(continued from previous page)

```
                time: [200x1 double]
  time_series_length: 200
                 var: 'state'
            var_inds: [1 2 3]
```





From the above `plot_ens_time_series` graphic, you can see the individual green ensemble members becoming more constrained with less spread as time evolves. If your figures look similar to these, you should feel confident that everything is working as intended. Don't miss the opportunity to rotate the "butterfly" plot for that classic chaos theory experience (perhaps while saying, "life, uh, finds a way").

Congratulations! You have now successfully configured DART and are ready to begin the next phase of your interaction with DART. You may wish to learn more about:

- *What is data assimilation?* — a brief introduction to ensemble data assimilation. This section includes more information about the Lorenz 63 model and how to configure the `input.nml` file to play with DA experiments in DART using the Lorenz 63 model.
- *What is DART?* — This section includes more information about DART and a basic flow chart of the overall DART workflow.
- *How do I run DART with my model?*
- *How do I add my observations to DART?*
- *How would I use DART for teaching students or myself?*
- *How can I contribute to DART?*

Note: In the case that the above instructions had one or more issues that either did not work for you as intended or were confusing, please contact the DART software development team at dart@ucar.edu. We value your input to make getting started as smooth as possible for new DART users!

6.7 Introduction to ensemble data assimilation

Data assimilation is a powerful and widely used computational technique that has many application areas throughout mathematics and science. At a very high level, data assimilation refers to the process of merging prior *forecasts* with new *observations*, creating a new analysis that is an "optimal" blending of the two by taking into account their relative uncertainties.

The following animated graphic describes the data assimilation process at a high level:

Shown here are three ensemble members, each of which gives a different initial prediction at the time t_k. Moving these predictions forward in time to t_{k+1} will give a new forecast distribution called a *prior*.

Suppose at this time there is also an observation, which will have some uncertainty due to instrument noise, etc. Mapping each of the ensemble members to the observations with a function h and applying **Bayes' theorem** will generate an update to the prior distribution, called here the *state increment*. Adding the state increment to the ensemble members will give the new *analysis* (also known as the *posterior*) at time t_{k+1}. This process can then be repeated for each set of observations as many times as necessary. For an introduction to Bayes' theorem, see *Conditional probability and Bayes' theorem* in the Theory section.

Expanding on this somewhat, the ith ensemble member is denoted x_i at the present time step. In the above graphic, there were three ensemble members, but in general there are usually many more, typically in the range of 20-1000 depending on the application. Each member x_i can have n components which together make up the **model state**. Each member contains all the variables you want to find the best fit for at a particular time. These variables are usually physically meaningful quantities; for example, this might include the 3D values of water vapor, temperature, wind speed, etc. for an atmospheric model. These values are expected to be advanced forward in time by a model, which is why they are called the "model state."

Note: In data assimilation, the "model state" is the minimum amount of information necessary to restart the model for a new forecast.

At any particular time step there may be m **observations** available. These observations are assumed to relate to the model state and provide "real world" checks against the model forecast. A "forward operator", represented in the above diagram by h, is a relationship that computes what an observation is most likely to be given a model state. In other words, h maps between x_i and y_j, giving the "expected observation" of the jth observation given the ith ensemble member. An observation may be of the same quantity as one found in the model state at a particular location, in which case the h function mapping them is trivial and the comparison is simple. The vector y may also contain more complex derived functions of the state x (for example, radar observations of precipitation), in which case the h function that models this mapping between x (in this example precipitation) and y (in this example radar returns) may be an algorithm that is quite complicated.

In practice, observations are never 100% reliable. The observations themselves will have some uncertainty for example arising from instrument noise. The instrument noise error variances are typically published by the instrument manufacturer, and these observation errors are usually assumed to be independent as true instrument "noise" should not be correlated in time or space. Furthermore, since models have a finite resolution (i.e. they are "fuzzy"), there is almost always an error that arises when comparing the model to the observations. This is called the **representativeness error**. Put together, the potential "likelihood" of the possible values of the observation forms the **observational error distribution** in the above graphic.

Finally, note that in real-world applications there are typically many fewer observations than state variables, i.e. m is typically **much much less** than n. In practice this means that the observations alone cannot be relied upon to predict the model state; the ensemble approach with Bayes' theorem is necessary.

DART makes it easy to find the optimal solution to the above problem using an ensemble filter algorithm (the most typically used algorithm is the **Ensemble Adjustment Kalman Filter**; see *Important capabilities of DART* for more

information). The user specifies which state variables make up the x ensemble vectors, which observations make up the y vector, and the observation error variances. The ensemble of model states is assumed to be representative of the uncertainty or spread in the model state. Finally, the user tells DART how to advance the model from one forecast to the next. Once DART has this information, it can proceed with optimally blending the observations and model forecasts — in other words, performing data assimilation.

The spread of the **ensemble** informs DART of the **uncertainty** in the model state. This allows for as rich, complex, and meaningful relationships as the data contained within the ensemble itself. By default, no implicit assumptions about the relative uncertainties are required, as the data can speak for itself. Areas of large uncertainty will naturally have large spread, as the ensemble members will contain very different values at those locations, while areas of low uncertainty will naturally have low spread due to the ensemble having relatively similar values at those locations. Furthermore, relationships in space and between variables can also be meaningfully derived. Of course this means that the quality of the ensemble is crucial to the success of the DA process, as uncertainty can only be accurately quantified if the ensemble is representative of the "true" uncertainty inherent in the system. Due to the fact that a relatively small number of ensemble members are typically used, estimated correlations between two distant locations may become unreliable due to sampling error. Thus, various techniques such as **covariance localization** may be employed to improve the quality of estimated relationships and increase skill in prediction. Furthermore, the ensemble spread may sometimes be deemed "too small" or "too large" by various criteria, in which case a multiplicative or additive **inflation** or **deflation**, respectively, may be applied. In practice the ensemble method is usually far more accurate and less error-prone than the main alternative of manually specifying uncertainty by some manually-designed algorithm, and it is certainly less labor-intensive to develop.

This was a brief introduction to the important concepts of DA. For more information, see the *DART Tutorial* and the *DART_LAB Tutorial*.

6.8 The Lorenz 63 model and its relevance to data assimilation

This section describes a consequential model in the development of humanity's understanding of the limits of predicting nature: the three-variable model of Lorenz (1963).[1] This model captures the essence of chaotic systems and will serve as an example to deepen your understanding of DART and data assimilation.

In 1963, Edward Lorenz developed a simplified three-variable model to investigate atmospheric convection. By making several simplifications to the Boussinesq approximation, the Lorenz model was derived for a single thin layer of fluid uniformly heated from below and cooled from above. The original paper has been cited over 20,000 times. The relatively simple, yet nonlinear, system of ordinary differential equations is:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= x(r - z) - y \\ \frac{dz}{dt} &= xy - bz \end{aligned}$$

Here, x is proportional to the rate of convection, y is related to the horizontal temperature variation, and z is the vertical temperature variation.

There are three constant parameters:

$$\sigma = 10, r = 28, b = 8/3$$

- σ relates to the Prandtl number
- r relates to the Rayleigh number

[1] Lorenz, Edward N., 1963: Deterministic Nonperiodic Flow. *Journal of the Atmospheric Sciences*, **20**, 130-141, doi:0.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2

- b relates to the physical dimensions of the layer

Note that two of the equations have nonlinear terms: $\frac{dy}{dt}$ has the $-xz$ term and $\frac{dz}{dt}$ has the xy term.

Lorenz 63 is a consequential model in the history of science because the numerical investigation of the **chaos** arising from this system of ordinary differential equations unexpectedly launched a revolution in humanity's understanding of nature. These investigations lead to numerous mathematical and scientific breakthroughs.

While the chaotic nature of certain systems such as the three-body problem had been investigated previously, it was the electronic computer, which could compute thousands of calculations per second, that allowed these ideas to be formalized.

In particular, Lorenz's model made it clear for the first time how an infinitesimally small change in the initial conditions of a system could end up having a dramatic effect on the subsequent behavior of the system. Lorenz discussed the strange behavior of this model in a popular science lecture, *The Essence of Chaos*[2]:

> At one point I decided to repeat some of the computations in order to examine what was happening in greater detail. I stopped the computer, typed in a line of numbers that it had printed out a while earlier, and set it running again. I went down the hall for a cup of coffee and returned after about an hour, during which the computer had simulated about two months of weather. The numbers being printed out were nothing like the old ones. I immediately suspected a weak vacuum tube or some other computer trouble, which was not uncommon, but before calling for service I decided to see just where the mistake had occurred, knowing that this could speed up the servicing process. Instead of a sudden break, I found that the new values at first repeated the old ones, but soon afterward had differed by one and then several units in the last decimal place…The numbers I had typed in were not the exact original numbers, but were the rounded-off values that appeared in the original printout. The initial round-off errors were the culprits; they were steadily amplifying until they dominated the solution. In today's terminology, there was chaos.

Lorenz discovered that even in a model with just three variables, a very small change in the initial conditions (in this case, the numbers he typed back into the computer, which were very slightly different from the original numbers) could cause the entire large-scale behavior to change. Lorenz's discovery has many important practical implications:

1. If tiny changes can grow to dominate a system, it is no longer possible to find the one set of "perfect" initial conditions and hope to allow the system to run forever with perfect forecasts. Instead, forecasting chaotic systems must be approached **statistically**.
2. There is a practical limit of predictability inherent in chaotic systems. In other words, the nonlinear dynamics of a chaotic model are inherently difficult to predict. Multiple evaluations (an **ensemble**) can be run with different plausible initial conditions to quantify this error growth.
3. In order to forecast chaotic systems effectively, periodic observations of the state are required to effectively guide the forecast and narrow the uncertainty. Since in real-world applications observations are almost always sparse compared to the number of state variables, merging observations and forecasts (i.e. **data assimilation**) is required to effectively forecast chaotic systems.

While Lorenz 63 is a simple example of a chaotic system, there are many other chaotic systems of real practical interest in areas such as weather prediction, climate, oceanography, hydrology, ecology, biology, and many other disciplines.

In short, while the Lorenz model is a simple set of equations that can easily be run on even the most basic of computers today, it is representative of the same problem of predictability that can be found throughout science. DART supports the investigation of forecasting chaotic systems in *any* field where periodic observations can be used to constrain the uncertainty using an ensemble.

[2] Lorenz, Edward N. *The Essence of Chaos*. University of Washington Press, 1995.

6.8.1 References

6.9 Data assimilation in DART using the Lorenz 63 model

In this section we open the "black box" of the Lorenz model that was previously used in *Compiling DART*. This section assumes you have successfully run the Lorenz 63 model with the example observation files that were distributed with the DART repository. In this section you will learn in more detail how DART interacts with the Lorenz 63 model to perform data assimilation.

6.9.1 The input.nml namelist

The `DART/models/lorenz_63/work/input.nml` file is the Lorenz model *namelist*, which is a standard Fortran method for passing parameters from a text file into a program without needing to recompile. There are many sections within this file that drive the behavior of DART while using the Lorenz 63 model for assimilation. Within `input.nml`, there is a section called *model_nml*, which contains the model-specific parameters:

```
&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600,
   solver = 'RK2'
   /
```

Here, you can see the values for the parameters `sigma`, `r`, and `b` that were discussed in the previous section. These are the original values Lorenz used in the 1963 paper to create the classic butterfly attractor.

6.9.2 The Lorenz 63 model code

The Lorenz 63 model code, which is under `DART/models/lorenz_63/model_mod.f90`, contains the lines:

```
subroutine comp_dt(x, dt)

real(r8), intent( in) ::  x(:)
real(r8), intent(out) :: dt(:)

! compute the lorenz model dt from standard equations

dt(1) = sigma * (x(2) - x(1))
dt(2) = -x(1)*x(3) + r*x(1) - x(2)
dt(3) = x(1)*x(2) - b*x(3)

end subroutine comp_dt
```

which directly translates the above ODE into Fortran.

Note that the routine `comp_dt` does not explicitly depend on the time variable, only on the state variables (i.e. the Lorenz 63 model is time invariant).

Note: By default, the `model_mod.f90` follows the Lorenz 63 paper to use the Runge-Kutta 2 scheme (otherwise known as RK2 or the midpoint scheme) to advance the model.

Since the Lorenz 63 model is time invariant, the RK2 code to advance the ODE in time can be written as follows, again following the Lorenz 63 paper, for a `fract` fraction of a time-step (typically equal to 1):

```
!-----------------------------------------------------------------
!> does single time step advance for lorenz convective 3 variable model
!> using two step rk time step

subroutine adv_single(x, fract)

real(r8), intent(inout) :: x(:)
real(r8), intent(in)    :: fract

real(r8) :: x1(3), x2(3), dx(3)

call comp_dt(x, dx)            !  compute the first intermediate step
x1 = x + fract * deltat * dx

call comp_dt(x1, dx)           !  compute the second intermediate step
x2 = x1 + fract * deltat * dx

!  new value for x is average of original value and second intermediate

x = (x + x2) / 2.0_r8

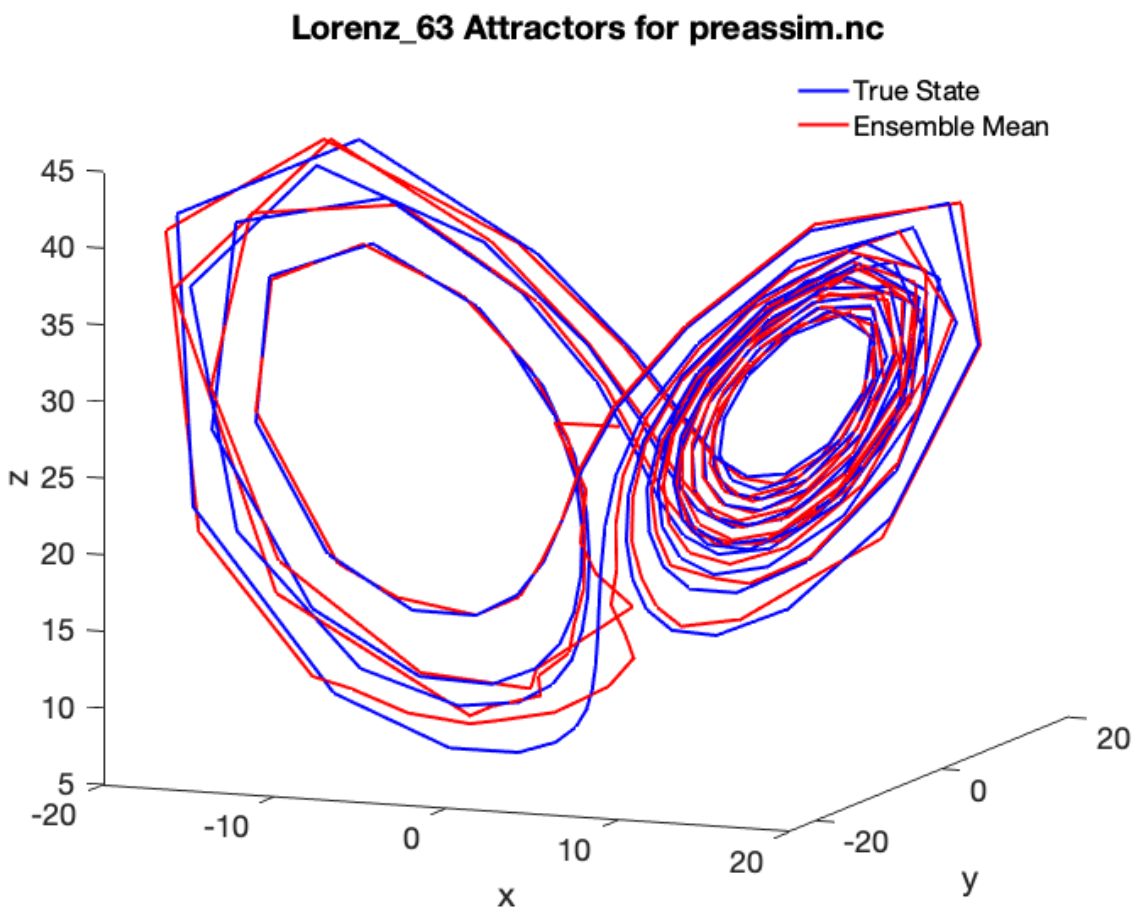
end subroutine adv_single
```

Together, these two code blocks describe how the Lorenz 63 model is advanced in time. You will see how DART uses this functionality shortly.

6.9.3 The model time step and length of the data assimilation

In the original Lorenz 63 paper, the model is run for 50 "days" using a non-dimensional time-step of 0.01, which is reproduced in the namelist above. This time-step was assumed equal to 3600 seconds, or one hour, in dimensional time. This is also set in the namelist above. The Lorenz 63 model observation file included with the DART repository uses observations of all three state variables every six hours (so every six model steps) to conduct the assimilation.

If you were previously able to run the Matlab diagnostic scripts, you may have noticed that the butterfly attractor for the included example does not look as smooth as might be desired:



This is because the model output was only saved once every six "hours" at the observation times. As an exercise, let's make a nicer-looking plot using the computational power available today, which even on the most humble of computers is many times greater than what Lorenz had in 1963. Let's change Lorenz's classic experiment to the following:

1. Make the non-dimensional timestep 0.001, a factor of 10 smaller, which will correspond to a dimensional timestep of 360 seconds (6 minutes). This smaller time-step will lead to a smoother model trajectory.
2. Keep the original ratio of time steps to observations included in the DART repository of assimilating observations every six time steps, meaning we now need observations every 36 minutes.

Therefore, in order to conduct our new experiment, we will need to regenerate the DART observation sequence files.

To change the time-step, change the `input.nml` file in `DART/models/lorenz_63/work` to the following:

```
&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.001,
   time_step_days = 0,
   time_step_seconds = 360
   /
```

Note: The changes are to `deltat` and `time_step_seconds`. Additionally: you do not need to recompile the DART

code as the purpose of namelist files is to pass run-time parameters to a Fortran program without recompilation.

6.9.4 Updating the observation sequence

Let's now regenerate the DART observation files with the updated timestep and observation ratio. In a typical large-scale application, the user will provide observations to DART in a standardized format called the *Observation Sequence* file. Since there are no real observations of the Lorenz 63 system, we must create our own synthetic observations - which may be done using *create_obs_sequence*, *create_fixed_network_seq*, and *perfect_model_obs* programs; each of which we will explain below. These helpful interactive programs are included with DART to generate these observation sequence files for typical research or education-oriented experiments. In such setups, observations (with noise added) will be generated at regular intervals from a model "truth". This "truth" will only be available to the experiment through the noisy observations but can later be used for comparison purposes. The number of steps necessary for the ensemble members to reach the true model state's "attractor" can be investigated and, for example, compared between different DA methods. This is an example of an "OSSE" — see *High-level data assimilation workflows in DART* for more information.

The three programs used in this example to create an observation sequence again are *create_obs_sequence*, *create_fixed_network_seq*, and *perfect_model_obs*. *create_obs_sequence* creates a template for the observations, *create_fixed_network_seq* repeats that template at multiple times, and finally *perfect_model_obs* harvests the observation values. These programs have many additional capabilities; if interested, see the corresponding program's documentation.

Let's now run the DART program *create_obs_sequence* to create the observation template that we will later replicate in time:

```
# Make sure you are in the DART/models/lorenz_63/work directory ./create_obs_sequence
```

The program *create_obs_sequence* will ask for the number of observations. Since we plan to have 3 observations at each time step (one for each of the state variables), input **3**:

```
set_nml_output Echo NML values to log file only

------------------------------------------------------------
-------------- ASSIMILATE_THESE_OBS_TYPES --------------
  RAW_STATE_VARIABLE
------------------------------------------------------------
-------------- EVALUATE_THESE_OBS_TYPES   --------------
  none
------------------------------------------------------------
---------- USE_PRECOMPUTED_FO_OBS_TYPES   --------------
  none
------------------------------------------------------------

Input upper bound on number of observations in sequence
3
```

For this experimental setup, we will not have any additional copies of the data, nor will we have any quality control fields. So use **0** for both.

```
Input number of copies of data (0 for just a definition)
0
Input number of quality control values per field (0 or greater)
0
```

We now will setup each of the three observations. The program asks to enter -1 if there are no additional observations, so input anything else instead (**1** below). Then enter **-1**, **-2**, and **-3** in sequence for the state variable index (the observation here is just the values of the state variable). Use **0 0** for the time (we will setup a regularly repeating observation after we finish this), and **8** for the error variance for each observation.

Finally, after inputting press enter to use the default output file `set_def.out`.

Input your values as follows:

```
input a -1 if there are no more obs
1
    Input -1 * state variable index for identity observations
    OR input the name of the observation kind from table below:
    OR input the integer index, BUT see documentation...
                1 RAW_STATE_VARIABLE
-1
input time in days and seconds (as integers)
0 0
Input the error variance for this observation definition
8
input a -1 if there are no more obs
1
    Input -1 * state variable index for identity observations
    OR input the name of the observation kind from table below:
    OR input the integer index, BUT see documentation...
                1 RAW_STATE_VARIABLE
-2
input time in days and seconds (as integers)
0 0
Input the error variance for this observation definition
8
input a -1 if there are no more obs
1
    Input -1 * state variable index for identity observations
    OR input the name of the observation kind from table below:
    OR input the integer index, BUT see documentation...
                1 RAW_STATE_VARIABLE
-3
input time in days and seconds (as integers)
0 0
Input the error variance for this observation definition
8
Input filename for sequence (<return> for set_def.out )

write_obs_seq  opening formatted observation sequence file "set_def.out"
write_obs_seq  closed observation sequence file "set_def.out"
create_obs_sequence Finished successfully.
```

6.9.5 Creating a regular sequence of observations

We will now utilize another DART program that takes this `set_def.out` file as input. The interactive program `create_fixed_network_seq` is a helper tool that can be used to generate a DART observation sequence file made of a set of regularly repeating observations.

```
# Make sure you are in the DART/models/lorenz_63/work directory ./create_fixed_network_seq
```

We want to use the default `set_def.out` file, so press return. We also want a regularly repeating time sequence, so input **1**.

```
set_nml_output Echo NML values to log file only

------------------------------------------------------------
-------------- ASSIMILATE_THESE_OBS_TYPES --------------
  RAW_STATE_VARIABLE
------------------------------------------------------------
-------------- EVALUATE_THESE_OBS_TYPES   --------------
  none
------------------------------------------------------------
---------- USE_PRECOMPUTED_FO_OBS_TYPES   --------------
  none
------------------------------------------------------------

Input filename for network definition sequence (<return> for set_def.out  )

To input a regularly repeating time sequence enter 1
To enter an irregular list of times enter 2
1
```

We now will input the number of observations in the file. The purpose of this exercise is to refine the time step used by Lorenz in 1963 by a factor of 10. Since we want to keep the ratio of six model steps per observation and run for 50 days, we will need 2000 model observations (360 seconds × 6 × 2000 = 50 days).

As we specified in `set_def.out`, there are 3 observations per time step, so a total of 6000 observations will be generated.

Note: The Lorenz 63 model dimensional time-step is related to the observational time *only* through this mechanism. In other words, `deltat` in the namelist could relate to virtually any dimensional time step `time_step_seconds` if the observation times were not considered. However, DART will automatically advance the model state to the observation times in order to conduct the data assimilation at the appropriate time, then repeat this process until no additional observations are available, thus indirectly linking `deltat` to `time_step_seconds`.

Enter **2000** for the number of observation times. The initial time will be **0 0**, and the input period will be **0** days and **2160** seconds (36 minutes).

```
Input number of observation times in sequence
2000
Input initial time in sequence
input time in days and seconds (as integers)
0 0
Input period of obs in sequence in days and seconds
0 2160
```

The numbers 1 to 2000 will then be output by `create_fixed_network_seq`. Press return to accept the default output name of `obs_seq.in`. The file suffix is `.in` as this will be the input to the next program, *perfect_model_obs*.

```
   1
   2
...
1998
1999
2000
What is output file name for sequence (<return> for obs_seq.in)

  write_obs_seq  opening formatted observation sequence file "obs_seq.in"
  write_obs_seq  closed observation sequence file "obs_seq.in"
  create_fixed_network_seq Finished successfully.
```

6.9.6 Running perfect_model_obs

We are now ready to run *perfect_model_obs*, which will read in `obs_seq.in` and generate the observations as well as create the "perfect" model trajectory. "Perfect" here is a synonym for the known "true" state which is used to generate the observations. Once noise is added (to represent observational uncertainty), the output is written to `obs_seq.out`.

```
# Make sure you are in the DART/models/lorenz_63/work directory
./perfect_model_obs
```

The output should look like the following:

```
set_nml_output Echo NML values to log file only
 initialize_mpi_utilities: Running single process

-------------------------------------------------------
-------------- ASSIMILATE_THESE_OBS_TYPES --------------
   RAW_STATE_VARIABLE
-------------------------------------------------------
-------------- EVALUATE_THESE_OBS_TYPES   --------------
   none
-------------------------------------------------------
---------- USE_PRECOMPUTED_FO_OBS_TYPES   --------------
   none
-------------------------------------------------------

 quality_control_mod: Will reject obs with Data QC larger than    3
 quality_control_mod: No observation outlier threshold rejection will be done
 perfect_main  Model size =                    3
 perfect_read_restart: reading input state from file
 perfect_main  total number of obs in sequence is         6000
 perfect_main  number of qc values is            1

 perfect_model_obs: Main evaluation loop, starting iteration    0
 move_ahead Next assimilation window starts    at:  day=       0 sec=     0
 move_ahead Next assimilation window ends      at:  day=       0 sec=   180
 perfect_model_obs: Model does not need to run; data already at required time
 perfect_model_obs: Ready to evaluate up to       3 observations
```

(continues on next page)

(continued from previous page)

```
 perfect_model_obs: Main evaluation loop, starting iteration    1
 move_ahead Next assimilation window starts    at:  day=       0 sec=  1981
 move_ahead Next assimilation window ends      at:  day=       0 sec=  2340
 perfect_model_obs: Ready to run model to advance data ahead in time
 perfect_model_obs: Ready to evaluate up to       3 observations

...

perfect_model_obs: Main evaluation loop, starting iteration 1999
move_ahead Next assimilation window starts    at:  day=      49 sec= 84061
move_ahead Next assimilation window ends      at:  day=      49 sec= 84420
perfect_model_obs: Ready to run model to advance data ahead in time
perfect_model_obs: Ready to evaluate up to       3 observations

perfect_model_obs: Main evaluation loop, starting iteration 2000
perfect_model_obs: No more obs to evaluate, exiting main loop
perfect_model_obs: End of main evaluation loop, starting cleanup
write_obs_seq  opening formatted observation sequence file "obs_seq.out"
write_obs_seq  closed observation sequence file "obs_seq.out"
```

You can now see the files `true_state.nc`, a netCDF file which has the perfect model state at all 2000 observation times; `obs_seq.out`, an ASCII file which contains the 6000 observations (2000 times with 3 observations each) of the true model state with noise added in; and `perfect_output.nc`, a netCDF file with the final true state that could be used to "restart" the experiment from the final time (49.75 days in this case).

We can now see the relationship between `obs_seq.in` and `obs_seq.out`: `obs_seq.in` contains a "template" of the desired observation locations and types, while `obs_seq.out` is a list of the actual observation values, in this case generated by the *perfect_model_obs* program.

Important: `create_obs_seq` is used for this low-order model because there are no real observations for Lorenz 63. For systems that have real observations, DART provides a variety of *observation converters* available to convert from native observation formats to the DART format. See *Available observation converter programs* for a list.

6.9.7 Running the filter

Now that `obs_seq.out` and `true_state.nc` have been prepared, DART can perform the actual data assimilation. This will generate an ensemble of model states, use the ensemble to estimate the prior distribution, compare to the "expected" observation of each member, and update the model state according to Bayes' rule.

Make sure you are in the DART/models/lorenz_63/work directory ./filter

```
set_nml_output Echo NML values to log file only
initialize_mpi_utilities: Running single process

-------------------------------------------------------------
-------------- ASSIMILATE_THESE_OBS_TYPES --------------
  RAW_STATE_VARIABLE
-------------------------------------------------------------
-------------- EVALUATE_THESE_OBS_TYPES   --------------
  none
-------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
---------- USE_PRECOMPUTED_FO_OBS_TYPES   --------------
  none
-------------------------------------------------------

quality_control_mod: Will reject obs with Data QC larger than    3
quality_control_mod: No observation outlier threshold rejection will be done
assim_tools_init: Selected filter type is Ensemble Adjustment Kalman Filter (EAKF)
assim_tools_init: The cutoff namelist value is     1000000.000000
assim_tools_init: ... cutoff is the localization half-width parameter,
assim_tools_init: ... so the effective localization radius is     2000000.000000
filter_main: running with an ensemble size of    20
parse_stages_to_write:  filter will write stage : preassim
parse_stages_to_write:  filter will write stage : analysis
parse_stages_to_write:  filter will write stage : output
set_member_file_metadata no file list given for stage "preassim" so using default names
set_member_file_metadata no file list given for stage "analysis" so using default names
Prior inflation: None
Posterior inflation: None
filter_main: Reading in initial condition/restart data for all ensemble members from␣
↪file(s)

filter: Main assimilation loop, starting iteration    0
move_ahead Next assimilation window starts    at:  day=       0 sec=     0
move_ahead Next assimilation window ends      at:  day=       0 sec=   180
filter: Model does not need to run; data already at required time
filter: Ready to assimilate up to       3 observations
comp_cov_factor: Standard Gaspari Cohn localization selected
filter_assim: Processed       3 total observations

filter: Main assimilation loop, starting iteration    1
move_ahead Next assimilation window starts    at:  day=       0 sec= 21421
move_ahead Next assimilation window ends      at:  day=       0 sec= 21780
filter: Ready to run model to advance data ahead in time
filter: Ready to assimilate up to       3 observations
filter_assim: Processed       3 total observations
...

filter: Main assimilation loop, starting iteration  199
move_ahead Next assimilation window starts    at:  day=      49 sec= 64621
move_ahead Next assimilation window ends      at:  day=      49 sec= 64980
filter: Ready to run model to advance data ahead in time
filter: Ready to assimilate up to       3 observations
filter_assim: Processed       3 total observations

filter: Main assimilation loop, starting iteration  200
filter: No more obs to assimilate, exiting main loop
filter: End of main filter assimilation loop, starting cleanup
write_obs_seq  opening formatted observation sequence file "obs_seq.final"
write_obs_seq  closed observation sequence file "obs_seq.final"
```

Based on the default Lorenz 63 `input.nml` namelist for *filter* included in the DART repository, the assimilation will have three stages:

1. The *preassim* stage, where the ensemble is updated by advancing the model. The file`preassim.nc`, which contains the pre-assimilation model trajectories for all the ensemble members, will be written.
2. The *analysis* stage, where the data assimilation is conducted. The post-assimilation model trajectories for all the ensemble members will be written to `analysis.nc`
3. The *output* stage, which writes the file `obs_seq.final` containing the actual observations as assimilated plus the ensemble forward-operator expected values and any quality-control values. This stage also writes the `filter_output.nc` file containing the ensemble state from the final cycle, which could be used to restart the experiment.

DART has now successfully assimilated our updated observations with a 6 minute model time step and assimilation every 36 minutes. *:tada:*

6.9.8 Verifying the nicer-looking results

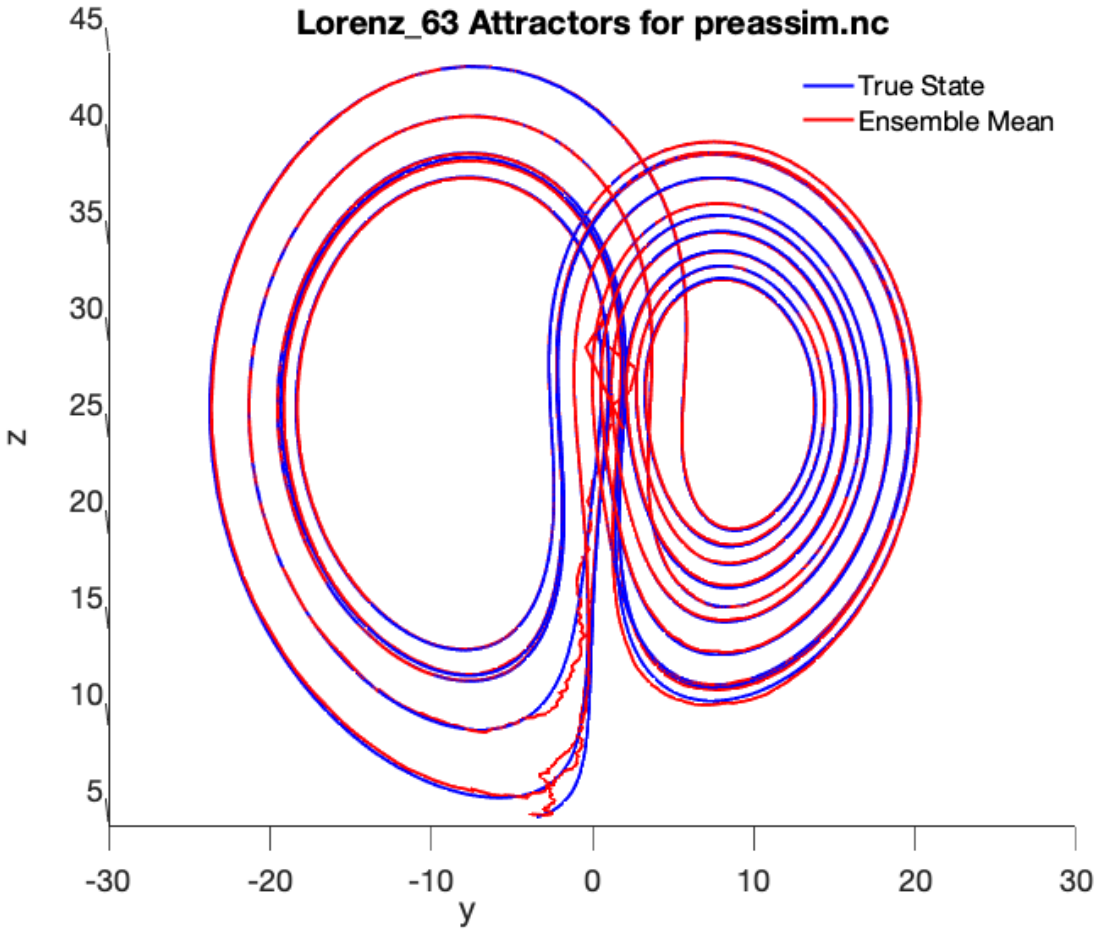
You can now run the verification scripts (as in the section *Verifying installation*) in Matlab with the following commands:

```
>> addpath ../../../diagnostics/matlab
>> plot_ens_time_series
```

Some additional commands to view the attractor from the ZY plane were used:

```
>> set(findall(gca, 'Type', 'Line'),'LineWidth',2);
>> set(gca,'FontSize',18)
>> xlabel('x')
>> ylabel('y')
>> zlabel('z')
>> view([90 0])
```

We can now see the following smooth Lorenz 63 true state and ensemble mean comparison with a 6 minute model time step and assimilation every 36 minutes:



As you can see, the ensemble mean in red matches the true state almost exactly, although it took a number of assimilation cycles before the blue ensemble mean was able to reach the red true state "attractor."

You should now be able to tinker with the Lorenz 63 model and other models in DART. For more detailed information on the theory of ensemble data assimilation, see the *DART Tutorial*. For more concrete information regarding DART's algorithms and capabilities, see the next section *The benefits of using DART*. To add your own model to DART, see *Assimilation in a complex model*. Finally, if you want to add your own observations to DART, see *Adding your observations to DART*.

6.10 What is DART?

The Data Assimilation Research Testbed (DART) is an open-source community facility that provides software tools for data assimilation research, development, and education. Using DART's carefully engineered ensemble data assimilation algorithms and diagnostic tools, atmospheric scientists, oceanographers, hydrologists, chemists, and other geophysicists can construct state-of-the-art data assimilation systems with unprecedented ease.

In this section we will introduce DART in further detail. This includes:

1. *The benefits of using DART*
2. *A brief history of DART*
3. *High-level data assimilation workflows in DART*
4. *DART's design philosophy*

5. *Important capabilities of DART*

6.11 The benefits of using DART

A common pitfall for graduate students and professionals alike is to look at the simplicity of data assimilation, in particular ensemble data assimilation, and decide they can easily write their own DA system. Indeed, this is true. After learning of the core algorithms, a talented programmer using their favorite language could write a functional DA system in a manner of weeks if not days. However, he or she will soon find that while the core of DA systems are easy to write, the more "real" the system needs to be, the more complex it will become. Writing a parallel DA system that can efficiently utilize multiple cores with MPI is not straight-forward, and adding covariance localization, observation operators, multiple models, and auxiliary tools such as quality control and pre-processing will quickly dwarf the amount of core DA code, not to mention the headaches involved in supporting multiple computing environments, compilers, etc.

DART employs a modular programming approach to apply an algorithm to move the underlying models toward a state that is more consistent with information from a set of observations. Models may be swapped in and out, as can different DA algorithms. The method requires running multiple instances of a model to generate an ensemble of states. A forward operator appropriate for the type of observation being assimilated is applied to each of the states to generate the model's estimate of the observation.

DART remains the top choice for scientists, educators, and mathematicians seeking mature and robust ensemble DA solutions without reinventing the wheel. Here are some of the many benefits of using DART:

1. DART is **freely available, open source, and released under the** Apache 2.0 License **.** In short this means that you are granted a copyright license stating you are free to use, modify, and redistribute any derivative works derived from the DART system provided that you maintain the license and copyright information. Of course, we also ask that you credit DART in your publications, and kindly ask that you contribute your modifications so that other users may benefit. See *How should I cite DART?* and *How can I contribute to DART?* for more information.
2. DART is **fully parallel and carefully engineered** to run on systems ranging from single-core research computers to the top performing multicore supercomputers in the world. Writing scalable parallel code is arguably the most difficult and time-consuming task in scientific computing today, but DART has already carefully implemented and tested this project, and the code is available for you to use out-of-the-box. For more information on how DART was written (and continues to be developed), see *DART's design philosophy*.
3. DART contains **numerous tools that accelerate getting started** on both research and "real-world" problems. Multiple rigorously tested inflation, localization, perturbation, and other auxiliary data assimilation algorithms are available for immediate use and testing. See *Important capabilities of DART* for more information.
4. DART **makes adding a new model straightforward**. A new model only needs to implement a list of (at most) 18 core functions or use the default behavior if applicable to take advantage of DART's mature and robust DA algorithms. A basic data assimilation system for a large model can be built in person-weeks, and comprehensive systems have been built in a few months. See *How do I run DART with my model?* for more information.
5. DART **makes it easy to add new observations** in order to test their potential beneficial impact. Incorporating new observation types only requires creating a forward operator that computes the expected value of an observation given a model's state. See *How do I add my observations to DART?* for more information.
6. DART **can be used to test new DA algorithms**. Many such algorithms have been successfully implemented, tested, and published using DART. This is not covered in this getting started guide as this is an "advanced user" functionality, so for this purpose it is best to first get in touch with the DART team at dart @ ucar.edu to make the process as smooth as possible.
7. Finally, and perhaps most importantly, DART **has world-class support** available from the DART team at NCAR. A talented team of dedicated software engineers and data assimilation scientists work together to continually improve DART and support user needs. See the About page for more information about the DART team.

6.12 A brief history of DART

The DART project was initiated in August 2001, and in 2003, the Data Assimilation Research Section (DAReS) was officially formed at NCAR. In 2004, the first officially supported version of DART was released. Consistent version control history is available back to 2005, making DART an extremely long-lived and well-supported software project. Since 2004, there have been more than a dozen releases. The first release, *Easter*, began the trend of naming the major releases after islands in alphabetical order in the following sequence:

Release	Date	Brief description
Easter	8 Mar 2004	Initial release
Fiji	29 Apr 2004	Enhanced portability; support for CAM and WRF
Guam	12 Aug 2004	New observation modules
Pre-Hawaii	20 Dec 2004	New filtering algorithms
Hawaii	28 Feb 2005	New filtering algorithms
DA Workshop 2005	13 Jun 2005	Tutorial, observation preprocessing
Pre-Iceland	20 Oct 2005	Huge expansion of real observation capability
Iceland	23 Nov 2005	Huge expansion of real observation capability
Post-Iceland	20 Jun 2006	Observation-space adaptive inflation
Pre-J	02 Oct 2006	Updated scalable filter algorithm
Jamaica	12 Apr 2007	Vertical localization, extensive MPI testing
Kodiak	30 Jun 2011	New obs types, new diagnostics, new utilities
Lanai	13 Dec 2013	Support for many new models, chemistry/aerosol types, new diagnostics, new utilities
Manhattan	15 May 2017	Native netCDF support, better scaling/performance

In September 2009, DART was featured on the cover of the Bulletin of the American Meteorological Society (BAMS):

To access the issue, see the September 2009 issue here. To read the DART article directly see the article here.

On the Publications page there are over 40 example publications that use DART, although there are many additional publications using DART not listed. The seminal BAMS paper has over 400 citations according to Google Scholar. The core algorithms used in DART have also been cited many more times. For example, the core EAKF algorithm (Anderson 2001) used in DART has over 1500 citations according to Google scholar.

6.13 High-level data assimilation workflows in DART

In this section we present two high-level data assimilation workflows that show the relevant DART programs with their inputs and outputs. These two workflows represent two different types of DA experiments typically run.

It is possible to run DART in **Observation System Simulation Experiment (OSSE)** mode. In OSSE mode, a perfect "true" model trajectory is created, and synthetic observations are generated from the "truth" with added noise. This is useful to test the theoretical capability of DA algorithms, observations, and/or models. In this document so far, we have conducted only OSSEs.

It is also possible to run DART in a more realistic **Observation System Experiment (OSE)** mode. In an OSE, there is no perfect model truth, which is similar to real-world situations where the true values of the model state will likely never be perfectly known. The observations (which again themselves are noisy and imperfect) are the only way to get a look at the "truth" that is estimated by the model state. In OSE mode, the user must provide observations to DART, which are usually from real-world observation systems (which come with all of their own idiosyncrasies and imperfections). DART can help generate ensemble perturbations, or the user can specify their own.

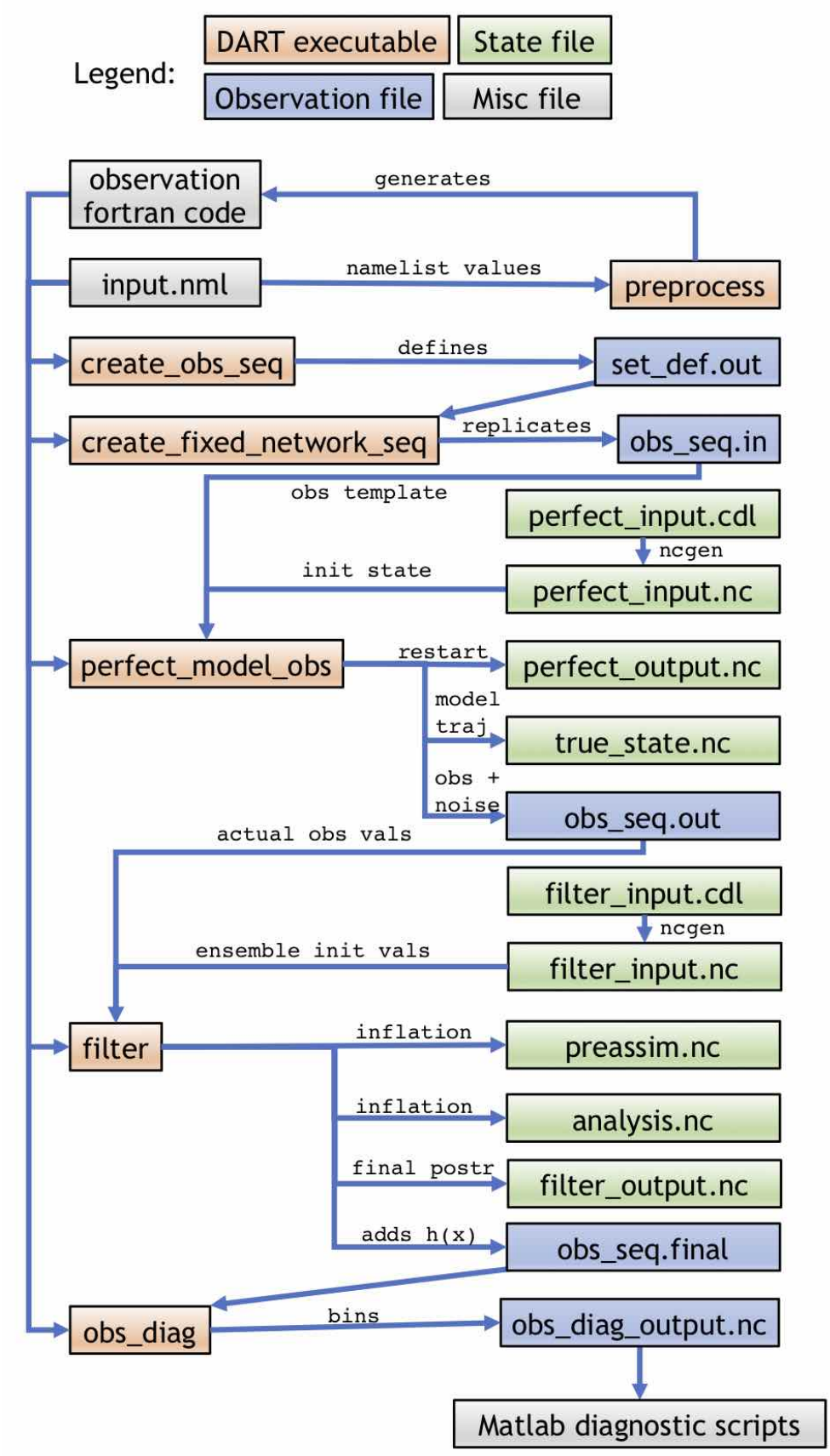
The filtering aspect is the same for both OSSE and OSE experiments, and many of the same tools for data assimilation are available in OSSE and OSE modes. The core difference, therefore, is the existence of the perfect model "truth."

For a simple model such as Lorenz 63 investigated above, DART can typically advance the model time explicitly through a Fortran function call, allowing the filtering to compute all necessary time steps in sequence without exiting the DART program. However, for larger models (or those that DART cannot communicate with through Fortran), a shell-script may be necessary to run the model and advance the time forward. For the largest models, the model state is typically advanced in parallel over many computing nodes on a supercomputer. In this more complex case, DART only considers one step at a time in order to combine the observations and the prior ensemble to find the posterior analysis, which will then be used to restart the model and continue the forecast.

For efficiency reasons, data from models with large states may be written in separate files for every ensemble member at every stage of the assimilation process. Data from models with small states may be conveniently be written as variables inside a single netCDF file.

6.13.1 Simple model workflow with an OSSE

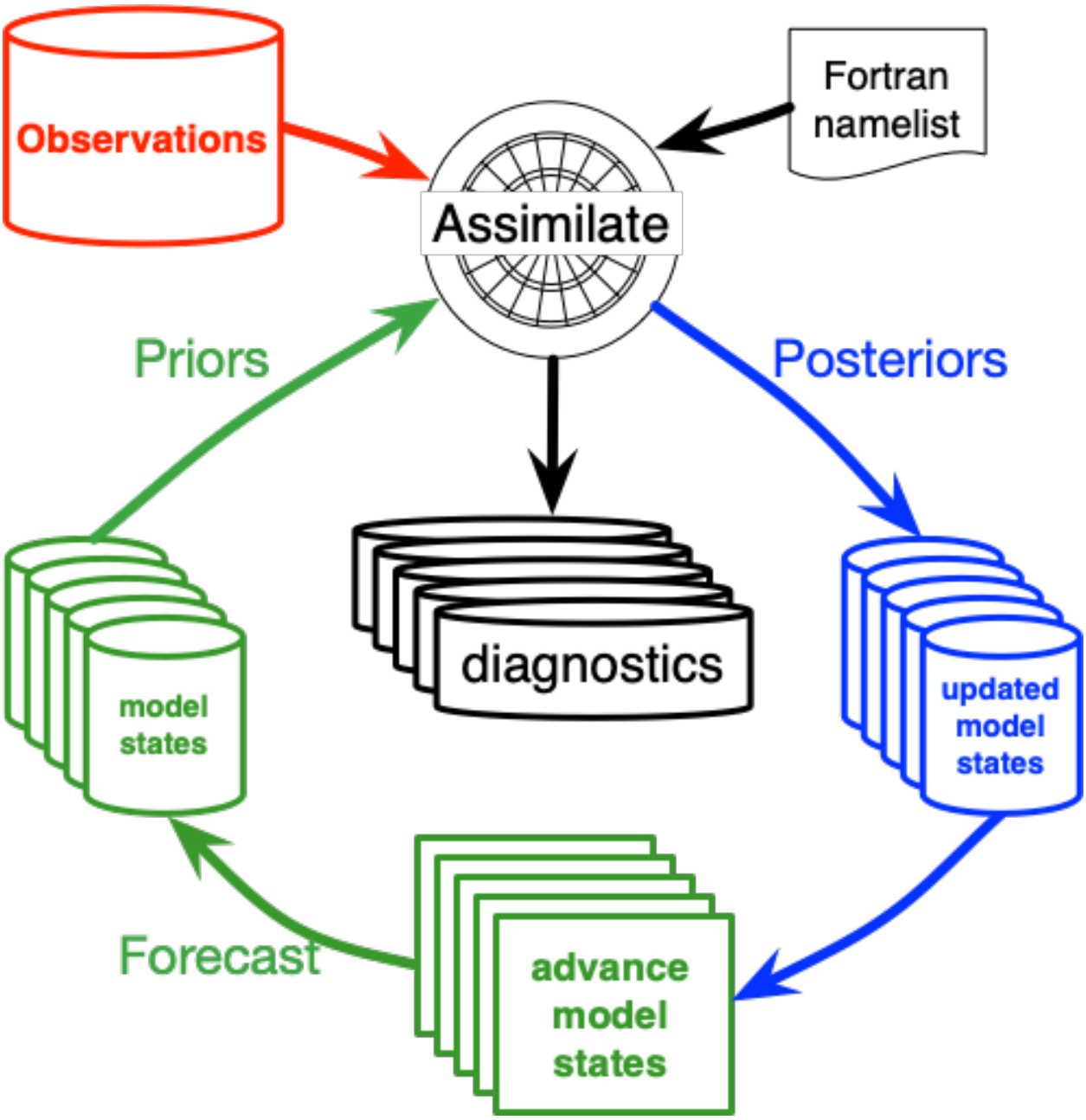
The first example DA workflow is for a model that can be advanced by DART with all ensemble members stored in a single file running an OSSE. Details of the executables mentioned below can be found in *Programs included in DART*.


Legend:
DART executable
State file
Observation file
Misc file
observation fortran code
generates
input.nml
namelist values
preprocess
create_obs_seq
defines
set_def.out
create_fixed_network_seq
replicates
obs_seq.in
obs template
perfect_input.cdl
ncgen
init state
perfect_input.nc
perfect_model_obs
restart
perfect_output.nc
model traj
true_state.nc
obs + noise
obs_seq.out
actual obs vals
filter_input.cdl
ncgen
ensemble init vals
filter_input.nc
filter
inflation
preassim.nc
inflation
analysis.nc
final postr
filter_output.nc
adds h(x)
obs_seq.final
obs_diag
bins
obs_diag_output.nc
Matlab diagnostic scripts

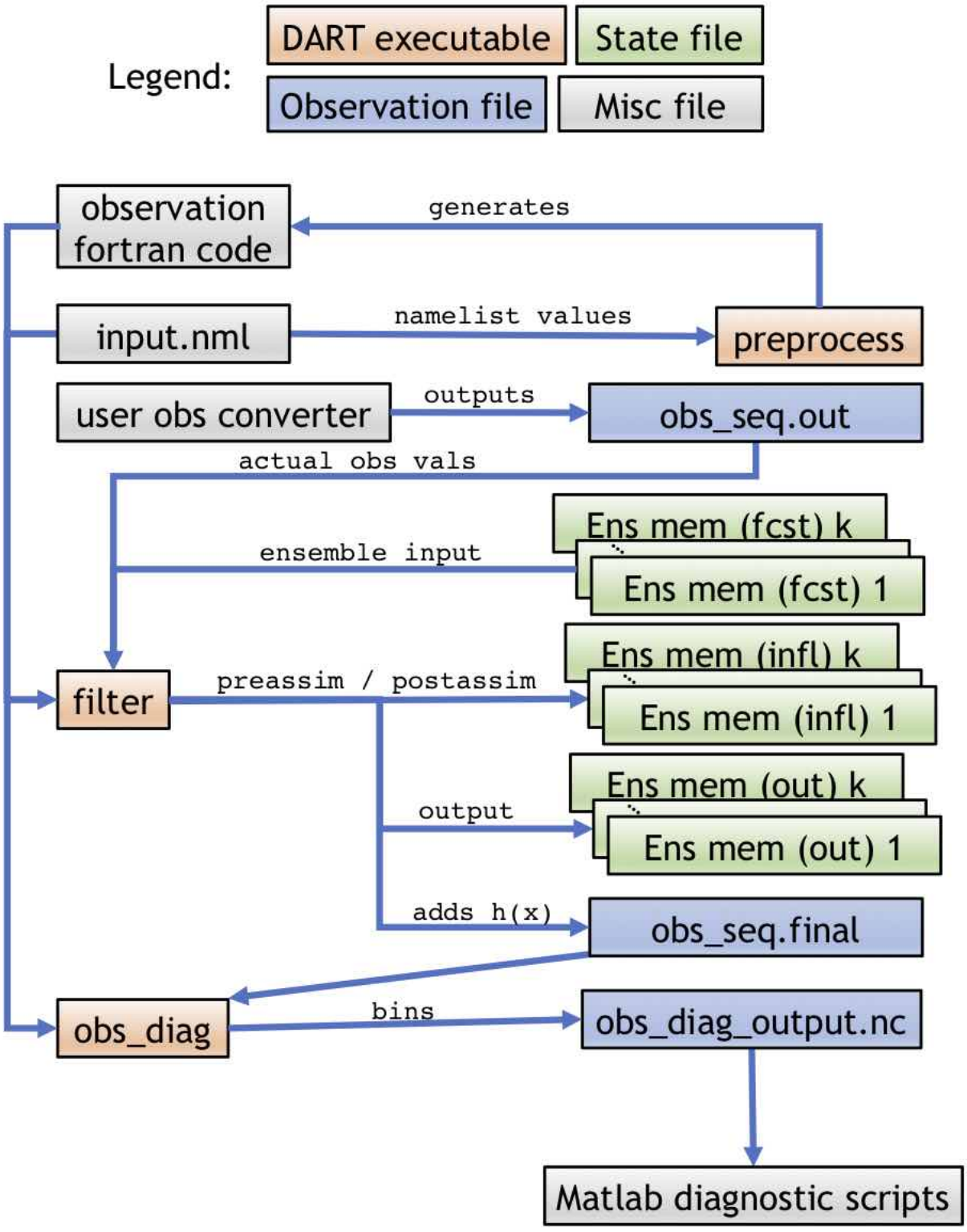
As shown, the program *preprocess* takes the `input.nml` namelist file and generates Fortran code for the observations. This code, along with the namelist, is used by all subsequent programs. *create_obs_seq* is used to define a set of observations in `set_def.out`, which can be replicated through the program *create_fixed_network_seq* to create a `obs_seq.in` file. There are two inputs to *perfect_model_obs*: the `obs_seq.in` file and `perfect_input.nc` (which here is generated by `perfect_input.cdl` via *ncgen*). `obs_seq.in` provides *perfect_model_obs* with the observation template (i.e. the location and type of observations), while `perfect_input.nc` provides the initial state that will be used to advance the model. On output, the "perfect" model state at the final time, which can be used as a restart for running this procedure again, will be written to `perfect_output.nc` (i.e. `perfect_output.nc` could be renamed to `perfect_input.nc` to extend the OSSE), while the entire state trajectory will be stored in `true_state.nc`. The noisy synthetic observations and noise-free truth (for verification only) will be stored in `obs_seq.out`. The observation values of `obs_seq.out` will be input to *filter* along with the `filter_input.nc` (generated by `filter_input.cdl` via *ncgen*), which contains the initial state for all the ensemble members. The output of *filter* is `preassim.nc`, which contains the prior state for all the ensemble members just before applying DA (so including prior inflation if it is being used); `analysis.nc`, which contains the posterior state for all the ensemble members after assimilation (and including inflation if it is being used); `filter_output.nc`, which is the final posterior that could be used to restart the OSSE process; and `obs_seq.final`, which adds the forward-calculated expected values $h(\mathbf{x})$ for each observation. The `obs_seq.final` file can be analyzed and binned by the *obs_diag* program, producing the file `obs_diag_output.nc` which can be used for diagnostics.

6.13.2 Complex model workflow with an OSE

The second workflow is for a complex model with all ensemble members stored in separate files running an OSE. In this case, DART will only operate on one model output at a time. External programs will advance the model states, generate the observations, and call DART again. Details of DART's internal programs, which are mentioned below, can be found in *Programs included in DART*. The following diagram in shows the high-level DART flow in this case:



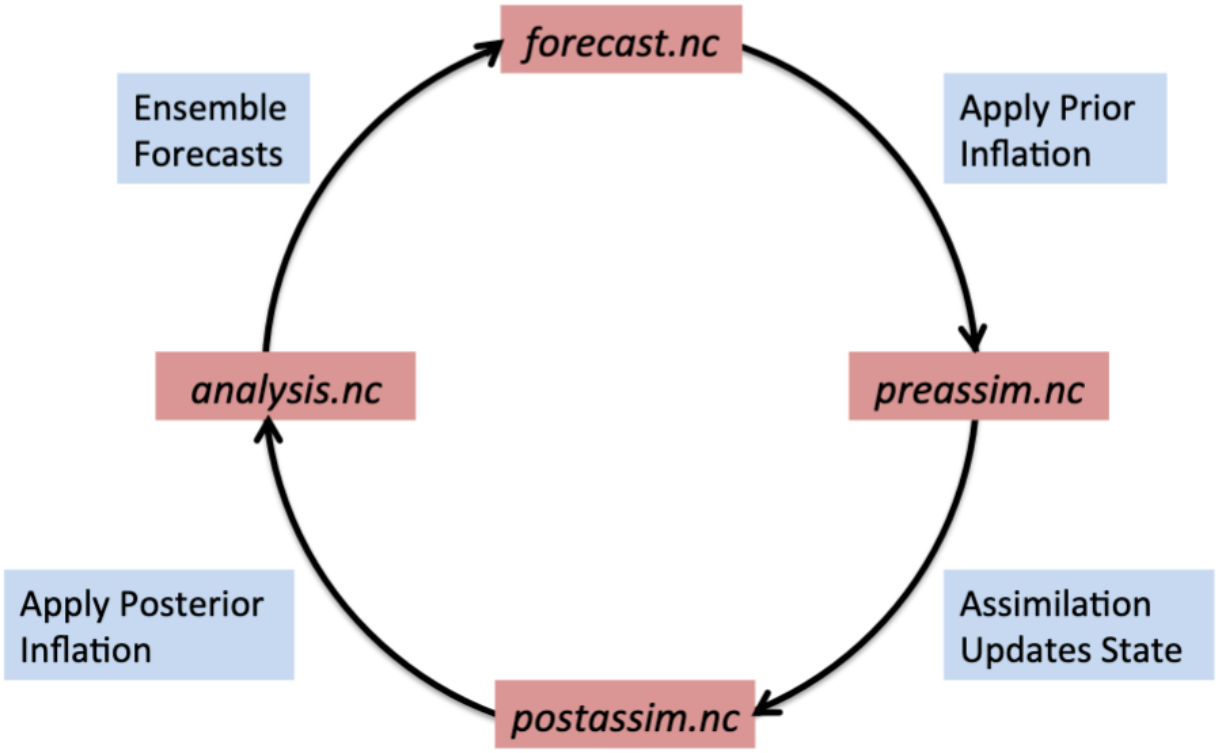
Within a single time step, DART will use the *filter* program to run the "Assimilate" portion of the above diagram and/or the "diagnostics" as follows:



The single time-step workflow for an OSE experiment within a single step is slightly simpler than the OSSE equivalent as DART handles less of the process. Like the OSSE case, the namelist and preprocessed observation source files are input to all other DART programs. In the OSE case, however, the user must provide an **obs converter** that will output a `obs_seq.out` file. There are many DART utilities to make this process easier, but for the OSE case the `obs_seq.`

`out` file is ultimately the user's responsibility (to avoid duplicating effort, see the list of existing observation types in *Important capabilities of DART*). Here, the option to run with one file for each ensemble member is demonstrated. There are k ensemble members used as input to *filter*, which also outputs k members for the prior and posterior. The `obs_seq.final` and `obs_diag_output.nc` are used in the same way as in the OSSE case. The names of the input files and output files can be controlled by the user through the `filter_input_list.txt` and `filter_output_list.txt` files, which can contain the user-specified list of the ensemble input or output files, respectively.

Another view of the stages of filter is shown in the following diagram:



As shown here, an ensemble forecast is stored in `forecast.nc` , to which prior inflation can be applied and stored in `preassim.nc`. Once assimilation is applied, the output can be stored in `postassim.nc`, and finally if posterior inflation is applied, the final analysis can be written in `analysis.nc` . The model forecast will start from the analysis to advance the model in order to start the cycle over again.

Note: The "forecast" will be the same as the "preassim" if prior inflation is not used, and the "postassim" will be the same as the "analysis" if posterior inflation is not used. The *stages_to_write* variable in the "&filter_nml" section of the `input.nml` namelist controls which stages are output to file. For a multi-file case, the potential *stages_to_write* are "input, forecast, preassim, postassim, analysis, output" while for a single file the same stages are available with the exception of "input."

Note: In the above cycling diagram, there will actually be one file per member, which is not shown here in order to simplify the process.

Important: The decision to store ensemble members as separate files and whether to run an OSSE or OSE are independent. An OSSE can be run with multiple files and an OSE can be run with all ensemble members stored in a single file.

6.14 DART's design philosophy

In this section we cover DART's design philosophy. Understanding this philosophy will make it easier to get started with DART, as you will quickly be able to predict how and where to find a particular feature of DART.

The main design goals of DART are to:

1. Create a system that is **coherent** and **easy to understand**. DART is carefully engineered to have self-contained programs that each do one job and do it well. Likewise, DART *just does DA, and does it well.*
2. Release source code that is **as compatible as possible** with the widest possible number of systems. The code is written in Fortran 90, which is one of the lowest possible common denominators available on virtually all systems. See the section *Why Fortran?* if this seems like a questionable decision to you in this modern world of Matlab, C++, Java, Python, Go, etc.
3. Strive to **limit library dependencies**. There is only one required dependency of DART: netCDF. Many modern systems have 10s or 100s of dependencies, each of which introduces complexity and the potential for bugs, lack of support, broken backwards compatibility, etc. If you've ever been frustrated struggling to debug relationships to packages you've never even heard of, you are likely to appreciate this DART design goal. Of course, there is nothing to stop you from using whatever dependencies you require, for example, to collect observations for the `obs_seq.out` in an OSE case, but DART by design will remain separate from that dependency for you and all other users.
4. Only **compile the code you need**. If you are only using a single model for your experiments, there is no reason to compile or even touch code for another model you never plan to use. Likewise, if you are not using a particular observation operator in your experiment, there is also no need to compile it or let it cause you headaches. DART recognizes this fact, and through the use of the *mkmf* utility and the *preprocess* program, only what you need will ever be compiled.
5. Use **explicit interfaces** to enforce contract programming. In practice this means that it is easy to add new models, observations operators, data assimilation algorithms, etc. as long as they can implement the required interface. This approach *allows all of the benefits of object-oriented programming without the added complexity for the end user.*
6. Provide results that are **reliable** and meaningful. The DART algorithms are carefully tested and maintained in order to be quickly published along with appropriate analysis. In a world of chaos, being able to quantify and shrink forecast uncertainty via data assimilation in a reliable way is a valuable tool for research and operations and everything in between.

In short, DART is designed at each step to make it as easy as possible for users to get up and running with their models, observations, and possibly even data assimilation algorithm advances.

6.14.1 Why Fortran?

Many users new to scientific computing such as graduate students raise their eyebrows when they first hear that a program uses Fortran for active development. Fortran is considered by many outside (and some inside) of the scientific computing community to be a dinosaur, old and decrepit, and not worthy of serious attention. However, this view is short-sighted. There is a Chinese idiom , which means "to love the new and loathe the old," indicating that just because something is old does not automatically make it bad.

While Fortran does have some outdated features that are far removed from the mainstream of software engineering (such as implicit typing by first initial of the variable), these can all be disabled, and the stylistic rules for easy-to-read, modern Fortran are always followed by DART. On the other hand, Fortran has many other attractive features that make it a top choice for modern scientific computing. In particular, Fortran offers vectorization of matrices that make it possible to operate on entire elements of an array at once or perform linear algebra operations on multi-dimensional arrays. With or without the use of the *colon operator* (:), Fortran multi-dimensional array support makes mathematical algorithms easier to read than the equivalent code written in many other languages. This highly intuitive Fortran syntax

was adopted by Matlab, NumPy, and other languages. Furthermore, for parallel programs using distributed memory in *MPI*, Fortran remains a top choice along with C and C++ when considering performance. Python code, for example, remains difficult to parallelize via MPI, not to mention the difficulties in supporting Python 2, Python 3, pip, anaconda, virtualenv, ...

Altogether, for large mathematically-oriented programs that need to be parallel, Fortran remains a top choice, especially considering the needs of DART:

1. DART does data assimilation, which is primarily mathematically-oriented operations on large data sets.
2. DART needs to be parallel with MPI to run on modern supercomputers.
3. Many users of DART are not software development professionals and appreciate straightforward and easily understandable code.
4. DART source distributions should be easy to compile and run reliably on many different systems. In practice this means avoiding software features that might not be supported on all compilers or systems.

With these considerations in mind, the choice of Fortran for DART development is clear. DART remains highly successful by keeping things simple and *not fixing what is not broken* even if it isn't shiny and new.

6.15 Important capabilities of DART

In this section we discuss the capabilities of DART that may be of interest to the user. This is a partial list of all of the functionality that is available in DART, and additional capabilities and improvements are continually being added.

As mentioned above, DART allows for both OSSE and OSE systems of models large and small. This allows users to test both theoretical limits of DA, models, and observations with idealized experiments as well as to improve actual real-world forecasts of chaotic systems with real observations.

6.15.1 Models supported by DART

A full list of models can be found *here*, but in brief the models supported by DART include:

Model	Latest version	Model	Latest version
lorenz_63	Manhattan	lorenz_84	Manhattan
lorenz_96	Manhattan	lorenz_96_2scale	Manhattan
lorenz_04	Manhattan	simple_advection	Manhattan
bgrid_solo	Manhattan	WRF	Manhattan
MPAS	Manhattan	ATM	Manhattan
ROMS	Manhattan	CESM	Manhattan
CAM-FV	Manhattan	CAM-CHEM	Manhattan
WACCM	Manhattan	WACCM-X	Manhattan
CICE	Manhattan	CM1	Manhattan
FESOM	Manhattan	NOAH-MP	Manhattan
WRF-Hydro	Manhattan	GCCOM	Lanai
LMDZ	Lanai	MITgcm_ocean	Lanai
NAAPS	Lanai	AM2	Lanai
CAM-SE	Manhattan	CLM	Manhattan
COAMPS	Lanai	COSMO	Lanai
Dynamo	Lanai	GITM	Lanai
Ikeda	Lanai	JULES	Lanai
MPAS_ocean	Lanai	null_model	Lanai
openggcm	Lanai	PARFLOW	Lanai
sqg	Lanai	TIE-GCM	Lanai
WRF-CHEM	Lanai	ECHAM	Prior to Lanai
PBL_1d	Prior to Lanai	MITgcm_annulus	Prior to Lanai
forced_barot	Prior to Lanai	pe2lyr	Prior to Lanai
ROSE	Prior to Lanai	CABLE	Prior to Lanai

The models listed as "Prior to Lanai" will take some additional work to integrate with a supported version of DART; please contact the dart @ ucar.edu team for more information. The versions listed as "Lanai" will be ported to the Manhattan version of DART depending on the needs of the user community as well as the availablity of resources on the DART team.

6.15.2 Observation converters provided by DART

Given a way to compute the expected observation value from the model state, in theory any and all observations can be assimilated by DART through the `obs_seq.out` file. In practice this means a user-defined observation converter is required. DART provides many observation converters to make this process easier for the user. Under the directory `DART/observations/obs_converters` there are multiple subdirectories, each of which has at least one observation converter. The list of these directories is as follows:

Observation	Directory	Format
Atmospheric Infrared Sounder satellite retrievals	AIRS	HDF-EOS
Advanced Microwave Sounding Unit brightness temperatures	AIRS	netCDF
Aviso: satellite derived sea surface height	Aviso	netCDF
Level 4 Flux Tower data from AmeriFlux	Ameriflux	Comma-separated text
Level 2 soil moisture from COSMOS	COSMOS	Fixed-width text
Doppler wind lidar	DWL	ASCII text
GPS retrievals of precipitable water	GPSPW	netCDF
GSI observation file	GSI2DART	Fortran binary
Global Temperature-Salinity Profile Program (GTSPP)	GTSPP	netCDF
Meteorological Assimilation Data Ingest System (MADIS)	MADIS	netCDF

continues on next page

Table 1 – continued from previous page

Observation	Directory	Format
MIDAS ionospheric obs	MIDAS	netCDF
MODIS satellite retrievals	MODIS	Comma-separated text
NCEP PREPBUFR	NCEP/prep_bufr	PREPBUFR
NCEP ASCII observations	NCEP/ascii_to_obs	NCEP text files
ROMS verification observations	ROMS	netCDF
Satellite winds from SSEC	SSEC	ASCII text
Sea surface temperature	SST	netCDF
Special Sensor Ultraviolet Spectrographic Imager (SSUSI) retrievals	SSUSI	netCDF
World Ocean Database (WOD)	WOD	World Ocean Database packed ASCII
National Snow and Ice Data Center sea ice obs	cice	Binary sea ice
VTEC Madrigal upper atmospheric obs	gnd_gps_vtec	ASCII text
GPS obs from COSMIC	gps	netCDF
Oklahoma Mesonet MDF obs	ok_mesonet	Oklahoma Mesonet MDF files
QuikSCAT scatterometer winds	quikscat	HDF 4
Radar reflectivity/radial velocity obs	Radar	WSR-88D (NEXRAD)
MODIS Snowcover Fraction obs	snow	General text
Text file (e.g. spreadsheet) obs	Text	General text
Total precipitable water from AQUA	tpw	HDF-EOS
Automated Tropical Cyclone Forecast (ATCF) obs	Tropical Cyclones	Fixed width text
LITTLE_R obs	var	little-r
MM5 3D-VAR radar obs	var	MM5 3D-VAR 2.0 Radar data files

6.15.3 Data assimilation algorithms available in DART

DART allows users to test the impact of using multiple different types of algorithms for filtering, inflation/deflation, and covariance localization.

DART offers numerous **filter algorithms**. These determine how the posterior distribution is updated based on the observations and the prior ensemble. The following table lists the filters supported in DART along with their type (set by *filter_kind* in `input.nml` under the "assim_tools_nml" section):

Filter #	Filter Name	References
1	EAKF (Ensemble Adjustment Kalman Filter)	**Anderson, J. L.**, 2001.[1] **Anderson, J. L.**, 2003.[2] **Anderson, J., Collins, N.**, 2007.[3]
2	ENKF (Ensemble Kalman Filter)	**Evensen, G.**, 2003.[4]
3	Kernel filter	
4	Observation Space Particle filter	
5	Random draw from posterior	None. **IMPORTANT**: *(contact dart @ ucar.edu before using)*
6	Deterministic draw from posterior with fixed kurtosis	None. **IMPORTANT**: *(contact dart @ ucar.edu before using)*
7	Boxcar kernel filter	
8	Rank Histogram filter	**Anderson, J. L.,** 2010.[5]
9	Particle filter	**Poterjoy, J.**, 2016.[6]

[1] Anderson, J. L., 2001: An Ensemble Adjustment Kalman Filter for Data Assimilation. *Monthly Weather Review*, **129**, 2884-2903. doi:10.1175/1520-0493(2001)129<2884:AEAKFF>2.0.CO;2

[2] Anderson, J. L., 2003: A local least squares framework for ensemble filtering. *Monthly Weather Review*, **131**, 634-642. doi:10.1175/1520-0493(2003)131<0634:ALLSFF>2.0.CO;2

[3] Anderson, J., Collins, N., 2007: Scalable Implementations of Ensemble Filter Algorithms for Data Assimilation. *Journal of Atmospheric and*

DART also has several **inflation algorithms** available for both prior (the first value in the namelist) and posterior (the second value in the namelist). The following table lists the inflation "flavors" supported in DART along with their type number (set by *inf_flavor* in `input.nml` under the "filter_nml" section):

Flavor #	Inflation flavor name	References
0	No inflation	n/a
1	(Not Supported)	n/a
2	Spatially-varying state-space (Gaussian)	**Anderson, J. L.**, 2009.[7]
3	Spatially-fixed state-space (Gaussian)	**Anderson, J. L.**, 2007.[8]
4	Relaxation to prior spread (posterior inflation only)	**Whitaker, J.S. and T.M. Hamill**, 2012.[9]
5	Enhanced spatially-varying state-space (inverse gamma)	**El Gharamti M.**, 2018.[10]

DART has the ability to correct for sampling errors in the regression caused by finite ensemble sizes. DART's sampling error correction algorithm (and localization algorithm) is described in **Anderson, J.L.**, 2012[11] Sampling error correction can be turned on or off via the *sampling_error_correction* variable in the `input.nml` under the "assim_tools_nml" section.

The following covariance localization options are available (set by *select_localization* in `input.nml` under the "cov_cutoff_nml" section):

Loc #	Localization type	References
1	Gaspari-Cohn eq. 4.10	**Gaspari, G. and Cohn, S. E.**, 1999.[12]
2	Boxcar	None
3	Ramped boxcar	None

The following image depicts all three of these options:

Oceanic Technology, **24**, 1452-1463. doi:10.1175/JTECH2049.1

[4] Evensen, G., 2003: The Ensemble Kalman Filter: Theoretical Formulation and Practical Implementation. *Ocean Dynamics*. **53(4)**, 343–367. doi:10.1007%2Fs10236-003-0036-9

[5] Anderson, J. L., 2010: A Non-Gaussian Ensemble Filter Update for Data Assimilation. *Monthly Weather Review*, **139**, 4186-4198. doi:10.1175/2010MWR3253.1

[6] Poterjoy, J., 2016: A localized particle filter for high-dimensional nonlinear systems. *Monthly Weather Review*, **144** 59-76. doi:10.1175/MWR-D-15-0163.1

[7] Anderson, J. L., 2009: Spatially and temporally varying adaptive covariance inflation for ensemble filters. *Tellus A*, **61**, 72-83, doi:10.1111/j.1600-0870.2008.00361.x

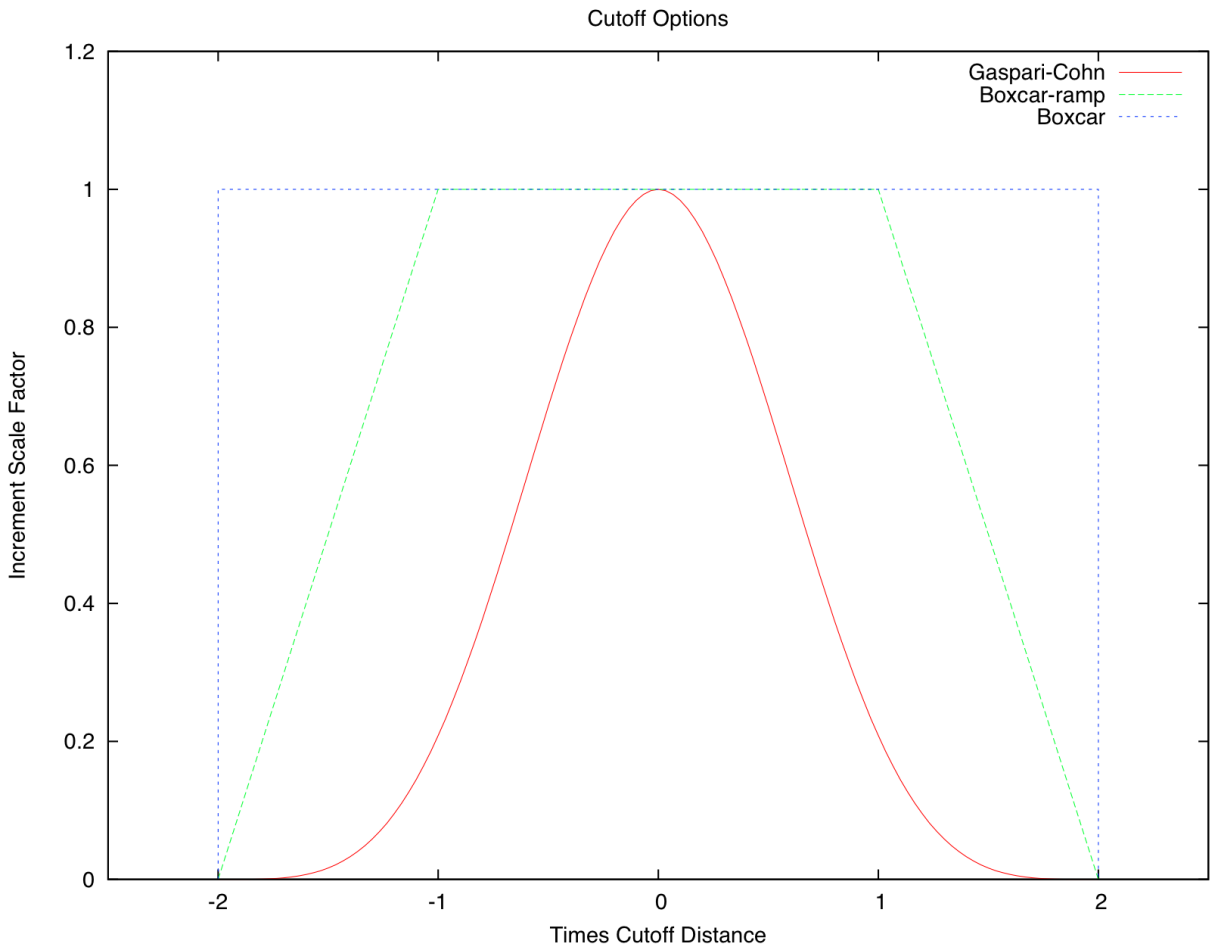
[8] Anderson, J. L., 2007: An adaptive covariance inflation error correction algorithm for ensemble filters. *Tellus A*, **59**, 210-224, doi:10.1111/j.1600-0870.2006.00216.x

[9] Whitaker, J.S. and T.M. Hamill, 2012: Evaluating Methods to Account for System Errors in Ensemble Data Assimilation. *Monthly Weather Review*, **140**, 3078–3089, doi:10.1175/MWR-D-11-00276.1

[10] El Gharamti M., 2018: Enhanced Adaptive Inflation Algorithm for Ensemble Filters. *Monthly Weather Review*, **2**, 623-640, doi:10.1175/MWR-D-17-0187.1

[11] Anderson, J.L., 2012: Localization and Sampling Error Correction in Ensemble Kalman Filter Data Assimilation. *Monthly Weather Review*, 140, 2359–2371. doi:10.1175/MWR-D-11-00013.1

[12] Gaspari, G. and Cohn, S. E., 1999: Construction of correlation functions in two and three dimensions. *Quarterly Journal of the Royal Meteorological Society*, **125**, 723-757. doi:10.1002/qj.49712555417



References

6.16 Working with collaborators on porting new models

The DART team often collaborates with other groups to help write the interface code to a new model. The most efficient way to get started is to meet with DAReS staff either virtually or in person, to discuss what is involved in supporting a different model.

If part of your team isn't familiar with data assimilation yet, you should review the introductory material in this documentation and and also look at work through the concepts in the *DART Tutorial*.

6.16.1 Goals of using DART

DART is the Data Assimilation Research Testbed. It is a collection of tools and routines and scripts that allow users to built custom solutions and explore a variety of DA related efforts. It is not a turnkey system; it must be built before use and is often customized based on needs and goals.

DART is often used for the following types of projects:

- Learning about Data Assimilation (DA)
- Using DART with an existing model and supported observations
- *Adding a DART interface to a new model*
- Using new observations with DART in an existing model
- Using both a new model and new observations with DART
- Using DART to teach DA

Everything on this "possible goals" list except adding support for a new model can generally be done by a single user with minimal help from the DART team. Therefore this discussion focuses only on adding a new model to DART.

6.16.2 Should I consider using DART?

DART is an ensemble-based DA system. It makes multiple runs of a model with slightly different inputs and uses the statistical distribution of the results to decide how to adjust the model state to be more consistent with the observations.

The advantage of ensemble systems is that no changes to the model itself are required. The disadvantage is that multiple runs of the model are needed and this can be computationally expensive.

Simple models can be added to DART with a single person effort, but larger, more complex models can require multiple person-months with support from the DART team to add the interfaces and scripts needed to perform a large-scale DA experiment.

The DART code is in Fortran. The supporting scripts and tools are a mix of shell scripts and python. The model can be written in any language; it will only be run and the input and output files will be used by DART.

6.16.3 Things to discuss before beginning

Is your model appropriate for any kind of DA?

If your model isn't chaotic, you don't need data assimilation. In non-chaotic models, you can improve your predictions by running the model, examining the difference between the prediction and the observations, inverting the equations inside the model to compute how different inputs would have produced outputs closer to the observations.

Chaotic models do not have a simple relationship between inputs and outputs. There are internal feedbacks and non-linear behaviors that make it difficult to adjust the inputs to make the outputs better match the observations.

What is your model state?

"Model state" has a very specific definition that can be the source of much confusion if someone running a model has not thought about DA before. Formally it is the minimal set of variables that must be saved when a model stops so it can be restarted again exactly.

At first glance this means all the variables on the right side of the equals sign for the governing equations of the system. However many models which have not been designed with DA in mind may have no clear time when all parts of the model are at a consistent time. e.g. some variables may be 1/2 timestep ahead or behind others. Some derived variables may be expensive to compute and so are precomputed and stored and not recomputed. If the DA process changes the state variables all derived variables must be recomputed before proceeding.

Restart files often store many more variables than the minimal set needed to restart the model. Often other variables are used in diagnostic routines or are of interest on their own. Generally these are not considered part of the model state.

How is your model execution controlled?

Generally larger and more complex models have an environment they are expecting to run in/with. e.g. scripts to control the execution parameters, or input parameter files; how many processors are used in a parallel system, how the tasks are distributed over the hardware; how long does the execution run, in model time, and what variables are written to the output files.

For DA, at a minimum there must be a way to control how long the model runs before it writes out the results and exits.

For large models, the DA filter process is a large parallel program generally requiring a multi-processor supercomputer or cluster. Many models themselves are large parallel programs, so there can be issues with how the switch between model and DA process is done.

New or adjusted scripting is generally required to include the DA process in the overall execution flow.

Cycling with a DA system

The DA process is generally a cycle of running the model for a certain amount of model time, then running the DA filter to adjust the model state before continuing.

These two steps happen over and over as observations are available to guide the adjustments to the model state.

Models may be written with the assumption that startup costs are only done once and then the model runs for a long period of time. When used with DA models are generally started and stopped after running a relatively short amount of model time. If model startup time is long this can result in unacceptably slow performance.

A small amount of round-off error is often introduced when a model writes restart files before stopping. So running a model N timesteps forward vs. running N/2, stopping, writing restart files, starting, reading restart files, and finishing the last N/2 timesteps will may not result in identical values. Large changes suggest that the model is not a good candidate for a cycling DA system.

The goal is to minimize the differences. This can require small or large changes to make the model behave as expected with repeated starting and stopping.

Some models include external forcing, for example boundary conditions from a separate model. If cycling the forcing files may need to be updated periodically outside of the DA system.

What coordinate system is used by your model?

Coordinate systems use a series of numbers to describe the relationship in space between parts of the model state and where observations are located. In Earth-system models, often a latitude-longitude-vertical coordinate system is used. X,Y,Z Cartesian coordinates are also used to describe 3D space. Other options include cyclindrical or spherical coordinates, and unit-line, -square or -cube coordinates with cyclical boundaries.

Only a single coordinate system can be selected and it applies to both the model state locations as well as the observations.

If the model coordinate system is based on some other space it may be necessary to transform it into physical coordinates before running DA. For example, some models compute in spectral space and the output must be translated into a physical space before DA can be done.

What file format is used for model restart files?

DART reads and writes NetCDF file format. Many earth-system models already use this format. If the model does not, converter programs from the native format to NetCDF and back are needed. NetCDF is a self-describing format with metadata that allows DART to read and process model data without additional configuration files.

What quantities are in the model state?

DART defines a "Quantity" as the fundamental physical object a value is measuring. Examples are Temperature, Pressure, Salinity, etc. Each value in a model state must be associated with a defined quantity.

What observations are you intending to assimilate?

Any observation you intend to assimilate requires a method to compute an "expected value" based on the model state. Often the observation is of the same quantity as exists in the model state, so computing the expected value is a direct process.

Other times the expected value is a function of quantities in the model state, and code called a "forward operator" uses one or more quantities from the model state and computes the expected value.

If the model state does not contain quantities that are needed to compute an expected value, auxiliary data values can be read and used to compute the expected value. But if the expected value cannot be computed or is not in some way a function of the model state, the observations cannot be assimilated.

How are you going to generate your initial ensemble?

Most models don't have an existing ensemble of states ready for ingestion into an ensemble DA system. Options for generating the initial ensemble include adding random perturbations to a single variable in a single state, perturbing forcing variables differently for each ensemble member, or perturbing the entire state.

For models which have a lot of error growth it may be enough to add a very small amount of noise to a single variable in the state to generate an ensemble of states and then run them forward in time with the model to generate states which have sufficient differences.

For models with slower error growth, larger perturbations may be needed, a longer model advance time before starting assimilation, or perturbations of forcing or boundary files may be needed.

The goal is to generate a set of model states which are different but contain internally-consistent values.

An ensemble of states without sufficient differences (spread) will reject assimilating observations.

6.16.4 What code is required to interface a model with DART?

There is a single FORTRAN module that hides the model details from the rest of the DART system. Generally the routines which require the most work are the interpolation routine, followed by the metadata routine and the "get close" localization routines.

Interpolation

Given an observation quantity and location, the model interface routines must return an array of values, one for each ensemble member. The values must be the best estimate of what a real instrument would return if the real state of the system were each of the ensemble values.

For a regular grid this can be computed fairly simply with routines already provided in the DART system. It involves locating the grid values that enclose the observation location, and doing bi- or tri-linear interpolation to the actual location.

However, many models have non-regular grid, especially in the vertical coordinates for an earth-system-based model. Or the grid can be an irregular mesh or deformed mesh. It may take searching or transforms to identify the closest values in the model state to use for interpolation.

Metadata

Given an offset into the model state, the model interface routines must return the location in the selected coordinate system, and the quantity at that offset.

There are routines provided which simplify this for regular or deformed grids, so this generally is not too complex but may require additional arrays for irregular grids or unstructured grids.

Localization

DART bases the impact of observations on the model state on the correlation between the array of predicted observation values, the actual observation value and error, and the array of model state values.

In practice observations are only correlated with model state values "close" to the observation. Spurrious correlations can occur which degrade the results after assimilation. Also there are efficiency gains if only parts of the model state which are "close" to the observation are processed.

DART includes routines which can compute what part of the state are close to a given observation. However some models have special considerations for whether they want to control the impact of observations on parts of the model state and this can be adjusted based on code added to the model-specific parts of getting close observations and model state.

Vertical issues

Most Earth System models use Latitude and Longitude for horizontal coordinates or can generate them if needed (e.g. spectral models can transform their state into Lat/Lon coords). But often vertical coordinates pose additional complications.

If the model and the observations both use the same coordinates for vertical, e.g. pressure or height, then there are no need for conversion routines. But some models use terrain-following coordinates, or a mix of pressure and terrain coordinates. Observation vertical locations can be reported in height or in pressure.

Additionally, if vertical localization is to be done in a different coordinate than the model or observations (e.g. scale height), then conversion routines are needed.

The interface code may need to read in additional arrays from the model in order to convert the vertical coordinates accurately.

During the run of filter there are two options for when vertical conversion is done: all at the start, or on demand. If the observations to be assimilated are expected to impact all or almost all of the state, doing all vertical conversion at the start is more efficient. If the observations are expected to impact only a small percentage of the state variables then doing it on demand is more efficient. The options here are namelist selectable at runtime and the impact on total runtime can be easily measured and compared.

6.16.5 Adding a DART interface to a new model

DART provides a script `new_model.sh` which will create the necessary files for a new model interface. Enter `./new_model.sh`, then the desired model name and location module separated by spaces. This will create the necessary files to get started.

For example to create a model interface for a model called BOUMME which uses the 3D sphere location module:

```
cd models
./new_model.sh BOUMME threed_sphere
```

This will create an BOUMME model directory with the following files:

```
BOUMME/
├── model_mod.f90
├── readme.rst
└── work
    ├── input.nml
    └── quickbuild.sh
```

- `model_mod.f90` is where to add the *required model_mod routines*.
- `readme.rst` is a stub to add documenation for your model interface.
- `quickbuild.sh` is used to compile DART for your model.

Templates are chosen based on location module input. The currently supported location templates are for 3D and 1D modules, with the possibility for more in the future. At the moment, `threed_sphere`, `threed_cartesian`, and `oned` will produce model_mod.f90 code that compile will sucessfully with `./quickbuild.sh`.

We recommend looking at the existing supported models and reusing code from them if possible. Models with similar grid types or vertical coordinates are good candidates.

6.17 DART build system

The DART build system consists of a `quickbuild.sh` script for each model or observation converter and several build functions in `DART/build_templates/`

```
buildconvfunctions.sh
buildfunctions.sh
buildpreprocess.sh
```

The DART build process is invoked by running `quickbuild.sh` and consists of the following steps:

1. Find the root of the DART git repository you are running in
2. Source the build functions

3. Parse the arguments to quickbuild.sh
4. Clean any existing .o .mod files
5. Run fixsystem to alter compiler dependent source code
6. Compile preprocess
7. Run preprocess to create an obs_def_mod.f90 and obs_kind_mod.f90 specific to the observations you are using in DART.
8. For each program
    - collect the source code needed
    - create a Makefile for the executable
    - compile the source code into the executable

The Makefile is created using `mkmf`, which maps out the dependencies between the source files. For more information on mkmf please see the mkmf documentation.

There is a quickbuild.sh script is each work directory. To view the usage information for quickbuild.sh,

```
./quickbuild.sh help
```

`quickbuild.sh` may be used to build all programs for a particular model or observation converter, or can be given a single program as an argument to build.

For example, you may want to build obs_sequence_tool

```
./quickbuild.sh obs_sequence_tool
```

In `quickbuild.sh` there are arrays containing the list of programs to build. In general the arrays will contain all programs needed, but you may want to add other *DART programs* to your build list. Edit `quickbuild.sh` to add your required program to the appropriate array.

For models there are four arrays in quickbuild.sh:

```
programs=(
DART programs that can be compiled with mpi go here
)

serial_programs=(
DART programs that do not use mpi go here
)

model_programs=(
Model program that can be compiled with mpi go here
)

model_serial_programs=(
model programs that do not use mpi go here
)
```

For observation converters, there is a single array.

```
programs=(
converter programs
)
```

6.17.1 For DART developers

If you are developing, and want to iterate over changing code and compiling a single program, for example filter, you can use `quickbuild.sh` in the following way:

```
./quickbuild.sh filter
*edit the code*
make
*edit the code*
make
```

Where does quickbuild.sh look for code?

The source code which gets compiled into DART executables is a function of

- Core DART code
- Location specific code (threed sphere, threed Cartesian, oned, . . .)
- Model/converter specific code
- External libraries
- mpi/null mpi utilities
- obs_def_mod.f90 and obs_kind_mod.f90 (created by preprocess)

The core DART code is collected from the `DART/assimilation_code/modules/` directory. Note the `DART/assimilation_code/modules/observations` directory is excluded from the search. This directory contains quantity files which are used as input to preprocess rather than being compiled directly.

The model directory and the location module to be used are defined in `quickbuild.sh`. For example, the Regional Ocean Model (ROMS) uses the threed_sphere location module.

```
MODEL=ROMS
LOCATION=threed_sphere
```

Similarly, for an observation converter, the converter directory and the location module are defined in `quickbuild.sh`

```
CONVERTER=MADIS
LOCATION=threed_sphere
```

The model/observation converter directory will be searched for .f90 files.

Additionally any .f90 files in the work directory where you are running `quickbuild.sh` will be added to the list of source files. .f90 files in the work directory will take precedence over .f90 files with the same name elsewhere.

To take a look how the .f90 files are collected, look at the `findsrc` and `findconvsrc` functions in the following files:

```
DART/build_templates/buildfunctions.sh
DART/build_templates/buildconvfunctions.sh
```

When adding new code, be sure to obey the following rules to make sure `quickbuild.sh` finds your new code and ignores any code you do not want compiled.

1. The {name} of the .f90 file must be the program {name}. For example the source code program called `red_mist` must be called `red_mist.f90`

2. Any .f90 files that you have in your work directory will take precedence over .f90 files with the same name elsewhere. For example if you have an `assim_tools_mod.f90` in your work directory, this will be compiled rather than the file `DART/assimilation_code/modules/assimilation/assim_tools_mod.f90`.

   In the example below, the file `assim_tools_mod.f90` from the work directory will be used when compiling the lorenz_96 programs.

```
DART/models/lorenz_96/work/
                            |-- quickbuild.sh
                            |-- assim_tools_mod.f90
```

3. If you have .f90 files that you do **not** want to compile into DART, you will need to exclude these files using one of these methods:

   - Put the code outside the directories quickbuild.sh searches, for example in a directory `DART/exclude/`
   - Explicitly exclude the .f90 files with the EXCLUDE variable in `quickbuild.sh`
   - Rename the .f90 files, e.g. `solar_flux.f90` renamed to `solar_flux.f90.exclude`

4. for core DART programs, use the following directory structure:

```
DART/assimilation_code/programs/{program_name}/
                                                |-- {program_name}.f90
                                                |-- {program_name}.rst
                                                |-- {program_name}.nml
```

   where {program_name}.rst is the documentation for the program and {program_name}.nml is a namelist with default values (if applicable to the program).

5. For observation converters, the program must be in the top level of the converter directory:

```
DART/observations/obs_converters/{converter}/{program_name}.f90
```

6. For programs specific to a particular model, the program must be in the model directory. For example programs that are specific to Weather and Research Forecasting model (WRF), must be in the `DART/models/wrf` directory. Model programs may be in subdirectories as shown in the example below.

```
DART/models/{model}/
                    |-- {program_one}.f90
                    |-- subdirectory/{program_two}.f90
```

   There may be code in the model directory that you do not want compiled into the DART executables. For example, the bgrid_solo model directory has all the .f90 code required to build bgrid_solo model (fms_src), which we do *not* want to compile into DART and so the bgrid_solo `quickbuild.sh` has the following line:

```
EXCLUDE=fms_src
```

   EXCLUDE is a directory of code to exclude.

7. For code that is outside of the above locations, you can use the variable EXTRA to add source files to be compiled. For example, the ROMS observation converter requires the ROMS model_mod.f90 code, so the ROMS `quickbuild.sh` has

```
EXTRA="$DART/models/ROMS/model_mod.f90"
```

   EXTRA is source code outside the work directory to include in the build. EXTRA can be a directory, a list of files, or a single file.

6.18 Assimilation in a complex model

6.18.1 Introduction

Running a successful assimilation takes careful diagnostic work and experiment iterations to find the best settings for your specific case.

The basic Kalman filter can be coded in only a handful of lines. The difficulty in getting an assimilation system working properly involves making the right choices to compensate for sampling errors, model bias, observation error, lack of model forecast divergence, variations in observation density in space and time, random correlations, etc. There are tools built into DART to deal with most of these problems but it takes careful work to apply them correctly.

6.18.2 Your first attempt

If you are adding a new model or a new observation type, you should assimilate exactly one observation, with no model advance, with inflation turned off, with a large cutoff, and with the outlier threshold off (see below for how to set these namelist items).

Run an assimilation. Look at the `obs_seq.final` file to see what the forward operator computed. Use ncdiff to difference the `preassim_mean.nc` and `postassim_mean.nc` (or `output_mean.nc`) diagnostic NetCDF files and look at the changes (the "innovations") in the various model fields. Is it in the right location for that observation? Does it have a reasonable value?

Then assimilate a group of observations and check the results carefully. Run the observation diagnostics and look at the total error and spread. Look carefully at the number of observations being assimilated compared to how many are available.

Assimilations that are not working can give good looking statistics if they reject all but the few observations that happen to match the current state. The errors should grow as the model advances and then shrink when new observations are assimilated, so a timeseries plot of the RMSE should show a sawtooth pattern. The initial error entirely depends on the match between the initial ensemble and the observations and may be large but it should decrease and then reach a roughly stable level. The ensemble spread should ultimately remain relatively steady, at a value around the expected observation error level. Once you believe you have a working assimilation, this will be your baseline case.

If the ensemble spread is too small, several of the DART facilities described below are intended to compensate for ensemble members getting too close to each other. Then one by one enable or tune each of the items below, checking each time to see what is the effect on the results.

6.18.3 Next attempts

High-level data assimilation workflows gives an overview of a variety of complete assimilation experiments, including the programs which need to be run and their input and output.

6.18.4 Important features of assimilations

Suggestions for the most common namelist settings and features built into DART for running a successful assimilation include:

Ensemble size

In practice, ensemble sizes between 20 and 100 seem to work best. Fewer than 20-30 members leads to statistical errors which are too large. More than 100 members takes longer to run with very little benefit, and eventually the results get worse again. Often the limit on the number of members is based on the size of the model since you have to run N copies of the model each time you move forward in time. If you can, start with 50-60 members and then experiment with fewer or more once you have a set of baseline results to compare it with. The namelist setting for ensemble size is `&filter_nml :: ens_size`

Localization

There are two main advantages to using localization. One is it avoids an observation impacting unrelated state variables because of spurious correlations. The other is that, especially for large models, it improves run-time performance because only points within the localization radius need to be considered. Because of the way the parallelization was implemented in DART, localization was easy to add and using it usually results in a very large performance gain. See here for a discussion of localization-related namelist items.

Inflation

Since the filter is run with a number of members which is usually small compared to the number of degrees of freedom of the model (i.e. the size of the state vector or the number of EOFs needed to characterize the variability), the model uncertainty is under-represented. Other sources of error and uncertainty are not represented at all. These factors lead to the ensemble being 'over-confident', or having too little spread. More observations leads to more over-confidence. This characteristic can worsen with time, leading to ensemble collapse to a single solution. Inflation increases the spread of the members in a systematic way to overcome this problem. There are several sophisticated options on inflation, including spatial and temporal adaptive and damping options, which help deal with observations which vary in density over time and location. See *Inflation* for a discussion of inflation-related namelist items.

Outlier rejection

Outlier rejection can be used to avoid bad observations (ones where the value was recorded in error or the processing has an error and a non-physical value was generated). It also avoids observations which have accurate values but the mean of the ensemble members is so far from the observation value that assimilating it would result in unacceptably large increments that might destablize the model run. If the difference between the observation and the prior ensemble mean is more than N standard deviations from the square root of the sum of the prior ensemble and observation error variance, the observation will be rejected. The namelist setting for the number of standard deviations to include is `&filter_nml :: outlier_threshold` and we typically suggest starting with a value of 3.0.

Sampling error

For small ensemble sizes a table of expected statistical error distributions can be generated before running DART. Corrections accounting for these errors are applied during the assimilation to increase the ensemble spread which can improve the assimilation results. The namelist item to enable this option is `&assim_tools_nml :: sampling_error_correction`. Additionally you will need to have the precomputed correction file `sampling_error_correction_table.nc`, in the run directory. See the description of the namelist item in the &assim_tools_nml namelist, and *PROGRAM gen_sampling_err_table* for instructions on where to find (or how to generate) the auxiliary file needed by this code. See Anderson (2011).

Free run/forecast after assimilation

Separate scripting can be done to support forecasts starting from the analyzed model states. After filter exits, the models can be run freely (with no assimilated data) further forward in time using one or more of the last updated model states from filter. Since all ensemble members are equally likely a member can be selected at random, or a member close to the mean can be chosen. See the *PROGRAM closest_member_tool* for one way to select a "close" member. The ensemble mean is available to be used, but since it is a combination of all the member states it may not have self-consistent features, so using a single member is usually preferred.

Evaluating observations without assimilation

Filter can be used to evaluate the accuracy of a single model state based on a set of available observations. Either copy or link the model state file so there appear to be 2 separate ensemble members (which are identical). Set the filter namelist ensemble size to 2 by setting `ens_size` to 2 in the &filter_nml namelist. Turn off the outlier threshold and both Prior and Posterior inflation by setting `outlier_threshold` to -1, and both the `inf_flavor` values to 0 in the same &filter_nml namelist. Set all observation types to be 'evaluate-only' and have no types in the 'assimilate' list by listing all types in the `evaluate_these_obs_types` list in the `&obs_kind_nml` section of the namelist, and none in the assimilation list. Run filter as usual, including model advances if needed. Run observation diagnostics on the resulting `obs_seq.final` file to compute the difference between the observed values and the predicted values from this model state.

Verification/comparison with and without assimilation

To compare results of an experiment with and without assimilating data, do one run assimilating the observations. Then do a second run where all the observation types are moved to the `evaluate_these_obs_types` list in the `&obs_kind_nml` section of the namelist. Also turn inflation off by setting both `inf_flavor` values to 0 in the &filter_nml namelist. The forward operators will still be called, but they will have no impact on the model state. Then the two sets of diagnostic state space netcdf files can be compared to evaluate the impact of assimilating the observations, and the observation diagnostic files can also be compared.

DART quality control flag added to output observation sequence file

The filter adds a quality control field with metadata 'DART quality control' to the `obs_seq.final` file. At present, this field can have the following values:

0:	Observation was assimilated successfully
1:	Observation was evaluated (as specified in namelist) and not used in the assimilation
2:	The observation was used but one or more of the posterior forward observation operators failed
3:	The observation was evaluated AND one or more of the posterior forward observation operators failed
4:	One or more prior forward observation operators failed so the observation was not used
5:	The observation was not used because it was not selected in the namelist to be assimilated or evaluated
6:	The prior quality control value was too high so the observation was not used.
7:	Outlier test failed (see below)
8:	Vertical conversion failed

The outlier test computes the difference between the observation value and the prior ensemble mean. It then computes a standard deviation by taking the square root of the sum of the observation error variance and the prior ensemble variance for the observation. If the difference between the ensemble mean and the observation value is more than the specified number of standard deviations, then the observation is not used and the DART quality control field is set to 7.

6.19 Message Passing Interface

6.19.1 Introduction

DART programs can be compiled using the *Message Passing Interface (MPI)*. MPI is both a library and run-time system that enables multiple copies of a single program to run in parallel, exchange data, and combine to solve a problem more quickly.

DART does **NOT** require MPI to run; the default build scripts do not need nor use MPI in any way. However, for larger models with large state vectors and large numbers of observations, the data assimilation step will run much faster in parallel, which requires MPI to be installed and used. However, if multiple ensembles of your model fit comfortably (in time and memory space) on a single processor, you need read no further about MPI.

MPI is an open-source standard; there are many implementations of it. If you have a large single-vendor system it probably comes with an MPI library by default. For a Linux cluster there are generally more variations in what might be installed; most systems use a version of MPI called MPICH. In smaller clusters or dual-processor workstations a version of MPI called either LAM-MPI or OpenMPI might be installed, or can be downloaded and installed by the end user.

Note: OpenMP is a different parallel system; OpenMPI is a recent effort with a confusingly similar name.

An "MPI program" makes calls to an MPI library, and needs to be compiled with MPI include files and libraries. Generally the MPI installation includes a shell script called `mpif90` which adds the flags and libraries appropriate for each type of fortran compiler. So compiling an MPI program usually means simply changing the fortran compiler name to the MPI script name.

These MPI scripts are built during the MPI install process and are specific to a particular compiler; if your system has multiple fortran compilers installed then either there will be multiple MPI scripts built, one for each compiler type, or there will be an environment variable or flag to the MPI script to select which compiler to invoke. See your system documentation or find an example of a successful MPI program compile command and copy it.

DART use of MPI

To run in parallel, only the DART 'filter' program (possibly the companion 'wakeup_filter' program), and the 'GSI2DART' observation converter need to be compiled with the MPI scripts. All other DART executables should be compiled with a standard F90 compiler and are not MPI enabled. (And note again that 'filter' can still be built as a single executable like previous releases of DART; using MPI and running in parallel is simply an additional option.) To build a parallel version of the 'filter' program, the 'mkmf_filter' command needs to be called with the '-mpi' option to generate a Makefile which compiles with the MPI scripts instead of the Fortran compiler.

Run `quickbuild.sh` will build DART with MPI. You will also need to edit the `$DART/mkmf/mkmf.template` file to call the proper version of the MPI compile script if it does not have the default name, is not in a standard location on the system, or needs additional options set to select between multiple Fortran compilers. To build without mpi, use `quickbuild.sh nompi`.

MPI programs generally need to be started with a shell script called 'mpirun' or 'mpiexec', but they also interact with any batch control system that might be installed on the cluster or parallel system. Parallel systems with multiple users generally run some sort of batch system (e.g. LSF, PBS, POE, LoadLeveler, etc). You submit a job request to this system and it schedules which nodes are assigned to which jobs. Unfortunately the details of this vary widely from system to system; consult your local web pages or knowledgeable system admin for help here. Generally the run scripts supplied with DART have generic sections to deal with LSF, PBS, no batch system at all, and sequential execution, but the details (e.g. the specific queue names, accounting charge codes) will almost certainly have to be adjusted.

The data assimilation process involves running multiple copies (ensembles) of a user model, with an assimilation computation interspersed between calls to the model. There are many possible execution combinations, including:

- Compiling the assimilation program 'filter' with the model, resulting in a single executable. This can be either a sequential or parallel program.
- Compiling 'filter' separately from the model, and having 2 separate executables. Either or both can be sequential or parallel.

The choice of how to combine the 'filter' program and the model has 2 parts: building the executables and then running them. At build time, the choice of using MPI or not must be made. At execution time, the setting of the 'async' namelist value in the filter_nml section controls how the 'filter' program interacts with the model.
Choices include:

- async = 0 The model and filter programs are compiled into a single executable, and when the model needs to advance, the filter program calls a subroutine. See a diagram which illustrates this option.
- async = 2 The model is compiled into a sequential (single task) program. If 'filter' is running in parallel, each filter task will execute the model independently to advance the group of ensembles. See a diagram which illustrates this option.
- async = 4 The model is compiled into an MPI program (parallel) and only 'filter' task 0 tells the startup script when it is time to advance the model. Each ensemble is advanced one by one, with the model using all the processors to run in parallel. See a diagram which illustrates this option.
- async ignored (sometimes referred to as 'async 5', but not a setting in the namelist) This is the way most large models run now. There is a separate script, outside of filter, which runs the N copies of the model to do the advance. Then filter is run, as an MPI program, and it only assimilates for a single time and then exits. The external script manages the file motion between steps, and calls both the models and filter in turn.

This release of DART has the restriction that if the model and the 'filter' program are both compiled with MPI and are run in 'async=4' mode, that they both run on the same number of processors; e.g. if 'filter' is run on 16 processors, the model must be started on 16 processors as well. Alternatively, if the user model is compiled as a single executable (async=2), 'filter' can run in parallel on any number of processors and each model advance can be executed independently without the model having to know about MPI or parallelism.
Compiling and running an MPI application can be substantially more complicated than running a single executable. There are a suite of small test programs to help diagnose any problems encountered in trying to run the new version of DART. Look in `DART/developer_tests/mpi_utilities/tests/README` for instructions and a set of tests to narrow down any difficulties.

Performance issues and timing results

Getting good performance from a parallel program is frequently difficult. Here are a few of reasons why:

- Amdahl's law You can look up the actual formula for this "law" in the Wikipedia, but the gist is that the amount of serial code in your program limits how much faster your program runs on a parallel machine, and at some point (often much sooner than you'd expect) you stop getting any speedup when adding more processors.
- Surface area to volume ratio Many scientific problems involve breaking up a large grid or array of data and distributing the smaller chunks across the multiple processors. Each processor computes values for the data on the interior of the chunk they are given, but frequently the data along the edges of each chunk must be communicated to the processors which hold the neighboring chunks of the grid. As you increase the number of processors (and keep the problem size the same) the chunk size becomes smaller. As this happens, the 'surface area' around

the edges decreases slower than the 'volume' inside that one processor can compute independently of other processors. At some point the communication overhead of exchanging edge data limits your speedup.

- Hardware architecture system balance Raw CPU speeds have increased faster than memory access times, which have increased faster than access to secondary storage (e.g. I/O to disk). Computations which need to read input data and write result files typically create I/O bottlenecks. There are machines with parallel filesystems, but many programs are written to have a single processor read in the data and broadcast it to all the other processors, and collect the data on a single node before writing. As the number of processors increases the amount of time spent waiting for I/O and communication to and from the I/O node increases. There are also capacity issues; for example the amount of memory available on the I/O node to hold the entire dataset can be insufficient.
- NUMA memory Many machines today have multiple levels of memory: on-chip private cache, on-chip shared cache, local shared memory, and remote shared memory. The approach is referred as Non-Uniform Memory Access (NUMA) because each level of memory has different access times. While in general having faster memory improves performance, it also makes the performance very difficult to predict since it depends not just on the algorithms in the code, but is very strongly a function of working-set size and memory access patterns. Beyond shared memory there is distributed memory, meaning multiple CPUs are closely connected but cannot directly address the other memory. The communication time between nodes then depends on a hardware switch or network card, which is much slower than local access to memory. The performance results can be heavily influenced in this case by problem size and amount of communication between processes.

Parallel performance can be measured and expressed in several different ways. A few of the relevant definitions are:

- Speedup Generally defined as the wall-clock time for a single processor divided by the wall-clock time for N processors.
- Efficiency The speedup number divided by N, which for perfect scalability will remain at 1.0 as N increases.
- Strong scaling The problem size is held constant and the number of processors is increased.
- Weak scaling The problem size grows as the number of processors increases so the amount of work per processor is held constant.

We measured the strong scaling efficiency of the DART 'filter' program on a variety of platforms and problem sizes. The scaling looks very good up to the numbers of processors available to us to test on. It is assumed that for MPP (Massively-Parallel Processing) machines with 10,000s of processors that some algorithmic changes will be required. These are described in this paper.

User considerations for their own configurations

Many parallel machines today are a hybrid of shared and distributed memory processors; meaning that some small number (e.g. 2-32) of CPUs share some amount of physical memory and can transfer data quickly between them, while communicating data to other CPUs involves slower communication across either some kind of hardware switch or fabric, or a network communication card like high speed ethernet.

Running as many tasks per node as CPUs per shared-memory node is in general good, unless the total amount of virtual memory used by the program exceeds the physical memory. Factors to consider here include whether each task is limited by the operating system to 1/Nth of the physical memory, or whether one task is free to consume more than its share. If the node starts paging memory to disk, performance takes a huge nosedive.

Some models have large memory footprints, and it may be necessary to run in MPI mode not necessarily because the computation is faster in parallel, but because the dataset size is larger than the physical memory on a node and must be divided and spread across multiple nodes to avoid paging to disk.

6.20 Filters

The different types of assimilation algorithms (EAKF, ENKF, Kernel filter, Particle filter, etc.) are determined by the `&assim_tools_nml:filter_kind` entry, described in *MODULE assim_tools_mod*. Despite having 'filter' in the name, they are assimilation algorithms and so are implemented in `assim_tools_mod.f90`.

6.21 Inflation

In pre-Manhattan DART, there were two choices for the basic type of inflation: observation-space or state-space. Observation-space inflation is no longer supported. (If you are interested in observation-space inflation, talk to Jeff first.) The rest of this discussion applies to state-space inflation.

State-space inflation changes the spread of an ensemble without changing the ensemble mean. The algorithm computes the ensemble mean and standard deviation for each variable in the state vector in turn, and then moves the member's values away from the mean in such a way that the mean remains unchanged. The resulting standard deviation is larger than before. It can be applied to the Prior state, before observations are assimilated (the most frequently used case), or it can be applied to the Posterior state, after assimilation. See Anderson (2007), Anderson (2009).

Historically, inflation was first introduced to address sampling errors (the fact that we are limited to a small ensemble size). Latest research, e.g. El Gharamti et al. (2019) suggests that prior and posterior inflation can be used to address different issues in the filtering problem. Prior inflation is able to address issues in the forecast step such as model errors while posterior inflation can help mitigate sampling errors in the analysis step.

Inflation values can vary in space and time, depending on the specified namelist values. Even though we talk about a single inflation value, the inflation has a probability density with a mean and standard deviation. We use the mean value when we inflate, and the standard deviation indicates how sure of the value we are. Larger standard deviation values mean "less sure" and the inflation value can increase more quickly with time. Smaller values mean "more sure" and the time evolution will be slower since we are more confident that the mean (inflation value) is correct.

The standard deviation of inflation allows inflation values to increase with time, if required by increasing density or frequency of observations, but it does not provide a mechanism to reduce the inflation when the frequency or density of observations declines. So there is also an option to damp inflation through time. In practice with large geophysical models using damped inflation has been a successful strategy.

The following namelist items which control inflation are found in the `input.nml` file, in the &filter_nml namelist. The detailed descriptions are in the filter_mod page. Here we try to give some basic advice about commonly used values and suggestions for where to start. Spatial variation is controlled by `inf_flavor`, which also controls whether there's any inflation, `inf_initial_from_restart`, and `inf_initial`, as described below. Time variation is controlled by `inf_sd_initial_from_restart`, `inf_sd_initial`, `inf_sd_lower_bound`, `inf_damping`, `inf_lower_bound` and `inf_upper_bound`.

In the namelist each entry has two values. The first is for Prior inflation and the second is for Posterior inflation.

`&filter_nml :: inf_flavor`
*valid values:*0, 2, 3, 4, 5

Set the type of Prior and Posterior inflation applied to the state vector. Values mean:

- **0:** No inflation (Prior and/or Posterior) and all other inflation variables are ignored
- [**1:** Deprecated: Observation space inflation]
- **2:** Spatially-varying state space inflation (gaussian)

- **3:** Spatially-uniform state space inflation (gaussian)
- **4:** Relaxation To Prior Spread (Posterior inflation only)
- **5:** Enhanced Spatially-varying state space inflation (inverse gamma)

Spatially-varying state space inflation stores an array of inflation values, one for each item in the state vector. If time-evolution is enabled, each value can evolve independently. Spatially-uniform state space inflation uses a single inflation value for all items in the state vector. If time-evolution is enabled, that single value can evolve. See `inf_sd_*` below for control of the time-evolution behavior. Enhanced spatially-varying inflation uses an inverse-gamma distribution which allows the standard deviation of the inflation to increase or decrease through time and may produce better results (see El Gharamti (2018)). In practice we recommend starting with no inflation (both values 0). Then try inflation type 2 or 5 prior inflation and no inflation (0) for posterior. WARNING: even if inf_flavor is not 0, inflation will be turned off if `inf_damping` is set to 0.

Important: Relaxation to prior spread (aka RTPS, i.e., `inf_flavor=4`) is a spatially varying **posterior** inflation algorithm.

When using RTPS you cannot set the prior inflation flavor to 4. The code will exit with an error messge. Unlike all other flavors, RTPS does not use files to handle inflation in time. So, if the user supplies `input_postinf_{mean,sd}.nc`, these will be **ignored**. The ONLY namelist option that RTPS uses (other than `inf_flavor=4`) is the second entry of `inf_initial`. This value is technically not the posterior inflation value but rather a *weighting* factor (denoted by α; in Whitaker and Hamill (2012)) that is used to relax the posterior spread to the prior spread. For instance, if $\alpha = 0.3$ then the inflated posterior spread is as follows: 70% of the analysis spread plus 30% of the prior spread. If $\alpha = 1.0$, then the inflated posterior spread is simply set to the prior spread. Using α, RTPS calculates the effective posterior inflation *under the hood* and writes out the inflation values to the user. These can be looked at for diagnostic purposes. The algorithm disregards them for the next data assimilation cycle. In short, RTPS is adaptive in time but unlike flavors 2, 3 and 5 it has no memory. The recommendation is to set the second entry of `inf_initial` to any number between 0.0 and 1.0.

`&filter_nml :: inf_initial_from_restart`
valid values: .true. or .false.

If true, read the inflation values from an inflation restart file named `input_{prior,post}inf_mean.nc`. An initial run could be done to let spatially-varying inflation values evolve in a spinup phase, and then the saved values can be read back in and used as fixed values in further runs. Or if time-varying inflation is used, then the restart file from the previous job step must be supplied as an input file for the next step.

`&filter_nml :: inf_initial`
valid values: real numbers, usually 1.0 or slightly larger If not reading in inflation values from a restart file, the initial value to set for the inflation. Generally we recommend starting with just slightly above 1.0, maybe 1.02, for a slight amount of initial inflation.

`&filter_nml :: inf_lower_bound`
valid values: real numbers, usually 1.0 or slightly larger

If inflation is time-evolving (see `inf_sd_*` below), then this sets the lowest value the inflation can evolve to. Setting a number less than one allows for deflation but generally in a well-observed system the ensemble needs more spread and not less. We recommend a setting of 1.0.

`&filter_nml :: inf_upper_bound`
valid values: real numbers, larger than 1.0

If inflation is time-evolving (see `inf_sd_*` below), then this sets the largest value the inflation can evolve to. We recommend a setting of 100.0, although if the inflation values reach those levels there is probably a problem with the assimilation.

&filter_nml :: inf_damping
valid values: 0.0 to 1.0

Applies to all state-space inflation types, but most frequently used with time-adaptive inflation variants. The difference between the current inflation value and 1.0 is multiplied by this factor before the next assimilation cycle. So the inflation values are pushed towards 1.0, from above or below (if inf_lower_bound allows inflation values less than 1.0). A value of 0.0 turns all inflation off by forcing the inflation value to 1.0. A value of 1.0 turns damping off by leaving the original inflation value unchanged. We have had good results in large geophysical models using time- and space-adaptive state-space inflation and setting the damping to a value of 0.9, which damps slowly.

&filter_nml :: inf_sd_initial_from_restart
valid values: .true. or .false.

If true, read the inflation standard deviation values from an restart file named `input_{prior,post}inf_sd.nc`. See the comments above about `inflation_initial_from_restart`.

&filter_nml :: inf_sd_initial
valid values: 0.0 to disable evolution of inflation, > 0.0 otherwise

The initial value to set for the inflation standard deviation, IF not reading in inflation standard deviation values from a file. This value (or these values) control whether the inflation values evolve with time or not. A negative value or 0.0 prevents the inflation values from being updated, so they are constant throughout the run. If positive, the inflation values evolve through time. We have had good results setting this and `inf_sd_lower_bound` to 0.6 for large geophysical models.

&filter_nml :: inf_sd_lower_bound
valid values: 0.0 to disable evolution of inflation, > 0.0 otherwise

If the setting of `inf_sd_initial` is 0 (to disable time evolution of inflation) then set this to the same value.

Otherwise, the standard deviation of the inflation cannot fall below this value. Smaller values will restrict the inflation to vary more slowly with time; larger values will allow the inflation to adapt more quickly. We have had good results setting this and `inf_sd_initial` to 0.6 for large geophysical models. Since the `inf_sd_lower_bound` is a scalar, it is not possible to set different lower bounds for different parts of the state vector.

Time-varying inflation with flavor 2 generally results in the inflation standard deviation for all state variables shrinking to the lower bound and staying there. For flavor 5, the inflation standard deviation value is allowed to increase and decrease.

&filter_nml :: inf_sd_max_change
valid values: 1.0 to 2.0

Used only with the Enhanced inflation (flavor 5). The Enhanced inflation algorithm allows the standard deviation to increase as well as decrease. The `inf_sd_max_change` controls the maximum increase of the standard deviation in an assimilation cycle. A value of 1.0 means it will not increase, a value of 2.0 means it can double; a value inbetween sets the percentage it can increase, e.g. 1.05 is a limit of 5%. Suggested value is 1.05 (max increase of 5% per cycle).

Because the standard deviation for original flavor 2 could never increase, setting the `inf_sd_initial` value equal to the `inf_sd_lower_bound` value effectively fixed the standard deviation at a constant value. To match the same behavior, if they are equal and Enhanced inflation (flavor 5) is used it will also use that fixed value for the standard deviation of the inflation. Otherwise the standard deviation will adapt as needed during each assimilation cycle.

&filter_nml :: inf_deterministic
valid values: .true. or .false.

Recommend always using `.true.`.

6.21.1 Guidance regarding inflation

First and foremost, if you are using one of the temporally-varying inflation options, save the entire series of inflation files to explore how inflation evolves through time. As part of the workflow, you have to take the output of one assimilation cycle and rename it to be the input for the next assimilation cycle. That is the time to make a copy that has a unique name - usually with some sort of date or timestamp. This also makes it possible to restart an experiment.

The suggested procedure for testing inflation options is to start without any (both `inf_flavor` values set to 0 and `inf_damping` > 0.). Then enable Prior state space, spatially-varying inflation, with no Posterior inflation (set `inf_flavor` to [2, 0]). Then try damped inflation (set `inf_damping` to 0.9 and set `inf_sd_initial` and `inf_sd_lower_bound` to 0.6). The inflation values and standard deviation are written out to files with `_{prior, post}inf_{mean,sd}` in their names. These NetCDF files can be viewed with common tools (we often use ncview). Expected inflation values are generally in the 1 to 30 range; if values grow much larger than this it usually indicates a problem with the assimilation.

PROGRAM fill_inflation_restart may be used to create netCDF files with initial values such that the input.nml settings for reading from file vs. reading from namelist can stay constant throughout the entire experiment.

It is possible to set inflation values in an existing netCDF file by using one of the standard NCO utilities like "`ncap2`" on a copy of a restart file. Inflation mean and sd values look exactly like restart values, arranged by variable type like T, U, V, etc.

Here's an example of using ncap2 to set the T,U and V inf values:

```
ncap2 -s 'T=1.0;U=1.0;V=1.0' wrfinput_d01 input_priorinf_mean.nc
ncap2 -s 'T=0.6;U=0.6;V=0.6' wrfinput_d01 input_priorinf_sd.nc
-or-
ncap2 -s 'T(:,:,:)=1.0;U(:,:,:)=1.0;V(:,:,:)=1.0' wrfinput_d01 input_priorinf_mean.nc
ncap2 -s 'T(:,:,:)=0.6;U(:,:,:)=0.6;V(:,:,:)=0.6' wrfinput_d01 input_priorinf_sd.nc
```

Some versions of the NCO utilities change the full 3D arrays into a single scalar. If that's your result (check your output with `ncdump -h`) use the alternate syntax or a more recent version of the NCO tools.

6.22 Required model_mod routines

There are 18 Fortran subroutines necessary to implement in order to successfully integrate a model in DART. You will place these routines in your `model_mod.f90` in a subdirectory with the name of your model in `DART/models`. There is often a sensible default implementation that can be used for each of these routines. For example, in the case of a model that starts at a time of "0", for the required routine **init_time()** the following code will use this default implementation:

```
use default_model_mod,     only : init_time
```

As in all Fortran programs, a comma-separated list of routines can be listed after the colon.

The following table lists each of the 18 routines, their default modules relative to `DART`, and the default behavior. If the default behavior is not desired, see the section *Suggestions for a "simple" model* for a model that DART can advance, or *Suggestions for a "complex" model* for a model that is advanced externally from DART.

Routine # / name	Purpose	Default module / directory	Default behavior
1. **init_time()**	Set the *initial time* if not read from the restart file.	`default_model_mod` / `models/utilities`	Sets the initial time to 0 days, 0 seconds
2. **init_conditions()**	For a "cold start" fill in an empty state vector with *initial conditions*. Many models cannot just make up values from thin air and thus choose to fail when this is requested.	`default_model_mod` / `models/utilities`	Sets the initial state to 0. To fail use `init_conditions => fail_init_conditions`.
3. **get_model_size()**	Return the *number of items in the state vector*.	`default_model_mod` / `models/utilities`	Returns 1; i.e. there is only one item in the state.
4. **static_init_model()**	*Initialize* DART with information about the model that will be used by the remaining `model_mod` routines. The procedure for doing this will depend on how complex the model is; see below for suggestions for implementation.	`default_model_mod` / `models/utilities`	Does nothing.
5. **get_state_meta_data()**	Takes an index into the state vector and returns the *location* corresponding to that value and optionally the *variable type*. See below for suggestions on implementation.	`default_model_mod` / `models/utilities`	Sets a missing location and the default variable type.
6. **end_model()**	*Deallocate* any arrays allocated in **static_init_model()**.	`default_model_mod` / `models/utilities`	Does nothing.
7. **adv_1step()**	If possible, *advance the model* state from one time to another. Complex models will be unable to implement this method and should fail.	`default_model_mod` / `models/utilities`	Call the error handler with the message "unable to advance model".
8. **shortest_time_between_assimilations()**	Return a namelist or a fixed value for the *minimum model advance time* between assimilations. Note that complex models will handle advancing the time externally.	`default_model_mod` / `models/utilities`	Returns a time period of 1 day.
9. **model_interpolate()**	*Interpolate* a requested quantity to the given location to get an array of expected values for all ensemble members. *NOTE*: this is often the most time consuming method to implement.	`default_model_mod` / `models/utilities`	Fail and set the expected observation to "missing."
10. **nc_write_model_atts()**	Add any *additional information* to the netCDF output diagnostic files. *NOTE*: the state will already be output by other	`default_model_mod` / `models/utilities`	Does nothing.

6.23 Suggestions for a "simple" model

A "simple" model is one where DART can advance the model through a function call. As we saw above, Lorenz 63 falls into this category and can be used as a reference. Here we provide some further advice on how to add this kind of model to DART.

The primary consideration with a simple model is how you will store the state. If you have only a *single type of variable* in your state vector (for example, the Lorenz 63 model), here are some hints on how to implement your initialization and meta data routines:

Routine # / name	Suggested implementation
4. **static_init_model()**	Your model_size will likely be set by namelist, so read it, allocate an array of that size, and precompute all the locations for each state vector item. Call **add_domain()** with the model size so DART knows how long the state vector is.
5. **get_state_meta_data()**	Return `QTY_STATE_VARIABLE` as the quantity, and return the location for that index through a look-up into the location array created during **static_init_ model()**.

If you have *more than a single type of variable* in the state vector (for example, "concentration", "wind", etc. as in the `DART/models/simple_advection` model):

Routine # / name	Suggested implementation
4. **static_init_model()**	Read from the namelist the number of fields to be used in the state vector. Use **add_domain()** to indicate which netCDF vars should be read. Read in any auxiliary data needed by interpolation code (for example, the grid topology). Cache the grid locations of the state variables as appropriate, and use **get_domain_size()** to compute the model_size.
5. **get_state_meta_data()**	Call **get_model_variable_indices()** and **get_state_kind()** to figure out the (*i*,*j*,*k*) indices and which variable this offset is. Use the (*i*,*j*,*k*) index to compute the grid location and return it along with the quantity.

Now, for either type of simple model, the following applies:

Routine # / name	Suggested implementation
6. **end_model()**	Deallocate any arrays allocated in **static_init_model()**
7. **adv_1step()**	If possible, embed the code that computes $\mathbf{x}(t+1) = \mathbf{F}(\mathbf{x}(t))$ or call a separate subroutine to advance the model state from one time to another.
8. **shortest_time_between_assimilations()**	Return a namelist or a fixed value for the minimum model advance time.
9. **model_interpolate()**	Find the (i,j,k) indices which enclose that location, or search for the cell number. For some models you can compute (i,j) directly from a regular lat/lon grid, and in others you may have to search over a deformed grid. Any model code or utilities available for this purpose may prove very helpful as a starting point. In the end, you will use **get_state()** to retrieve an ensemble-sized array of values for each offset into the state vector, and then do interpolation to get an array of expected values.
10. **nc_write_model_atts()**	Optionally add any desired attributes to the output diagnostic files.

The remaining routines can mostly use the defaults, except possibly for 11. **read_model_time()** and 12. **write_model_time()**, which may need to be customized if using a model restart file that already stores time in a particular format.

Note that there is often no need to convert vertical obs or states in a simple model without vertical coordinate choices.

6.24 Suggestions for a "complex" model

A "complex" model is typically a large geophysical model where the model must be advanced outside of DART execution control. Here we provide some advice on how to integrate this kind of model with DART.

First of all, the 4. **static_init_model**, 5. **get_state_meta_data()** and 6. **end_model()** suggestions will match the multiple state variable in the previous section as complex models will typically have multiple fields.

An additional twist is that complex models may have different grid locations for different variables, (i.e. grid staggering), but the above instructions still apply.

The 7. **adv_1step()** method for a complex model should fail, so the default behavior is sufficient.

The advice for the 8. **shortest_time_between_assimilations()** routine is similar to the advice for a simple model: read the value from the namelist or return a fixed time as appropriate.

Note: Since the model will not be advanced by DART, the value returned here is irrelevant except for user information purposes.

For the remaining routines, we give the following implementation suggestions:

Routine # / name	Suggested implementation
9. **model_interpolate()**	Find the (*i*,*j*,*k*) indices which enclose that location, or search for the cell number. For some models you can compute (*i*,*j*) directly from a regular lat/lon grid, and in others you may have to search over a deformed grid. Any model code or utilities available for this purpose may prove very helpful as a starting point. In the end, you will use **get_state()** to retrieve an ensemble-sized array of values for each offset into the state vector, and then do interpolation to get an array of expected values.
10. **nc_write_model_atts()**	It is very helpful (but optional) to add grid information to assist in plotting your results.
11. **read_model_time()**	(see **write_model_time()** below)
12. **write_model_time()**	If the model time is stored in the netCDF files, supply routines that can read and write it in the correct format. The default routines will work if the model time matches what those routines expect: a time variable with an optional calendar variable. If no calendar is provided, the routine assumes fractional days. If the time variable is an array (i.e. more than one time step is stored in the file), read/write the last one.
13. **pert_model_copies()**	The default of adding Gaussian noise to all state variables may be undesirable. Complex models often have a method to perturb a state according to a particular formula or method. Otherwise, it may be necessary to perturb each variable with separate noise levels, only perturb certain variables, etc.
14. **convert_vertical_obs()**	(see **convert_vertical_state()** below)
15. **convert_vertical_state()**	Add code to convert between vertical coordinates (e.g. pressure, height, sigma levels, etc.) if appropriate. Code from the model or a model utility may be a very helpful starting point.
16. **get_close_obs()**	(see **get_close_state()** below)
17. **get_close_state()**	If you want to change the localization impact based on something other than the type or kind, put code here. You should test for vertical type and do the conversion on demand if it hasn't already been done.

As mentioned above, the most difficult routine to implement for a complex model is typically 9. **model_interpolate()**.

6.25 How to test your model_mod routines

The program `model_mod_check.f90` can be used to test the routines individually before running them with *filter*. Add a `mkmf_model_mod_check` and `path_names_model_mod_check` to your `DART/models/your_model/work` subdirectory. You might find it helpful to consult another model matching your model type (simple or complex). See the documentation for *model_mod_check* in `DART/assimilation_code/programs/model_mod_check` for more information on the tests available.

6.26 Controlling which files are output by filter

DART provides you with fine-grained control over how and when files are output. You can instruct DART whether or not to output files after each stage in an assimilation cycle. Since most experiments are run for more than one assimilation cycle, you can also instruct DART to aggregate all of the output for a specific stage into a single file.

These options are controlled by three settings in the `filter_nml` namelist in `input.nml`:

- `stages_to_write` specifies the stages during an assimilation cycle during which state files may be output. The possible stages are `'input'`, `'forecast'`, `'preassim'`, `'postassim'`, `'analysis'` and `'output'`. The input strings are case-insensitive, but the corresponding output files are always lowercase.
- `single_file_in` specifies how input state files are structured. If `.true.` the state of all ensemble members is expected to be read from single file. If `.false.` the state of each ensemble member expected to be read from its own file.
- `single_file_out` specifies how output state files are structured. If `.true.` the state of all ensemble members is output to a single file. If `.false.` the state of each ensemble members is output to its own file.

Caution: `single_file_out` only refers to the output **for a particular stage**. So even if you set `single_file_out = .true.`, you can get *several* output files - one per stage. If you set `single_file_out = .false.` filter will output a deluge of files. Be careful about what stages you choose to write.

6.26.1 Two common assimilation workflows

There are many ways to configure your data assimilation workflows. However, the following two workflows are sensible for small models and large models, respectively.

Small models

For models that read and write small state files and complete their numerical integrations relatively quickly, it makes sense to configure filter to:

1. complete multiple assimilation cycles
2. read from and write to a single output file for all ensemble members

This workflow requires setting `single_file_in = .true.` and `single_file_out = .true.`.

When `filter` is used for a long assimilation experiment, setting `single_file_out = .true.` will consolidate all the information for a particular stage into a single file that contains all the ensemble members, the mean, spread, inflation, etc.

This results in far fewer files, and each file may contain multiple timesteps to encompass the entirety of the experiment. Take note: since a single task must write each file, this setting engenders some computational overhead.

Large models

For models that read and write large state files and complete their numerical integrations relatively slowly, it make sense to configure filter to:

1. complete a single assimilation cycle at a time
2. read from and write to a seperate output file for each ensemble member

This workflow requires setting `single_file_out = .false.` and makes sense for large models or in cases where it is beneficial to run different number of MPI tasks for the model advances and the assimilation. In this case, there can be a substantial computational efficiency to have each ensemble member write its information to a separate file, and each file can be written simultaneously by different tasks. The tradeoff (at the moment) is that each of the files can only have a single timestep in them. Consequently, some files are redundant and should not be output.

6.26.2 Output and diagnostic files produced by filter

In the case when `single_file_out = .false.`

from *perfect_model_obs*		
`obs_seq.out`		the synthetic observations at some predefined times and locations
`perfect_output.nc`	1 timestep	a netCDF file containing the model trajectory - the true state

There are some namelist settings that control what files are output. Depending on the settings for *input.nml&filter_nml:stages_to_write* and others ...

from *filter*		
forecast_member_####.nc	1 timestep	the ensemble forecast, each ensemble member is a separate file
forecast_[mean,sd].nc	1 timestep	the mean and standard deviation (spread) of the ensemble forecast
forecast_priorinf_[mean,sd].nc	1 timestep	the prior inflation information before assimilation
forecast_postinf_[mean,sd].nc	1 timestep	the posterior inflation information before assimilation
preassim_member_####.nc	1 timestep	the model states after any prior inflation but before assimilation
preassim_[mean,sd].nc	1 timestep	the mean and standard deviation (spread) of the ensemble after any prior inflation but before assimilation
preassim_priorinf_[mean,sd].nc	1 timestep	the prior inflation information before assimilation
preassim_postinf_[mean,sd].nc	1 timestep	the posterior inflation information before assimilation
postassim_member_####.nc	1 timestep	the model states after assimilation but before posterior inflation
postassim_[mean,sd].nc	1 timestep	the mean and standard deviation (spread) of the ensemble after assimilation but before posterior inflation
postassim_priorinf_[mean,sd].nc	1 timestep	the (new) prior inflation information after assimilation
postassim_postinf_[mean,sd].nc	1 timestep	the (new) posterior inflation information after assimilation
analysis_member_####.nc	1 timestep	the model states after assimilation and after any posterior inflation
analysis_[mean,sd].nc	1 timestep	the mean and standard deviation (spread) of the ensemble after assimilation and after posterior inflation
analysis_priorinf_[mean,sd].nc	1 timestep	the (new) prior inflation information after assimilation
analysis_postinf_[mean,sd].nc	1 timestep	the (new) posterior inflation information after assimilation
output_[mean,sd].nc	1 timestep	the mean and spread of the posterior ensemble
output_priorinf_[mean,sd].nc	1 timestep	the (new) prior inflation information after assimilation
output_priorinf_[mean,sd].nc	1 timestep	the (new) posterior inflation information after assimilation
obs_seq.final		the model estimates of the observations (an integral part of the data assimilation process)

from both	
dart_log.out	the 'important' run-time output (each run of *filter* appends to this file; remove it or start at the bottom to see the latest values)
dart_log.nml	the input parameters used for an experiment

In the case when `single_file_out = .true.`

All the information for each stage is contained in a single file that *may* have multiple timesteps.

from *perfect_model_obs*		
`obs_seq.out`		the synthetic observations at some predefined times and locations
`perfect_output.nc`	N timesteps	a netCDF file containing the model trajectory - the true state

There are some namelist settings that control what files are output. Depending on the settings for `input.nml &filter_nml:stages_to_write` and others.

from *filter*		
`filter_input.nc`	1 timestep	The starting condition of the experiment. All ensemble members, [optionally] the input mean and standard deviation (spread), [optionally] the prior inflation values, [optionally] the posterior inflation values
`forecast.nc`	N timesteps	The ensemble forecast. All ensemble members, the mean and standard deviation (spread), the prior inflation values, the posterior inflation values
`preassim.nc`	N timesteps	After any prior inflation but before assimilation. All ensemble members, the mean and standard deviation (spread) of the ensemble, the prior inflation values, the posterior inflation values
`postassim.nc`	N timesteps	After assimilation but before posterior inflation. All ensemble members, the mean and standard deviation (spread) of the ensemble, the (new) prior inflation values, the (new) posterior inflation values
`analysis.nc`	N timesteps	After assimilation and after any posterior inflation. All ensemble members, the mean and standard deviation (spread) of the ensemble, the (new) prior inflation values, the (new) posterior inflation values
`filter_output.nc`	1 timestep	After assimilation and after any posterior inflation. All ensemble members, the mean and standard deviation (spread) of the ensemble, the (new) prior inflation values, the (new) posterior inflation values
`obs_seq.final`		the model estimates of the observations (an integral part of the data assimilation process)

from both	
`dart_log.out`	the 'important' run-time output (each run of *filter* appends to this file; remove it or start at the bottom to see the latest values)
`dart_log.nml`	the input parameters used for an experiment

6.27 Advice for models with multiple vertical coordinate options

6.27.1 DART vertical types for the 3D sphere locations type (threed_sphere)

A location when using the *MODULE location_mod (threed_sphere)* location module consists of a Latitude (-90 to 90), a Longitude (0 to 360), and a vertical value and type. The value is a real number. Possible types are:

- Height (in meters)
- Pressure (in Pascals)
- Model Level (index number)
- Scale Height (unitless)

- Surface (if value used, elevation of surface in meters)
- Undefined (entire vertical column)

If the model grid locations, all observation locations, and the choice of localization coordinate are all using the identical vertical type then no vertical conversion routines are needed. However, this is seldom the case.

6.27.2 Multiple vertical coordinate types

Most Earth System models and observations use latitude and longitude for horizontal coordinates or can generate them if needed (e.g. spectral models can transform their state into Lat/Lon coords). But often vertical coordinates pose additional complications.

Some models use terrain-following vertical coordinates, or a mix of pressure and terrain coordinates. Observation vertical locations are frequently reported in height or in pressure.

Additionally, if vertical localization is to be done in a different coordinate than the model or observations (e.g. scale height), then conversion routines are needed.

Vertical conversion routines typically take a DART `location_type` derived type and a desired output vertical coordinate type as inputs, and either update the location derived type or return a separate location type with the value converted to the requested type.

The conversion code may require additional auxilliary arrays from the model in order to convert the vertical coordinates accurately.

6.27.3 Varying vertical levels

If the computation of the vertical location depends on any of the fields in the state (e.g. pressure), then different ensemble members may compute different vertical locations.

Forward operators

During computation of expected values (Forward Operators), each ensemble member should compute the most accurate value regardless of whether the location in the model grid is consistent from member to member.

Localization

During assimilation the distance between model state values and the observation must be computed and only a single value can be returned, not an ensemble of distances. If part of the state is needed to compute the vertical location the ensemble mean is available to compute a single value which is representative of the entire state.

6.27.4 Choice of when conversion is done

During the assimilation phase of filter there are two options for when vertical conversion is done: all at the start, or on demand. If the observations to be assimilated are expected to impact all or almost all of the state, doing all vertical conversion at the start is more efficient. If the observations are expected to impact only a small percentage of the state variables then doing it on demand is more efficient.

The options here are namelist selectable at runtime and the impact on total runtime can be easily measured and compared.

6.28 Data management in DART

One of the more challenging aspects of an ensemble Data Assimilation (DA) system is the need to manage large amounts of memory to store ensembles of the model state.

Most contemporary large models run in parallel on multi-processor computer systems and distribute the data across multiple memory nodes to support finer grids, smaller timesteps and longer modeling time periods. Common computer science strategies include using shared memory on individual nodes and using the Message Passing Interface (MPI) libraries on distributed memory nodes.

Ensemble DA exacerbates this memory problem by requiring multiple copies, often 20-100x, of the model data to do the assimilation.

DART uses the MPI libraries to distribute ensembles of model state data across distributed memory nodes. For models with small amounts of data the code can be compiled and run as a serial program but when compiled with MPI it can scale up to 10,000s of nodes using Giga to Petabytes of memory.

Memory usage and internode communication time are mutually incompatible items to minimize. DART has different strategies that can be selected at runtime to use less memory per node at the cost of more time spent in communication of data between nodes, or use more memory per node and minimize communication time.

The following descriptions detail the different phases of the main assimilation program in DART, called `filter`, and what options exist for memory layout and management.

6.28.1 Ensembles of data

State data

- N ensemble members times X items in the state vector, always resident.
- 6 additional copies of X items for inflation, ensemble mean & sd, etc.

Observations

Allocated and deallocated if looping over multiple assimilation windows within a single run of filter.

- Only observations within the current assimilation window, O
- O observations times N ensemble members for the Forward Operator (FO) results
- O observations times N ensemble members for the QC results

Delayed writing option

If selected in the namelist, up to P phases (input, forecast, preassim, postassim, analysis, output) of the state data are stored in memory and written out at the end of filter.

6.28.2 Filter run phases

FO computation, prior and posterior

Run-time options include allocating spaces for two layouts and transposing between them, or running distributed in 'all copies' mode.

Assimilation

Distributed FO and QC observation ensembles

Runtime option to either replicate the model state ensemble mean on each MPI task or run with that ensemble fully distributed.

6.28.3 Ensemble memory usage and layout

Transposable

Data is distributed over T MPI tasks but during the program execution the data is communicated between tasks to alternate between two different data layouts.

Allocations are needed for two different 2D arrays:

- N ensemble members times (X items/T tasks)
- X items times (N ensemble members/T tasks)

Distributed

Data is distributed over T MPI tasks but only a single data array is used:

- N ensemble members times (X items/T tasks)

Replicated

The same data array is replicated on each MPI task:

- X items per task

6.29 Programs included with DART

This list of programs is separated into groups which have similar functionality. Within each group they are sorted by the order in which they might be used and/or by how widely they are used.

6.29.1 Setting Up Experiments

In many cases, you won't need to use any programs in Setting Up Experiments except for `preprocess`, because you're using an existing model interface and have the observation sequence files. In that case, you the programs you're looking for are probably in *Assimilation Programs*.

preprocess
Program to insert observation specific code into DART before filter or perfect_model_obs is compiled.

fill_inflation_restart
Create inflation restart files with constant values taken from `fill_inflation_restart_nml`.

obs_impact_tool
Construct a table that is read by filter at run-time to localize the impact of sets of observation types on sets of state vector quantities.

model_mod_check
Program to test some of the more fundamental routines in any `model_mod`, especially a for a new model.

perturb_single_instance
Generate an ensemble of perturbed ensemble member restart files. (Alternatively, you might perturb the model state using `model_nml` variables).

gen_sampling_err_table
Computes a table of values needed to apply Sampling Error Correction (SEC), which corrects covariances based on small sample size statistics.

6.29.2 Creating Observation Sequence Files

create_obs_sequence
Creates a short andor synthetic observation sequence file using values read from standard input.

create_fixed_network_seq
Reads observation sequence file information from standard input and replicates it multiple times in a second observation sequence file, at user specified dates.

obs_utils/create_obs_grid
Create a set of observations located on a regular grid. Obs have no data values, but they are time ordered.

obs_utils/obs_timejitter
Randomly perturb the times of the observations in a (usually) set_def.out file. Writes the results to (usually) obs_seq.in.

6.29.3 Querying Observation Sequence Files

obs_utils/obs_info
Summarize obs types, times, counts found in observation sequence file(s).

obs_utils/obs_assim_count
Prints out a quick table of obs types and counts, overall start and stop times, and metadata strings and counts. See obs_diag for more. There is an older version in the obs_assim_count directory.

obs_seq_coverage
Queries a set of observation sequence files to determine which observation locations report frequently enough to be useful for a verification study.

obs_total_error
Prints the total error in the mean and spread from an obs_seq file which has been through both perfect_model_obs and filter, so it has copies 'truth', 'ensemble mean', and 'ensemble spread'. You can get more information by running the obs_diag program.

6.29.4 Changing Observation Sequence Files

obs_sequence_tool
Subsets, combines, or alters observations from one or more observation sequence files and optionally writes them into a single output obs_seq file.

obs_loop
A template to read in observations from one obs_seq file and write them, optionally modified by user supplied code, to another obs_seq file.

obs_utils/obs_sort
Do a complete sort of an obs_seq file by location, observation type, then variance. An ancestor of obs_remove_dups.

obs_utils/obs_remove_dups
Removes duplicate observations from an obs_seq file, which involves a complete sort by time, location, observation type, then variance.

obs_selection
Extracts observations out of one or more obs_sequence files according to a list of observation types, times, and locations. The list is usually created by *obs_seq_coverage*, but can be an observation sequence file.

obs_common_subset
Select the subset of observations, which were successfully assimilated, from two or more assimilation cases (which used the same obs_seq.out file).

obs_keep_a_few
Creates an output observation sequence file that is shorter than the input obs_seq file.

obs_seq_verify
Reorders the observations from a forecast run of DART into a structure that is amenable for the evaluation of the forecast.

obs_utils/obs_data_denial
THIS IS NOT YET DONE! Help implement a data-denial experiment by randomly changing the error variance of N of each obs type in an observation sequence file to a huge value.

6.29.5 Assimilation Programs

perfect_model_obs
Creates synthetic observation sequences from a hindcast model.

filter
Main Fortran program for driving ensemble filter assimilations.

advance_time
Provides a shell-scripting-friendly way to increment and decrement calendar dates and times.

integrate_model
Generic main program which advances a single ensemble member in `perfect_model_obs` or the serial or parallel version of the `filter` program.

6.29.6 Evaluating Results

obs_diag
Reads obs_seq.final files, calculates statistics, and writes them to NetCDF files for use by Matlab (or other) plotting scripts. There are separate versions for models with different coordinate systems:

- *1D*
- *3D Cartesian*
- *3D spherical*
- 3D spherical with streamflow.

obs_seq_to_netcdf
Extracts the observation components from observation sequence files and writes out netCDF files that can be used by other applications. such as `diagnostics/matlab/plot_obs_netcdf*` There are two versions; the *standard version* and one which filters out radiance metadata which is not needed by the scripts which use the resulting NetCDF file.

compare_states
Compare fields in two NetCDF files and print out the min and max values from each file and of the difference between the two files.

compute_error
Compute the time-mean ensemble error and spread in the same manner as the DART MATLAB diagnostic routine `plot_total_err`; in state space from true_state.nc and preassim.nc (or analysis.nc).

closest_member_tool
Prints out a sorted order of which ensemble members are 'closest' to the mean, where the method for computing the 'close' metric is selectable by namelist option.

6.29.7 Historical and Deprecated

system_simulation
A collection of standalone programs for simulating various properties of ensembles. Talk to Jeff Anderson about the programs in this directory.

wakeup_filter
For use in the "async=4" case where both the main filter program and the hindcast model are MPI programs. The main MPI job script runs each of the model advances for the ensemble members, and then runs this program to restart the filter program.

6.30 Adding your observations to DART

First, you should understand that DART already supports a tremendous variety of observations. To fully support an observation means that the observation can be converted from its native format to the DART observation sequence format and that the observation forward operator is already implemented. Keep in mind that forward operators are not specific to any one model.

The observation converters are in the *observations/obs_converter* directory and you should look there for the documentation describing which converters are available.

The forward operators are functionally or logically grouped into Fortran modules in the *observations/forward_operator* directory. DART employs a 'contractual' style of programming in that the forward operator requests information from the model, and if the model cannot provide it, the forward operator may request different information in an attempt to

collect the information needed to apply the operator. If the model cannot provide any of the required information, the forward operator fails, the DART QC for that observation is set to the appropriate value, and the program continues.

6.31 How DART supports different types of observations: the preprocess program

DART's *preprocess program* actually writes the source code that supports observations. This source code is then used by other modules.

6.31.1 The rationale for `preprocess`

Certain types of data require additional metadata in order to be assimilated. For example, while radiosondes only require the observation location in order to be assimilated, radar observations need extra metadata to specify the location of the radar in addition to the location of the observation. GPS occultations need the locations of the two satellites so the forward operator can integrate along the raypath. Cosmic ray soil moisture sensors have forward operators that require site-specific calibration parameters that are not part of the model and must be included in the observation metadata.

The potential examples are numerous.

Since each 'observation quantity' may require different amounts of metadata to be read or written, any routine to read or write an observation sequence **must** be compiled with support for those particular observations. This is the rationale for the inclusion of `preprocess` in DART. The supported observations are listed in the `obs_kind_nml` namelist in `input.nml`.

For this reason, we strongly recommend that you use the DART routines to read and process DART observation sequence files.

Important: You **must** actually run `preprocess` before building any executables. It is an essential part of DART that enables the same code to interface with multiple models and observation types. For example, `preprocess` allows DART to assimilate synthetic observations for the Lorenz_63 model and real radar reflectivities for WRF without needing to specify a set of radar operators for the Lorenz_63 model.

`preprocess` combines multiple `obs_def` and `obs_quantity` modules into one `obs_def_mod.f90` that is then used by the rest of DART. Additionally, a new `obs_kind_mod.f90` is built that will provide support for associating the specific observation **TYPES** with corresponding (generic) observation **QUANTITIES**.

The list of `obs_def` and `obs_quantity` module source codes are contained in the `&preprocess_nml` namelist in `input.nml`. These modules determine what observations and operators are supported.

Warning: If you want to add another `obs_def` module, you **must** rerun `preprocess` and recompile the rest of your project.

6.31.2 Example preprocess namelist

As an example, if a preprocess_nml namelist in input.nml looks like:

```
&preprocess_nml
    input_obs_kind_mod_file  = '../../../assimilation_code/modules/observations/DEFAULT_
↪obs_kind_mod.F90'
    output_obs_kind_mod_file = '../../../assimilation_code/modules/observations/obs_kind_
↪mod.f90'
    quantity_files           = '../../../assimilation_code/modules/observations/
↪atmosphere_quantities_mod.f90',
    input_obs_def_mod_file   = '../../../observations/forward_operators/DEFAULT_obs_def_
↪mod.F90'
    obs_type_files           = '../../../observations/forward_operators/obs_def_gps_mod.
↪f90',
                               '../../../observations/forward_operators/obs_def_QuikSCAT_
↪mod.f90',
                               '../../../observations/forward_operators/obs_def_GWD_mod.
↪f90',
                               '../../../observations/forward_operators/obs_def_
↪altimeter_mod.f90',
                               '../../../observations/forward_operators/obs_def_
↪reanalysis_bufr_mod.f90'
    output_obs_def_mod_file  = '../../../observations/forward_operators/obs_def_mod.f90'
    /
```

preprocess will combine the following modules:

- DEFAULT_obs_def_mod.F90
- obs_def_gps_mod.f90
- obs_def_QuikSCAT_mod.f90
- obs_def_GWD_mod.f90
- obs_def_altimeter_mod.f90
- and obs_def_reanalysis_bufr_mod.f90

into obs_def_mod.f90. This resulting module can be used by the rest of the project.

6.31.3 Building and running preprocess

Since preprocess is an executable, it must be compiled following the procedure of all DART executables:

1. The DART/build_templates/mkmf.template must be correct for your environment.
2. The preprocess_nml namelist in input.nml must be set properly with the modules you want to use.

If those two conditions are met, you can build and run preprocess using these commands:

```
$ csh mkmf_preprocess
$ make
$ ./preprocess
```

The first command generates an appropriate Makefile and the input.nml.preprocess_default file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces the preprocess

executable file. The third command actually runs preprocess - which builds the new `obs_kind_mod.f90` and `obs_def_mod.f90` source code files. Once these source code files are created, you can now build the rest of DART.

6.32 How DART stores observations: observation sequence (obs_seq) files

Since DART is designed to assimilate observations from any data source, it includes a set of programs to convert observations from their original format to DART's own observation sequence, or `obs_seq`, format. The `obs_seq` format is designed to allow DART to accomodate a myriad of source observation file formats, structure and metadata. Many original source observation files don't contain the necessary information about the error characteristics and spatial structure of the data needed to perform an assimilation.

There are three types of `obs_seq` files.

6.32.1 obs_seq.in

An `obs_seq.in` file actually contains no observation quantities. It may be best thought of as a perfectly laid-out notebook waiting for an observer to fill in the actual observation quantities.

All the rows and columns are ready, labelled, and repeated for every observation time and platform. The `obs_seq.in` file is generally the start of a "perfect model" experiment.

In a perfect model experiment, one instance of the model is run through the DART program `perfect_model_obs` - which applies the appropriate forward operators to the model state and writes down the observations generated by the model in the writes them down in the perfectly laid-out notebook.

The completed notebook is then renamed `obs_seq.out`.

6.32.2 obs_seq.out

An `obs_seq.out` file contains a linked list of observations. The observations can potentially be (and usually are) from different platforms and of different quantities, each with their own error characteristics and metadata.

An `obs_seq.out` file containing real data can be generated by using one of DART's many observation converter programs. Additionally, an `obs_seq.out` file containing synthetic data can be created by running DART's `perfect_model_obs` program.

The observations in the `obs_seq.out` files are assimilated into the model ensemble by DART's `filter` program.

To learn more about the structure of the `obs_seq.out` file, see *Detailed structure of an obs_seq file*.

If you want to create an observation sequence file from real observations, you should contact DAReS staff by emailing dart@ucar.edu for advice regarding your specific types of observations.

6.32.3 obs_seq.final

When running an assimilation, DART's `filter` program assimilates the observations contained in the `obs_seq.out` file and generates an `obs_seq.final` file.

The `obs_seq.final` file contains everything in the `obs_seq.out` file and also contains a few additional 'copies' of the observation.

Since DART is an ensemble algorithm, each ensemble member must compute its own estimate of the observation for the algorithm. You can save the ensemble members' estimates of the observation in the `obs_seq.final` file by setting the `num_output_obs_members` entry in the `filter_nml` namelist of `input.nml` to a value greater than zero.

Minimally, `filter` will record the mean and spread of the ensemble estimates in the `obs_seq.final` file.

To learn more about the structure of the `obs_seq.final` file, see *Detailed structure of an obs_seq file*.

6.32.4 Using obs_seq.final for observation-space diagnostics

The best method to determine the performance of an experiment in which you assimilate data from real-world sources is to compare the ensemble estimates of the observation to your real-world data. You can estimate the bias and error of the ensemble mean or gauge how many of the real-world observations are actually being assimilated. These diagnostics are known as observation-space diagnostics.

DART provides *programs obs_diag* and *MATLAB® observation space diagnostics* for you use to quickly assess the performance of your experiment.

Note: Since each 'observation type' may require different amounts of metadata to be read or written, any routine to read or write an observation sequence **must** be compiled with support for those particular observations. The supported observations are listed in the `obs_kind_nml` namelist of `input.nml`. For more information, see *How DART supports different types of observations: the preprocess program*.

6.33 Detailed structure of an obs_seq file

Since the configuration of model ensembles and the characteristics of assimilated observations are highly variable, observation sequence files are highly adjustable to accomodate these variations.

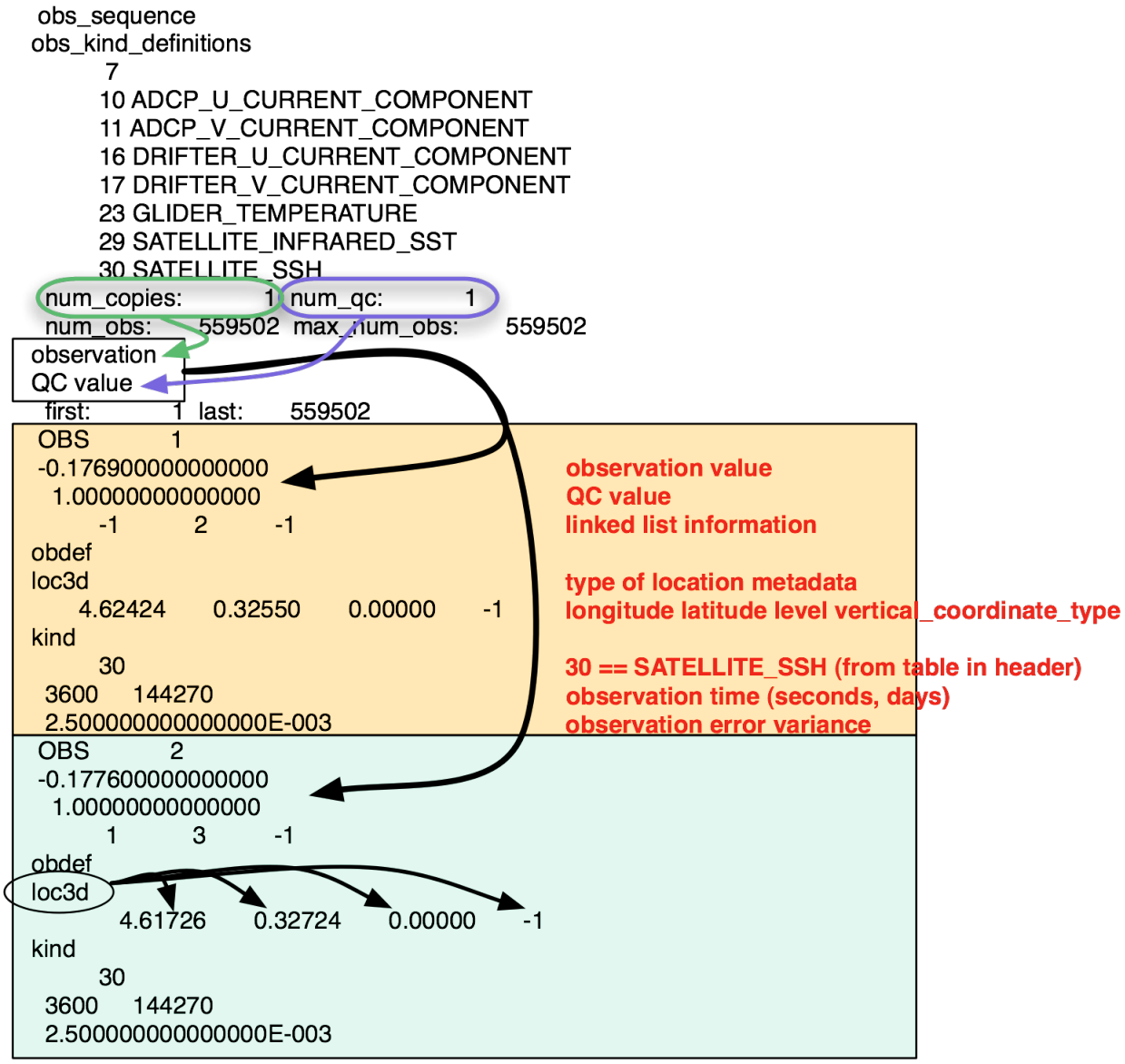
There are many extensible parts of an observation sequence file. The following aspects of an observation sequence file can be adjusted:

- the number of observation kinds contained in the file
- whether the locations have one or more components
- how many quality control values are available for each observation
- where those quality control values come from
- how many copies of each observation there are

The following two diagrams demonstrate the structure of an `obs_seq.out` and an `obs_seq.final` file, respectively.

Note: These example files are from entirely separate experiments. They were selected to demonstrate the flexibility of the observation sequence file format.

6.33.1 obs_seq.out structure



6.33.2 obs_seq.final structure

```
 obs_sequence
obs_kind_definitions
          15
           1 RADIOSONDE_U_WIND_COMPONENT
           2 RADIOSONDE_V_WIND_COMPONENT
           3 RADIOSONDE_SURFACE_PRESSURE
           4 RADIOSONDE_TEMPERATURE
           5 RADIOSONDE_SPECIFIC_HUMIDITY
           6 AIRCRAFT_U_WIND_COMPONENT
           7 AIRCRAFT_V_WIND_COMPONENT
           8 AIRCRAFT_TEMPERATURE
           9 ACARS_U_WIND_COMPONENT
          10 ACARS_V_WIND_COMPONENT
          11 ACARS_TEMPERATURE
          12 MARINE_SFC_U_WIND_COMPONENT
          13 MARINE_SFC_V_WIND_COMPONENT
          14 MARINE_SFC_TEMPERATURE
          15 MARINE_SFC_SPECIFIC_HUMIDITY
  num_copies:            5  num_qc:            2
  num_obs:        37695  max_num_obs:        37695
NCEP BUFR observation
prior ensemble mean
posterior ensemble mean
prior ensemble spread
posterior ensemble spread
NCEP QC index
DART quality control
  first:            1  last:        37695
 OBS            1
 1009.76377118761002
 1008.61783794436531
 1009.92390496581413
 0.799858860231082436
 0.202591644167762347
 2.00000000000000000
 0.000000000000000000E+00
 -1 2 -1
obdef
loc3d
      4.433480      0.858041     917.000000      -1
kind
 3
 64800     148425
 1.00000000000000000
 OBS            2
 1013.65517759213003
 1014.38817672596736
 1015.08278344676989
 1.21335841600085725
 0.300925406401195361
 2.00000000000000000
 0.000000000000000000E+00
 1 3 -1
obdef
loc3d
      4.494620      0.894229     498.000000      -1
kind
 3
 64800     148425
 1.00000000000000000
```

More kinds of observations

5 copies

2 kinds of QC

5 copies

2 kinds of QC

6.34 Creating an obs_seq file of synthetic observations

There are several steps to create an observation sequence file, which follows directly from the modular nature of the DART programming philosophy. This procedure may be used to create synthetic observations from *any model*.

1. Decide what observations you want to investigate and edit the `input.nml&obs_kind_nml` block.
2. Build and run *preprocess* to create code that supports the observations you want.
3. Build and run *create_obs_sequence* to define the specifics about the observation you want.
4. Build and run *create_fixed_network_sequence* to replicate those specifics through time.
5. Build and run *perfect_model_obs* to create an observation consistent with the model state and specified error distribution at the requested times and locations.

These programs are described in *Programs included in DART*.

6.34.1 Example: generating observations for the Lorenz '63 model.

While this procedure works with any model, the responses in 'create_obs_sequence' will vary based on what observations are supported. You should not expect the responses for observations for L63 can be used to produce radar observations from WRF, for example. When compiled with support for radar observations, *create_obs_sequence* **will** prompt you for the required metadata.

1) There are no 'real' observations for the Lorenz '63 model, so the appropriate namelist settings are:

```
&obs_kind_nml
    assimilate_these_obs_types = 'RAW_STATE_VARIABLE'  /

&preprocess_nml
     input_obs_def_mod_file = '../../../observations/forward_operators/DEFAULT_obs_def_
↪mod.F90'
    output_obs_def_mod_file = '../../../observations/forward_operators/obs_def_mod.f90'
    input_obs_kind_mod_file = '../../../assimilation_code/modules/observations/DEFAULT_
↪obs_kind_mod.F90'
   output_obs_kind_mod_file = '../../../assimilation_code/modules/observations/obs_kind_
↪mod.f90'
               input_files = '../../../observations/forward_operators/obs_def_1d_state_
↪mod.f90'
  /
```

2) Run *preprocess* in the normal fashion.

3) *create_obs_sequence* creates an *observation set definition* (typically named `set_def.out`), the time-independent part of an observation sequence. It may help to think of it as trying to define what sorts of observations will be taken at one 'reading' … you walk out to the box and take temperature, humidity, and wind observations all at the same time and place, for example. You can think of it as one page in an observer's notebook, and only contains the *location, type,* and *observational error characteristics* (normally just the diagonal observational error variance) for a related set of observations. There are no actual observation values, nor are there any times associated with the definition. The program is interactive and queries the user for the information it needs. Begin by creating a minimal observation set definition in which each of the 3 state variables of L63 is directly observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Starting program create_obs_sequence
 Initializing the utilities module.
 Trying to log to unit   10
 Trying to open file dart_log.out

 ------------------------------------
 Starting ... at YYYY MM DD HH MM SS =
                 2017  3 28 10 15 30
 Program create_obs_sequence
 ------------------------------------

 set_nml_output Echo NML values to log file only
 Trying to open namelist log dart_log.nml
 ------------------------------------------------------


 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RAW_STATE_VARIABLE
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 ------------------------------------------------------

 ---------- USE_PRECOMPUTED_FO_OBS_TYPES --------------
 ------------------------------------------------------

 Input upper bound on number of observations in sequence
4
 Input number of copies of data (0 for just a definition)
0
 Input number of quality control values per field (0 or greater)
0
 input a -1 if there are no more obs
0
       Input -1 * state variable index for identity observations
       OR input the name of the observation type from table below:
       OR input the integer index, BUT see documentation...
         1 RAW_STATE_VARIABLE
-1
 input time in days and seconds
0 0
 Input error variance for this observation definition
1.0
 input a -1 if there are no more obs
0

 { this gets repeated ... until you tell it to stop ... }

 input a -1 if there are no more obs
-1
 Input filename for sequence (  set_def.out   usually works well)
 set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

Rest assured that if you requested to assimilate more realistic observation types, you will be queried for appropriate information by *create_obs_sequence*. Below is a table that explains all of the input you should need to supply for observations of the L63 model state.

```
4            # upper bound on num of observations in sequence
0            # number of copies of data (0 for just a definition)
0            # number of quality control values per field (0 or greater)
0            # -1 to exit/end observation definitions

-1           # observe state variable 1
0   0        # time -- days, seconds
1.0          # observational variance
0            # -1 to exit/end observation definitions

-2           # observe state variable 2
0   0        # time -- days, seconds
1.0          # observational variance
0            # -1 to exit/end observation definitions

-3           # observe state variable 3
0   0        # time -- days, seconds
1.0          # observational variance
-1           # -1 to exit/end observation definitions

set_def.out  # Output file name
```

4) *create_fixed_network_sequence* takes the observation set definition and repeats it in time, essentially making multiple pages in our notebook. Again, the program is interactive and queries the user for information. You should be able to simply follow the prompts. The table below represents the input needed for the L63 example:

```
set_def.out    # Input observation set definition file
1              # Regular spaced observation interval in time
1000           # 1000 observation times
0, 43200       # First observation after 12 hours (0 days, 12 * 3600 seconds)
0, 43200       # Observations every 12 hours
obs_seq.in     # Output file for observation sequence definition
```

5) *perfect_model_obs* advances the model from the state defined by the initial conditions file specified in the `input.nml` and 'applies the forward operator' to harvest observations to fill in the observation sequence specified in `obs_seq.in`. The observation sequence finally has values for the observations and is saved in a file generally named *obs_seq.out*. *perfect_model_obs* is namelist-driven, as opposed to the previous two (whose input is a lot harder to specify in a namelist). Take a look at (and modify if you like) the `input.nml&perfect_model_obs_nml` section of the namelist.

The End. Not only should you have an observation sequence file (usually `obs_seq.out`) , you also have a file containing the exact evolution of the model consistent with those observations - the true state: `perfect_output.nc`.

6.35 Creating an obs_seq file from real observations

Real observations come in a mind-boggling diversity of formats. We have converters for many formats in the `DART/observations/obs_converters` directory. The documentation for that directory is listed in *DART Observations*.

The converters are designed to work on one input file format and create (or add to) an output observation sequence. It may be desirable to post-process multiple observation sequence files with the *program obs_sequence_tool* to select for timeframe, geographic region, etc.

Many of the formats require their own libraries (like HDF), and require intimate knowledge of the data format to extract the portions required for the *DART observation sequence file*.

You should feel free to browse the converters and their companion documentation. If you create a new observation coverter for a format that DART doesn't already support, please follow the *Contributors' guide* to add your code to DART. These types of contributions are greatly appreciated by DAReS staff and by the geoscience community!

The DART framework enforces a clean separation between observations and the models used for assimilation. The same observations can be used in any model which understands how to generate a value for the requested type of observation from the models' state-space values (i.e. the forward observation operator must exist - DART provides many for the most common state variables).

In many cases, the original datasets are in a standard scientific format like netCDF, HDF, or BUFR, and library routines for those formats can be used to read in the original observation data. The DART software distribution includes Fortran subroutines and functions to help create a sequence of observations in memory, and then a call to the DART observation sequence write routine will create an entire *obs_seq* file in the correct format.

In many cases, a single, self-contained program can convert directly from the observation location, time, value, and error into the DART format. In other cases, especially those linking with a complicated external library (e.g. BUFR), there is a two-step process with two programs and an ASCII intermediate file. We are currently leaning towards single-step conversions but either approach can be used for new programs.

The DART system comes with several types of location modules for computing distances appropriately. The two most commonly used are for data in a 1D system and for data in a 3D spherical coordinate system. All the programs in the `DART/observations` directory assume the `assimilation_code/location/threed_sphere/location_mod.f90` 3D sphere location module is being used.

With the myriad of observation file formats, HDF, Grib, BUFR, netCDF, ... we simply have not had the time nor need to support all of them. The converters are a work in progress. There are currently about 10 other observation sources and types which we are in the process of collecting information and conversion programs for and which will eventually be added to this directory. In the meantime, if you have converters for data or interest in something that is not in the repository, please email the DART group. Your best bet is to contact our group at *dart@ucar.edu* with a specific request and we can steer you to the most similar process.

6.35.1 Overview

Real-world observations of earth-system data come from a variety of sources, including radiosondes, satellites, ships, aircraft, weather stations, etc. The files in this `observations` directory can be used to convert data from a variety of native formats into a common DART *observation sequence* format.

Synthetic observations are those not based on an actual instrument reading of a system, but instead are fabricated to have a known value, or have values computed by running a model, possibly with a fixed amount of simulated noise added. These observations can be used for testing, determining the sensitivity of the model to assimilation, and for designing new observation systems. The DART system includes several ways to create synthetic observations. For more information, see *Creating an obs_seq file of synthetic observations*.

The DART framework enforces a clean separation between observations and the models they are assimilated into. The same observations can be used in any model which understands how to generate a value for the requested type of

observation from its state space values.

In many cases a single, self-contained program can convert directly from the observation location, time, value, and error into the DART format. In other cases, especially those linking with a complicated external library (e.g. BUFR), there is a two-step process with two programs and an ASCII intermediate file. We are currently leaning towards single-step conversions but either approach can be used for new programs.

Frequently the original datasets are in a standard scientific format like netCDF, HDF, or BUFR, and library routines for those formats can be used to read in the original observation data.

The DART software distribution includes Fortran subroutines and functions to help create a sequence of observations in memory, and then a call to the DART observation sequence write routine will create an entire *obs_seq* file in the correct format.

The DART system comes with several types of location modules for computing distances appropriately. Two of the ones most commonly used are for data in a 1D system and for data in a 3D spherical coordinate system. All the programs here assume the `location/threed_sphere/location_mod.f90` 3D sphere location module is being used.

There are currently some additional observation sources and types which we are in the process of collecting information and conversion programs for and which will eventually be added to this directory. In the meantime, if you have converters for data or interest in something that is not in the repository, please contact DAReS staff by emailing dart@ucar.edu.

6.35.2 Data sources and formats

See the various subdirectories here, which generally include information on where the example data was obtained and in what format it is distributed. Most data is available for download off the web. The Data Support Section (DSS) at NCAR has large data repositories, the MADIS data center distributes observations in netCDF format, GTS real-time weather data is available from various sources. For new converters, if you can find what format the data is distributed in you may be able to adapt one of the existing converters here for your own use. Formats read by the existing converters include netCDF, HDF, little-r, text, Prepbufr, amongst others.

See the current list of *converter programs*

If you have looked and none of the existing converters are right for your data, here are some suggestions for where to start creating a new converter. Create a new subdirectory in the *observations* directory. Copy with the recursive option (*cp -r*) one of the existing converters and adapt to your needs. Our suggestions for which converter to start from depends on the format of your input observations to be converted. If your input data format is:

format	advice
netCDF	Start with the *MADIS* converters, and in particular try the `convert_madis_profiler.f90` file because it is the most straightforward. Another good option is `SST/oi_sst_to_obs.f90`.
Comma separated text	Start with the *Ameriflux* converter.
Generic text	Start with the *text* converter.
HDF-EOS5	Start with the *AIRS* converter.
BUFR or prepBUFR	Start with the *NCEP* converter.
Dense data, like Satellite swaths	Start with the *tpw* converter, which includes code that averages the raw data in space and time.
Ray-path integrated data	Start with the *GPS* converter, which includes code that traces a path and integrates values along the ray.
World Ocean Database packed ASCII	Start with the *WOD* converter.

6.35.3 Decisions you might need to make

Time

Time enters into the assimilation system in 3 places: the timestamp of the state vector data (the current model time when this data was produced), the time of each observation, and the minimum time period the model should be called to advance (the assimilation window size). The internal timestepping of the model is unrelated to any of these times and is outside the scope of the assimilation system.

The basic time type in DART is a pair of integers; one for the day number and one for the number of seconds. Generally the low order models, which aren't direct geophysical models, use time directly as a sequence of days starting at 0 and incrementing in any appropriate number of seconds or days. The observations assimilated into these systems do not need to use a calendar.

Observations of a real-world system usually are distributed with a year/month/day, hour/min/seconds timestamp. There are routines in DART to convert back and forth between the (day-number/seconds) format and a variety of (year/month/day) calendars. For more details on how DART stores time information and the types of available calendars, see *MODULE time_manager_mod*.

Some climate models which do long runs (100s or 1000s of years) use a modified calendar for simplicity in computation, e.g. months which always have 30 days, or no leap years. When trying to assimilate real observations into these models there may be calendar issues to solve.

The smallest resolvable unit of time in DART is a second. To model a system which operates on sub-second time scales the time can be scaled up by some factor. As long as the observation time, the state data time, and the minimum model advance time are expressed in the same scaled time units, there is no problem.

Error variances

Observations must specify an associated expected error variance. Each individual observation stores its own error variance value, so it can be a constant value for all observations of that type or it can vary by location, by height, by magnitude of the observed value, etc. This value is the expected instrument error variance plus the representativeness error variance of the model. The model error variance includes deficiencies in the equations representing the processes of the system as well as errors introduced by representing a continuous system as a series of discrete points. While the instrument error and the representativeness error could be specified separately, they each have the same impact on the assimilation and can be difficult to determine with any real accuracy. For simplicity, in DART (and most current assimilation software) they are combined and specified as a single value, which we frequently call the 'observation error'. Keep in mind we really mean 'observation error variance'.

The instrument error is generally supplied by the instrument maker. Sadly, it is frequently surprisingly difficult to find these values. For the representativeness error, you can generate a set of artificial observations with the *program perfect_model_obs* and then run an assimilation experiment to generate an estimate of the error in the model.

In practice, however, most people make an educated guess on the values of the error and then start with a larger than expected value and decrease it based on the results of running some test assimilations.

For these tests, the namelist for the outlier threshold in the `filter_nml` namelist of `input.nml` should be disabled by setting it to -1 (the default value is 3). This value controls whether the observation is rejected because the observed value is too far from the ensemble mean.

If the diagnostics show that the difference between the mean of the forward operators and the observed value is consistently smaller than the specified observation error, then the error is probably too large. A error that is too large reduces the impact of an observation on the state. If the specified observation error is too small it is likely the observation will be rejected when the outlier threshold is enabled, and the observation will not be assimilated. It is important to look at the output observation sequence files after an assimilation to see how many observations were assimilated or rejected, and also at the RMSE (root mean squared error) versus the total spread. DART includes Matlab diagnostic routines to create these types of plots. The observation RMSE and total spread should be roughly commensurate. The total spread

includes contributions from both the ensemble variance and the observational error variance, so it can be adjusted by changing the error values on the incoming observations.

There are other ways to adjust the ensemble spread, including *Inflation*, so the observation error is not the only factor to consider.

One last recommendation: if possible, the Prior forward operator values should be compared against the observations after several assimilation cycles. If you plot results using the Posterior values it is always possible for the assimilation to overfit the observations and look good on the diagnostic plots. But the actual test is to then advance the model and look at how the forecast of the state compares to the observations.

Observation types

All observations have to have a specific 'type'. There are namelist controls to turn on and off the assimilation of observations at run-time by type, or to only evaluate the forward operator for an observation but have no impact on the state. Several of the diagnostics also group observations by type to give aggregate statistics after an assimilation. Generally types are based on both the observing platform or instrument as well as the 'kind' of observation, e.g. RADIOSONDE_TEMPERATURE, ARGO_SALINITY, etc. Each type is associated with a single underlying generic 'kind', which controls what forward operator code is called inside the model, e.g. QTY_TEMPERATURE, QTY_DENSITY, etc.

For more details on how to use and add new DART types, see the *MODULE obs_def_mod*.

The DART `obs_kind_mod.f90` defines a list of already defined observation types, and users can either use existing observation types in 'obs_def_xxx_mod.f90' files, or define their own. Be aware that `obs_kind_mod.f90` is auto-generated by the *PROGRAM preprocess*, so until you configure and run `preprocess`, `obs_kind_mod.f90` will not exist.

Observation locations

The two most common choices for specifying the location of an observation are the *MODULE location_mod (threed_sphere)* and the *MODULE (1D) location_mod* locations.

For observations of a real-world system, the 3D Sphere is generally the best choice. For low-order, 1D models, the 1D locations are the most commonly used. The observation locations need to match the type of locations used in the model in that you cannot read observations on a unit circle (1D) when using models that require 3D Sphere locations.

The choice of the vertical coordinate system may also be important. For the 3D Sphere, the vertical coordinate system choices are:

string	integer value	meaning
VERTISUNDEF	-2	has no specific vertical location (undefined)
VERTISSURFACE	-1	surface value (value is surface elevation in m)
VERTISLEVEL	1	by model level
VERTISPRESSURE	2	by pressure (in pascals)
VERTISHEIGHT	3	by height (in meters)
VERTISSCALEHEIGHT	4	by scale height (unitless)

The choice of the vertical coordinate system may have ramifications for vertical localization, depending on your model's ability to convert from one coordinate system to another. `VERTISUNDEF` is typically used for column-integrated quantities. `VERTISLEVEL` only makes sense for synthetic observations.

When observations are declared to be `VERTISSURFACE` or `VERTISUNDEF` it is not possible to compute a vertical distance between the observation and anything else. Consequently, the distance between that observation and everything else (state, other observations) is strictly a horizontal distance, and the observation will impact the entire column (all levels) within the horizontal localization radius.

6.36 Available observation converter programs

The DART/observations/obs_converters directory contains a variety of converter programs to read various external formats and convert the observations into the format required by DART.

Each directory has at least one converter:

- AIRS: *AIRS and AMSU*
- AURA: See DART/observations/obs_converters/AURA
- Aviso+/CMEMS: *Aviso+/CMEMS Observations*
- Ameriflux: *PROGRAM level4_to_obs*
- CHAMP: *CHAMP*
- cice: *PROGRAM cice_to_obs*
- CNOFS: See DART/observations/obs_converters/CNOFS
- CONAGUA: *CONAGUA*
- COSMOS: *PROGRAM COSMOS_to_obs*
- DWL: *PROGRAM dwl_to_obs*
- GMI: *GMI Brightness Temperatures*
- GOES: *NOAA GOES-R Series Advanced Baseline Imager (ABI) Level 1b Radiances*
- GPSPW: *GPSPW*
- GRACE: See DART/observations/obs_converters/GRACE
- GSI2DART: *GSI2DART*
- GTSPP: *GTSPP Observations*
- MADIS: *MADIS Data Ingest System*
- MIDAS: *PROGRAM MIDAS_to_obs*
- MODIS: *PROGRAM MOD15A2_to_obs*
- MPD: See DART/observations/obs_converters/MPD
- NCEP: (prepbufr -> ascii) *PROGRAM prepbufr*
- NCEP: (ascii -> obs_seq) *PROGRAM create_real_obs*
- ROMS: *ROMS observations to DART observation sequences*
- SIF: *PROGRAM SIF_to_obs_netcdf*
- SSEC: *SSEC Data Center*
- SST: *PROGRAM sst_to_obs, oi_sst_to_obs*
- SSUSI: *SSUSI F16 EDR-DSK format to observation sequence converters*
- WOD: *WOD Observations*
- gnd_gps_vtec: *GND GPS VTEC*
- GPS: *GPS Observations*
- ok_mesonet: *Oklahoma Mesonet MDF Data*
- QuikSCAT: *QuikSCAT SeaWinds Data*

- `Radar`: *Radar observations*
- `snow`: *PROGRAM snow_to_obs*
- `Text`: *PROGRAM text_to_obs*
- `text_GITM`: See `DART/observations/obs_converters/text_GITM`
- `tpw`: *Total Precipitable Water Observations*
- `Tropical Cyclones`: *PROGRAM tc_to_obs*
- `Var (little-r)`: *PROGRAM littler_tf_dart*
- `Var (radar)`: *PROGRAM rad_3dvar_to_dart*

There are also a couple utilities of note:

- *Even Sphere* - a utility for generating evenly-spaced observation locations that can then be used in a perfect model experiment.
- *Obs Error* - modules that specify observation errors based on what is used by ECMWF and NCEP

In addition the following external program produces DART observation sequence files:

- Observation Processing And Wind Synthesis (OPAWS): OPAWS can process NCAR Dorade (sweep) and NCAR EOL Foray (netCDF) radar data. It analyzes (grids) data in either two-dimensions (on the conical surface of each sweep) or three-dimensions (Cartesian). Analyses are output in netCDF, Vis5d, and/or DART (Data Assimilation Research Testbed) formats.

For generating synthetic observations, see the documentation for the *program create_obs_sequence*. You can also generate observation files based on text input. See the documentation for the *PROGRAM text_to_obs*. Or for simulating a large complex observing system, you can use the DART library routines in a Fortran program to compute the observation information and have the DART routines write the output file.

To learn how to run a model with a set of observations that have only locations, types, and times, and have the forward operators compute the observation values, see the documentation for the *program perfect_model_obs*.

6.37 Manipulating obs_seq files with the obs_sequence_tool

Please see the *program obs_sequence_tool* document for detailed information and examples.

The `obs_sequence_tool` is the primary means to manipulate observation sequence files.

Observations sequence files are linked lists of observations organized by time. The observations may appear in any order in the file, but traversing the linked list will result in observations ordered by time.

The `obs_sequence_tool` can be used to combine observation sequences, convert from ASCII to binary or vice-versa, extract a subset of observations, etc.

When you are testing your DA application, you should use the `obs_sequence_tool` to extract one or a small number of observations from an existing observation sequence file for assimilation. Testing your application using a small number of observations will allow you to test and troubleshoot problems much faster than performing a full-scale assimilation.

6.38 The difference between observation TYPE and QUANTITY

Since DART is designed to assimilate data from any data source into any model, the assimilation algorithms need a way to define how observational data sources relate to model state variables.

DART does this by defining a single generic observation QUANTITY, such as zonal wind, and mapping many specific observation TYPEs, corresponding to source observations, to the single QUANTITY.

For example, QuikSCAT and radiosondes are both capable of measuring zonal wind. DART defines two observation TYPEs:

- `QKSWND_U_WIND_COMPONENT` for the QuikSCAT observations of zonal wind
- `RADIOSONDE_U_WIND_COMPONENT` for the radiosonde observations of zonal wind

and relates both of these TYPES to a single QUANTITY: QTY_U_WIND_COMPONENT.

Thus TYPE and QUANTITY have a many-to-one relationship. This distinction enables you to assimilate or evaluate observation platforms independently of one another with a single observation sequence file; reducing the possibility of error.

The forward observation operators are implemented based on observation QUANTITY. When requested, the model generates a QTY_U_WIND_COMPONENT, it doesn't need to know that it will be compared to an observation from QuikSCAT or one from a radiosonde.

Tip: It is usually scientifically very interesting to be able to compare the assimilations one TYPE of observation verus another. An observation sequence file can have many types of observations. DART has the capability to assimilate (or evaluate) any combination of observation types without getting bogged down in dataset management. The same observation sequence can be used for experiments that include or exclude certain observation types. This procedure can ensure that you are actually performing the experiment that you think you are performing.

6.39 Adding support for a new observation TYPE

If you would like to add support for a new observation TYPE, see *MODULE obs_def_mod* for detailed information.

6.40 Introduction to DART's support for RTTOV

This document serves as an orientation for DART's support for satellite radiance assimilation. At the current time, only ECMWF's RTTOV radiative transfer model is supported.

DART now includes the ability to use the RTTOV forward operators for satellite radiance assimilation. This is a new capability for DART, please submit issues with the DART Issues facility.

Note that DART support for RTTOV does not mean that all issues regarding satellite data assimilation with an ensemble system have been solved. Rather, the DART team hopes to provide the tools necessary for researchers to investigate the relevant issues with multiple models and data assimilation methodologies.

DART supports RTTOV version 12.3. Support for RTTOV v13 has been added to DART by Lukas Kugler. Use obs_def_rttov13_mod.f90 to compile DART with RTTOV v13.

Both RTTOV-direct for visible/infrared/microwave without scattering as well as RTTOV-scatt for microwave computations with full scattering are supported for v12.3. DART supports all features of RTTOV 12.3 as a pass-through from the models to RTTOV. This includes aerosols, trace gases, clouds, and atmospheric variables. It also includes directly specifying scattering properties.

However, a particular model may not have all of the variables necessary for these functions depending on the model and model setup. In some cases RTTOV default climatologies can be used, but at a minimum the following quantities must be supplied by the model_mod interpolate:

Quantity	Description
QTY_PRESSURE	atmospheric pressure in hPa at the model levels
QTY_TEMPERATURE	atmospheric temperature in K at the model levels
QTY_VAPOR_MIXING_RATIO	atmospheric humidity mixing ratio in kg/kg at the model levels
QTY_SURFACE_PRESSURE	the surface pressure in hPa
QTY_SURFACE_ELEVATION	the surface elevation in km
QTY_2M_TEMPERATURE	the atmospheric temperature in K at 2 m above the surface
QTY_SKIN_TEMPERATURE	the surface (skin) temperature in K
QTY_SURFACE_TYPE	0 = land, 1 = water, 2 = sea ice

If a DART model_mod cannot provide these required quantities, the RTTOV forward operator will fail and cannot be used. It may be possible to look up surface elevation or surface type through an look-up table or "atlas," although DART does not yet provide such functionality. 2M temperature in theory could be interpolated based on skin temperature and the lowest-level model temperature.

Beyond these fields, there are many other optional fields (such as clouds, trace gases, and aerosols) that can be specified. See *MODULE obs_def_rttov_mod* for a complete list of values.

6.40.1 Setting up DART+RTTOV

The RTTOV code and coefficients can be downloaded from this page:

https://www.nwpsaf.eu/site/software/rttov

Be aware that there are more coefficient files available once you download the RTTOV package. There is a `rtcoef_rttov12/rttov_coef_download.sh` script that assists in the process and you can select specific coefficient files or large batches. There is also a website https://nwp-saf.eumetsat.int/site/software/rttov/download/coefficients/rttov-v12-coefficient-download/

You will need to register for a free account before downloading the code.

You should read the RTTOV user guide carefully as DART primarily acts as a pass through. Refer to the setup instructions included with the RTTOV documentation.

It may also be useful to refer to:

https://github.com/NCAR/DART/wiki/Getting-Started-with-DART-RTTOV

Once you have successfully installed RTTOV, you should customize the mkmf.template.rttov.gfortran file to your own build system, possibly referring to the other mkmf.template examples for additional information if you are not using gfortran.

There are many namelist options available through input.nml that control the run-time behavior of the RTTOV model. These are documented in *MODULE obs_def_rttov_mod*.

To get RTTOV to work with your model, you will need to follow these steps:

1. Install RTTOV as above
2. Customize your mkmf.template to include the RTTOV libraries and include directories
3. Go into the models//work directory for your model of choice
4. Add your observation types (which are listed in obs_def_rttov_mod.f90) to the input.nml namelist (assimilate_ / evaluate_these_obs_types)

5. Include observations/forward_operators/obs_def_rttov_mod.f90 in the input_files section under &preprocess
6. In your model of choice, run ./quickbuild.sh and ensure the RTTOV libraries are built
7. For OSSE runs with perfect_model_obs:
    - Create an observation sequence file using ./create_obs_sequence and ./create_fixed_network_seq as detailed in the DART Getting_Started documentation
    - Run perfect_model_obs
    - Setup your ensemble as appropriate
    - Run filter and analyze the results in the usual way
8. For OSE runs:
    - Run the observation converter for your desired observations
    - Setup your ensemble as appropriate
    - Run filter and analyze the results in the usual way

Note that currently obervation converters are only provided for AIRS, AMSU/A, GOES, and GMI. These converters can be found in the observations/obs_converters directories. The L1 converters are the appropriate converters for the radiance or brightness temperatures (rather than retrievals). If you need real L1 data for another satellite (as opposed to running an OSSE with perfect_model_obs where you can generate your own data), you may be able to use one of these converters to get you started. We welcome your contributions back to the DART public repository. Please issue a pull request to https://github.com/NCAR/DART.

Note that some of the observation converters may require the HDF-EOS libraries. See the BUILDME script in each directory for help in building these observation converters.

6.40.2 Current list of known issues

DART support for satellite radiances cannot be considered 100% complete. The following details the known issues that are being considered with DART's support for satellite radiances.

- DART does not yet provide satellite bias correction capabilities. This will be released in the near future.
- Cross-channel error correlations are not yet supported. A principal component approach has been discussed. For now, the best bet is to use a subset of channels that are nearly independent of one another.
- Vertical localization is an issue for satellite radiances. The main choices are to turn off vertical localization, use the maximum peak of the weighting function or the cloud-top may be appropriate, or explore other options. We consider this an open research problem.

6.41 DART Observations

6.41.1 Overview

Real-world observations of earth-system data come from a variety of sources, including radiosondes, satellites, ships, aircraft, weather stations, etc. The files in this *observations* directory can be used to convert data from a variety of native formats into a common DART *observation sequence* format.

Synthetic observations are those not based on an actual instrument reading of a system, but instead are fabricated to have a known value, or have values computed by running a model, possibly with a fixed amount of simulated noise added. These observations can be used for testing, determining the sensitivity of the model to assimilation, and for

designing new observation systems. The DART system includes several ways to create synthetic observations. See the *programs* section below for more details.

The DART framework enforces a clean separation between observations and the models they are assimilated into. The same observations can be used in any model which understands how to generate a value for the requested type of observation from its state space values.

In many cases a single, self-contained program can convert directly from the observation location, time, value, and error into the DART format. In other cases, especially those linking with a complicated external library (e.g. BUFR), there is a two-step process with two programs and an ASCII intermediate file. We are currently leaning towards single-step conversions but either approach can be used for new programs.

Frequently the original datasets are in a standard scientific format like netCDF, HDF, or BUFR, and library routines for those formats can be used to read in the original observation data.

The DART software distribution includes Fortran subroutines and functions to help create a sequence of observations in memory, and then a call to the DART observation sequence write routine will create an entire *obs_seq* file in the correct format.

The DART system comes with several types of location modules for computing distances appropriately. Two of the ones most commonly used are for data in a 1D system and for data in a 3D spherical coordinate system. Most of the programs here assume the *location/threed_sphere/location_mod.f90* 3D sphere location module is being used.

There are currently some additional observation sources and types which we are in the process of collecting information and conversion programs for and which will eventually be added to this directory. In the meantime, if you have converters for data or interest in something that is not in the repository, please email the DART group.

6.41.2 Data Sources and Formats

See the various subdirectories here, which generally include information on where the example data was obtained and in what format it is distributed. Most data is available for download off the web. The Data Support Section (DSS) at NCAR has large data repositories, the MADIS data center distributes observations in NetCDF format, GTS real-time weather data is available from various sources. For new converters, if you can find what format the data is distributed in you may be able to adapt one of the existing converters here for your own use. Formats read by the existing converters include NetCDF, HDF, little-r, text, Prepbufr, amongst others.

See the *programs* section below for a list of the current converter programs. It might save you from reinventing the wheel.

If you have looked and none of the existing converters are right for your data, here are some suggestions for where to start creating a new converter. Create a new subdirectory in the *observations* directory. Copy with the recursive option (*cp -r*) one of the existing converters and adapt to your needs. Our suggestions for which converter to start from depends on the format of your input observations to be converted. If your input data format is:

Start with the MADIS converters, and in particular try the convert_madis_profiler.f90 file because it is the most straightforward. Another good option is SST/oi_sst_to_obs.f90

netCDF	Start with the MADIS converters, and in particular try the convert_madis_profiler.f90 file because it is the most straightforward. Another good option is SST/oi_sst_to_obs.f90
Comma separated text	Start with the Ameriflux converter.
Generic text	Start with the text converter.
HDF-EOS	Start with the AIRS converter.
BUFR or prep-BUFR	Start with the NCEP converter.
Dense data, like Satellite swaths	Start with the tpw converter, which includes code that averages the raw data in space and time.
Ray-path integrated data	Start with the GPS converter, which includes code that traces a path and integrates values along the ray.
World Ocean Database packed ASCII	Start with the WOD converter.

6.41.3 Decisions You May Need to Make

Time

Time enters into the assimilation system in 3 places: the time of the state vector data (the current model time when this data was produced), the time of each observation, and the assimilation window length. The window length is set by the model-dependent routine `shortest_time_between_assimilations()`. The internal timestepping of the model is unrelated to any of these times and is outside the scope of the assimilation system.

The basic time type in DART is a pair of integers; one for the day number and one for the number of seconds. Generally the low order models, which aren't direct geophysical models, use time directly as a sequence of days starting at 0 and incrementing in any appropriate number of seconds or days. The observations assimilated into these systems do not need to use a calendar.

Observations of a real-world system usually are distributed with a year/month/day, hour/min/seconds timestamp. There are routines in DART to convert back and forth between the (day-number/seconds) format and a variety of (year/month/day) calendars. See the time manager documentation for more details on how DART stores time information and the types of available calendars. Some climate models which do long runs (100s or 1000s of years) use a modified calendar for simplicity in computation, e.g. months which always have 30 days, or no leap years. When trying to assimilate real observations into these models there may be calendar issues to solve.

The smallest resolvable unit of time in DART is a second. To model a system which operates on sub-second time scales the time can be scaled up by some factor. As long as the observation time, the state data time, and the minimum model advance time are expressed in the same scaled time units, there is no problem.

Error

Observations must specify an associated expected error. Each individual observation stores its own error value, so it can be a constant value for all observations of that type or it can vary by location, by height, by magnitude of the observed value, etc. This value is the expected instrument error plus the representativeness error of the model. The model error includes deficiencies in the equations representing the processes of the system as well as errors introduced by representing a continuous system as a series of discrete points. While the instrument error and the representativeness error could be specified separately, they each have the same impact on the assimilation and can be difficult to determine with any real accuracy. For simplicity, in DART (and most current assimilation software) they are combined and specified as a single value.

The instrument error is generally supplied by the instrument maker. Sadly, it is frequently surprisingly difficult to find these values. For the representativeness error, a set of artificial observations could be generated with the perfect_model_obs program and an assimilation experiment could be run to generate an estimate of the error in the model. In practice however most people make an educated guess on the values of the error and then start with a larger than expected value and decrease it based on the results of running some test assimilations. For these tests the namelist for the outlier threshold should be disabled by setting it to -1 (the default value is 3). This value controls whether the observation is rejected because the observed value is too far from the ensemble mean.

If the diagnostics show that the difference between the mean of the forward operators and the observed value is consistently smaller than the specified observation error, then the error is probably too large. A too-large error reduces the impact of an observation on the state. If the specified observation error is too small it is likely the observation will be rejected when the outlier threshold is enabled, and the observation will not be assimilated. It is important to look at the output observation sequence files after an assimilation to see how many observations were assimilated or rejected, and also at the RMSE (root mean squared error) versus the total spread. DART includes Matlab diagnostic routines to create these types of plots. The observation RMSE and total spread should be roughly commensurate. The total spread includes contributions from both the ensemble variance and the observational error variance, so it can be adjusted by changing the error values on the incoming observations. There are other ways to adjust the ensemble spread, including inflation, so the observation error is not the only factor to consider.

One last recommendation: if possible, the Prior forward operator values should be compared against the observations after several assimilation cycles. If you plot results using the Posterior values it is always possible for the assimilation to overfit the observations and look good on the diagnostic plots. But the actual test is to then advance the model and look at how the forecast of the state compares to the observations.

Types

All observations have to have a specific 'type'. There are namelist controls to turn on and off the assimilation of observations at run-time by type, or to only evaluate the forward operator for an observation but have no impact on the state. Several of the diagnostics also group observations by type to give aggregate statistics after an assimilation. Generally types are based on both the observing platform or instrument as well as the kind of observation, e.g. RADIOSONDE_TEMPERATURE, ARGO_SALINITY, etc. Each type is associated with a single underlying generic 'kind', which controls what forward operator code is called inside the model, e.g. QTY_TEMPERATURE, QTY_DENSITY, etc.

See here for more details on how to use and add new DART types. The DART obs_kind_mod.f90 defines a list of already defined observation kinds, and users can either use existing observation types in 'obs_def_xxx_mod.f90' files, or define their own.

Locations

The two most common choices for specifying the location of an observation are the threed_sphere and the oned locations. For observations of a real-world system, the 3D Sphere is generally the best choice. For low-order, 1D models, the 1D locations are the most commonly used. The observation locations need to match the type of locations used in the model.

6.41.4 Converting a series of observations

If you are running a series of assimilation steps you may need a separate observation sequence (obs_seq) file per step. The suggested process is to create the first few files by hand to check the resulting obs_seq files and then write scripts (python, shell) to automate the creation of the remainder of the files. The following are some of the considerations to take into account when creating scripts for a series of obs_seq files.

Looping in Time

Often observations are distributed in files that contain observations from a particular time period, e.g. a file per day or per week. The output obs_seq files need to include observations from the same time period as the assimilation window; how often the assimilation is stopped and the model is advanced in time. The conversion process can either convert all the observations from an input file into a single output file and in a subsequent step break the file into the required time ranges, or the conversion process can extract and convert only the observations required for a single output file and loop multiple times over the same input file.

Generally earth system models use calendar dates, including months, days, years, hours, minutes and seconds. The `advance_time` program is very useful in adding or subtracting time periods from calendar dates taking into account changing months and years, accounting for leap days, etc.

Observation conversion programs usually take one of two strategies for their input and output filenames.

- Have fixed input and output filenames for the converter. Have the script make symbolic links from the actual filenames to the fixed names for the files for each conversion run.
- Have a Fortran namelist variable that sets the input and output filenames for the converter. Have the script generate or edit the namelist file (e.g. with the *sed* stream editor) to set the actual filenames for each conversion run.

Generally it is a good idea to encode the date information in the output filename so each file is guarenteed to be unique. This can also make it simpler at filter runtime to generate the required input observation sequence filenames using a program like `advance_time`.

Multiple Observation Files

It is common that an assimilation will want to use observations from different sources. Generally it is easier to convert observations from each source separately and then merge them together with the `obs_sequence_tool`.

Creating filenames and directory names which follow a pattern that can be generated with the `advance_time` program makes this easier to do.

The `obs_sequence_tool` can read the input filenames from a separate ascii file. This makes generating the filenames easy from a script; it can simply concatinate the input filenames echo'd to an ascii file and then run the obs_sequence_tool. The output file can either be set by using `sed` on the namelist, or a fixed output filename can be used and then the file renamed after the tool has run.

Conversion Run Time for Large File Counts

If 100s of files need to be generated and a supercomputer or other multiple-CPU resource is available, batch files which convert multiple files at the same time can be a large time savings. Care must be taken that each conversion has its own settings and unique filenames. Often a separate working directory from other conversions running at the same time simplifies the scripting needed.

Verification

Observations taken from real-world sources can have missing values, illegal values, missing files, duplicated data, etc. The list is as long as your imagination. It can be very useful to write or adapt programs like `obs_info` to print out the first and last obs times in a file, the count of each obs type, etc. Especially for observations which are close to the start/end of a month or year, it is easy to find truncated data files.

If converting a large number of files it is also common for computer system failures to occur at random times. File systems fill up, batch jobs exit early, power glitches stop programs before they finish. Look for anomolous observation counts, unexpected first and last times of obs in a file, missing files, files with many fewer bytes than others, and anything else you can think of.

Output Formats

There are options to write output obs_seq files in binary, which are roughly half the size of ascii files. However it greatly increases the effort to examine the contents of a file for problems. Generally we have used the ascii format. It is portable between systems of different "endians" (order of bytes in a multi-byte number) and can be browsed much more easily.

6.42 Converter programs

The *DART/observations/obs_converters* directory contains a variety of converter programs to read various external formats and convert the observations into the format required by DART.

The current list of converters (some directories contain multiple converters) include:

- `AIRS`: *AIRS and AMSU*
- `AURA`: See `./AURA`
- `Aviso+/CMEMS`: *Aviso+/CMEMS Observations*
- `Ameriflux`: *PROGRAM level4_to_obs*
- `CHAMP`: *CHAMP*
- `cice`: *PROGRAM cice_to_obs*
- `CNOFS`: See `./CNOFS`
- `CONAGUA`: *CONAGUA*
- `COSMOS`: *PROGRAM COSMOS_to_obs*
- `DWL`: *PROGRAM dwl_to_obs*
- `GMI`: *GMI Brightness Temperatures*
- `GOES`: *NOAA GOES-R Series Advanced Baseline Imager (ABI) Level 1b Radiances*
- `GPSPW`: *GPSPW*

- GRACE: See ./GRACE
- GSI2DART: *GSI2DART*
- GTSPP: *GTSPP Observations*
- MADIS: *MADIS Data Ingest System*
- MIDAS: *PROGRAM MIDAS_to_obs*
- MODIS: *PROGRAM MOD15A2_to_obs*
- MPD: See ./MPD
- NASA_Earthdata*PROGRAMS LPRM_L3_to_obs.f90 AMSR_E_L2_to_obs.f90*
- NCEP: (prepbufr-> ascii) *PROGRAM prepbufr*
- NCEP: (ascii-> obs_seq) *PROGRAM create_real_obs*
- NSIDC*PROGRAM SMAP_L2_to_obs*
- ROMS: *ROMS observations to DART observation sequences*
- SIF: *PROGRAM SIF_to_obs_netcdf*
- SSEC: *SSEC Data Center*
- SST: *PROGRAM sst_to_obs, oi_sst_to_obs*
- ocean color: *VIIRS/AQUA Satellite Ocean Color*
- SSUSI: *SSUSI F16 EDR-DSK format to observation sequence converters*
- WOD: *WOD Observations*
- gnd_gps_vtec: *GND GPS VTEC*
- GPS: *GPS Observations*
- ok_mesonet: *Oklahoma Mesonet MDF Data*
- QuikSCAT: *QuikSCAT SeaWinds Data*
- Radar: *Radar observations*
- snow: *PROGRAM snow_to_obs*
- Text: *PROGRAM text_to_obs*
- text_GITM: See ./text_GITM
- tpw: *Total Precipitable Water Observations*
- Tropical Cyclones: *PROGRAM tc_to_obs*
- 3DVAR/4DVAR: *3DVAR/4DVAR Observation Converters*
- Var (little-r): *PROGRAM littler_tf_dart*
- Var (radar): *PROGRAM rad_3dvar_to_dart*

In addition the following external program produces DART observation sequence files:

- Observation Processing And Wind Synthesis (OPAWS): OPAWS can process NCAR Dorade (sweep) and NCAR EOL Foray (netcdf) radar data. It analyzes (grids) data in either two-dimensions (on the conical surface of each sweep) or three-dimensions (Cartesian). Analyses are output in netcdf, Vis5d, and/or DART (Data Assimilation Research Testbed) formats.

For generating synthetic observations, see the create_obs_sequence program documentation. You can also generate observation files based on text input. See the text_to_obs program documentation and even_sphere. Or for simulating a large complex observing system, you can use the DART library routines in a Fortran program to compute the observation information and have the DART routines write the output file.

There are a couple utilities of note:

- even_sphere - a utility for generating a text file of evenly-spaced observation locations that can then be used in a perfect model experiment.
- obs_error - modules that specify observation errors based on what is used by ECMWF and NCEP

See the perfect_model program documentation on how to run a model with a set of observations that have only locations, types, and times, and have the forward operators compute the observation values.

Contact the DART development group if you have observations in a different format that you want to convert. We can give you advice and pointers on how to approach writing the code.

6.43 AIRS and AMSU

Caution: Before you begin: Installing the libraries needed to read these files can be fairly troublesome. The NASA Earthdata Data Access Services website is the download site for the necessary libraries. An example build script (*AIRS/Build_HDF-EOS.sh*) is intended to provide some guidance.

This directory covers two observation converters:

- *Program convert_airs_L2* for temperature and moisture retrievals.
- *Program convert_amsu_L1* for radiances.

Both converters are in the AIRS directory because of the complicated history of the data used to create the AIRS L2 product (which includes some AMSU observations). Since both datasets are HDF - it was believed that some of the routines could be used by both converters. Alas, that has not proven to be the case.

6.43.1 Atmospheric Infrared Sounder (AIRS) Level 2 observations

The AIRS instrument is an Atmospheric Infrared Sounder flying on the Aqua spacecraft. Aqua is one of a group of satellites flying close together in a polar orbit, collectively known as the "A-train". The programs in this directory help to extract the data from the distribution files and put them into DART observation sequence (obs_seq) file format.

AIRS data includes atmospheric temperature in the troposphere, derived moisture profiles, land and ocean surface temperatures, surface emissivity, cloud fraction, cloud top height, and ozone burden in the atmosphere.

6.43.2 Advanced Microwave Sounding Unit (AMSU-A) L1B Brightness Temperatures

The *DART/observations/obs_converters/AIRS* directory contains the code to convert the L1B AMSU-A Brightness Temperatures in HDF-EOS2 format to the DART observation sequence file format.

There is a little bit of confusing history to be aware of for AMSU/A:

https://en.wikipedia.org/wiki/Advanced_microwave_sounding_unit#History

AMSU/A was flown on NOAA 15-17. It is also on the Aqua satellite (that also houses AIRS) as well as the European MetOp. It has been replaced by ATMS on NOAA-20.

6.43.3 Dependencies

Both *convert_airs_L2* and *convert_amsu_L1* require the HDF-EOS libraries. *convert_amsu_L1* also requires HDF5 support because of the RTTOV libraries. HDF5 is incompatible with HDF-EOS, so a two-step conversion is necessary for the AMSU observations. The data must be converted from HDF to netCDF (which can be done without HDF5) and then the netCDF files can be converted to DART radiance observation format - which requires `obs_def_rttov_mod.f90`, which depends on HDF5. To simplify things, An example build script (*DART/observations/obs_converters/AIRS/Build_HDF-EOS.sh*) is supplied and may provide some guidance on downloading and building the libraries required by NASA.

The NASA Earthdata Data Access Services website is the download site, at press time, the following packages were required to build HDF-EOS Release v2.20:

- hdf-4.2.13.tar.gz
- HDF-EOS2.20v1.00.tar.Z
- HDF-EOS2.20v1.00_TestDriver.tar.Z
- HDF-EOS_REF.pdf
- HDF-EOS_UG.pdf
- jpegsrc.v9b.tar.gz
- zlib-1.2.11.tar.gz

Similarly for HDF-EOS5 Release v5.1.16:

- HDF-EOS5.1.16.tar.Z
- HDF-EOS5.1.16_TESTDRIVERS.tar.Z
- HDF-EOS5_REF.pdf
- HDF-EOS5_UG.pdf
- hdf5-1.8.19.tar.gz
- szip-2.1.1.tar.gz

BUILD_HDF-EOS.sh may help you build these libraries. You *will* have to modify it for your system, and you *probably will* have to iterate on that process. The script takes the stance that if you have to build HDF4, HDF-EOS, HDF5 … you might as well build HDF-EOS5 too. The HDF-EOS5 is entirely optional. The HDF5 will be needed by RTTOV.

6.43.4 Converting from HDF4 to netCDF

There are multiple ways to convert from HDF4 to netCDF. The HDF-EOS Tools and Information Center provides binaries for several common platforms as well as source code should you need to build your own.

HDF4 CF CONVERSION TOOLKIT

The HDF-EOS Tools and Information Center provides the HDF4 CF CONVERSION TOOLKIT

> The HDF4 CF (H4CF) Conversion Toolkit can access various NASA HDF4 external and HDF-EOS2 external files by following the CF conventions external. The toolkit includes a conversion library for application developers and a conversion utility for NetCDF users. We have translated the information obtained from various NASA HDF-EOS2 and HDF4 files and the corresponding product documents into the information required by CF into the conversion library. We also have implemented an HDF4-to-NetCDF (either NetCDF-3 or NetCDF-4 classic) conversion tool by using this conversion library. In this web page, we will first introduce how to build the conversion library and the tool from the source. Then, we will provide basic

usage of the tool and the conversion library APIs. The information for the supported NASA HDF-EOS2 and HDF4 products and visualization screenshots of some converted NetCDF files will also be presented.

If you download a binary, it's a good habit to verify the checksum. The download page has a link to a .pdf that has the known checksums. Here's how to generate the checksum. Be aware that when I downloaded the file (via Chrome or 'wget') on an OSX system, the checksum did not match. When I downloaded the file on a linux system, the checksum *did* match.

If you download the source, the tar file comes with a `README` and an `INSTALL`. Please become familiar with them. DART also has a build script: `AIRS/shell_scripts/Build_HDF_to_netCDF.csh` that you can customize after you read the `INSTALL` document.

6.44 Program `convert_airs_L2`

Caution: Before you begin: Installing the libraries needed to read these files can be fairly troublesome. The NASA Earthdata Data Access Services website is the download site for the necessary libraries. An example build script (*AIRS/Build_HDF-EOS.sh*) is intended to provide some guidance.

6.44.1 Overview

The Atmospheric Infrared Sounder (AIRS) is a facility instrument aboard the second Earth Observing System (EOS) polar-orbiting platform, EOS Aqua. In combination with the Advanced Microwave Sounding Unit (AMSU) and the Humidity Sounder for Brazil (HSB), AIRS constitutes an innovative atmospheric sounding group of visible, infrared, and microwave sensors. AIRS data will be generated continuously. Global coverage will be obtained twice daily (day and night) on a 1:30pm sun synchronous orbit from a 705-km altitude.

The AIRS Standard Retrieval Product consists of retrieved estimates of cloud and surface properties, plus profiles of retrieved temperature, water vapor, ozone, carbon monoxide and methane. Estimates of the errors associated with these quantities will also be part of the Standard Product. The temperature profile vertical resolution is 28 levels total between 1100 mb and 0.1 mb, while moisture profile is reported at 14 atmospheric layers between 1100 mb and 50 mb. The horizontal resolution is 50 km. An AIRS granule has been set as 6 minutes of data, 30 footprints cross track by 45 lines along track. The Shortname for this product is AIRX2RET. (AIRS2RET is the same product but without the AMSU data.)

Atmospheric Infrared Sounder (AIRS) Level 2 observations

Several types of AIRS data, with varying levels of processing, are available. The following descriptions are taken from the V5_Data_Release_UG document:

> The L1B data product includes geolocated, calibrated observed microwave, infrared and visible/near infrared radiances, as well as Quality Assessment (QA) data. The radiances are well calibrated; however, not all QA data have been validated. Each product granule contains 6 minutes of data. Thus there are 240 granules of each L1B product produced every day.
>
> The L2 data product includes geolocated, calibrated cloud-cleared radiances and 2-dimensional and 3-dimensional retrieved physical quantities (e.g., surface properties and temperature, moisture, ozone, carbon monoxide and methane profiles throughout the atmosphere). Each product granule contains 6 minutes of data. Thus there are 240 granules of each L2 product produced every day.
>
> The L3 data are created from the L2 data product by binning them in 1°x1° grids. There are three products: daily, 8-day and monthly. Each product provides separate ascending (daytime) and descending (nighttime) binned data sets.

The converter in this directory processes level 2 (L2) data files, using data set AIRS_DP and data product AIRX2RET or AIRS2RET without HSB (the instrument measuring humidity which failed).

Getting the data currently means putting in a start/stop time at this web page. The keyword is AIRX2RET and put in the time range of interest and optionally a geographic region. Each file contains 6 minutes of data, is about 2.3 Megabytes, and globally there are 240 files/day (about 550 Megabytes/day). There are additional options for getting only particular variables of interest, but the current reader expects whole files to be present. Depending on your connection to the internet, there are various options for downloading. We have chosen to download a wget script which is created by the web page after adding the selected files to a 'cart' and 'checking out'. The script has a series of wget commands which downloads each file, one at a time, which is run on the machine where you want the data.

6.44.2 convert_airs_L2.f90

The convert_airs_L2 converter is for **temperature and moisture retrievals** from the L2 data. The temperature observations are at the corresponding vertical pressure levels. However, the moisture obs are the mean for the layer, so the location in the vertical is the midpoint, in log space, of the current layer and the layer above it. There is an alternative computation for the moisture across the layer which may be more accurate, but requires a forward operator subroutine to be written and for the observation to contain metadata. The observation could be defined with a layer top, in pressure, and a number of points to use for the integration across the layer. Then the forward operator would query the model at each of the N points in the vertical for a given horizontal location, and compute the mean moisture value. This code has not been implemented yet, and would require a different QTY_xxx to distinguish it from the simple location/value moisture obs. See the GPS non-local operator code for an example of how this would need to be implemented.

The temperature observations are located on standard levels; there is a single array of heights in each file and all temperature data is located on one of these levels. The moisture observations, however, are an integrated quantity for the space between the levels; in their terminology the fixed heights are 'levels' and the space between them are 'layers'. The current converter locates the moisture obs at the midpoint, in log space, between the levels.

The hdf files need to be downloaded from the data server, in any manner you choose. The converter program reads each hdf granule and outputs a DART obs_seq file containing up to 56700 observations. Only those with a quality control of 0 (Best) are kept. The resulting obs_seq files can be merged with the *program obs_sequence_tool* into larger time periods.

It is possible to restrict the output observation sequence to contain data from a region of interest throught the use of the namelist parameters. If you need a region that spans the Prime Meridian lon1 can be a larger number than lon2, for example, a region from 300 E to 40 E and 60 S to 30 S (some of the South Atlantic), would be *lon1 = 300, lon2 = 40, lat1 = -60, lat2 = -30*.

The DART/observations/obs_converters/AIRS/shell_scripts directory includes scripts (download_L2.sh and oneday_down.sh) that make use of the fact that the AIRS data is also archived on the NCAR HPSS (tape library) in daily tar files. oneday_down.sh has options to download a day of granule files, convert them, merge them into daily files, and remove the original data files and repeat the process for any specified time period.

6.44.3 Namelist

This namelist is read in a file called input.nml. We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. The default values are shown below. More realistic values are provided in AIRS/work/input.nml

```
&convert_airs_L2_nml
   l2_files          = ''
   l2_file_list      = ''
```

(continues on next page)

(continued from previous page)

```
   outputfile         = ''
   lon1               =   0.0
   lon2               = 360.0
   lat1               = -90.0
   lat2               =  90.0
   min_MMR_threshold  = 1.0e-30
   top_pressure_level = 0.0001
   cross_track_thin   = 0
   along_track_thin   = 0
   use_NCEP_errs      = .false.
   version            = 6
/
```

Contents	Type	Description
l2_files	character(len=256), dimension(512)	A list of one or more names of the HDF file(s) to read, NOT including the directory. If multiple files are listed, each will be read and the results will be placed in a separate file with an output filename constructed based on the input filename.
l2_file_list	character(len=256)	The name of an ascii text file which contains one filename per line, NOT including the directory. Each file will be read and the observations converted into an output file where the output filename is based on the input filename. Only one of 'l2_files' and 'l2_file_list' can be specified. The other must be ' ' (empty).
output_file	character(len=256)	The name of the output observation sequence file.
lon1	real(r8)	the West-most longitude of interest in degrees. [0.0, 360]
lon2	real(r8)	the East-most longitude of interest in degrees. [0.0, 360]
lat1	real(r8)	the South-most latitude of interest in degrees. [-90.0,90.0]
lat2	real(r8)	the North-most latitude of interest in degrees. [-90.0,90.0]
min_MMR_threshold	real(r8)	The data files contains 'Retrieved Water Vapor Mass Mixing Ratio'. This is the minimum threshold, in gm/kg, that will be converted into a specific humidity observation.
top_pressure_level	real(r8)	The highest pressure level of interest (in mb).
cross_track_thin	integer	provides ability to thin the data by keeping every Nth data value in the cross-track scan. [0,30] e.g. 3 == keep every third value. 0 is no thinning.
along_track_thin	integer	provides ability to thin the data by keeping every Nth data value in the along-track scan. [0,45] e.g. 4 == keep only every 4th row. 0 is no thinning.
use_NCEP_errs	logical	if .true. use the maximum observation error from either the granule or the NCEP equivalent (from `obs_error_mod.f90`)
version	integer	The AIRS file format version.

Dependencies

See the *Dependencies Section* of the AIRS/README.

Known Bugs

Earlier versions of this converter mistakenly put the moisture obs at level heights, in the same location as the temperature observations. The moisture observations are in fact an integrated value across the distance between two levels. This means the location was shifted 1/2 level in the vertical from the center of the layer. The fixed converter outputs the location at the center, in log space, of each layer.

Future Plans

If a more accurate moisture observation was needed, the observation value could be computed by actually integrating multiple values between the levels. At this point it doesn't seem necessary.

6.45 Program `convert_amsu_L1`

Caution: Before you begin: Installing the libraries needed to read these files can be fairly troublesome. The NASA Earthdata Data Access Services website is the download site for the necessary libraries. An example build script (*AIRS/Build_HDF-EOS.sh*) is intended to provide some guidance.

6.45.1 Overview

There is a little bit of confusing history to be aware of for AMSU/A:

https://en.wikipedia.org/wiki/Advanced_microwave_sounding_unit#History

AMSU/A was flown on NOAA 15-17. It is also on the Aqua satellite (that also houses AIRS) as well as the European MetOp. It has been replaced by ATMS on NOAA-20.

The datset of interest is: "AIRS/Aqua L1B AMSU (A1/A2) geolocated and calibrated brightness temperatures V005 (AIRABRAD) at GES DISC" The *short name* for this dataset is 'AIRABRAD'

The introductory paragraph for the dataset is:

> Version 5 is the current version of the data set.tmospheric Infrared Sounder (AIRS) is a grating spectrometer (R = 1200) aboard the second Earth Observing System (EOS) polar-orbiting platform, EOS Aqua. In combination with the Advanced Microwave Sounding Unit (AMSU) and the Humidity Sounder for Brazil (HSB), AIRS constitutes an innovative atmospheric sounding group of visible, infrared, and microwave sensors. The AMSU-A instrument is co-aligned with AIRS so that successive blocks of 3 x 3 AIRS footprints are contained within one AMSU-A footprint. AMSU-A is primarily a temperature sounder that provides atmospheric information in the presence of clouds, which can be used to correct the AIRS infrared measurements for the effects of clouds. This is possible because non-precipitating clouds are for the most part transparent to microwave radiation, in contrast to visible and infrared radiation which are strongly scattered and absorbed by clouds. AMSU-A1 has 13 channels from 50 - 90 GHz and AMSU-A2 has 2 channels from 23 - 32 GHz. The AIRABRAD_005 products are stored in files (often referred to as "granules") that contain 6 minutes of data, 30 footprints across track by 45 lines along track.

The citation information for this dataset is:

Title: AIRS/Aqua L1B AMSU (A1/A2) geolocated and calibrated brightness temperatures V005 Version: 005 Creator: AIRS project Publisher: Goddard Earth Sciences Data and Information Services Center (GES DISC) Release Date: 2007-07-26T00:00:00.000Z Linkage: https://disc.gsfc.nasa.gov/datacollection/AIRABRAD_005.html

NASA provides a README.AIRABRAD.pdf through the Goddard Earth Sciences Data and Information Services Center.

6.45.2 convert_amsua_L1.f90

`convert_amsua_L1` converts the L1B AMSU-A Brightness Temperatures in netCDF format to the DART observation sequence file format. The native HDF-EOS2 format files must be converted to netCDF. The conversion from HDF-EOS2 to netCDF is easily performed by the h4tonccf_nc4 converter.

As you can imagine, you need to download each satellite's data in a different way. Also, just for your information, AMSU/B has been replaced on newer satellites by MHS and HSB, but especially MHS is almost identical.

Namelist

DARTs design structure has the support for radiance observations (like brightness temperatures) provided by the *MODULE obs_def_rttov_mod* which depends on HDF5 libraries. Consequently, the `obs_def_rttov_mod_nml` namelist must appear in the `input.nml`. However, only two options are used when converting the observations: *use_zeeman* and *rttov_sensor_db_file*.

Be aware that if the RTTOV namelist option `use_zeeman = .true.` certain metadata must be available in the observation. This is not fully implemented in the AMSU-A observation converter. For more information, please see GitHub Issue 99 "AIRS AMSUA observation converter … Zeeman coefficients and channels"

Namelists are read in a file called `input.nml`. We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. The default values are shown below. More realistic values are provided in `AIRS/work/input.nml`

```
&convert_amsua_L1_nml
   l1_files           = ''
   l1_file_list       = ''
   outputfile         = ''
   append_output      = .false.
   channel_list       = 'null'
   along_track_thin   = 0
   cross_track_thin   = 0
   lon1               =   0.0
   lon2               = 360.0
   lat1               = -90.0
   lat2               =  90.0
   verbose            = 0
/
```

Contents	Type	Description
l1_files	character(len=256), dimension(512)	A list of one or more names of the netCDF file(s) to read.
l1_file_list	character(len=256)	The name of an ascii text file which contains one filename per line. Each file will be read and the observations converted into a single output file. Only one of 'l1_files' and 'l1_file_list' can be specified. The other must be ' ' (empty).
output_file	character(len=256)	The name of the output observation sequence file.
append_output	logical	If the output observation sequence file exists it is possible to add to it. The observations are added consistent with the paradigm that the observation linked list will be traversed in temporally-ascending fashion, no matter the physical location of the observation in the file. `.true.` adds the new observations to the existing file, `.false.` will cause an existing output file to be overwritten.
channel_list	character(len=8), dimension(15)	The AMSU channels desired. See the table below for valid input.
along_track_thin	integer	provides ability to thin the data by keeping every Nth data value in the along-track scan. [0,45] e.g. 4 == keep only every 4th row. 0 is no thinning.
cross_track_thin	integer	provides ability to thin the data by keeping every Nth data value in the cross-track scan. [0,30] e.g. 3 == keep every third value. 0 is no thinning.
lon1	real(r8)	the West-most longitude of interest in degrees. [0.0, 360]
lon2	real(r8)	the East-most longitude of interest in degrees. [0.0, 360]
lat1	real(r8)	the South-most latitude of interest in degrees. [-90.0,90.0]
lat2	real(r8)	the North-most latitude of interest in degrees. [-90.0,90.0]
verbose	integer	Controls the amount of run-time output. 0 == bare minimum. 3 is very verbose. Only use 3 if converting one or two files for testing.

Channel Specification

> "AMSU-A primarily provides temperature soundings. It is a 15-channel microwave temperature sounder implemented as two independently operated modules. Module 1 (AMSU-A1) has 12 channels in the 50-58 GHz oxygen absorption band which provide the primary temperature sounding capabilities and 1 channel at 89 GHz which provides surface and moisture information. Module 2 (AMSU-A2) has 2 channels: one at 23.8 GHz and one at 31.4 GHz which provide surface and moisture information (total precipitable water and cloud liquid water)."

To facilitate the selection of channels, either the 'Integer' or 'String' values may be used to specify `channel_list`. The 'Documentation' and 'netCDF' values are provided for reference only. The 'Documentation' values are from the README.AIRABRAD.pdf document.

Integer	String	Documentation Frequency	netCDF *center_freq*
Module 2 - surface and moisture information			
1	'A2-1'	23.8	23.8
2	'A2-2'	31.4	31.4
Module 1 - primary temperature sounding capability			
3	'A1-1'	50.3	50.3
4	'A1-2'	52.8	52.8
5	'A1-3'	53.596	53.596
6	'A1-4'	54.4	54.4
7	'A1-5'	54.94	54.94
8	'A1-6'	55.5	55.5
9	'A1-7'	57.29034	57.29034
10	'A1-8'		57.29034
11	'A1-9'		57.29034
12	'A1-10'		57.29034
13	'A1-11'		57.29034
14	'A1-12'		57.29034
15	'A1-13'	89	89

Known Bugs

None.

Future Plans

None.

Instructions to download the AIRABRAD dataset

1. Go to https://earthdata.nasa.gov
2. Log in (or create an account if necessary)
3. Search for AIRABRAD
4. Scroll down past datasets to "Matching results."

- Follow the link to "AIRS/Aqua L1B AMSU (A1/A2) geolocated and calibrated brightness temperatures V005 (AIRABRAD) at GES DISC"

5. You should now be at 'https://cmr.earthdata.nasa.gov/search/concepts/C1243477366-GES_DISC.html' (unless they've changed the site).

- Select the 'Download data' tab
- Select 'Earthdata search'
- Select the AIRS link under 'Matching datasets' (I have not tested the NRT products)

6. You can now select 'Granule filters' to choose your start and end dates.
7. Select the granules you want, then click 'download all' and 'download data'
8. Click download access script

9. Follow the instructions on that page to download the data.

Each granule is about 560K and has names like

```
AIRS.2019.06.22.236.L1B.AMSU_Rad.v5.0.0.0.G19174110442.hdf
```

Build

See the *Dependencies Section* of the AIRS/README.

Because the data are distributed in HDF-EOS format, and the RTTOV libraries require HDF5 (incompatible with HDF-EOS) a two-step conversion is necessary. The data must be converted from HDF to netCDF (which can be done without HDF5) and then the netCDF files can be converted to DART radiance observation format - which is the part that requires `obs_def_rttov_mod.f90`, which is the part that requires HDF5.

The NASA Earthdata Data Access Services website is the download site, at press time, the following packages were required to build HDF-EOS Release v2.20:

- hdf-4.2.13.tar.gz
- HDF-EOS2.20v1.00.tar.Z
- HDF-EOS2.20v1.00_TestDriver.tar.Z
- HDF-EOS_REF.pdf
- HDF-EOS_UG.pdf
- jpegsrc.v9b.tar.gz
- zlib-1.2.11.tar.gz

Similarly for HDF-EOS5 Release v5.1.16:

- HDF-EOS5.1.16.tar.Z
- HDF-EOS5.1.16_TESTDRIVERS.tar.Z
- HDF-EOS5_REF.pdf
- HDF-EOS5_UG.pdf
- hdf5-1.8.19.tar.gz
- szip-2.1.1.tar.gz

DART provides a script `DART/observations/obs_converters/AIRS/BUILD_HDF-EOS.sh` that may help provide support for these libraries. You *will* have to modify it for your system, and you *probably will* have to iterate on that process. The script takes the stance that if you have to build HDF4, HDF-EOS, HDF5 … you might as well build HDF-EOS5 too. The HDF-EOS5 is entirely optional. The HDF5 will be needed by RTTOV.

6.45.3 Converting from HDF4 to netCDF

There are multiple ways to convert from HDF4 to netCDF. The HDF-EOS Tools and Information Center provides binaries for several common platforms as well as source code should you need to build your own.

HDF4 CF CONVERSION TOOLKIT

The HDF-EOS Tools and Information Center provides the HDF4 CF CONVERSION TOOLKIT

> The HDF4 CF (H4CF) Conversion Toolkit can access various NASA HDF4 external and HDF-EOS2 external files by following the CF conventions external. The toolkit includes a conversion library for application developers and a conversion utility for NetCDF users. We have translated the information obtained from various NASA HDF-EOS2 and HDF4 files and the corresponding product documents into the information required by CF into the conversion library. We also have implemented an HDF4-to-NetCDF (either NetCDF-3 or NetCDF-4 classic) conversion tool by using this conversion library. In this web page, we will first introduce how to build the conversion library and the tool from the source. Then, we will provide basic usage of the tool and the conversion library APIs. The information for the supported NASA HDF-EOS2 and HDF4 products and visualization screenshots of some converted NetCDF files will also be presented.

If you download a binary, it's a good habit to verify the checksum. The download page has a link to a .pdf that has the known checksums. Here's how to generate the checksum. Be aware that when I downloaded the file (via Chrome or 'wget') on an OSX system, the checksum did not match. When I downloaded the file on a linux system, the checksum *did* match.

If you download the source, the tar file comes with a `README` and an `INSTALL`. Please become familiar with them. DART also has a build script: `AIRS/shell_scripts/Build_HDF_to_netCDF.csh` that you can customize after you read the `INSTALL` document.

Actually converting to netCDF

While the converter creates very nice netCDF files, there are two global attributes that are exceedingly large and uninformative. Should you want to remove them, I suggest using the `ncatted` command from NCO.

```
h4tonccf_nc4 AIRS.2019.06.22.236.L1B.AMSU_Rad.v5.0.0.0.G19174110442.hdf bob.nc
ncatted -a coremetadata,global,d,, -a StructMetadata_0,global,d,, bob.nc bill.nc
```

The DART `L1_AMSUA_to_netcdf.f90` program

Before I became aware of `h4tonccf_nc4`, I was in the process of writing my own converter `L1_AMSUA_to_netcdf.f90`. *It is not finished.* Furthermore, at this stage, I don't know which variables are needed to be a viable DART observation sequence file, and I don't see the point in converting EVERYTHING.

6.46 Aviso+/CMEMS Observations

6.46.1 Overview

This short description of the `SEALEVEL_GLO_SLA_L3_REP_OBSERVATIONS_008_018` product is repeated from the **INFORMATION** tab from the Copernicus Marine Environment Monitoring Service online catalogue (in April 2017).

> For the Global Ocean- Mono altimeter satellite along-track sea surface heights computed with respect to a twenty-year mean. Previously distributed by Aviso+, no change in the scientific content. All the missions are homogenized with respect to a reference mission which is currently Jason-2. This product is computed

with an optimal and centered computation time window (6 weeks before and after the date). Two kinds of datasets are proposed: filtered (nominal dataset) and unfiltered.

The main researcher for this project was Fred Castruccio.

The `convert_aviso.f90` program is designed to read a netCDF file containing the (Level 3) sea surface anomalies from any of the following platforms: "Jason-1", "Envisat", or "Geosat Follow On". One of those platforms must be listed in the netCDF file global attribute: `platform`

The data files have names like:

- dt_global_j1_sla_vfec_20080101_20140106.nc,
- dt_global_en_sla_vfec_20080101_20140106.nc, or
- dt_global_g2_sla_vfec_20080101_20140106.nc

corresponding to the "Jason-1", "Envisat", and the "Geosat Follow On" platforms. The DART observation TYPE corresponding to each of these platforms are `J1_SEA_SURFACE_ANOMALY`, `EN_SEA_SURFACE_ANOMALY`, and `GFO_SEA_SURFACE_ANOMALY`, respectively and are defined in obs_def_ocean_mod.f90.

Fred wrote a python script (`shell_scripts/convert_aviso.py`) to repeatedly call `convert_aviso` and decided it was easiest to simply provide the input file name as a command line argument and always have the output file have the name `obs_seq.aviso`. As such, there is no input namelist specifically for these parameters, but other DART modules still require run-time crontrol specified by `input.nml`.

After creating a large number of output observation sequence files, it is usually necessary to consolidate the files and subset them into files containing just the timeframe required for a single assimilation. **NOTE**: the `obs_sequence_tool` is constructed for just this purpose.

The `shell_scripts/makedaily.sh` script attempts to consolidate all the SLA observations and those that may have been (separately) converted from the World Ocean Database into 24-hour segments centered at midnight GMT. You will have to modify the `makedaily.sh` script to suit your filesystem and naming convention. It is provided as a starting point.

Reminder: (according to the data providers): In order to compute Absolute Dynamic Topography, the Mean Dynamic Topography (MDT) can be added. It is distributed by Aviso+ (http://www.aviso.altimetry.fr/en/data/products/auxiliary-products/mdt.html). Fred was using this product in assimilations with POP, so he chose a different source for MDT - consistent with POP's behavior.

6.46.2 Data sources

The Copernicus Marine and Environment Monitoring Service (CMEMS) has taken over the processing and distribution of the Ssalto/Duacs multimission altimeter products formerly administered by Aviso+. After a registration process, the along-track sea level anomalies (SLA) may be downloaded from http://marine.copernicus.eu/services-portfolio/access-to-products/ - search for the `SEALEVEL_GLO_SLA_L3_REP_OBSERVATIONS_008_018` if it does not come up directly.

6.46.3 Programs

convert_aviso.f90	does the actual conversion from netCDF to a DART observation sequence file, which may be ASCII or binary.
shell_scripts/convert_aviso.py	python script to convert a series of input files and datestamp the output files.
shell_scripts/makedaily.sh	shell script to repeatedly call obs_sequence_tool to consolidate multiple observation sequence files into an observation sequence file that has ALL the observations from ALL platforms in a single file. makedaily.sh is capable of looping over time ranges and creating observation sequences for each time range.

6.46.4 Namelist

There is no namelist for convert_aviso, but other namelists control aspects of the execution, namely &obs_sequence_nml:write_binary_obs_sequence. see *MODULE obs_sequence_mod*.

6.46.5 Modules used

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
models/template/model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/obs_converters/AVISO/convert_aviso.f90
observations/obs_converters/utilities/obs_utilities_mod.f90
```

6.47 PROGRAM level4_to_obs

6.47.1 Overview

AmeriFlux level 4 data to DART observation sequence converter

This routine is designed to convert the flux tower Level 4 data from the AmeriFlux network of observations from micrometeorological tower sites. AmeriFlux is part of FLUXNET and the converter is hoped to be a suitable starting point for the conversion of observations from FLUXNET. As of May 2012, I have not yet tried to work with any other observations from FLUXNET.

The AmeriFlux Level 4 products are recorded using the local time. DART observation sequence files use GMT. For more information about AmeriFlux data products, go to http://ameriflux.lbl.gov.

Warning: There was a pretty severe bug in the converter that swapped latent heat flux and sensible heat flux. The bug was present through revision 7200. It was corrected on 30 Dec 2016.

The workflow is usually:

1. download the Level 4 data for the towers and years in question (see DATA SOURCES below)
2. record the TIME ZONE, latitude, longitude, and elevation for each tower
3. build the DART executables with support for the tower observations. This is done by running `preprocess` with `obs_def_tower_mod.f90` in the list of `input_files` for `preprocess_nml`.
4. provide basic tower information via the `level4_to_obs_nml` namelist since this information is not contained in the Level 4 data file
5. convert each Level 4 data file individually using `level4_to_obs`
6. combine all output files for the region and timeframe of interest into one file using *program obs_sequence_tool*

For some models (CLM, for example), it is required to reorganize the observation sequence files into a series of files that contains ONLY the observations for each assimilation. This can be achieved with the makedaily.sh script.

6.47.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&level4_to_obs_nml
   text_input_file = 'textdata.input',
   obs_out_file    = 'obs_seq.out',
   year            = -1,
   timezoneoffset  = -1,
   latitude        = -1.0,
   longitude       = -1.0,
   elevation       = -1.0,
   flux_height     = -1.0,
   maxgoodqc       = 3,
   verbose         = .false.
   /
```

Contents	Type	Description
text_input_file	character(len=128)	Name of the Level 4 ASCII file of comma-separated values. This may be a relative or absolute filename.
obs_out_file	character(len=128)	Name of the output observation sequence file.
year	integer	The year of the observations in the Level 4 text file.
timezoneoffset	real	the time zone offset (in hours) of the station. The tower observation times are local time, we need to convert them to GMT.
latitude	real	Latitude (in degrees N) of the tower.
longitude	real	Longitude (in degrees E) of the tower. For internal consistency, DART uses longitudes in the range [0,360]. An input value of -90 will be converted to 270, for example.
elevation	real	surface elevation (in meters) of the tower.
flux_height	real	height (in meters) of the flux instrument on the tower.
maxgoodqc	real	maximum value of any observation quality control flag to pass through to the output observation sequence. Keep in mind that `filter` has the ability to discriminate on the value, so there is really little to be gained by rejecting them during the conversion.
verbose	logical	Print extra information during the `level4_to_obs` execution.

6.47.3 Data sources

The data was acquired from http://cdiac.ornl.gov/ftp/ameriflux/data/Level4/Sites_ByName
and have names like

```
USBar2004_L4_h.txt, USHa12004_L4_h.txt, USNR12004_L4_h.txt,
USSP32004_L4_h.txt, USSRM2004_L4_h.txt, USWCr2004_L4_h.txt, USWrc2004_L4_h.txt, ...
```

The Level 4 products in question are ASCII files of comma-separated values taken every 30 minutes for an entire year. The first line is a comma-separated list of column descriptors, all subsequent lines are comma-separated numerical values. The converter presently searches for the columns pertaining to *NEE_or_fMDS*, *H_f*, *LE_f*, their corresponding quality control fields, and those columns pertaining to the time of the observation. These values are mapped as follows:

Level 4 units	Level 4 variable	description	DART type	DART kind	DART units
W/m^2	LE_f	Latent Heat Flux	TOWER_LATENT_HEAT_FLUX	LATENT_HEAT_FLUX	W/m^2
[0-3]	LE_fqc	QC for LE_f	N/A	N/A	same
W/m^2	H_f	Sensible Heat Flux	TOWER_SENSIBLE_HEAT_FLUX	SENSIBLE_HEAT_FLUX	W/m^2
[0-3]	H_fqc	QC for H_f	N/A	N/A	same
umolCO2/m^2/s	NEE_or_fMDS	Net Ecosystem Production	TOWER_NETC_ECO_EXCHANGE	NET_CARBON_PRODUCTION	gC/m^2/s
[0-3]	NEE_or_fMDSqc	QC for NEE_or_fMDS	N/A	N/A	same

The `LE_fqc`, `H_fqc`, and `NEE_or_fMDSqc` variables use the following convention:

> 0 = original, 1 = category A (most reliable), 2 = category B (medium), 3 = category C (least reliable). (Refer to Reichstein et al. 2005 Global Change Biology for more information)

I am repeating the AmeriFlux Data Fair-Use Policy because I believe it is important to be a good scientific citizen:

> "The AmeriFlux data provided on this site are freely available and were furnished by individual AmeriFlux scientists who encourage their use. Please kindly inform in writing (or e-mail) the appropriate AmeriFlux scientist(s) of how you intend to use the data and of any publication plans. It is also important to contact the AmeriFlux investigator to assure you are downloading the latest revision of the data and to prevent potential misuse or misinterpretation of the data. Please acknowledge the data source as a citation or in the acknowledgments if no citation is available. If the AmeriFlux Principal Investigators (PIs) feel that they should be acknowledged or offered participation as authors, they will let you know and we assume that an agreement on such matters will be reached before publishing and/or use of the data for publication. If your work directly competes with the PI's analysis they may ask that they have the opportunity to submit a manuscript before you submit one that uses unpublished data. In addition, when publishing please acknowledge the agency that supported the research. Lastly, we kindly request that those publishing papers using AmeriFlux data provide reprints to the PIs providing the data and to the AmeriFlux archive via ameriflux.lbl.gov."

6.47.4 Programs

The `level4_to_obs.f90` file is the source for the main converter program. Look at the source code where it reads the example data file. You will almost certainly need to change the "read" statement to match your data format. The example code reads each text line into a character buffer and then reads from that buffer to parse up the data items.

To compile and test, go into the work subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

To change the observation types, look in the `DART/obs_def` directory. If you can find an obs_def_XXX_mod.f90 file with an appropriate set of observation types, change the 'use' lines in the converter source to include those types. Then add that filename in the `input.nml` namelist file to the &preprocess_nml namelist, the 'input_files' variable. Multiple files can be listed. Then run quickbuild.sh again. It remakes the table of supported observation types before trying to recompile the source code.

An example script for converting batches of files is in the `shell_scripts` directory. A tiny example data file is in the `data` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the script saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

6.47.5 Decisions you might need to make

See the discussion in the *Creating an obs_seq file from real observations* page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.48 CHAMP

This is a modification of a standard `text_to_obs` converter that comes that comes with DART.

This observation converter reads CHAMP and GRACE density files, as described in Sutton (2011)[1] and outputs `obs_seq` files that can be assimilated using DART.

Warning: If an `obs_seq.out` file already exists, this converter automatically adds new observations to that file without deleting it. This is done to allow the wrapper script (work/convert.sh) to process sequentially numbered Density_*.ascii files, as documented in the comments in convert.sh. If you don't want this behavior, comment out lines 129-132 in text_to_obs.f90 and rebuild.

6.48.1 Namelist

Please inspect the `text_to_obs_nml` namelist in `work/input.nml` to ensure the input and output filenames are specified properly.

Note: The *work/Density_3deg_02_335.ascii* file is truncated to 2 datapoints merely to demonstrate the format. It isn't meant to be used for real experiments.

6.48.2 Author

Thank you to Alexey Morozov for contributing this observation converter.

6.48.3 References

6.49 PROGRAM `cice_to_obs`

6.49.1 Overview

Sea ice percentage observations to DART converter

This converter reads the binary sea ice observations from the snow and ice data center files and outputs DART obs_seq format files. It will loop over multiple days inside a single run of the converter program.

[1] Sutton, Erik K., 2011: Accelerometer-Derived Atmospheric Density from the CHAMP and GRACE Satellites.

6.49.2 Data sources

The National Snow and Ice Data Center supplies the data files read by this converter. (I believe it is this format?)

6.49.3 Programs

The `cice_to_obs.f90` file is the source for the main converter program. More documentation is in the source code file especially around where the namelist variables are declared.

6.50 CONAGUA

The streamflow observations from CONAGUA are naturally in a Microsoft database format. Mirce converts these one-at-a-time to a csv format. The filenames have a gage identifier in them, there is another file that has the lat/lon of the gage.

```
/glade/scratch/mirce/LaSierra/Observations/
```

The existing DART csv readers are:

```
vi -R Ameriflux/level4_to_obs.f90 \
CHAMP/CHAMP_density_text_to_obs.f90 \
CNOFS/CNOFS_text_to_obs.f90 \
COSMOS/COSMOS_development.f90 \
COSMOS/COSMOS_to_obs.f90 \
MODIS/MOD15A2_to_obs.f90 \
ROMS/convert_roms_obs.f90 \
gnd_gps_vtec/gnd_gps_vtec_text_to_obs.f90 \
gps/convert_cosmic_gps_cdf.f90 \
gps/convert_cosmic_ionosphere.f90 \
quikscat/quikscat_JPL_mod.f90 \
snow/snow_to_obs.f90 \
text/text_to_obs.f90 \
text_GITM/text_to_obs.f90
```

One of these should be close enough. Some are more sophisticated in that they try to determine which column contains the string that identifies the year, mondy, day, etc. - as opposed to hardcoding the knowledge about which column is which.

These are the meanings for each of the column headers in the daily observation files: pk_anio = Year pk_mes = Month ngasto_d01, d02 . . . and so on up to d31 = Streamflow in day 01, day 02 . . . day 31 The streamflow is in cms

6.51 PROGRAM `COSMOS_to_obs`

6.51.1 Overview

COSMOS "level 2" text file to DART converter

COSMOS is an NSF supported project to measure soil moisture on the horizontal scale of hectometers and depths of decimeters using cosmic-ray neutrons. The data for each station is available from the COSMOS data portal with

several levels of processing. The metadata for each station (location, height, etc) is also available from the data portal. The **Level 2 Data** is most suited for use with DART.

Since each site has a separate input data file, and the metadata for each site must essentially be hand-input to the converter program, it is generally easiest to convert the observations for each site separately and then use the *program obs_sequence_tool* to combine the observations from multiple sites and restrict the DART observation sequence file to contain just the observations of the timeframe of interest.

FYI - in DART, the soil moisture profile is converted to expected neutron counts using the **CO**smic-ray **S**oil **M**oisture **I**nteraction **C**ode (COSMIC), developed at the University of Arizona by Rafael Rosolem and Jim Shuttleworth.

The workflow is usually:

1. get the site metadata and enter it in the `input.nml` *&COSMOS_to_obs_nml*
2. download the Level 2 Data and prefix the filename with the station name (or else they all get named `corcounts.txt`) and enter the filename into *&COSMOS_to_obs_nml*
3. make sure the station soil parameters and COSMIC parameters are contained in the `observations/COSMOS/data/COSMIC_parlist.nc` (more on this in the section on COSMIC parameters)
4. run `COSMOS_to_obs` to generate a DART observation sequence file for the station and rename the output file if necessary (you can explicity name the output file via the namelist).
5. repeat steps 1-4 for this converter to generate a DART observation sequence file for each station.
6. use the *program obs_sequence_tool* to combine the observations from multiple sites

6.51.2 Data sources

The COSMOS data portal can be found at: http://cosmos.hwr.arizona.edu/Probes/probemap.php The data for each station is available from the data portal with several levels of processing. The metadata for each station (location, height, etc) is also available from the data portal. The **Level 2 Data** is most suited for use with DART. An example of the Level 2 Data follows:

```
YYYY-MM-DD HH:MM  MOD PROBE PRESS  SCALE SANPE INTEN OTHER CORR ERR
2009-10-23 18:34 5996 0.800 1.087 06.901 2.486 1.062 1.000 1768 022
2009-10-23 19:34 5885 0.800 1.080 06.901 2.486 1.059 1.000 1729 022
2009-10-23 20:34 6085 0.800 1.072 06.901 2.486 1.059 1.000 1774 022
2009-10-23 21:34 6339 0.800 1.068 06.901 2.486 1.059 1.000 1843 023
...
```

6.51.3 Programs

The `COSMOS_to_obs.f90` file is the source code for the main converter program. At present there is an uncomfortable assumption that the order of the columns in the Level 2 data is fixed. I hope to relax that requirement in the near future. `COSMOS_to_obs` reads each text line into a character buffer and then reads from that buffer to parse up the data items. The items are then combined with the COSMIC parameters for that site and written to a DART-format observation sequence file. The DART format allows for the additional COSMIC parameters to be contained as metadata for each observation.

To compile and test, go into the `COSMOS/work` subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. The *program obs_sequence_tool* manipulates (i.e. combines, subsets) DART observation files once they have been created. The default observations supported are those defined in `DART/observations/forward_operators/obs_def_land_mod.f90` and `DART/observations/forward_operators/obs_def_COSMOS_mod.f90`. If you need additional observation types, you will have to add the appropriate `obs_def_XXX_mod.f90` file to the `input.nml &preprocess_nml:input_files`

variable and run `quickbuild.sh` again. It rebuilds the table of supported observation types before compiling the source code.

Guidance on COSMIC parameters

Additional information is needed by DART to convert soil moisture profiles to neutron counts. Each COSMOS instrument has site-specific parameters describing soil properties etc. Those parameters have been inserted into the observation file as metadata for each observation to simplify the DART observation operator. It is a bit redundant as currently implemented, but it is convenient.

`COSMOS_to_obs` reads the site name from the input namelist and the known station information from `COSMIC_parlist.nc`. The simplest way to add a new station to `COSMIC_parlist.nc` is probably to:

1. manually enter the information into the "data" section of `COSMIC_parlist_station.txt`
2. then use `ncgen` to convert `COSMIC_parlist_station.txt` to a netCDF file.
3. That netCDF file can be concatenated onto `COSMIC_parlist.nc` with a simple `ncrcat` command.

Listing the sites already supported is easy:

```
observations/COSMOS/data % ncdump -v sitenames COSMIC_parlist.nc
netcdf COSMIC_parlist {
dimensions:
        nsites = UNLIMITED ; // (42 currently)
        strlength = 21 ;
variables:
        char sitenames(nsites, strlength) ;
                sitenames:long_name = "COSMOS Site Names" ;
        double longitude(nsites) ;
                longitude:long_name = "Longitude" ;
                longitude:units = "degrees" ;
        double latitude(nsites) ;
                latitude:long_name = "Latitude" ;
                latitude:units = "degrees" ;
        double elevation(nsites) ;
                elevation:long_name = "Elevation" ;
                elevation:units = "m" ;
        double bd(nsites) ;
                bd:long_name = "Dry Soil Bulk Density" ;
                bd:units = "g cm{-3}" ;
        double lattwat(nsites) ;
                lattwat:long_name = "Lattice Water Content" ;
                lattwat:units = "m{3} m{-3}" ;
        double N(nsites) ;
                N:long_name = "High Energy Neutron Intensity" ;
                N:units = "relative counts" ;
        double alpha(nsites) ;
                alpha:long_name = "Ratio of Fast Neutron Creation Factor (Soil to Water)
↪" ;
                alpha:units = "-" ;
        double L1(nsites) ;
                L1:long_name = "High Energy Soil Attenuation Length" ;
                L1:units = "g cm{-2}" ;
```

(continues on next page)

(continued from previous page)

```
        double L2(nsites) ;
                L2:long_name = "High Energy Water Attenuation Length" ;
                L2:units = "g cm{-2}" ;
        double L3(nsites) ;
                L3:long_name = "Fast Neutron Soil Attenuation Length" ;
                L3:units = "g cm{-2}" ;
        double L4(nsites) ;
                L4:long_name = "Fast Neutron Water Attenuation Length" ;
                L4:units = "g cm{-2}" ;

// global attributes:
                :website = "COsmic-ray Soil Moisture Observing System (COSMOS) -
                            http://cosmos.hwr.arizona.edu" ;
data:

 sitenames =
  "ARM-1                ",
  "Austin_Cary          ",
  "Bondville            ",
  "Brookings            ",
  "Chestnut_Ridge_NOAA  ",
  "Coastal_Sage_UCI     ",
  "Daniel_Forest        ",
  "Desert_Chaparral_UCI ",
  "Fort_Peck            ",
  "Harvard_Forest       ",
  "Hauser_Farm_North    ",
  "Hauser_Farm_South    ",
  "Howland              ",
  "Iowa_Validation_Site ",
  "Island_Dairy         ",
  "JERC                 ",
  "Kendall              ",
  "KLEE                 ",
  "Manitou_Forest_Ground",
  "Metolius             ",
  "Morgan_Monroe        ",
  "Mozark               ",
  "Mpala_North          ",
  "Neb_Field_3          ",
  "P301                 ",
  "Park_Falls           ",
  "Pe-de-Gigante        ",
  "Rancho_No_Tengo      ",
  "Reynolds_Creek       ",
  "Rietholzbach         ",
  "Rosemount            ",
  "San_Pedro_2          ",
  "Santa_Rita_Creosote  ",
  "Savannah_River       ",
  "Silver_Sword         ",
  "SMAP-OK              ",
```

(continues on next page)

(continued from previous page)

```
  "Soaproot             ",
  "Sterling             ",
  "Tonzi_Ranch          ",
  "UMBS                 ",
  "UVA                  ",
  "Wind_River           " ;
}
```

The observation sequence files will look something like the following, the attributes on the "cosmic" record are the information from `COSMIC_parlist.nc` (in their closes 64-bit real representation):

```
 obs_sequence
obs_kind_definitions
           1
          20 COSMOS_NEUTRON_INTENSITY
  num_copies:            1  num_qc:            1
  num_obs:         3840  max_num_obs:         3840
observation
COSMOS QC
  first:            1  last:         3840
 OBS            1
   1048.0000000000000
   1.0000000000000000
          -1           2          -1
obdef
loc3d
     4.154723123116714        0.7997185899100618         0.000000000000000     -1
kind
          20

cosmic 0.88500000000000001    5.840999999999999966E-002    336.95696938999998    0.
↪3191802587700000
        161.98621864285701     129.14558984999999         55.311849408000000     3.
↪8086191933000002
              1

 77340     150034
   1225.0000000000000
   ...
```

6.51.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&COSMOS_to_obs_nml
   site_metadata_file = 'COSMIC_parlist.nc'
   text_input_file    = 'textdata.input',
   obs_out_file       = 'obs_seq.out',
   sitename           = 'missing',
```

(continues on next page)

(continued from previous page)

```
   maxgoodqc          =  3,
   verbose            = .false.
   /
```

Contents	Type	Description
site_metadata_file	character(len=256)	The netCDF file containing the parameter values for each site.
text_input_file	character(len=128)	The text file containing the raw observations for each site.
obs_out_file	character(len=128)	The output observation sequence file for DART.
sitename	character(len=128)	The name of the site. Must match one of the site names in the `site_metadata_file`. Case-insensitive match, trailing blanks ignored. Use *ncdump -v sitenames COSMIC_parlist.nc*
maxgoodqc	integer	left for future implementation.
verbose	logical	A switch to specify the amount of run-time output. `.true.` the most amount of output. `.false.` the least amount of output.

Cosmos_to_obs namelist

```
&COSMOS_to_obs_nml
   site_metadata_file = 'COSMIC_parlist.nc',
   text_input_file    = 'SantaRita_corcounts.txt',
   obs_out_file       = 'SantaRita_obs_seq.out',
   sitename           = 'Santa_Rita_Creosote',
```

6.51.5 References

- The COSMOS web page.
- Franz, T.E, M. Zreda, T.P.A. Ferre, R. Rosolem, C. Zweck, S. Stillman, X. Zeng and W.J. Shuttleworth, 2012: Measurement depth of the cosmic-ray soil moisture probe affected by hydrogen from various sources. Water Resources Research 48, W08515, doi:10.1029/2012WR011871
- Franz, T.E, M. Zreda, R. Rosolem, T.P.A. Ferre, 2012: Field validation of cosmic-ray soil moisture probe using a distributed sensor network. Vadose Zone Journal (in press), doi:10.2136/vzj2012.0046

6.51.6 Future Plans

- Implement a routine to automatically determine the column indices of the columns of interest.
- Implement a QC encoding that reflects the uncertainty of the measurement. Presently, all Level 2 data have an incoming QC of 1.

6.52 PROGRAM COSMOS_development

6.52.1 Overview

Trial COSMOS text file to DART converter

COSMOS is an NSF supported project to measure soil moisture on the horizontal scale of hectometers and depths of decimeters using cosmic-ray neutrons. The data for each station is available from the COSMOS data portal with several levels of processing. The metadata for each station (location, height, etc) is also available from the data portal. The **Level 2 Data** is most suited for use with DART, but does not currently have a correction for the amount of hydrogen in the atmospheric volume near the probe. To this end, Rafael Rosolem has a separate data stream. `COSMOS_development` reads Rafaels data streams and converts them to DART observation sequence files. **Since these data streams are not widespread, we recommend using** *PROGRAM COSMOS_to_obs*.
The workflow is usually:

1. get the site metadata and enter it in the `input.nml` *&COSMOS_development_nml*
2. acquire the development observation data and prefix the filename with the station name (or else they all get named `corcounts.txt`) and enter the filename into *&COSMOS_development_nml*
3. make sure the station soil parameters and COSMIC parameters are contained in the `observations/COSMOS/data/COSMIC_parlist.nc` (more on this in the section on COSMIC parameters)
4. run `COSMOS_development` to generate a DART observation sequence file for the station and rename the output file if necessary (you can explicity name the output file via the namelist).
5. repeat steps 1-4 for this converter to generate a DART observation sequence file for each station.
6. use the *program obs_sequence_tool* to combine the observations from multiple sites

6.52.2 Data sources

The COSMOS data portal can be found at: http://cosmos.hwr.arizona.edu/Probes/probemap.php The development observation data for each station is generally not available. The metadata for each station (location, height, etc) is also available from the data portal. The **Level 2 Data** is most suited for use with DART. We recommend using *PROGRAM COSMOS_to_obs*. An example of the development observation data follows:

```
month,day,hour,doy,neutron_fluxAVE,neutron_fluxSTD,neutron_fluxQC
 1, 1, 0,  1,-9999,9999,3
 1, 1, 1,  1,-9999,9999,3
 1, 1, 2,  1,-9999,9999,3
 1, 1, 3,  1,-9999,9999,3
...
```

6.52.3 Programs

The COSMOS_development.f90 file is the source code for the main converter program. At present there is an uncomfortable assumption that the order of the columns in the Level 2 data is fixed. I hope to relax that requirement in the near future. COSMOS_development reads each text line into a character buffer and then reads from that buffer to parse up the data items. The items are then combined with the COSMIC parameters for that site and written to a DART-format observation sequence file. The DART format allows for the additional COSMIC parameters to be contained as metadata for each observation.

To compile and test, go into the COSMOS/work subdirectory and run the quickbuild.sh script to build the converter and a couple of general purpose utilities. The *program obs_sequence_tool* manipulates (i.e. combines, subsets) DART observation files once they have been created. The default observations supported are those defined in DART/observations/forward_operators/obs_def_land_mod.f90 and DART/observations/forward_operators/obs_def_COSMOS_mod.f90. If you need additional observation types, you will have to add the appropriate obs_def_XXX_mod.f90 file to the input.nml &preprocess_nml:input_files variable and run quickbuild.sh again. It rebuilds the table of supported observation types before compiling the source code.

COSMIC parameters

Additional information is needed by DART to convert soil moisture profiles to neutron counts. Each COSMOS instrument has site-specific parameters describing soil properties etc. Those parameters have been inserted into the observation file as metadata for each observation to simplify the DART observation operator. It is a bit redundant as currently implemented, but it is convenient.
COSMOS_development reads the site name from the input namelist and the known station information from COSMIC_parlist.nc. The simplest way to add a new station to COSMIC_parlist.nc is probably to:

1. manually enter the information into the "data" section of COSMIC_parlist_station.txt
2. then use ncgen to convert COSMIC_parlist_station.txt to a netCDF file.
3. That netCDF file can be concatenated onto COSMIC_parlist.nc with a simple ncrcat command.

Listing the sites already supported is easy:

```
observations/COSMOS/data % ncdump -v sitenames COSMIC_parlist.nc
netcdf COSMIC_parlist {
dimensions:
        nsites = UNLIMITED ; // (42 currently)
        strlength = 21 ;
variables:
        char sitenames(nsites, strlength) ;
                sitenames:long_name = "COSMOS Site Names" ;
        double longitude(nsites) ;
                longitude:long_name = "Longitude" ;
                longitude:units = "degrees" ;
        double latitude(nsites) ;
                latitude:long_name = "Latitude" ;
                latitude:units = "degrees" ;
        double elevation(nsites) ;
                elevation:long_name = "Elevation" ;
                elevation:units = "m" ;
        double bd(nsites) ;
                bd:long_name = "Dry Soil Bulk Density" ;
```

(continues on next page)

(continued from previous page)

```
                bd:units = "g cm{-3}" ;
        double lattwat(nsites) ;
                lattwat:long_name = "Lattice Water Content" ;
                lattwat:units = "m{3} m{-3}" ;
        double N(nsites) ;
                N:long_name = "High Energy Neutron Intensity" ;
                N:units = "relative counts" ;
        double alpha(nsites) ;
                alpha:long_name = "Ratio of Fast Neutron Creation Factor (Soil to Water)
↪" ;
                alpha:units = "-" ;
        double L1(nsites) ;
                L1:long_name = "High Energy Soil Attenuation Length" ;
                L1:units = "g cm{-2}" ;
        double L2(nsites) ;
                L2:long_name = "High Energy Water Attenuation Length" ;
                L2:units = "g cm{-2}" ;
        double L3(nsites) ;
                L3:long_name = "Fast Neutron Soil Attenuation Length" ;
                L3:units = "g cm{-2}" ;
        double L4(nsites) ;
                L4:long_name = "Fast Neutron Water Attenuation Length" ;
                L4:units = "g cm{-2}" ;

// global attributes:
                :website = "COsmic-ray Soil Moisture Observing System (COSMOS) -
                            http://cosmos.hwr.arizona.edu" ;
data:

 sitenames =
  "ARM-1                ",
  "Austin_Cary          ",
  "Bondville            ",
  "Brookings            ",
  "Chestnut_Ridge_NOAA  ",
  "Coastal_Sage_UCI     ",
  "Daniel_Forest        ",
  "Desert_Chaparral_UCI ",
  "Fort_Peck            ",
  "Harvard_Forest       ",
  "Hauser_Farm_North    ",
  "Hauser_Farm_South    ",
  "Howland              ",
  "Iowa_Validation_Site ",
  "Island_Dairy         ",
  "JERC                 ",
  "Kendall              ",
  "KLEE                 ",
  "Manitou_Forest_Ground",
  "Metolius             ",
  "Morgan_Monroe        ",
  "Mozark               ",
```

(continues on next page)

(continued from previous page)

```
  "Mpala_North          ",
  "Neb_Field_3          ",
  "P301                 ",
  "Park_Falls           ",
  "Pe-de-Gigante        ",
  "Rancho_No_Tengo      ",
  "Reynolds_Creek       ",
  "Rietholzbach         ",
  "Rosemount            ",
  "San_Pedro_2          ",
  "Santa_Rita_Creosote  ",
  "Savannah_River       ",
  "Silver_Sword         ",
  "SMAP-OK              ",
  "Soaproot             ",
  "Sterling             ",
  "Tonzi_Ranch          ",
  "UMBS                 ",
  "UVA                  ",
  "Wind_River           " ;
}
```

The observation sequence files will look something like the following, the attributes in yellow are the information from `COSMIC_parlist.nc`:

```
 obs_sequence
obs_kind_definitions
           1
          20 COSMOS_NEUTRON_INTENSITY
  num_copies:            1  num_qc:            1
  num_obs:         3840  max_num_obs:         3840
observation
COSMOS QC
  first:            1  last:         3840
 OBS            1
   1048.0000000000000
   1.0000000000000000
          -1           2          -1
obdef
loc3d
     4.154723123116714        0.7997185899100618         0.000000000000000     -1
kind
          20
```

cosmic 0.88500000000000001 5.84099999999999966E-002 336.95696938999998 0.31918025877000000 161.98621864285701 129.14558984999999 55.311849408000000 3.8086191933000002 1

```
77340     150034
  1225.0000000000000
  ...
```

6.52.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&COSMOS_development_nml
   site_metadata_file = 'COSMIC_parlist.nc'
   text_input_file    = 'textdata.input',
   obs_out_file       = 'obs_seq.out',
   sitename           = 'missing',
   year               = -1
   maxgoodqc          =  3,
   verbose            = .false.
   /
```

Contents	Type	Description
site_metadata_file	character(len=256)	The netCDF file containing the parameter values for each site.
text_input_file	character(len=128)	The text file containing the raw observations for each site.
obs_out_file	character(len=128)	The output observation sequence file for DART.
sitename	character(len=128)	The name of the site. Must match one of the site names in the `site_metadata_file`. Case-insensitive match, trailing blanks ignored. Use *ncdump -v sitenames COSMIC_parlist.nc*
year	integer	The year of the data.
maxgoodqc	integer	left for future implementation.
verbose	logical	A switch to specify the amount of run-time output. `.true.` the most amount of output. `.false.` the least amount of output.

COSMOS development namelist

```
&COSMOS_development_nml
   site_metadata_file = '../data/COSMIC_parlist.nc',
   text_input_file    = 'SantaRita_corcounts.txt',
   obs_out_file       = 'SantaRita_obs_seq.out',
   sitename           = 'Santa_Rita_Creosote',
```

6.52.5 References

- The COSMOS web page.
- Franz, T.E, M. Zreda, T.P.A. Ferre, R. Rosolem, C. Zweck, S. Stillman, X. Zeng and W.J. Shuttleworth, 2012: Measurement depth of the cosmic-ray soil moisture probe affected by hydrogen from various sources. Water Resources Research 48, W08515, doi:10.1029/2012WR011871
- Franz, T.E, M. Zreda, R. Rosolem, T.P.A. Ferre, 2012: Field validation of cosmic-ray soil moisture probe using a distributed sensor network. Vadose Zone Journal (in press), doi:10.2136/vzj2012.0046

6.52.6 Future Plans

- Implement a routine to automatically determine the column indices of the columns of interest.
- Implement a QC encoding that reflects the uncertainty of the measurement. Presently, all Level 2 data have an incoming QC of 1.

6.53 PROGRAM `dwl_to_obs`

6.53.1 Overview

DWL to DART converter

These are Doppler Wind Lidar measurements which have previously been extracted from the incoming format and output in ascii format, one pair of wind component observations per line. This converter reads in the ascii file and outputs the data in DART observation sequence (obs_seq) format.

This is OSSE data from a satellite which is expected to be launched in 2015. Information on the satellite mission is here at http://en.wikipedia.org/wiki/ADM-Aeolus.

The workflow is:

- read in the needed information about each observation - location, time, observation values, obs errors - from an ascii file
- call a series of DART library routines to construct a derived type that contains all the information about a single observation
- call another set of DART library routines to put it into a time-sorted series
- repeat the last 2 steps until all observations are processed
- finally, call a write subroutine that writes out the entire series to a file in a format that DART can read in

6.53.2 Data sources

Matic Savli at University of Ljubljana has programs which read the expected instrument formats, do the proper conversions, and write out ascii lines, one per wind observation.

6.53.3 Programs

The `dwl_to_obs.f90` file is the source for the main converter program. There is a sample data file in the "data" directory. The converter reads each text line into a character buffer and then reads from that buffer to parse up the data items.

To compile and test, go into the work subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

The observation types are defined in `DART/obs_def/obs_def_dwl_mod.f90`. That filename must be added to the `input.nml` namelist file, to the &preprocess_nml namelist, the 'input_files' variable before compiling any program that uses these observation types. Multiple files can be listed. Then run quickbuild.sh again. It remakes the table of supported observation types before trying to recompile the source code.

An example script for converting batches of files is in the `shell_scripts` directory. It will need customization before being used.

6.54 GMI Brightness Temperatures

This directory contains the code to convert the GMI Brightness Temperatures in HDF5 format to the DART observation sequence file format.

The dataset of interest is: "GPM GMI Common Calibrated Brightness Temperatures Collocated L1C 1.5 hours 13 km V05 (GPM_1CGPMGMI) at GES DISC" **not** the _R set! The *short name* for this dataset is 'GPM_1CGPMGMI'.

The introductory paragraph for the dataset is:

> Version 5 is the current version of the data set. Version 4 is no longer available and has been superseded by Version 5. All 1C products have a common L1C data structure, simple and generic. Each L1C swath includes scan time, latitude and longitude, scan status, quality, incidence angle, Sun glint angle, and the intercalibrated brightness temperature (Tc). One or more swaths are included in a product. The radiometer data are recalibrated to a common basis so that precipitation products derived from them are consistent. 1CGMI contains common calibrated brightness temperatures from the GMI passive microwave instrument flown on the GPM satellite. 1C-R GMI is a remapped version of 1CGMI which is explained at the end of this section. Swath S1 has 9 channels which are similar to TRMM TMI (10V 10H 19V 19H 23V 37V 37H 89V 89H). Swath S2 has 4 channels similar to AMSU-B (166V 166H 183+/-3V 183+/-8V). Data for both swaths is observed in the same revolution of the instrument.

The citation information for this dataset is:

> Title: GPM GMI Common Calibrated Brightness Temperatures Collocated L1C 1.5 hours 13 km V05 Version: 05 Creator: Wesley Berg Publisher: Goddard Earth Sciences Data and Information Services Center (GES DISC) > Release Date: 2016-03-03T00:00:00.000Z

Linkage: https://disc.gsfc.nasa.gov/datacollection/GPM_1CGPMGMI_05.html

6.54.1 Instructions to download the GPM_1CGPMGMI dataset for the GMI converter

1. Go to https://earthdata.nasa.gov
2. Log in (or create an account if necessary)
3. Search for GMI L1C (the "c" here is for cross-calibrated with other satellites)
4. Scroll down past datasets to "Matching results."

- Follow the link to the GMI common calibrated data set: "GPM GMI Common Calibrated Brightness Temperatures Collocated L1C 1.5 hours 13 km V05 (GPM_1CGPMGMI) at GES DISC" dataset (**NOT** the _R set)

5. You should now be at the https://cmr.earthdata.nasa.gov/search/concepts/C1383813813-GES_DISC.html page.

- Select the 'Download data' tab
- Select 'Earthdata search'
- Select the GPM link under 'Matching datasets'

6. You can now select 'Granule filters' to choose your start and end dates.
7. Select the granules you want, then click 'download all' and 'download data'
8. Click download access script
9. Follow the instructions on that page to download the data.

Each granule is about 28M and has names like:
`1C.GPM.GMI.XCAL2016-C.20160621-S001235-E014508.013137.V05A.HDF5`

Guidelines for converting the observations, thinning, superobbing, etc. are forthcoming. For more background on assimilating radiances in DART, please read https://dart.ucar.edu/pages/Radiance_support.html

When running the DART converter, two swaths (S1, S2) are converted to observations. S1 and S2 have different channels and different "postings," meaning actual observation locations. They are more or less right next to each other …

https://disc.gsfc.nasa.gov/datasets/GPM_1CGPMGMI_05/summary

Swath S1 has 9 channels which are similar to TRMM TMI (10V 10H 19V 19H 23V 37V 37H 89V 89H).
Swath S2 has 4 channels similar to AMSU-B (166V 166H 183+/-3V 183+/-8V).
Data for both swaths is observed in the same revolution of the instrument.

Partial run-time output for one file (no thinning, whole globe, i.e. about 8 million observations):

```
 ...
  Data Metadata: observation
    QC Metadata: GMI QC
First timestamp: day=151747, sec=6309
  calendar Date: 2016 Jun 21 01:45:09
 Last timestamp: day=151747, sec=11863
  calendar Date: 2016 Jun 21 03:17:43
  Number of obs processed  :               5734296
  ------------------------------------------------------------
                    GPM_1_GMI_TB 5734296 obs

 add_swath_observations:  Converted      5734296  obs for swath /S1; total GMI obs =
↪ 5734296

  Data Metadata: observation
    QC Metadata: GMI QC
First timestamp: day=151747, sec=6309
  calendar Date: 2016 Jun 21 01:45:09
 Last timestamp: day=151747, sec=11863
  calendar Date: 2016 Jun 21 03:17:43
  Number of obs processed  :               8279480
  ------------------------------------------------------------
                    GPM_1_GMI_TB 8279480 obs

 add_swath_observations:  Converted      2545184  obs for swath /S2; total GMI obs =
↪ 8279480

 write_obs_seq  opening unformatted observation sequence file "obs_seq.gmi"
 write_obs_seq  closed observation sequence file "obs_seq.gmi"
 convert_gmi_L1.f90 Finished successfully.
 ...
```

6.55 NOAA GOES-R Series Advanced Baseline Imager (ABI) Level 1b Radiances

The data are available from NOAA-NCEI

The **convert_goes_ABI_L1b** program converts ABI Level 1b Radiances in netCDF format to a DART observation sequence file with `GOES_16_ABI_RADIANCE` observations (there is a namelist option to select other GOES satellites, which will have the appropriate observation type).

> The Advanced Baseline Imager (ABI) instrument samples the radiance of the Earth in sixteen spectral bands using several arrays of detectors in the instrument's focal plane. Single reflective band ABI Level 1b Radiance Products (channels 1 - 6 with approximate center wavelengths 0.47, 0.64, 0.865, 1.378, 1.61, 2.25 microns, respectively) are digital maps of outgoing radiance values at the top of the atmosphere for visible and near-infrared (IR) bands. Single emissive band ABI L1b Radiance Products (channels 7 - 16 with approximate center wavelengths 3.9, 6.185, 6.95, 7.34, 8.5, 9.61, 10.35, 11.2, 12.3, 13.3 microns, respectively) are digital maps of outgoing radiance values at the top of the atmosphere for IR bands. Detector samples are compressed, packetized and down-linked to the ground station as Level 0 data for conversion to calibrated, geo-located pixels (Level 1b Radiance data). The detector samples are decompressed, radiometrically corrected, navigated and resampled onto an invariant output grid, referred to as the ABI fixed grid.
>
> Cite as: GOES-R Calibration Working Group and GOES-R Series Program, (2017): NOAA GOES-R Series Advanced Baseline Imager (ABI) Level 1b Radiances. [indicate subset used]. NOAA National Centers for Environmental Information. doi:10.7289/V5BV7DSR. [access date].

6.55.1 Specifying a vertical location

Jeff Steward added (PR 48) the capability to specify a vertical location if desired. This allows for localization in the vertical.

> It's sometimes helpful, even though definitely wrong from a theoretical standpoint, to give a vertical location to satellite observations (which are integrated quantities). This has been an issue with observation-space localization for some time, and this is the standard workaround pioneered by Lili Lei and Jeff Whittaker.

6.55.2 A short description of the namelist options

This table is meant to familiarize you with some of the options available. Until we fully implement automatic documentation generation, you would be well advised to familiarize yourself with the code. This is not the full list of namelist variables ...

variable	default	meaning
x_thin	0	Skip this many per X scan.
y_thin	0	Skip this many per Y scan.
goes_num	16	GOES Satellite number.
reject_dqf	.true.	Bad scan rejection criteria. If .true. and DQF /= 0, the scan is rejected. If .false. any DQF > 1 rejects the scan.
verbose	.false.	Run-time output verbosity
obs_err	MISSING_R8	The observation error standard deviation (std dev, in radiance units) TODO: make this more sophisticated. You must supply a value other than MISSING_R8. Be aware that the observation sequence files convert this to a variance.
vloc_pres	-1.0	The vertical location of this observation (hPa). A negative means there is no vertical location (which is typical for a ve rtically-integrated quantity).

6.56 GPSPW

convert GPS observations of Precipitable Water into DART obs_sequence format.

6.57 GSI2DART

6.57.1 Overview

The GSI2DART converter was contributed by **Craig Schwartz** and **Jamie Bresch** of the Mesoscale & Microscale Meteorology Lab at NCAR. *Thanks Craig and Jamie!*

This converter is designed to convert observation files created by the Gridpoint Statistical Interpolation (GSI) system maintained by the National Oceanic and Atmospheric Administration (NOAA) into DART observation sequence files. The files created by GSI are 'BIG_ENDIAN' and have filenames such as:

- diag_amsua_metop-a_ges.ensmean
- diag_amsua_metop-a_ges.mem001
- diag_amsua_metop-a_ges.mem002
- diag_amsua_n18_ges.ensmean
- diag_amsua_n18_ges.mem001
- diag_amsua_n18_ges.mem002
- diag_amsua_n19_ges.ensmean
- diag_amsua_n19_ges.mem001
- diag_amsua_n19_ges.mem002
- diag_conv_ges.ensmean

- diag_conv_ges.mem001
- diag_conv_ges.mem002

The DART converter uses routines from the GSI system that use the Message Passing Interface (MPI) to process observations in parallel (even when converting a small amount of observations) so MPI is required to execute this observation converter.

Due to these prerequisites, we provide a detailed description of this directory to guide the user.

This directory contains copies of several source code files from GSI. The GSI source code is available via a Github repository managed by NOAA's Environmental Modeling Center (EMC):

https://github.com/NOAA-EMC/GSI

To differentiate between the sets of code, we refer to the root directory of the NOAA-EMC repository as `GSI` and refer to the root directory of this observation converter as `GSI2DART`.

`GSI2DART/enkf` copies seven files from `GSI/src` mostly without modification:

1. `GSI2DART/enkf/constants.f90` from `GSI/src/gsi/constants.f90`
2. `GSI2DART/enkf/kinds.F90` from `GSI/src/gsi/kinds.F90`
3. `GSI2DART/enkf/mpi_readobs.f90` from `GSI/src/enkf/mpi_readobs.f90`
4. `GSI2DART/enkf/readconvobs.f90` from `GSI/src/enkf/readconvobs.f90`
5. `GSI2DART/enkf/read_diag.f90` from `GSI/src/gsi/read_diag.f90`
6. `GSI2DART/enkf/readozobs.f90` from `GSI/enkf/readozobs.f90`
7. `GSI2DART/enkf/readsatobs.f90` from `GSI/enkf/readsatobs.f90`

Note that within `GSI` the source file `kinds.F90` has an upper-case `F90` suffix. Within the `GSI2DART` observation converter, it gets preprocessed into `mykinds.f90` with a lower-case `f90` suffix. Case-insensitive filesystems should be banned . . . until then, it is more robust to implement some name change during preprocessing. The path name specified in `GSI2DART/work/path_names_gsi_to_dart` reflects this processed filename.

The following three files had their open() statements modified to read 'BIG_ENDIAN' files without the need to compile EVERYTHING with the `-convert big_endian` compiler option. Using the DART open_file() routine also provides some nice error handling.

- original: `open(iunit,form="unformatted",file=obsfile,iostat=ios)`
- modified: `iunit = open_file(obsfile,form='unformatted',action='read', convert='BIG_ENDIAN')`

1. `GSI2DART/enkf/readconvobs.f90`
2. `GSI2DART/enkf/readozobs.f90`
3. `GSI2DART/enkf/readsatobs.f90`

6.57.2 DART Modifications

Within GSI2DART

The source files within GSI2DART are:

1. gsi_to_dart.f90: the main program.
2. dart_obs_seq_mod.f90: the DART obs_seq output subroutine.
3. params.f90: the same module name as GSI/src/enkf/params.f90 but with different content. This version is used to avoid modifying GSI2DART/enkf/read*.f90.
4. radinfo.f90: the same module name as GSI/src/gsi/radinfo.f90 but with different content. This version is used to avoid modifying GSI2DART/enkf/read*.f90.
5. mpisetup.f90: the same module name as GSI/src/enkf/mpisetup.f90 but with different content. This version is used to avoid dependency on GSI.

Elsewhere in the repository

This observation converter required modifying two files and adding a module for radiance observation types.

- Modified ../../forward_operators/DEFAULT_obs_def_mod.F90
- Modified ../../DEFAULT_obs_kind_mod.F90
- Added ../../forward_operators/obs_def_radiance_mod.f90 which has radiance observation types

Compiler notes

When using ifort, the Intel Fortran compiler, you may need to add the compiler flag -nostdinc to avoid inserting the standard C include files which have incompatible comment characters for Fortran. You can add this compiler flag in the the GSI2DART/work/mkmf_gsi_to_dart file by adding it to the "-c" string contents.

Please note: this was NOT needed for ifort version 19.0.5.281.

Additional files and directories

1. satinfo is a file read by radinfo.f90 and must exist in the GSI2DART/work directory.
2. datapath specifies the directory containing the data to be converted – it is specified in the gsi_to_dart_nml namelist in GSI2DART/work/input.nml.
3. submit.csh is contained in GSI2DART/work/ – it runs the gsi_to_dart converter once it has been compiled. Again, since GSI requires MPI, multiple processors must be requested to run the gsi_to_dart executable.

6.57.3 Issues

1. The converter requires an ensemble size greater than one and will MPI_Abort() if only one ensemble member is requested.

The following are issues previously recorded in the README:

1. Radiance and surface pressure bias correction
2. Surface pressure altimeter adjustment?
3. Specific humidity obs are transformed to relative humidity. What to do? [Just run EnSRF with psuedo_rh=.false. and assimilate RH obs]
4. DART must use W and PH as control variables [okay, EnSRF can do this too (nvars=6 for WRF-ARW)]
5. Does DART not do vertical localization for surface obs?

```
! If which_vert has no vertical definition for either location do only horizontal
if(loc1%which_vert == VERTISUNDEF .or. loc2%which_vert == VERTISUNDEF) comp_h_only = .
↪true.
! If both verts are surface, do only horizontal
if(loc1%which_vert == VERTISSURFACE .and. loc2%which_vert == VERTISSURFACE) comp_h_only␣
↪= .true.
```

Running with 32 bit reals

The converter has been tested with 64-bit reals as well as 32-bit reals (i.e. r8=r4 and -D_REAL_4). The answers are different only at the roundoff level.

This requires changes in two places:

1. `DART/assimilation_code/modules/utilities/types_mod.f90` change required: r8 = r4
2. `GSI2DART/work/mkmf_gsi_to_dart` change required: `-D_REAL4_`

If these are not set in a compatible fashion, you will fail to compile with the following error (or something similar):

```
../../../../observations/obs_converters/GSI2DART/dart_obs_seq_mod.f90(213): error #6284:
There is no matching specific function for this generic function reference.   [SET_
↪LOCATION]
location = set_location(lon, lat, vloc, which_vert)
-----------------^
```

6.58 GTSPP Observations

6.58.1 Overview

GTSPP (Global Temperature-Salinity Profile Program) data measures vertical profiles of ocean temperature and salinity. The GTPSS home page has detailed information about the repository, observations, and datasets. The programs in this directory convert from the netcdf files found in the repository into DART observation sequence (obs_seq) file format.

6.58.2 Data sources

Data from the GTSPP can be downloaded interactively from the GTSPP data server. It is delivered in netCDF file format, one vertical profile per netCDF file.

Currently each vertical profile is stored in a separate file, so converting a months's worth of observations involves downloading many individual files. The converter program can take a list of input files, so it is easy to collect a month of observations together into a single output file with one execution of the converter program.

The units in the source file are degrees C for temperature, g/kg for salinity, and so far we have not found any error information (not quality control, but observation instrument error values). There is probably instrument source information encoded in these files, but so far we don't have the key. The quality control values are read and only those with a QC of 1 are retained.

6.58.3 Programs

The data is distributed in netCDF file format. DART requires all observations to be in a proprietary format often called DART "obs_seq" format. The files in this directory, a combination of C shell scripts and a Fortran source executable, do this data conversion.

6.58.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&gtspp_to_obs_nml
   gtspp_netcdf_file     = '1234567.nc'
   gtspp_netcdf_filelist = 'gtspp_to_obs_filelist'
   gtspp_out_file        = 'obs_seq.gtspp'
   avg_obs_per_file      = 500
   debug                 = .false.
/
```

Item	Type	Description
gtspp_netcdf_file	character(len=128)	The input filename when converting a single profile. Only one of the two file or filelist items can have a valid value, so to use the single filename set the list name 'gtspp_netcdf_filelist' to the empty string (' ').
gtspp_netcdf_filelist	character(len=128)	To convert a series of profiles in a single execution create a text file which contains each input file, in ascii, one filename per line. Set this item to the name of that file, and set 'gtspp_netcdf_file' to the empty string (' ').
gtspp_out_file	character(len=128)	The output file to be created. To be compatible with earlier versions of this program, if this file already exists it will be read in and the new data will be inserted into that file.
avg_obs_per_file	integer	The code needs an upper limit on the number of observations generated by this program. It can be larger than the actual number of observations converted. The total number of obs is computed by multiplying this number by the number of input files. If you get an error because there is no more room to add observations to the output file, increase this number.
debug	logical	If true, output more debugging messages.

6.58.5 Modules used

```
types_mod
time_manager_mod
utilities_mod
location_mod
obs_sequence_mod
obs_def_mod
obs_def_ocean_mod
obs_kind_mod
netcdf
```

6.58.6 Known Bugs

Does not have correct code for setting observation error variance yet. Also, not sure if the incoming data qc is strict enough.

6.59 MADIS Data Ingest System

6.59.1 Overview

The MADIS (Meteorological Assimilation Data Ingest System) service provides access to real-time and archived data of a variety of types, with added Quality Control (QC) and integration of data from a variety of sources.

To convert a series of MADIS data files (where different types of observations are distributed in separate files), one high level view of the workflow is:

1. convert each madis file, by platform type, into an obs_seq file. one file in, one file out. no time changes. use the `shell_scripts/madis_conv.csh` script. there are script options for hourly output files, or a single daily output file.
2. if you aren't using the wrf preprocessing program, you're ready to go.
3. if you do want to do subsequent wrf preprocessing, you need to:
    1. decide on the windowing. each platform has a different convention and if you're going to put them into the wrf preprocessing you'll need to have the windowing match. use the `shell_scripts/windowing.csh` script.
    2. the wrf preprocessing takes a list of files and assumes they will all be assimilated at the same time, for superob'ing purposes, so it should match the expected assimilation window when running filter.

6.59.2 Data sources

http://madis.noaa.gov

There are two satellite wind converter programs; the one in this directory and one in the *SSEC Data Center* directory. The observations distributed here come from NESDIS. The SSEC observations are processed by SSEC itself and will differ from the observations converted here.

6.59.3 Programs

The programs in the `DART/observations/MADIS/` directory extract data from the distribution files and create DART observation sequence (obs_seq) files. Build them in the `work` directory by running the `./quickbuild.sh` script. In addition to the converters, the `advance_time` and `obs_sequence_tool` utilities will be built.

There are currently converters for these data types:

ACARS aircraft T,U,V,Q data	convert_madis_acars
Marine surface data	convert_madis_marine
Mesonet surface data	convert_madis_mesonet
Metar data	convert_madis_metar
Wind Profiler data	convert_madis_profiler
Rawinsonde/Radiosonde data	convert_madis_rawin
Satellite Wind data	convert_madis_satwnd

Example data files are in the `data` directory. Example scripts for converting batches of these files are in the `shell_scripts` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the scripts saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

Several converters have compile-time choices for outputting various types of moist variables. Check the source code for more details. Some converters also read multiple T/F strings from the console (standard input) to control at run-time what types of observations to convert. Again, check the source code for more details.

Each converter has hard-coded input and output filenames:

convert_madis_acars:	acars_input.nc	obs_seq.acars
convert_madis_marine:	marine_input.nc	obs_seq.marine
convert_madis_mesonet:	mesonet_input.nc	obs_seq.mesonet
convert_madis_metar:	metar_input.nc	obs_seq.metar
convert_madis_profiler:	profiler_input.nc	obs_seq.profiler
convert_madis_rawin:	rawin_input.nc	obs_seq.rawin
convert_madis_satwnd:	satwnd_input.nc	obs_seq.satwnd

The expected usage pattern is that a script will copy, rename, or make a symbolic link from the actual input file (which often contains a timestamp in the name) to the fixed input name before conversion, and move the output file to an appropriate filename before the next invocation of the converter. If an existing observation sequence file of the same output name is found when the converter is run again, it will open that file and append the next set of observations to it.

6.60 PROGRAM `MIDAS_to_obs`

6.60.1 Overview

MIDAS netCDF file to DART observation converter

Alex Chartier (University of Bath, UK) contributed the code.

> "MIDAS runs in Matlab. The raw observations come from GPS receivers as RINEX files, but we can't use them directly just yet … Currently, the 'slant' (satellite-to-receiver path) observations are inverted by MIDAS to make vertical, column-integrated 'observations' of plasma density."

6.60.2 Data sources

The original files have been converted to netCDF files that are then converted to DART observation sequence files. The netCDF files have a pretty simple format:

```
netcdf Test {
dimensions:
        latitude = 5 ;
        longitude = 6 ;
        height = 30 ;
        time = UNLIMITED ; // (1 currently)
variables:
        double latitude(latitude) ;
                latitude:units = "degrees_north" ;
                latitude:long_name = "latitude" ;
                latitude:standard_name = "latitude" ;
        double longitude(longitude) ;
                longitude:units = "degrees_east" ;
                longitude:long_name = "longitude" ;
                longitude:standard_name = "longitude" ;
        double height(height) ;
                height:units = "metres" ;
                height:long_name = "height" ;
                height:standard_name = "height" ;
        double time(time) ;
```

(continues on next page)

(continued from previous page)

```
                time:units = "Days since 1601-01-01" ;
                time:long_name = "Time (UT)" ;
                time:standard_name = "Time" ;
        double Ne(height, latitude, longitude) ;
                Ne:grid_mapping = "standard" ;
                Ne:units = "1E11 e/m^3" ;
                Ne:long_name = "electron density" ;
                Ne:coordinates = "latitude longitude" ;
        double TEC(time, latitude, longitude) ;
                TEC:grid_mapping = "standard" ;
                TEC:units = "1E16 e/m^2" ;
                TEC:long_name = "total electron content" ;
                TEC:coordinates = "latitude longitude" ;
        double Variance(time, latitude, longitude) ;
                Variance:grid_mapping = "standard" ;
                Variance:units = "1E16 e/m^2" ;
                Variance:long_name = "Variance of total electron content" ;
                Variance:coordinates = "latitude longitude" ;
                Variance:standard_name = "TEC variance" ;
// global attributes:
                :Conventions = "CF-1.5" ;
}
```

6.60.3 Programs

The `MIDAS_to_obs.f90` file is the source code for the main converter program.

To compile and test, go into the `MIDAS/work` subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. The *program obs_sequence_tool* manipulates (i.e. combines, subsets) DART observation files once they have been created. The default observations supported are those defined in observations/forward_operators/obs_def_upper_atm_mod.f90. If you need additional observation types, you will have to add the appropriate `obs_def_XXX_mod.f90` file to the `input.nml` `&preprocess_nml:input_files` variable and run `quickbuild.sh` again. It rebuilds the table of supported observation types before compiling the source code.

6.60.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&MIDAS_to_obs_nml
   input_file    = 'infile.nc'
   obs_out_file  = 'obs_seq.out',
   verbose       = .false.
   /
```

Item	Type	Description
input_file	character(len=256)	Name of the input netCDF MIDAS file to read.
obs_out_file	character(len=256)	Name of the output observation sequence file that is created.
verbose	logical	Controls how much informational output is printed during a conversion. `.true.` the most amount of output. `.false.` the least amount of output.

Example

```
&MIDAS_to_obs_nml
   input_file    = '../data/Test.nc',
   obs_out_file  = 'obs_seq.out',
   verbose       = .TRUE.,
```

6.60.5 References

6.61 DART observations and MODIS products.

There are many MODIS products, in many formats. This document will list all of the data products and formats that have DART programs to convert them to observation sequence files.

6.61.1 Programs

PROGRAM MOD15A2_to_obs	Converts MODIS Land Product Subsets Leaf Area Index (**LAI**) and Fraction of Photosynthetically Active Radiation (**FPAR**) 8 day composite [MOD15A2]

6.61.2 Plans

1. Support MOD15A2 'Global Tool' records.
2. The work that remains is to get the IGBP landcover code for the site and incorporate that into the observation metadata. I *almost* have everything I need. Once that happens, the forward observation operator can be made to be much more accurate by only using model landunits that have the right landcover class.
3. Support more products. Put in a request to help me prioritize.

6.62 PROGRAM `MOD15A2_to_obs`

6.62.1 MODIS land product subsets (collection 5) to DART observation sequence converter

Overview

This routine is designed to convert the MODIS Land Product Subsets data of Leaf Area Index (**LAI**) and Fraction of Photosynthetically Active Radiation (**FPAR**) 8 day composite [MOD15A2] to a DART observation sequence file. According to the MODIS LAI/FPAR Product User's Guide:

> Leaf area index (LAI; dimensionless) is defined as the one-sided green leaf area per unit ground area in broadleaf canopies and as one-half the total needle surface area per unit ground area in coniferous canopies. Fraction of Photosynthetically Active Radiation absorbed by vegetation (FPAR; dimensionless) is defined as the fraction of incident photosynthetically active radiation (400-700 nm) absorbed by the green elements of a vegetation canopy.

Specifically, the composites are comma-separated-values (.csv format) ASCII files where each line is a record. The input `.csv` files are directly from the Oak Ridge National Laboratory DAAC. There are two streams to download the data formats we support, they differ only in the very first line of the file. One of the formats has a header record, the other does not. Other than that, the file formats are identical. The format with the header record is fully described in https://lpdaac.usgs.gov/dataset_discovery/modis. Please remember to cite the data in your publications, specific instructions from LP DAAC are available here. This is an example:

> Data Citation: Oak Ridge National Laboratory Distributed Active Archive Center (ORNL DAAC). 2012. MODIS subsetted land products, Collection 5. Available on-line [http://daac.ornl.gov/MODIS/modis.html] from ORNL DAAC, Oak Ridge, Tennessee, U.S.A. Accessed *Month dd, yyyy*.

For more information on *downloading* the data, see DATA SOURCES below. The MODIS Land Product Subsets page indicates that the Collection 5 MODIS Subsets are available three ways:

1. Field Site and Flux tower. Since the files are preprocessed, the download is immediate. The current state of the converter supports this format.
2. Global Tool. This requires exact knowledge of the location(s) of interest. Because some processing to fulfill the request is needed, a job is scheduled on the DAAC server and an email notification is sent with instuctions on how to retrieve the file(s) of interest. The converter **does not** currently support this format, but will soon. Worst case scenario is that you make your own header file and add your 'site' to the metadata file described below.
3. Web Service. I have not used the Web Service.

The DART workflow is usually:

1. download the MOD15A2 data for the sites and years in question (see DATA SOURCES below)
2. build the DART executables with support for `MODIS_LEAF_AREA_INDEX` and `MODIS_FPAR` observations. This is done by running `preprocess` with `obs_def_land_mod.f90` in the list of `input_files` for `preprocess_nml` and then building `MOD15A2_to_obs` in the usual DART way.
3. provide basic information via the `input.nml`:`MOD15A2_to_obs_nml` namelist
4. convert each MODIS data file individually using `MOD15A2_to_obs`
5. combine all output files for the region and timeframe of interest into one file using *program obs_sequence_tool*

For some models (CLM, for example), it is required to reorganize the observation sequence files into a series of files that contains ONLY the observations for each assimilation. This can be achieved with the `DART/observations/obs_converters/MODIS/shell_scripts/makedaily.sh` script.

6.62.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&MOD15A2_to_obs_nml
   text_input_file = 'MOD15A2.fn_usbouldr.txt',
   metadata_file   = 'MOD15A2_site_metadata.txt',
   obs_out_file    = 'obs_seq.out',
   maxgoodqc       = 10,
   verbose         = .false.
   /
```

Contents	Type	Description
text_input_file	character(len=256)	Name of the MODIS file of comma-separated values. This may be a relative or absolute filename.
metadata_file	character(len=256)	Name of the file that contains the location information for the specific sites. This may be a relative or absolute filename. If this file does not exist, it is **presumed** that the location information is part of the 'site' column. If this is not true, the program will fail. For more information see the section Presumed Format
obs_out_file	character(len=128)	Name of the output observation sequence file.
maxgoodqc	real	maximum value of any observation quality control flag to pass through to the output observation sequence. Keep in mind that `filter` has the ability to discriminate on the value, so there is really little to be gained by rejecting them during the conversion. The QC value is passed through in its native value, i.e. it is not converted to play nicely with observations that have values 0,1,2,3,4,5 etc.
verbose	logical	Print extra information during the `MOD15A2_to_obs` execution.

6.62.3 Data sources

Field site and flux tower

The download site for the 'Field Site and Flux tower' data is http://daac.ornl.gov/cgi-bin/MODIS/GR_col5_1/mod_viz.html. Since the files are preprocessed, the download is immediate. This method results in files **with** the header record, and requires a small amount of additional work:

- Download the metadata file containing the locations for the Field Sites ftp://daac.ornl.gov/data/modis_ascii_subsets/5_MODIS_SUBSETS_C5_&_FLUXNET.csv
- I usually convert this to UNIX format with the UNIX utility `dos2unix` and rename it to `MOD15A2_site_metadata.txt`

The data files have names like `MOD15A2.fn_uswiirpi.txt` or `MOD15A2.fn_dehambur.txt` and have very long lines. The first line (i.e. record) of the file is a comma-separated list explaining the file format for all the remaining lines/records.

These files contain records with 49 pixel values where each pixel represents the values for a 1km by 1km voxel. The center pixel is the only value converted to a DART observation value.

```
MODIS_LAI % head -1 MOD15A2.fn_dehambur.txt
HDFname,Product,Date,Site,ProcessDate,Band,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
↪19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
↪48,49
```

The format of the `Site` in these files is the predominant difference between the files from the download methods. The `Site` fields in these files have specified site names that must have a case-sensitive match to a site in the metadata file specified by `input.nml:metadata_file` .

Global tool

This format is not supported yet.

The download site for the 'Global Tool' data is http://daac.ornl.gov/cgi-bin/MODIS/GLBVIZ_1_Glb/modis_subset_order_global_col5.pl. Because some processing to fulfill the request is needed, a job is scheduled on the DAAC server and an email notification is sent with instuctions on how to retrieve the file(s) of interest. **This method requires exact knowledge of the location(s) of interest.** `MOD15A2_to_obs` presumes prior knowledge of the file format and that the latitude and longitude are coded in the site name (which is the default behavior). **Do not change the format of the file.** Please follow the download instructions below - **exactly.** These instructions were accurate as of 11 April 2014.

1. go to the DAAC download site for MODIS global data.
2. Select either
    1. "Country" (it helps to FIRST clear out the values from the "lat/lon" boxes)
    2. or a specific latitude and longitude. Be precise. This will specify the center pixel location.
3. click "Continue"
4. Select the "[MOD15A2] Leaf Area Index (LAI) and Fraction of Photsyntetically Active Radiation (FPAR) 8 Day Composite" from the pull-down menu.
5. **Important:** Specify 3 **and only 3** kilometers to encompass the center location. This results in the 7 km by 7 km resolution required by `MOD15A2_to_obs`.
6. click "Continue"
7. select the Starting Date and Ending Date from the list. You can convert the entire dataset into one long DART observation sequence file and then subset it later if need be.
8. **Important:** Make sure you check the button "Generate GeoTIFF and Reproject to Geographic Lat/long"
9. Supply your REAL email address
10. click "Continue"
11. Review the confirmation page. Make sure the requested resolution and area is correct. You should see something like "The Requested Data Area is Approximately 7 Kilometers Wide and 7 Kilometers High"
12. click "Continue"
13. At some point later (perhaps even days), you will get an email with the subject "ORNL DAAC MODIS MOD15A2 order", follow the instructions to complete the download.

The resulting ASCII files will have the same format as described below. The 'site name' column for these files is of the form: `Lat47.61666667Lon12.58333333Samp7Line7` which provides the location information otherwise provided by the `MOD15A2_site_metadata.txt` file for the predefined sites.

Web service

I have not used the Web Service.

6.62.4 Format

The data product "Leaf Area Index - Fraction of Photosynthetically Active Radiation 8-Day L4 Global 1km" (**MOD15A2**) is described in https://lpdaac.usgs.gov/products/modis_products_table/mod15a2 (**expand the 'Layers' tab**). The units and the QC values are described there. What I have not been able to determine is how to interpret the 'Date' … if it is 2000049 … It is day 49 of year 2000. Is that the start of the 8 day composite, the middle, the end? If you know the answer, please let me know.

Taken (almost) directly from https://lpdaac.usgs.gov/tools/lp_daac_web_services and modified only slightly with examples more appropriate for the LAI/FPAR product.

The MODIS MOD15A2 products in question are ASCII files of comma-separated values. If the file contains a header record/line, all columns are interpreted based on this header column. If the file does not contain a header, the following format is REQUIRED.

- ASCII values are comma delimited
- Row 1 is the header row (which may not exist for products generated by the Global Tool)
- Data values start in row 2 if the header row is present.
- Rows of QC data are interleaved with measurement data as indicated in Column 6.
- Note that values may contain embedded periods, dashes, and underscores (".,-, _").

Column	Column Description	Example Values
1	Unique row identifier	MOD15A2.A2000049.fn_ruyakuts.005.2006268205917.Fpar_1km MOD15A2.A2000049.fn_ruyakuts.005.2006268205917.Lai_1km
2	MODIS Land Product Code	MOD15A2
3	MODIS Acquisition Date A(YYYYDDD)	A2000049 (?this is an 8 day average) What does 49 indicate? start? middle? end?
4	SiteID Each site is assigned a unique ID. To get the Site name information from SiteID, click here	fn_ustnwalk, Lat47.61666667Lon12.58333333Samp7Line7
5	MODIS Processing Date (YYYYDDDHHMMSS)	2006269073558
6	Product Scientific Data Set (Band): Indicates type of values to follow. Specific values vary by Product. Data quality information are interleaved.	MOD15A2: FparExtra_QC, FparLai_QC, FparStdDev_1km, Fpar_1km, LaiStdDev_1km, Lai_1km
7 to N	Data values of type as specified. Number of data columns as given in Column 4. Definition of QC component values vary by Scientific Data Set	QC: 00100001,01100001,01100001, … Measurement: 2,2,1,1,1,1,1,0,0,0,1,1,0,0, to N

QC flags are binary-coded ascii strings e.g., 10011101 bits 5,6,7 (the last three) are decoded as follows:

- 000 … Main(RT) method used, best result possible (no saturation)
- 001 … Main(RT) method used with saturation, Good, very usable

- 010 … Main(RT) method failed due to bad geometry, empirical algorithm used
- 011 … Main(RT) method failed due to other problems
- 100 … pixel not produced at all

Consequently, the last three digits are used by DART's data processing logic.

6.62.5 Programs

The `MOD15A2_to_obs.f90` file is the source for the main converter program. Look at the source code where it reads the example data file. You will almost certainly need to change the "read" statement to match your data format. The example code reads each text line into a character buffer and then reads from that buffer to parse up the data items.
FIXME Explain the 10% for the obs error for FPAR and question the LAIStddev …

To compile and test, go into the work subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

To change the observation types, look in the `DART/obs_def` directory. If you can find an obs_def_XXX_mod.f90 file with an appropriate set of observation types, change the 'use' lines in the converter source to include those types. Then add that filename in the `input.nml` namelist file to the &preprocess_nml namelist, the 'input_files' variable. Multiple files can be listed. Then run quickbuild.sh again. It remakes the table of supported observation types before trying to recompile the source code.

An example script for converting batches of files is in the `shell_scripts` directory. A tiny example data file is in the `data` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the script saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

6.62.6 Decisions you might need to make

See the general discussion in the *Creating an obs_seq file from real observations* page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.62.7 Future plans

- Support for the data records without the header, as created by the Global Tool.
- The work that remains is to get the IGBP landcover code for the site and incorporate that into the observation metadata. I *almost* have everything I need. Once that happens, the forward observation operator can be made to be much more accurate by only using model landunits that have the right landcover class.

6.63 PROGRAM MOD15A2_to_obs

MODIS land product subsets (collection 5) to DART observation sequence converter

6.64 MPD

The Micro Pulse Differential Absorption Lidar (MPD) data were collected during field campaigns and testing periods by the Earth Observing Laboratory (EOL).

The differential absorption lidar (DIAL) technique uses two separate laser wavelengths: an absorbing wavelength (on-line) and a non-absorbing wavelength (offline). The ratio of the range-resolved backscattered signals between the online and offline wavelengths is proportional to the amount of water vapor in the atmosphere, which allows the retrieval of absolute humidity profiles above the lidar site.

This observation converter takes absolute humidity (g/m3) profiles retrieved from the MPD data and converts them to the format used by DART. The `obs_converter/MPD/work/convert_to_text.py` script reads the netCDF files from each MPD site and combines them into text files, one for each date and time. The `obs_converter/MPD/work/MPD_text_to_obs` program translates the text files to the DART `obs_seq.out` format.

Test data for a single site and an example output can be downloaded from https://www.image.ucar.edu/pub/DART/MPD/MPD.tar.gz

For more details of the retrieval and quality control process, and inquire about data availability for your research project, please contact Tammy Weckwerth at EOL, NCAR.

6.65 PROGRAMS LPRM_L3_to_obs.f90 AMSR_E_L2_to_obs.f90

This is a brief description of the converters and utilities in this directory retrieved from the NASA Earthdata portal This is a front end for many (Distributed Active Archive Center) DAAC portals and for the Goddard Earth Sciences Data and Information Services Center

These directories contain satellite retrieval data for land surface soil moisture and leaf area index (LAI).

The general workflow for each of the observation converters (described in more detail below) is usually:

1. Download the data for the period in question (see DATA SOURCES below)
2. Build the DART executables with support for the soil moisture observations. This is done by running preprocess with `obs_def_land_mod.f90` in the list of input_files for `preprocess_nml`.
3. Convert each data file individually (e.g. executing `LPRM_L3_to_obs`)
4. Combine or subset all output files for the region and timeframe of interest into one file using *program obs_sequence_tool*

For some models (CLM, for example), it is required to reorganize the observation sequence files into a series of files that contains ONLY the observations for each assimilation. This can be achieved with the *~/models/clm/shell_scripts/makedaily.sh* script. Since there are subtleties for each model, makedaily.sh is generally found in the shell_scripts directory of the model.

6.65.1 Soil Moisture Observation Converters

Program `AMSR_E_L2_to_obs.f90`

Description

AMSR-E/Aqua surface soil moisture (LPRM) L2B V002 is a Level 2 (swath) data set. Its land surface parameters, surface soil moisture, land surface (skin) temperature, and vegetation water content are derived from passive microwave remote sensing data from the Advanced Microwave Scanning Radiometer-Earth Observing System (AMSR-E), using the Land Parameter Retrieval Model (LPRM). Each swath is packaged with associated geolocation fields. The data set covers the period from June 2002 to October 2011 (when the AMSR-E on the NASA EOS Aqua satellite stopped producing data due to a problem with the rotation of its antenna), at the spatial resolution (nominally 56 and 38 km, respectively) of AMSR-E's C and X bands (6.9 and 10.7 GHz, respectively).

NAMELIST

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&AMSRE_L2_to_obs_nml
   input_file          = 'LPRM-AMSR_E_L2_D_SOILM2_V002_20030630025503.nc'
   obs_out_file        = 'obs_seq.out'
   max_rfi_code        = 2
   amsre_rep_error     = 2
   /
```

Description of namelist variables:

Contents	Type	Description
input_file	character(len=256)	Name of the netcdf soil moisture data file.
obs_out_file	character(len=256)	Name of the output observation sequence file.
max_rfi_code	integer	Maximum Radio Frequency Interference. Soil moisture values with a max rfi above this threshold are excluded from obs_out_file
amsre_rep_error	integer	Representativeness Error (standard deviation units). This value is added to the instrument error provided from the input file to calculate the total observation error variance

Important: The total error (instrument error + representativeness error) is used to calculate the observation error variance in the `obs_out_file`. The instrument error is taken from the data file variable (sm_c_error or sm_x_error), whereas the represenativeness error (amsre_rep_error) is a namelist variable input by the user at runtime.

Data Source

The dataset (LPRM_AMSRE_SOILM2: AMSR-E/Aqua surface soil moisture (LPRM) L2B V002) can be found here.

Program `LPRM_L3_to_obs.f90`

Description

TMI/TRMM surface soil moisture (LPRM) L3 1 day 25 km x 25 km nighttime V001 is Level 3 (gridded) data set. Its land surface parameters, surface soil moisture, land surface (skin) temperature, and vegetation water content, are derived from passive microwave remote sensing data from the Tropical Rainfall Measuring Mission (TRMM) Microwave Imager (TMI), using the Land Parameter Retrieval Model (LPRM). There are two files per day, one daytime and one nighttime, archived as two different products. This document is for the nighttime product. The data set covers the period from December 1997 to April 2015 (when the instruments on the TRMM satellite were shut down in preparation for its reentry into the earth's atmosphere).

The LPRM is based on a forward radiative transfer model to retrieve surface soil moisture and vegetation optical depth. The land surface temperature is derived separately from TMI's Ka-band (37 GHz). A unique feature of this method is that it can be applied at any microwave frequency, making it very suitable to exploit all the available passive microwave data from various satellites.

NAMELIST

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&LPRM_L3_to_obs_nml
   input_file        = 'LPRM-TMI_L3_NT_SOILM3_V001-20120411T144345Z_20120407.nc'
   output_file       = 'obs_seq.out'
   lon_bounds        = 0.0, 360.0
   lat_bounds        = -90.0,  90.0
   /
```

Description of namelist variables:

Contents	Type	Description
input_file	character(len=256)	Name of the netcdf soil moisture data file.
output_file	character(len=256)	Name of the output observation sequence file.
lon_bounds	real(r8)	Longitude bounds. Observations outside these bounds are excluded from the output_file
lat_bounds	real(r8)	Latitude bounds. Observations outside these bounds are excluded from the output_file

Important: The total error (instrument error + representativeness error) is used to calculate the observation error variance in the `output_file`. The instrument error is taken from the data file variable (sm_x_error), whereas the representativeness error is set to 0.1 within the `LPRM_L3_to_obs`.

Data Source

The dataset (LPRM_TMI_NT_SOILM3: TMI/TRMM surface soil moisture (LPRM) L3 1 day 25km x 25km nighttime V001) can be found here.

6.65.2 Leaf Area Index Observation Converter

Program `netCDF_to_obs.f90`

Description

This dataset provides a global 0.25 degree x 0.25 degree gridded monthly mean leaf area index (LAI) climatology as averaged over the period from August 1981 to August 2015. The data were derived from the Advanced Very High Resolution Radiometer (AVHRR) Global Inventory Modeling and Mapping Studies (GIMMS) LAI3g version 2, a bi-weekly data product from 1981 to 2015 (GIMMS-LAI3g version 2). The LAI3g version 2 (raw) data were first regridded from 1/12 x 1/12 degree to 0.25 x 0.25 degree resolution, then processed to remove missing and unreasonable values, scaled to obtain LAI values, and the bi-weekly LAI values were averaged for every month. Finally, the monthly long-term mean LAI (1981-2015) was calculated.

The Global Monthly Mean Leaf Area Index Climatology, (1981-2015) dataset may be converted with the `netCDF_to_obs` program. Since these are monthly means, each timestep is read and output as their own observation sequence file that has the date and time appended to the filename.

NAMELIST

```
&netCDF_to_obs_nml
   input_file        = 'LAI_mean_monthly_1981-2015.nc4'
   output_file_base  = 'obs_seq.out'
   lon_bounds        = 0.0, 360.0
   lat_bounds        = -90.0,  90.0
   debug               = .FALSE.
   observation_varname          = 'LAI'
   observation_type             = 'GIMMS_LEAF_AREA_INDEX'
   obs_error_standard_deviation = 0.2
   /
```

Description of namelist variables:

Contents	Type	Description
input_file	character(len=256)	Name of the netcdf LAI data file.
output_file_base	character(len=256)	Name of the output observation sequence file.
lon_bounds	real(r8)	Longitude bounds. Observations outside these bounds are excluded from the output file
lat_bounds	real(r8)	Latitude bounds. Observations outside these bounds are excluded from the output file
debug	logical	If .TRUE. prints out extra information on data file characteristics
observation_varname	character(len=256)	Name of of the leaf area variable within the netcdf data file
observation_type	character(len=256)	Name of the DART observation type
obs_error_standard_deviation	character(len=256)	The observation error standard deviation (not provided within data file)

Data Source

The Global Monthly Mean Leaf Area Index Climatology, (1981-2015) data can be found here.

6.66 PROGRAM `prepbufr`

6.66.1 Overview

Translating NCEP PREPBUFR files into DART obs_seq.out files (input file to filter) is a 2 stage process. The first stage uses NCEP software to translate the PREPBUFR file into an intermediate text file. This is described in this document. The second step is to translate the intermediate files into obs_seq.out files, which is done by create_real_obs, as described in *PROGRAM create_real_obs* .

6.66.2 Instructions

The prep_bufr package is free-standing and has not been completely assimilated into the DART architecture. It also requires adaptation of the sources codes and scripts to the computing environment where it will be run. It is not so robust that it can be controlled just with input parameters. It may not have the same levels of error detection and warning that the rest of DART has, so the user should very careful about checking the end product for correctness.

Overview of what needs to be built and run

More detailed instructions follow, but this section describes a quick overview of what programs you will be building and running.

Building

Running the install.sh script will build the library and main executable. You will probably have to edit this script to set which fortran compiler is available on your system.

If you have raw unblocked PREPBUFR files you will need to convert them to blocked format (what prepbufr expects as input). The blk/ublk section of the build script compiles the `cword.x` converter program.

If you are running on an Intel (little-endian) based machine you will need the `grabbufr` byte swapping program that is also built by this script.

One-shot execution

If you are converting a single obs file, or are walking through the process by hand for the first time, you can follow the more detailed build instructions below, and then run the prep_bufr.x program by hand. This involves the following steps:

- building the executables.
- running the blocker if needed (generally not if you have downloaded the blocked format PREPBUFR files).
- running the binary format converter if you are on an Intel (little-endian) machine.
- linking the input file to a fixed input filename
- running prepbufr.x to convert the file
- copying the fixed output filename to the desired output filename

Production mode

If you have multiple days (or months) of observations that you are intending to convert, there is a script in the work subdirectory which is set up to run the converter on a sequence of raw data files, and concatenate the output files together into one output file per day. Edit the work/prepbufr.csh script and set the necessary values in the 'USER SET PARAMETERS' section near the top. This script can either be run from the command line, or it can be submitted to a batch queue for a long series of conversion runs.

Installation of the ncep prepbufr decoding program

This package is currently organized into files under the DART/observations/NCEP/prep_bufr directory:

```
src           Source code of the NCEP PREPBUFR decoder
lib           NCEP BUFR library source
install.sh    A script to install the NCEP PREPBUFR decoder and the NCEP BUFR library.
exe           Executables of the decoder and converter.
data          Where the NCEP PREPBUFR files (prepqm****) could be loaded into
              from the NCAR Mass Store (the script assumes this is the default location).
work          Where we run the script to do the decoding.
convert_bufr  Source code (grabbufr) to convert the binary big-endian PREPBUFR files to
              little-endian files, and a script to compile the program.
blk_ublk      Source code (cwordsh) to convert between blocked and unblocked format.
docs          Some background information about NCEP PREPBUFR observations.
```

The decoding program: src/prepbufr.f

The program prepbufr.f is used to decode the NCEP reanalysis PREPBUFR data into intermediate text files. This program was originally developed by NCEP. It has been modified to output surface pressure, dry temperature, specific humidity, and wind components (U/V) of conventional radiosonde, aircraft reports, and satellite cloud motion derived wind. There are additional observation types on the PREPBUFR files, but using them they would require significant modifications of prepbufr and require detailed knowledge of the NCEP PREPBUFR files. The NCEP quality control indexes for these observations based on NCEP forecasts are also output and used in DART observation sequence files. The NCEP PREPBUFR decoding program is written in Fortran 77 and has been successfully compiled on Linux computers using pgi90, SGI® computers with f77, IBM® SP® systems with xlf, and Intel® based Mac® with gfortran.

If your operating system uses modules you may need to remove the default compiler and add the one desired for this package. For example

- which pgf90 (to see if pgf90 is available.)
- module rm intel64 netcdf64 mpich64
- module add pgi32

To compile the BUFR libraries and the decoding program, set the CPLAT variable in the install.sh script to match the compilers available on your system. CPLAT = linux is the default. Execute the install.sh script to complete the compilations for the main decoding program, the NCEP BUFR library, and the conversion utilities.

The executables (i.e., prepbufr.x, prepbufr_03Z.x) are placed in the ../exe directory.

Platforms tested:

- Linux clusters with Intel, PGI, Pathscale, GNU Fortran,
- Mac OS X with Intel, GNU Fortran,
- SGI Altix with Intel
- Cray with Intel, Cray Fortran.

The byte-swapping program convert_bufr/grabbufr.f

For platforms with little-endian binary file format (e.g. Intel, AMD®, and non-MIPS SGI processors) the program grabbufr.f is used to convert the big-endian format NCEP PREPBUFR data into little-endian format. The grabbufr.f code is written in Fortran 90, and has been compiled can be compiled with the pgf90 compiler on a Linux system, with gfortran on an Intel based Mac, and the ifort compiler on other Linux machines. More detailed instructions for building it can be found in convert_bufr/README, but the base install script should build this by default. In case of problems, cd into the convert_bufr subdirectory, edit convert_bufr.csh to set your compiler, and run it to compile the converter code (grabbufr).

This program reads the whole PREPBUFR file into memory, and needs to know the size of the file (in bytes). Unfortunately, the system call STAT() returns this size as one number in an array, and the index into that array differs depending on the system and sometimes the word size (32 vs 64) of the compiler. To test that the program is using the right offset into this array, you can compile and run the stat_test.f program. It takes a single filename argument and prints out information about that file. One of the numbers will be the file size in bytes. Compare this to the size you see with the 'ls -l' command for that same file. If the numbers do not agree, find the right index and edit the grabbufr.f source file. Look for the INDEXVAL line near the first section of executable code.

If grabbufr.f does not compile because the getarg() or iargc() subroutines are not found or not available, then either use the arg_test.f program to debug how to get command line arguments into a fortran program on your system, or simply go into the grabbufr.f source and comment out the section which tries to parse command line arguments and comment in the hardcoded input and output filenames. Now to run this program you must either rename the data files to these predetermined filenames, or you can use links to temporarily give the files the names needed.

The blocking program blk_ublk/cword.x

The prepbufr.x program expects to read a blocked input file, which is generally what is available for download. However, if you have an unblocked file that you need to convert, there is a conversion program. The install.sh script will try to build this by default, but in case of problems you can build it separately. Change directories into the blk_ublk subdirectory and read the README_cwordsh file for more help. The cwordsh shell-script wrapper shows how to run the executable cwordsh.x executable.

Note that if you can get the blocked file formats to begin with, this program is not needed.

Getting the ncep reanalysis prepbufr format data from ncar hpss

The NCEP PREPBUFR files (prepqmYYMMDDHH) can be found within the NCEP reanalysis dataset, ds090.0, on NCAR Mass Store System (HPSS).

To find the files:

- go to the NCAR/NCEP reanalysis archive.
- Click on the "Inventories" tab.
- Select the year you are interested in.
- Search for files with the string "prepqm" in the name.
- Depending on the year the format of the filenames change, but they should contain the year, usually as 2 digits, the month, and then either the start/stop day for weekly files, or the letters A and B for semi-monthly files.

Depending on the year you select, the prepqm files can be weekly, monthly, or semi-monthly. Each tar file has a unique dataset number of the form "A#####". For example, for January of 2003, the 4 HPSS TAR files are: A21899, A21900, A21901, A21902. After September 2003, these files include AIRCRAFT data (airplane readings taken at cruising elevation) but not ACARS data (airplane readings taken during takeoff and landing). There are different datasets which include ACARS data but their use is restricted and you must contact the RDA group to get access.

If you are running on a machine with direct access to the NCAR HPSS, then change directories into the prep_bufr/data subdirectory and run:
> *hsi get /DSS/A##### rawfile*
where ##### is the data set number you want.

These files may be readable tar files, or they may require running the `cosconvert` program first. See if the `tar` command can read them:
> *tar -tvf rawfile*
If you get a good table of contents then simply rename the file and untar it:
> *mv rawfile data.tar*
> *tar -xvf data.tar*
However, if you get an error from the tar command you will need to run the `cosconvert` program to convert the file into a readable tar file. On the NCAR machine *yellowstone*, run:
> */glade/u/home/rdadata/bin/cosconvert -b rawfile data.tar*
On other platforms, download the appropriate version from: http://rda.ucar.edu/libraries/io/cos_blocking/utils/ . Build and run the converter and then you should have a tar file you can unpack.

The output of tar should yield individual 6-hourly NCEP PREPBUFR data files for the observations in the +/- 3-hour time windows of 00Z, 06Z, 12Z, and 18Z of each day. Note that DART obs_seq files are organized such that a 24 hour

file with 4 observation times would contain observations from 3:01Z to 3:00Z of the next day, centered on 6Z, 12Z, 18Z and "24Z". In addition, there are some observations at 3:00Z on the PREPBUFR file labelled with 06Z. Then, in order to make a full day intermediate file incorporating all the required obs from the "next" day, you'll need the PREPBUFR files through 6Z of the day after the last day of interest. For example, to generate the observation sequence for Jan 1, 2003, the decoded NCEP PREPBUFR text files for Jan 1 and 2, 2003 are needed, and hence the PREPBUFR files

- prepqm03010106
- prepqm03010112
- prepqm03010118
- prepqm03010200
- prepqm03010206

are needed.

Running the ncep prepbufr decoding program

In prep_bufr/work/prepbufr.csh set the appropriate values of the year, month, first day, and last day of the period you desire, and the variable "convert" to control conversion from big- to little-endian. Confirm that the raw PREPBUFR files are in ../data, or that prepbufr.csh has been changed to find them. Execute prepbufr.csh in the work directory. It has code for running in the LSF batch environment, but not PBS.

Currently, this script generates decoded PREPBUFR text data each 24 hours which contains the observations within the time window of -3:01 hours to +3:00Z within each six-hour synoptic time. These daily output text files are named as temp_obs.yyyymmdd. These text PREPBUFR data files can then be read by DART/observations/NCEP/ascii_to_obs/work/*PROGRAM create_real_obs* to generate the DART daily observation sequence files.

There is an alternate section in the script which creates a decoded PREPBUFR text data file each 6 hours (so they are 1-for-1 with the original PREPBUFR files). Edit the script prepbufr.csh and look for the commented out code which outputs 4 individual files per day. Note that if you chose this option, you will have to make corresponding changes in the create_obs_seq.csh script in step 2.

6.66.3 Other modules used

This is a piece of code that is intended to be 'close' to the original, as such, we have not modified it to use the DART build mechanism. This code does not use any DART modules.

6.66.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&prep_bufr_nml
   obs_window       = 1.5,
   obs_window_upa   = 1.5,
   obs_window_air   = 1.5,
   obs_window_sfc   = 0.8,
   obs_window_cw    = 1.5,
   land_temp_error  = 2.5,
   land_wind_error  = 3.5,
```

(continues on next page)

(continued from previous page)

```
   land_moist_error = 0.2,
   otype_use        = missing,
   qctype_use       = missing,
/
```

Item	Type	Description
obs_window	real	Window of time to include observations. If > 0, overrides all the other more specific window sizes. Set to -1.0 to use different time windows for different obs types. The window is +/- this number of hours, so the total window size is twice this value.
obs_window_upa	real	Window of time to include sonde observations (+/- hours) if obs_window is < 0, otherwise ignored.
obs_window_air	real	Window of time to include aircraft observations (+/- hours) if obs_window is < 0, otherwise ignored.
obs_window_sfc	real	Window of time to include surface observations (+/- hours) if obs_window is < 0, otherwise ignored.
obs_window_cw	real	Window of time to include cloud wind observations (+/- hours) if obs_window is < 0, otherwise ignored.
otype_use	real(300)	Report Types to extract from bufr file. If unspecified, all types will be converted.
qctype_use	integer(300)	QC types to include from the bufr file. If unspecified, all QC values will be accepted.
land_temp_error	real	observation error for land surface temperature observations when none is in the input file.
land_wind_error	real	observation error for land surface wind observations when none is in the input file.
land_moist_error	real	observation error for land surface moisture observations when none is in the input file.

6.66.5 Files

- input file(s); NCEP PREPBUFR observation files named using ObsBase with the "yymmddhh" date tag on the end. Input to grabbufr if big- to little-endian is to be done. Input to prepbufr if not.
- intermediate (binary) prepqm.little; output from grabbufr, input to prepbufr.
- intermediate (text) file(s) "temp_obs.yyyymmddhh"; output from prepbufr, input to create_real_obs

6.66.6 References

DART/observations/NCEP/prep_bufr/docs/* (NCEP text files describing the PREPBUFR files)

6.67 PROGRAM create_real_obs

6.67.1 Overview

Translating NCEP BUFR files into DART obs_seq.out files (input file to filter) is a 2 stage process. The first stage uses NCEP software to translate the BUFR file into an "intermediate" text file. This is described in *PROGRAM prepbufr*. The second step is to translate the intermediate files into an `obs_seq.out` files, which is done by `create_real_obs`, as described in this document.

This program provides a number of options to select several observation types (radiosonde, aircraft, and satellite data, etc.) and the DART observation variables (U, V, T, Q, Ps) which are specified in its optional namelist interface `&ncepobs_nml` which may be read from file `input.nml`.

6.67.2 Instructions

- Go to DART/observations/NCEP/ascii_to_obs/work
- Use `quickbuild.sh` to compile all executable programs in the directory. To rebuild just one program:
  - Use `mkmf_create_real_obs` to generate the makefile to compile `create_real_obs.f90`.
  - Type `make` to get the executable.
- Make appropriate changes to the `&ncep_obs_nml` namelist in `input.nml`, as follows.
- run `create_real_obs`.

The selection of any combinations of the specific observation fields (T, Q, U/V, and surface pressure) and types (radiosonde, aircraft reports, or satellite wind, etc.) is made in the namelist `&ncepobs_nml`. All the available combinations of fields X types (i.e. ADPUPA and obs_U) will be written to the obs_seq file. (You will be able to select which of those to use during an assimilation in another namelist (`assimilate_these_obs`, in `&obs_kind_nml`), so be sure to include all the fields and types you might want.) You should change `Obsbase` to the pathname of the decoded PREPBUFR text data files. Be sure that `daily_file` is set to .TRUE. to create a single 24 hour file; .FALSE. converts input files one-for-one with output files. The default action is to tag each observation with the exact time it was taken and is the recommended setting. However, if you want to bin the observations in time, for example to do additional post-processing, the time on all observations in the window can be overwritten and set to the nearest synoptic time (e.g. 0Z, 6Z, 12Z, or 18Z), by setting `obs_time` to false.

Generally you will want to customize the namelist for your own use. For example, here is a sample namelist:

```
&ncepobs_nml
  year = 2007,
  month = 3,
  day = 1,
  tot_days = 31,
  max_num = 700000,
  ObsBase = '../prep_bufr/work/temp_obs.'
  select_obs  = 1,
  ADPUPA = .true.,
  AIRCAR = .false.,
```

(continues on next page)

(continued from previous page)

```
  AIRCFT = .true.,
  SATEMP = .false.,
  SFCSHP = .false.,
  ADPSFC = .false.,
  SATWND = .true.,
  obs_U  = .true.,
  obs_V  = .true.,
  obs_T  = .true.,
  obs_PS = .false.,
  obs_QV = .false.,
  daily_file = .true.
  obs_time = .true.,
/

&obs_sequence_nml
  write_binary_obs_sequence = .false.
/
```

This will produce daily observation sequence files for the period of March 2007, which have the selected observation types and fields; T, U, and V from radiosondes (ADPUPA) and aircraft (AIRCFT). No surface pressure or specific humidity would appear in the obs_seq files, nor observations from ACARS, satellites, and surface stations. The output files look like "obs_seq200703dd", with dd = 1,...,31.

6.67.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&ncepobs_nml
   year       = 2003,
   month      = 1,
   day        = 1,
   tot_days   = 31,
   max_num    = 800000,
   select_obs = 0,
   ObsBase    = 'temp_obs.',
   ADPUPA     = .false.,
   AIRCAR     = .false.,
   AIRCFT     = .false.,
   SATEMP     = .false.,
   SFCSHP     = .false.,
   ADPSFC     = .false.,
   SATWND     = .false.,
   obs_U      = .false.,
   obs_V      = .false.,
   obs_T      = .false.,
   obs_PS     = .false.,
   obs_QV     = .false.,
   daily_file = .true.,
   obs_time   = .true.,
```

(continues on next page)

(continued from previous page)

```
   lon1       =   0.0,
   lon2       = 360.0,
   lat1       = -90.0,
   lat2       =  90.0
/
```

Item	Type	Description
year, month, day	integer	Beginning year, month, day of the observation period.
tot_days	integer	Total days in the observation period. The converter cannot cross month boundaries.
max_num	integer	Maximum observation number for the current one day files.
select_obs	integer	Controls whether to select a subset of observations from the NCEP BUFR decoded daily ascii files. • 0 = All observations are selected. • 1 = Select observations using the logical parameters below.
daily_file	logical	Controls timespan of observations in each obs_seq file: • true = 24 hour spans (3:01Z to 3:00Z of the next day). Filenames have the form obs_seqYYYYMMDD. • false = 6 hour spans (3:01Z to 9:00Z, 9:01Z to 15:00Z, 15:01Z to 21:00Z, and 21:01Z to 3:00Z of the next day. Filenames have the form obs_seqYYYYMMDDHH, where HH is 06, 12, 18, and 24.
ObsBase	character(len=129)	Path that contains the decoded NCEP BUFR daily observation files. To work with the example scripts this should be 'temp_obs.', or if it includes a pathname then it should end with a '/temp_obs.'
include_specific_humidity, include_relative_humidity, include_dewpoint	logical	Controls which moisture observations are created. The default is to create only specific humidity obs, but any, all, or none can be requested. Set to .TRUE. to output that obs type, .FALSE. skips it.
ADPUPA	logical	Select the NCEP type ADPUPA observations which includes land and ship launched radiosondes and pibals as well as a few profile dropsonde. This involves, at 00Z and 12Z, about 650 - 1000 stations, and at 06Z and 18Z (which are mostly pibals), about 150 - 400 stations.
AIRCFT	logical	Select the NCEP type AIRCFT observations, which includes commercial, some military and reconnaissance reports. They are flight level reports.
AIRCAR	logical	Select the NCEP type AIRCAR observations, which includes data from aircraft takeoff and landings. Some-

6.67.4 Modules used

```
types_mod
utilities_mod
obs_utilities_mod
obs_sequence_mod
obs_kind_mod
obs_def_mod
assim_model_mod
model_mod
cov_cutoff_mod
location_mod
random_seq_mod
time_manager_mod
null_mpi_utilities_mod
real_obs_mod
```

6.67.5 Files

- path_names_create_real_obs; the list of modules used in the compilation of create_real_obs.
- temp_obs.yyyymmdd; (input) NCEP BUFR (decoded/intermediate) observation file(s) Each one has 00Z of the next day on it.
- input.nml; the namelist file used by create_real_obs.
- obs_seqYYYYMMDD[HH]; (output) the obs_seq files used by DART.

6.67.6 References

- …/DART/observations/NCEP/prep_bufr/docs/* (NCEP text files describing the BUFR files)

6.68 PROGRAM `SMAP_L2_to_obs`

6.68.1 Overview

Soil Moisture Active Passive (SMAP) passive microwave radiometer

to DART Observation Sequence Converter

This routine is designed to convert the SMAP L2 Radiometer Half-Orbit 36 km EASE-Grid Soil Moisture, Soil moisture product (Versions 4-8) to a DART observation sequence file.

Quoting the NSIDC:

This Level-2 (L2) soil moisture product provides estimates of global land surface conditions retrieved by the Soil Moisture Active Passive (SMAP) passive microwave radiometer during 6:00 a.m. descending and 6:00 p.m. ascending half-orbit passes. SMAP L-band brightness temperatures are used to derive soil moisture data, which are then resampled to an Earth-fixed, global, cylindrical 36 km Equal-Area Scalable Earth Grid, Version 2.0 (EASE-Grid 2.0).

Data Set ID: SPL2SMP SMAP L2 Radiometer Half-Orbit 36 km EASE-Grid Soil Moisture

Surface soil moisture (0-5 cm) in m3/m3 derived from brightness temperatures (TBs) is output on a fixed global 36 km EASE-Grid 2.0. Also included are brightness temperatures in kelvin representing the weighted average of Level-1B brightness temperatures whose boresights fall within a 36 km EASE-Grid 2.0 cell.

Important: `SMAP_L2_to_obs` uses an observation error standard deviation of 0.01 m3/m3 or 20% of the soil moisture value, whatever is higher. **These numbers have no scientific basis and should be thoroughly explored.** The data files I have explored have a variable soil_moisture_error but these appear to be empty - i.e. full of _FillValue values. A better way of specifying the observation error standard deviation is needed.

Important: SMAP_L2_to_obs has only been thoroughly tested with the half-orbit files - these files have names like 'SMAP_L2_SM_P_02526_D_20150723T070211_R12170_001.h5'.

Important: SMAP_L2_to_obs is not compatible with SMAP L3 files as they are formatted differently

---

The workflow is usually:

1. Download the data for the period in question (see DATA SOURCES below)
2. Build the DART executables with support for the soil moisture observations. This is done by running preprocess with `obs_def_land_mod.f90` in the list of input_files for `preprocess_nml`.
3. Convert each data file individually using `SMAP_L2_to_obs`
4. Combine or subset all output files for the region and timeframe of interest into one file using *program obs_sequence_tool*

For some models (CLM, for example), it is required to reorganize the observation sequence files into a series of files that contains ONLY the observations for each assimilation. This can be achieved with the *~/models/clm/shell_scripts/makedaily.sh* script. Since there are subtleties for each model, makedaily.sh is generally found in the shell_scripts directory of the model.

6.68.2 NAMELIST

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&SMAP_L2_to_obs_nml
   input_file_list     = 'file_list.txt'
   obs_out_file        = 'obs_seq.out'
   max_num_input_files = 10
   verbose             = .false.
   /
```

Description of namelist variables:

Contents	Type	Description
input_file_list	character(len=256)	Name of the file containing the list of input data files. Each input data file name must be on a separate line. No blank lines are allowed. This may be a relative or absolute filename.
obs_out_file	character(len=256)	Name of the output observation sequence file.
max_num_input_files	integer	The maximum number of filenames in 'input_file_list' to convert. This should be a small number - converting all frequencies, all polarizations, both passes into one file is not recommended
verbose	logical	Print extra information during the *SMAP_L2_to_obs* execution.

6.68.3 Data Sources

The SMAP L2 Radiometer Half-Orbit 36 km EASE-Grid Soil Moisture, Version 4 data

6.68.4 Data Citation

The following example shows how to cite the use of this data set in a publication. For more information, see the Use and Copyright web page

> O'Neill, P. E., S. Chan, E. G. Njoku, T. Jackson, and R. Bindlish. 2016. SMAP L2 Radiometer Half-Orbit 36 km EASE-Grid Soil Moisture, Version 4. [Indicate subset used]. Boulder, Colorado USA. NASA National Snow and Ice Data Center Distributed Active Archive Center. doi: http://dx.doi.org/10.5067/XPJTJT812XFY. [Date Accessed].

6.68.5 PROGRAMS

The `SMAP_L2_to_obs.f90` file is the source for the converter program. To compile and test, go into the work subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. *PROGRAM advance_time* helps with calendar and time computations, and the *program obs_sequence_tool* manipulates DART observation files once they have been created.

6.68.6 DECISIONS YOU MIGHT NEED TO MAKE

See the discussion in the *Creating an obs_seq file from real observations* introduction page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.68.7 Terms of Use

DART software - Copyright UCAR. This open source software is provided by UCAR, "as is", without charge, subject to all terms of use at http://www.image.ucar.edu/DAReS/DART/DART_download

6.69 ROMS observations to DART observation sequences

6.69.1 Overview

The relationship between ROMS and DART is slightly different than most other models. ROMS has the ability to apply its own forward operator as the model is advancing (a capability needed for variational assimilation) which produces something the ROMS community calls '*verification*' observations. The observation file that is input to ROMS is specified by the `s4dvar.in`:`OBSname` variable. The verification obs are written out to a netcdf file whose name is specified by the `s4dvar.in`:`MODname` variable. Since each ROMS model is advancing independently, a set of verification observation files are created during a DART/ROMS assimilation cycle. This set of files can be converted using `convert_roms_obs` to produce a DART observation sequence file that has precomputed forward operators (FOs). `convert_roms_obs` can also convert `s4dvar.in`:`OBSname`,`MODname` files to a DART observation sequence file that does not have the precomputed FOs.

The ROMS verification observation files **must** contain the **obs_provenance as a global attribute** and the following variables:

- *obs_lat, obs_lon, obs_depth*
- *obs_value*
- *obs_error*
- *obs_time*
- *NLmodel_value*
- *obs_scale*
- *obs_provenance*

Note that the *obs_provenance:flag_values*, and *obs_provenance:flag_meanings* attributes are totally ignored - those relationships are specified by the global attribute **obs_provenance**.

Locations only specified by *obs_Xgrid, obs_Ygrid, obs_depth* are **not** supported.

The conversion of a (set of) ROMS verification observations requires metadata to coordinate the relationship of the ROMS observation provenance to a DART observation TYPE. ROMS provides significant flexibility when specifying the observation provenance and it is simply impractical for DART to try to support all of them. An example of the current practice is described in the PROGRAMS section below.

Important: `filter` and `perfect_model_obs` must also be informed which DART observation types use precomputed forward operators. This is done by setting the `input.nml&obs_kind_nml` namelist. An example is shown at the end of the PROGRAMS section below.

6.69.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&convert_roms_obs_nml
   ens_size              = 1
   roms_mod_obs_files    = ''
   roms_mod_obs_filelist = 'filelist.txt'
   dart_output_obs_file  = 'obs_seq.out'
```

(continues on next page)

(continued from previous page)

```
  append_to_existing     = .false.
  use_precomputed_values = .true.
  add_random_noise       = .false.
  pert_amplitude         = 0.01
  verbose                = 0
  type_translations      = 'NULL'
 /
```

Item	Type	Description
ens_size	integer	Number of ensemble members which are expected to be found when creating the expected obs values. This must match the number of ROMS "mod" files listed in either the 'roms_mod_obs_files' or 'roms_mod_obs_filelist' namelist items. It is an error if they are not the same length.
roms_mod_obs_files	character(len=256), dimension(100)	List of filenames, one per ensemble member, that contain the observation values for each ensemble member. These are output from the ROMS program. If listing the files explicitly in this list, 'roms_mod_obs_filelist' must be ' ' (null).
roms_mod_obs_filelist	character(len=256)	The name of an ASCII file which contains, one per line, a list of filenames, one per ensemble member, that contain the expected obs values for each ensemble member. The filenames should NOT be quoted. These are output from the ROMS program. If using a filelist, then 'roms_mod_obs_files' must be ' ' (null).
dart_output_obs_file	character(len=256)	The name of the DART obs_seq file to create. If a file already exists with this name, it is either appended to or overwritten depending on the 'append_to_existing' setting below.
append_to_existing	logical	If an existing 'dart_output_obs_file' is found, this namelist item controls how it is handled. If .true. the new observations are appended to the existing file. If .false. the new observations overwrite the existing file.
use_precomputed_values	logical	Flag to indicate that the output DART observation sequence file should include the verification observation values from all of the ROMS observation files. If `.true.` this will result in the DART file having the precomputed FOs to be used in the DART assimilation. If `.false.` this will result in DART files having the instrument values only.
add_random_noise	logical	Almost always should be .false. . The exception is the first cycle of an assimilation if all the ROMS input files are identical (no ensemble currently exists). To create differences in the forward operator values (since they are computed by ROMS), we can add gaussian noise here to give them perturbed values. This should be set as well as the "perturb_from_single_instance = .true." namelist in the `&filter_nml` namelist. After the first cycle, both these should be set back to .false. .
pert_amplitude	real(8)	Ignored unless 'add_random_noise' is .true. . Controls the range of random values added to the expected obs values. Sets the width of a gaussian.
verbose	integer	If greater than 0, prints more information during the conversion.
type_translations	character(256), dimension(2, 100)	A set of strings which control the mapping of ROMS observation types to DART observation types. These should be specified in pairs. The first column should be a string that occurs in the global attribute '`obs_provenance`'. Note that the `obs_provenance:flag_values` and `obs_provenance:flag_meanings` attributes are ignored. The second column should be a DART specific obs type that is found in `DART/assimilation_code/modules/observations/obs_kind_mod.f90`, which is created by the DART `preprocess` program.

6.69.3 Data sources

The origin of the input observation files used by ROMS are completely unknown to me.

6.69.4 Programs

- `convert_roms_obs`
- *PROGRAM obs_seq_to_netcdf*
- *program obs_sequence_tool*
- *PROGRAM preprocess*
- *PROGRAM advance_time*

Only `convert_roms_obs` will be discussed here.

The **global attribute** `obs_provenance` is used to relate the observation provenance to DART observation TYPES. The ROMS 'MODname' netCDF file(s) must have both the `obs_provenance` variable and a `obs_provenance` **global attribute**. The **exact** strings must be repeated in the DART `convert_roms_obs_nml:type_translations` variable to be able to convert from the integer value of the obs_provenance to th DART type in the following example:

`ncdump -h roms_mod_obs.nc` (the output has been pruned for clarity)

```
netcdf roms_mod_obs {
dimensions:
        record = 2 ;
        survey = 5376 ;
        state_var = 8 ;
        datum = 2407217 ;
variables:
        {snip}
        int obs_provenance(datum) ;
                obs_provenance:long_name = "observation origin" ;
                obs_provenance:flag_values = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ;
        double obs_time(datum) ;
                obs_time:long_name = "time of observation" ;
                obs_time:units = "days since 1900-01-01 00:00:00 GMT" ;
                obs_time:calendar = "gregorian" ;
        double obs_lon(datum) ;
                obs_lon:long_name = "observation longitude" ;
                obs_lon:units = "degrees_east" ;
        double obs_lat(datum) ;
                obs_lat:long_name = "observation latitude" ;
                obs_lat:units = "degrees_north" ;
        double obs_depth(datum) ;
                obs_depth:long_name = "ROMS internal depth of observation variable" ;
                obs_depth:units = "meters or fractional z-levels" ;
                obs_depth:negative_value = "downwards" ;
                obs_depth:missing_value = 1.e+37 ;
        double obs_error(datum) ;
                obs_error:long_name = "observation error covariance" ;
        double obs_value(datum) ;
                obs_value:long_name = "observation value" ;
        double obs_scale(datum) ;
```

(continues on next page)

(continued from previous page)

```
                obs_scale:long_name = "observation screening/normalization scale" ;
                obs_scale:_FillValue = 0. ;
        double NLmodel_value(datum) ;
                NLmodel_value:long_name = "nonlinear model at observation locations" ;
                NLmodel_value:_FillValue = 1.e+37 ;
        {snip}
    :obs_provenance = "\n",
              "1: gridded AVISO sea level anomaly (zeta)\n",
              "2: gridded Aquarius SSS (salinity)\n",
              "3: XBT from Met Office (temperature)\n",
              "4: CTD from Met Office (temperature)\n",
              "5: CTD from Met Office (salinity)\n",
              "6: ARGO floats (temperature)\n",
              "7: ARGO floats (salinity)\n",
              "8: glider UCSD (temperature)\n",
              "9: glider UCSD (salinity)\n",
              "10: blended satellite SST (temperature)" ;
        {snip}
```

Note the integer values that start the obs_provenance strings are used to interpret the integer contents of the obs_provenance variable. They need not be consecutive, nor in any particular order, but they must not appear more than once.

The following is the relevent section of the DART `input.nml`:

```
&convert_roms_obs_nml
   ens_size               = 32
   roms_mod_obs_filelist  = 'precomputed_files.txt'
   dart_output_obs_file   = 'obs_seq.out'
   append_to_existing     = .false.
   use_precomputed_values = .true.
   add_random_noise       = .false.
   verbose                = 1
   type_translations = "gridded AVISO sea level anomaly (zeta)", "SATELLITE_SSH",
                       "gridded Aquarius SSS (salinity)",        "SATELLITE_SSS",
                       "XBT from Met Office (temperature)",      "XBT_TEMPERATURE",
                       "CTD from Met Office (temperature)",      "CTD_TEMPERATURE",
                       "CTD from Met Office (salinity)",         "CTD_SALINITY",
                       "ARGO floats (temperature)",              "ARGO_TEMPERATURE",
                       "ARGO floats (salinity)",                 "ARGO_SALINITY",
                       "glider UCSD (temperature)",              "GLIDER_TEMPERATURE",
                       "glider UCSD (salinity)",                 "GLIDER_SALINITY",
                       "blended satellite SST (temperature)",    "SATELLITE_BLENDED_SST"
  /
```

A complete list of DART observation TYPES for oceans is described in *MODULE obs_def_ocean_mod*

Any or all of the DART observation types that appear in the second column of `type_translations` must also be designated as observations that have precomputed forward operators. This is done by setting the `input.nml&obs_kind_nml` namelist as follows:

```
&obs_kind_nml
   assimilate_these_obs_types =          'SATELLITE_SSH',
                                         'SATELLITE_SSS',
                                         'XBT_TEMPERATURE',
                                         'CTD_TEMPERATURE',
                                         'CTD_SALINITY',
                                         'ARGO_TEMPERATURE',
                                         'ARGO_SALINITY',
                                         'GLIDER_TEMPERATURE',
                                         'GLIDER_SALINITY',
                                         'SATELLITE_BLENDED_SST'
   use_precomputed_FOs_these_obs_types = 'SATELLITE_SSH',
                                         'SATELLITE_SSS',
                                         'XBT_TEMPERATURE',
                                         'CTD_TEMPERATURE',
                                         'CTD_SALINITY',
                                         'ARGO_TEMPERATURE',
                                         'ARGO_SALINITY',
                                         'GLIDER_TEMPERATURE',
                                         'GLIDER_SALINITY',
                                         'SATELLITE_BLENDED_SST'
  /
```

6.70 PROGRAM `SIF_to_obs_netcdf`

6.70.1 Overview

Harmonized SIF data product to DART observation sequence converter

This routine converts a harmonized satellite SIF product (Harmonized long-term SIF; SIF005) to DART `obs_seq` format. The SIF product is described by JPL and combines GOME-2 and SCIAMACHY SIF retrievals, along with MODIS data to produce a single continuous, monthly, 0.05 degree SIF data set. See Wen et al., 2020 RSE for a more detailed description. The conversion script was designed and tested for version SIF005v2. Download instructions can be found in the *Data Sources* section below.

This SIF data product also comes with its own uncertainty value, and quality control flag described below. Namelist options also include a wavelength option (e.g. 740 nm or 755 nm) to specify the location the SIF value is centered upon.

Standard workflow:

1. Download the Level 3 data for the months of interest. Years 2002-2018 are available as of 5/18/21. (see *Data Sources* below)
2. Make note of the SIF wavelength the data is centered upon. This information is included in the SIF variable of netcdf file `SIF_740_daily_corr`
3. Build the DART executables with support for land observations. This is done by running `preprocess` with `obs_def_land_mod.f90` in the list of `input_files` for `preprocess_nml`.
4. Provide basic information via the `SIF_to_obs_netcdf_nml` (e.g. verbose, wavelength)
5. Convert single or multiple SIF netcdf data files using `SIF_to_obs_netcdf`. Converting one file at a time results in better memory management, but this is unlikely to be an issue in most cases.

6. Combine all output files for the region and timeframe of interest into one file using *program obs_sequence_tool*

For some models (CLM, for example), it is required to reorganize the observation sequence files into a series of files that contains ONLY the observations for each assimilation. This can be achieved with the *makedaily.sh* script which can be found in the *DART/models/clm/shell_scripts* directory.

6.70.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&SIF_to_obs_netcdf_nml
   input_file_list = 'SIF.input.txt',
   verbose         = 0
   wavelength      = 740
   /
```

Contents	Type	Description
input_file_list	character(len=256)	Name of the Level 3 netcdf containing with SIF data. This may be a relative or absolute filename.
verbose	integer	Print more/less information during the `SIF_to_obs_netcdf` execution.
wavelength	integer	The wavelength at which SIF irradiance is centered upon (e.g. 740 nm)

6.70.3 Data Sources

The datasets are available from the Cornell University Box service, and have names like:

`SIF005_YYYYMM.nc`, `SIF005_200504.nc`, `SIF005_201808.nc`

The Level 3 SIF product is provided within netcdf files for monthly average values from 2002-2018. This `SIF_obs_to_netcdf` converter was tested using SIF005v2 files, although older (SIF005v1) and newer (SIF005v2.2) versions exist with similar format.

The data product variables are provided in global, gridded (lat/lon) format:

Units	Variable	Description	Observation TYPE	DART QUANTITY	DART units
mW/m^2/nm/sr	SIF_740_daily_corr	Solar Induced Fluorescence Irradiance	HARMONIZED_SIF	QTY_SOLAR_INDUCED_FLUORESCENCE	mW/m^2/nm/sr
mW/m^2/nm/sr	SIF_740_daily_corr_SD	Solar Induced Fluorescence Irradiance Standard Deviation	N/A	N/A	mW/m^2/nm/sr
See below	EVI_Quality	MODIS EVI Quality Flag	N/A	N/A	See below
degrees	lat	latitude	N/A	N/A	radians
degrees	lon	longitude	N/A	N/A	radians

The `SIF_740_daily_corr` value is the SIF satellite derived irradiance value. It is most closely related to the 'top of the vegetation canopy' emitted SIF as simulated from land surface models. This is distinct from 'leaf-level' emitted SIF.

The `SIF_740_daily_corr_SD` value is an algorithm based uncertainty estimate provided by the data product providers. It is most closely related to instrument uncertainty inherent to the SIF retrievals and does not account for representativeness error when compared to the simulated SIF from a land surface model. We recommend this uncertainty value be used as a minimum baseline when performing data assimilation.

The `EVI_Quality` is a data quality estimate for the `SIF_740_daily_corr`. The `EVI_Quality` is derived from the MODIS retrieval of EVI (enhanced vegetation index) which is one of the explanatory variables used in the algorithm to calculate `SIF_740_daily_corr`. The `EVI_Quality` is an integer (representing a 16 bit field) that evaluates quality through 9 parameters that include VI (Vegetation Index) Quality, VI Usefulness, Aerosol Quantity, Adjacent Cloud Detection, Atmosphere BRDF correction, Mixed Clouds, Land/Water Mask, possible snow/ice, possible shadow. See Table 5 of the MODIS Vegetation Index Users Guide for more information.

The DART-compatible QC value assigned to the *obs_seq.out* uses the criteria from the MODIS EVI Quality and EVI Usefulness only. The DART-compatible QC is based on NCEP-like error codes and `SIF_to_obs_netcdf` assigns values as follows:

0 = best quality
1 = less quality
"....."
17 = least quality
50 = faulty, no utility

The *input_qc_threshold* namelist value can be used to test whether or not lesser quality observations improve the result or not. Thus, all observations (except those that are defined as faulty/no utility) are included in *obs_seq.out* and the exclusion of observations is left up to the user based upon the *input_qc_threshold*.

The qc value assignment is such where values given an EVI quality value of 'good' (00), are assigned a QC from 1-7 based on the EVI Quality Usefulness Parameter (see table below). Values where the 'EVI is produced, but should be checked with additional QA' (01) are assigned a QC from 10-17. Anything with an EVI Quality Usefulness Parameter of '1101' or higher is given a QC of 50 (or more) and is currently skipped and **not written** to the output observation sequence file.

EVI Quality Usefulness Parameter		QC	EVI Quality Value (00)	QC	EVI Quality Value (01)
0000	Highest quality	0	Highest quality	10	Decreasing quality
0001	Lower quality	1	Lower quality	11	Decreasing quality
0010	Decreasing quality	2	Decreasing quality	12	Decreasing quality
0100	Decreasing quality	3	Decreasing quality	13	Decreasing quality
1000	Decreasing quality	4	Decreasing quality	14	Decreasing quality
1001	Decreasing quality	5	Decreasing quality	15	Decreasing quality
1010	Decreasing quality	6	Decreasing quality	16	Decreasing quality
1100	Lowest quality	7	Decreasing quality	17	Least quality
1101	Quality so low that it is not useful	50	Not used	50	Not used
1110	L1B data faulty	50	Not used	50	Not used
1111	Not useful for any other reason/not processed	50	Not used	50	Not used

6.70.4 Citation

Wen, J., P. Köhler, G. Duveiller, N. C. Parazoo, T. S. Magney, G. Hooker, L. Yu, C. Y. Chang, and Y. Sun. "A framework for harmonizing multiple satellite instruments to generate a long-term global high spatial-resolution solar-induced chlorophyll fluorescence (SIF)." Remote Sensing of Environment 239 (2020): 111644.https://doi.org/10.1016/j.rse.2020.111644

6.70.5 Programs

The `SIF_to_obs_netcdf.f90` file is the source for the main converter program. To compile and test, go into the work subdirectory and run `mkmf_preprocess`, run the `Makefile` and finally run `preprocess`. Be sure that `obs_def_land_mod.f90` is included as an input file within `&preprocess_nml` of the `input.nml`.

Next compile the observation converter by running `mkmf_SIF_to_obs_netcdf`, run `Makefile`, and finally run `SIF_to_obs_netcdf`.

6.71 SSEC Data Center

6.71.1 Overview

The program in this directory takes satellite wind data from the University of Wisconsin-Madison Space Science and Engineering Center, and converts it into DART format observation sequence files, for use in assimilating with the DART filter program.

6.71.2 Data sources

The Space Science and Engineering Center (SSEC) at University of Wisconsin-Madison has an online data center with both real-time and archival weather satellite data.

The last 2 day's worth of data is available from ftp://cyclone.ssec.wisc.edu/pub/fnoc.

There is a second satellite wind DART converter in the *MADIS Data Ingest System* directory which converts wind observations which originate from NESDIS. The data from this converter is processed at the SSEC and the observations will be different from the ones distributed by MADIS.

6.71.3 Programs

Conversion program `convert_ssec_satwnd` converts the ascii data in the input files into a DART observation sequence file. Go into the `work` directory and run the `quickbuild.sh` script to compile the necessary files.

The program reads standard input for the data time range, which types of observations to convert, and then, if quality control information is found in the input file, what type of quality control algorithm to use when deciding whether the observation is of good quality or not. See the references below.

6.71.4 References

- RF method: Velden, C. S., T. L. Olander, and S. Wanzong, 1998: The impact of multispectral GOES-8 wind information on Atlantic tropical cyclone track forecasts in 1995. Part I: Dataset methodology, description, and case analysis. Mon. Wea. Rev., 126, 1202-1218.
- QI method: Holmlund, K., 1998: The utilization of statistical properties of satellite-derived atmospheric motion vectors to derive quality indicators. Wea. Forecasting, 13, 1093-1104.
- Comparison of two methods: Holmlund, K., C.S. Velden, and M. Rohn, 2001: Enhanced Automated Quality Control Applied to High-Density Satellite-Derived Winds. Mon. Wea. Rev., 129, 517-529.

6.72 PROGRAM `sst_to_obs, oi_sst_to_obs`

6.72.1 Overview

There are two gridded SST observation converters in this directory, one for data from PODAAC, and one from NOAA/NCDC. `sst_to_obs` converts data from PODAAC and has been used by Romain Escudier for regional studies with ROMS. `oi_sst_to_obs` converts data from NOAA/NCDC and has been used by Fred Castruccio for global studies with POP.

sst_to_obs – GHRSST to DART observation sequence converter

These routines are designed to convert the GHRSST Level 4 AVHRR_OI Global Blended Sea Surface Temperature Analysis (GDS version 2) from NCEI data distributed by the Physical Oceanography Distributed Active Archive Center. Please remember to cite the data in your publications, specific instructions from PODAAC are available here. This is an example:

> National Centers for Environmental Information. 2016. GHRSST Level 4 AVHRR_OI Global Blended Sea Surface Temperature Analysis (GDS version 2) from NCEI. Ver. 2.0. PO.DAAC, CA, USA. Dataset accessed [YYYY-MM-DD] at http://dx.doi.org/10.5067/GHAAO-4BC02.

Many thanks to Romain Escudier (then at Rutgers) who did the bulk of the work and graciously contributed his efforts to the DART project. Romain gave us scripts and source code to download the data from the PODAAC site, subset the global files to a region of interest, and convert that subsetted file to a DART observation sequence file. Those scripts and programs have been only lightly modified to work with the Manhattan version of DART and contain a bit more documentation.

The workflow is usually:

1. compile the converters by running `work/quickbuild.sh` in the usual way.
2. customize the `shell_scripts/parameters_SST` resource file to specify variables used by the rest of the scripting.
3. run `shell_scripts/get_sst_ftp.sh` to download the data from PODAAC.
4. provide a mask for the desired study area.
5. run `shell_scripts/Prepare_SST.sh` to subset the PODAAC data and create the DART observation sequence files. Be aware that the `Prepare_SST.sh` modifies the `shell_scripts/input.nml.template` file and generates its own `input.nml`. `work/input.nml` is not used.
6. combine all output files for the region and timeframe of interest into one file using the *program obs_sequence_tool*

Example

It is worth describing a small example. If you configure `get_sst_ftp.sh` to download the last two days of 2010 and then specify the mask to subset for the NorthWestAtlantic (NWA) and run `Prepare_SST.sh` your directory structure should look like the following:

```
0[1234] cheyenne6:/<6>obs_converters/SST
.
|-- ObsData
|   `-- SST
|       |-- ncfile
|       |   `-- 2010
|       |       |-- 20101230120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0.nc
|       |       `-- 20101231120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0.nc
|       `-- nwaSST
|           `-- 2010
|               |-- 20101230120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0_
↪NWA.nc
|               `-- 20101231120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0_
↪NWA.nc
|-- oi_sst_to_obs.f90
|-- oi_sst_to_obs.nml
|-- sst_to_obs.f90
|-- sst_to_obs.nml
|-- shell_scripts
|   |-- Prepare_SST.sh
|   |-- functions.sh
|   |-- get_sst_ftp.sh
|   |-- input.nml
|   |-- input.nml.template
|   |-- my_log.txt
|   |-- parameters_SST
|   `-- prepare_SST_file_NWA.sh
|-- masks
|   |-- Mask_NWA-NCDC-L4LRblend-GLOB-v01-fv02_0-AVHRR_OI.nc
|   `-- Mask_NWA120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0.nc
`-- work
    |-- Makefile
    |-- advance_time
    |-- input.nml
    |-- obs_sequence_tool
    |-- oi_sst_to_obs
    |-- preprocess
    |-- quickbuild.sh
    `-- sst_to_obs
```

The location of the DART observation sequence files is specified by `parameter_SST:DIR_OUT_DART`. That directory should contain the following two files:

```
0[1236] cheyenne6:/<6>v2/Err30 > ls -l
'total 7104
-rw-r--r-- 1 thoar p86850054 3626065 Jan 10 11:08 obs_seq.sst.20101230
-rw-r--r-- 1 thoar p86850054 3626065 Jan 10 11:08 obs_seq.sst.20101231
```

6.72.2 oi_sst_to_obs – noaa/ncdc to DART observation sequence converter

oi_sst_to_obs is designed to convert the NOAA High-resolution Blended Analysis: Daily Values using AVHRR only data. The global metadata of a typical file is shown here:

```
:Conventions = "CF-1.5" ;
:title = "NOAA High-resolution Blended Analysis: Daily Values using AVHRR only" ;
:institution = "NOAA/NCDC" ;
:source = "NOAA/NCDC  ftp://eclipse.ncdc.noaa.gov/pub/OI-daily-v2/" ;
:comment = "Reynolds, et al., 2007:
     Daily High-Resolution-Blended Analyses for Sea Surface Temperature.
     J. Climate, 20, 5473-5496.
     Climatology is based on 1971-2000 OI.v2 SST,
     Satellite data: Navy NOAA17 NOAA18 AVHRR, Ice data: NCEP ice." ;
:history = "Thu Aug 24 13:46:51 2017: ncatted -O -a References,global,d,, sst.day.mean.
↪2004.v2.nc\n",
        "Version 1.0" ;
:references = "https://www.esrl.noaa.gov/psd/data/gridded/data.noaa.oisst.v2.highres.html
↪" ;
:dataset_title = "NOAA Daily Optimum Interpolation Sea Surface Temperature" ;
```

The workflow is usually:

1. compile the converters by running work/quickbuild.sh in the usual way.
2. download the desired data.
3. customize the work/input.nml file.
4. run work/oi_sst_to_obs to create a single DART observation sequence file.
5. combine all output files for the region and timeframe of interest into one file using the *program obs_sequence_tool*

6.72.3 sst_to_obs namelist

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&sst_to_obs_nml
   sst_netcdf_file     = '1234567.nc'
   sst_netcdf_filelist = 'sst_to_obs_filelist'
   sst_out_file        = 'obs_seq.sst'
   subsample_intv      = 1
   sst_rep_error       = 0.3
   debug               = .false.
   /
```

Contents	Type	Description
sst_netcdf_file	character(len=256)	Name of the (usually subsetted) netcdf data file. This may be a relative or absolute filename. If you run the scripts 'as is', this will be something like: `../ObsData/SST/nwaSST/2010/20101231120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0_NWA.nc`
sst_netcdf_filelist	character(len=256)	Name of the file that contains a list of (usually subsetted) data files, one per line. **You may not specify both sst_netcdf_file AND sst_netcdf_filelist.** One of them must be empty.
sst_out_file	character(len=256)	Name of the output observation sequence file.
subsample_intv	integer	It is possible to 'thin' the observations. `subsample_intv` allows one to take every Nth observation.
sst_rep_error	real	In DART the observation error variance can be thought of as having two components, an instrument error and a representativeness error. In `sst_to_obs` the instrument error is specified in the netCDF file by the variable `analysis_error`. The representativeness error is specified by `sst_rep_error`, which is specified as a standard deviation. These two values are added together and squared and used as the observation error variance. **Note:** This algorithm maintains backwards compatibility, but is technically not the right way to combine these two quantities. If they both specified variance, adding them together and then taking the square root would correctly specify a standard deviation. Variances add, standard deviations do not. Since the true observation error variance (in general) is not known, we are content to live with an algorithm that produces useful observation error variances. If your research comes to a more definitive conclusion, please let us know.
debug	logical	Print extra information during the `sst_to_obs` execution.

6.72.4 oi_sst_to_obs namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&oi_sst_to_obs_nml
   input_file       = '1234567.nc'
   output_file_base = 'obs_seq.sst'
   subsample_intv   = 1
   sst_error_std    = 0.3
   debug            = .false.
   /
```

Contents	Type	Description
input_file	character(len=256)	Name of the input netcdf data file. This may be a relative or absolute filename. If you run the scripts 'as is', this will be something like: `../ObsData/SST/nwaSST/2010/20010231120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0_NWA.nc`
output_file_base	character(len=256)	Partial filename for the output file. The date and time are appended to `output_file_base` to construct a unique filename reflecting the time of the observations in the file.
subsample_intv	integer	It is possible to 'thin' the observations. `subsample_intv` allows one to take every Nth observation.
sst_error_std	real	This is the total observation error standard deviation.
debug	logical	Print extra information during the `oi_sst_to_obs` execution.

6.72.5 Decisions you might need to make

See the general discussion in the *Creating an obs_seq file from real observations* page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.72.6 Known Bugs

I do not believe `sst_to_obs` will work correctly if given multiple files in `sst_netcdf_filelist`. The number of observation used to declare the length of the output observation sequence is based on a single file ... yet seems to be used by many. I have not tested this configuration, since the scripting does not use the `sst_netcdf_filelist` mechanism.

6.73 SSUSI F16 EDR-DSK format to observation sequence converters

6.73.1 Overview

The Special Sensor Ultraviolet Spectrographic Imager SSUSI is designed to remotely sense the ionosphere and thermosphere. The following is repeated from the SSUSI home page:

> *Overview Beginning in 2003, the Defense Meteorological Satellite Program (DMSP) satellites began carrying the SSUSI instrument - a combination of spectrographic imaging and photometric systems designed to remotely sense the ionosphere and thermosphere. The long term focus of the SSUSI program is to provide data concerning the upper atmospheric response to the sun over the changing conditions of the solar cycle. Data collected by SSUSI instrument can help identify structure in the equatorial and polar regions. Mission SSUSI was designed for the DMSP Block 5D-3 satellites. These satellites are placed into nearly polar, sun-synchronous orbits at an altitude of about 850 km. SSUSI is a remote-sensing instrument which measures ultraviolet (UV) emissions in five different wavelength bands from the Earth's upper atmosphere. SSUSI is mounted on a nadir-looking panel of the satellite. The multicolor images from SSUSI cover the visible Earth disk from horizon to horizon and the anti-sunward limb up to an altitude of approximately 520 km. The UV images and the derived environmental data provide the Air Force Weather Agency (Offutt Air Force Base, Bellevue, NE) with near real-time information that can be utilized in a number of applications, such as maintenance of high frequency (HF) communication links and related systems and assessment of the environmental hazard to astronauts on the Space Station.*

convert_f16_edr_dsk.f90 will extract the ON2 observations from the F16 "edr-dsk" format files and create DART observation sequence files. There is one additional preprocessing step before the edr-dsk files may be converted.

The ON2_UNCERTAINTY variable in the netcdf files have IEEE NaN values, but none of the required metadata to interpret them correctly. These 2 lines will add the required attributes so that NaNs are replaced with a fill value that can be queried and manipulated. Since the ON2_UNCERTAINTY is a standard deviation, it is sufficient to make the fill value negative. See the section on Known Bugs

```
ncatted -a _FillValue,ON2_UNCERTAINTY,o,f,NaN        input_file.nc
ncatted -a _FillValue,ON2_UNCERTAINTY,m,f,-1.0       input_file.nc
```

6.73.2 Data sources

http://ssusi.jhuapl.edu/data_products

Please read their data usage policy.

6.73.3 Programs

DART/observations/SSUSI/convert_f16_edr_dsk.f90 will extract ON2 data from the distribution files and create DART observation sequence (obs_seq) files. Build it in the SSUSI/work directory by running the ./quickbuild.sh script located there. In addition to the converters, the advance_time and obs_sequence_tool utilities will be built.

An example data file is in the data directory. An example scripts for adding the required metadata to the ON2_UNCERTAINTY variable in the shell_scripts directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the scripts saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

6.73.4 Errors

The code for setting observation error variances is using fixed values, and we are not certain if they are correct. Incoming QC values larger than 0 are suspect, but it is not clear if they really signal unusable values or whether there are some codes we should accept.

6.73.5 Known Bugs

The netCDF files - as distributed - have NaN values to indicate "MISSING". This makes it exceptionally hard to read or work with, as almost everything will core dump when trying to perform any math with NaNs. convert_f16_edr_dsk.f90 tries to count how many values are missing. If the NaN has not been replaced with a numerically valid MISSING value, the following FATAL ERROR is generated (by the Intel compiler, with debug and traceback enabled):

```
set_nml_output Echo NML values to log file only
Trying to open namelist log dart_log.nml
forrtl: error (65): floating invalid
Image              PC                Routine            Line        Source
convert_f16_edr_d  000000000051717D  MAIN__                  143  convert_f16_edr_dsk.
↪f90
convert_f16_edr_d  0000000000409B3C  Unknown               Unknown  Unknown
libc.so.6          0000003101E1ED5D  Unknown               Unknown  Unknown
```

(continues on next page)

(continued from previous page)

```
convert_f16_edr_d  0000000000409A39  Unknown               Unknown  Unknown
Abort (core dumped)
```

The solution is to replace the NaN values with a viable MISSING value using the `shell_scripts/netcdf_manip.csh` script. It relies on the netCDF Operators, freely available http://nco.sourceforge.net

6.74 WOD Observations

6.74.1 Overview

The World Ocean Database (WOD) is a collection of data from various sources, combined into a single format with uniform treatment. WOD is created by the National Centers for Environmental Information (NCEI) of the National Oceanic and Atmospheric Administration (NOAA).

An updated version of the dataset is released approximately every four years. It was first produced in 1994 and has been released in 1998, 2001, 2005, 2009, 2013 and 2018.

The WOD website has detailed information about the repository, observations, and datasets. The programs in this directory convert from the packed ASCII files found in the repository into DART observation sequence (obs_seq) file format.

There are two sets of available files: the raw observations and the observations binned onto standard levels.

Note: DAReS staff recommend using the datasets on standard levels for assimilation. The raw data can be very dense in the vertical and are not truly independent observations. The correlation between nearby observations leads to too much certainty in the updated values during the assimilation.

6.74.2 Data sources

Use already existing obs_seq files

NCAR staff have prepared datasets already converted to DART's obs_seq file format for the World Ocean Database 2013 (WOD13) and the World Ocean Database 2009 (WOD09).

WOD13

The already-converted WOD13 dataset comprises data from 2005-01-01 to 2016-12-31 and was created by *Fred Castruccio*. Thanks Fred! The files are stored in the following directory on GLADE:

```
/glade/p/cisl/dares/Observations/WOD13
```

The subdirectories are formatted in `YYYYMM` order and contain the following observation types:

FLOAT_SALINITY	FLOAT_TEMPERATURE
DRIFTER_SALINITY	DRIFTER_TEMPERATURE
GLIDER_SALINITY	GLIDER_TEMPERATURE
MOORING_SALINITY	MOORING_TEMPERATURE
BOTTLE_SALINITY	BOTTLE_TEMPERATURE
CTD_SALINITY	CTD_TEMPERATURE
XCTD_SALINITY	XCTD_TEMPERATURE
APB_SALINITY	APB_TEMPERATURE
XBT_TEMPERATURE	

If you use WOD13, please cite Boyer et al. (2013).[1]

WOD09

The already-converted WOD09 dataset, which comprises data from 1960-01-01 to 2008-12-31, is stored in the following directory on GLADE:

```
/glade/p/cisl/dares/Observations/WOD09
```

If you use WOD09, please cite Johnson et al. (2009).[2]

Download WOD from NCEI

Data from each of the WOD releases can be downloaded interactively from the WOD website.

Download WOD from NCAR

WOD09 can also be downloaded from NCAR's research data archive (RDA) dataset 285.0.

6.74.3 Programs

The data is distributed in a specialized packed ASCII format. In this directory is a program called `wodFOR.f` which is an example reader program to print out data values from the files. The program `wod_to_obs` converts these packed ASCII files into DART obs_sequence files.

As with most other DART directories, the `work` directory contains a `quickbuild.sh` script to build all necessary executables.

[1] Boyer, T.P., J. I. Antonov, O. K. Baranova, C. Coleman, H. E. Garcia, A. Grodsky, D. R. Johnson, R. A. Locarnini, A. V. Mishonov, T.D. O'Brien, C.R. Paver, J.R. Reagan, D. Seidov, I. V. Smolyar, and M. M. Zweng, 2013: World Ocean Database 2013, NOAA Atlas NESDIS 72, S. Levitus, Ed., A. Mishonov, Technical Ed.; Silver Spring, MD, 209 pp., doi:10.7289/V5NZ85MT.

[2] Johnson, D.R., T.P. Boyer, H.E. Garcia, R.A. Locarnini, O.K. Baranova, and M.M. Zweng, 2009. World Ocean Database 2009 Documentation. Edited by Sydney Levitus. NODC Internal Report 20, NOAA Printing Office, Silver Spring, MD, 175 pp., http://www.nodc.noaa.gov/OC5/WOD09/pr_wod09.html.

6.74.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&wod_to_obs_nml
   wod_input_file       =  'XBTS2005',
   wod_input_filelist   =  '',
   wod_out_file         =  'obs_seq.wod',
   avg_obs_per_file     =  500000,
   debug                =  .false.,
   timedebug            =  .false.,
   print_qc_summary     =  .true.,
   max_casts            =  -1,
   no_output_file       =  .false.,
   print_every_nth_cast =  -1,
   temperature_error    =  0.5,
   salinity_error       =  0.5,
 /
! temperature error is in degrees C, salinity error in g/kg.
```

Item	Type	Description
wod_input_file	character(len=128)	The input filename when converting a single file. Only one of the two namelist items that specify input files can have a valid value, so to use a single filename set the list name 'wod_input_filelist' to the empty string (' ').
wod_input_filelist	character(len=128)	To convert one or more files in a single execution create a text file which contains each input filename, in ascii, one filename per line. Set this item to the name of that file, and set 'wod_input_file' to the empty string (' ').
wod_output_file	character(len=128)	The output file to be created. Note that unlike earlier versions of some converters, this program will overwrite an existing output file instead of appending to it. The risk of replicated observations, which is difficult to detect since most of the contents are floating point numbers, outweighed the possible utility.
avg_obs_per_file	integer	The code needs an upper limit on the number of observations generated by this program. It can be larger than the actual number of observations converted. The total number of obs is computed by multiplying this number by the number of input files. If you get an error because there is no more room to add observations to the output file, increase this number. Do not make this an unreasonably huge number, however, since the code does preallocate space and will be slow if the number of obs becomes very large.
print_every_nth_obs	integer	If a value greater than 0, the program will print a message after processing every N casts. This allows the user to monitor the progress of the conversion.
print_qc_summary	logical	If .TRUE. the program will print out a summary of the number of casts which had a non-zero quality control values (current files appear to use values of 1-9).
debug	logical	If .TRUE. the program will print out debugging information.
timedebug	logical	If .TRUE. the program will print out specialized time-related debugging information.
max_casts	integer	If a value greater than 0 the program will only convert at most this number of casts from each input file. Generally only expected to be useful for debugging. A negative value will convert all data from the input file.
no_output	logical	If .TRUE. the converter will do all the work needed to convert the observations, count the number of each category of QC values, etc, but will not create the final obs_seq file. Can be useful if checking an input file for problems, or for getting QC statistics without waiting for a full output file to be constructed, which can be slow for large numbers of obs. Only expected to be useful for debugging.
temperature_error	real(r8)	The combined expected error of temperature observations from all sources, including instrument error, model bias, and representativeness error (e.g. larger or smaller grid box sizes affecting expected accuracy), in degrees Centigrade. Values in output file are error variance, which will be this value squared.
salinity_error	real(r8)	The combined expected error of salinity observations from all sources, including instrument error, model bias, and representativeness error (e.g. larger or smaller grid box sizes affecting expected accuracy) in g/kg (psu). Values in output file are error variance, and use units of msu (kg/kg), so the numbers will be this value / 1000.0, squared.

6.74.5 Modules used

```
types_mod
time_manager_mod
utilities_mod
location_mod
obs_sequence_mod
obs_def_mod
obs_def_ocean_mod
obs_kind_mod
```

6.74.6 Errors and known bugs

The code for setting observation error variances is using fixed values, and we are not certain if they are correct. Incoming QC values larger than 0 are suspect, but it is not clear if they really signal unusable values or whether there are some codes we should accept.

6.74.7 Future Plans

- This converter is currently being used on WOD09 data, but the standard files generally stop with early 2009 data. There are subsequent additional new obs files available from the download site.
- The fractional-time field, and sometimes the day-of-month field in a small percentage of the obs have bad values. The program currently discards these obs, but it may be possible to recover the original good day number and/or time of day. There is a subroutine at the end of the *wod_to_obs.f90* file which contains all the reject/accept/correction information for the year, month, day, time fields. To accept or correct the times on more obs, edit this subroutine and make the necessary changes.

6.74.8 References

6.75 GND GPS VTEC

This is a modification of a standard "text" converter that comes with DART.

`gnd_gps_vtec_text_to_obs.f90` reads VTEC text files (from OpenMadrigal at http://madrigal.haystack.mit.edu/) and outputs DART obs_seq.out files.

Please examine `work/input.nml:&text_to_obs_nml` as it specifies the name of the input and the output files

The provided file work/gps021201g.002.txt is only for example (only 2 datapoints are shown) and not for real estimation.

6.76 GPS Observations

6.76.1 Overview

The COSMIC project provides data from a series of satellites. There are two forms of the data that are used by DART: GPS Radio Occultation data and Electron Density. The programs in this directory extract the data from the distribution files and put them into DART observation sequence (obs_seq) file format.

Radio occultation

The COSMIC satellites measure the phase delay caused by deviation of the straight-line path of the GPS satellite signal as it passes through the Earth's atmosphere when the GPS and COSMIC satellites rise and set relative to each other. This deviation results from changes in the angle of refraction of light as it passes through regions of varying density of atmosphere. These changes are a result of variations in the temperature, pressure, and moisture content. Vertical profiles of temperature and moisture can be derived as the signal passes through more and more atmosphere until it is obscured by the earth's horizon. There are thousands of observations each day distributed around the globe, including in areas which previously were poorly observed. These data are converted with the `convert_cosmic_gps_cdf.f90` program and create DART observations of GPSRO_REFRACTIVITY.

Electron density

The COSMIC satellites also provide ionospheric profiles of electron density. The accuracy is generally about 10^{-4} 10^{-5} cm^{-3}. These data are converted with the `convert_cosmic_ionosphere.f90` program and create DART observations tagged as COSMIC_ELECTRON_DENSITY.

6.76.2 Data sources

Data from the COSMIC Program are available by signing up on the data access web page. We prefer delivery in netCDF file format.

Radio occultation

The files we use as input to these conversion programs are the Level 2 data, Atmospheric Profiles (filenames include the string 'atmPrf').

Each vertical profile is stored in a separate netCDF file, and there are between 1000-3000 profiles/day, so converting a day's worth of observations used to involve downloading many individual files. There are now daily tar files available which makes it simpler to download the raw data all in a single file and then untar it to get the individual profiles.

The scripts in the `shell_scripts` directory can now download profiles from any of the available satellites that return GPS RO data to the CDAAC web site. See the `gpsro_to_obsseq.csh` or `convert_many_gpsro.csh` script for where to specify the satellites to be included.

Electron density

The files we have used as input to these conversion programs are from the COSMIC 2013 Mission and have a data type of 'ionPrf'.

The file naming convention and file format are described by COSMIC here and there can be more than 1000 profiles/day. Like the GPS radio occultation data, the profiles are now available in a single daily tar file which can be downloaded then be unpacked into the individual files. COSMIC has instructions on ways to download the data at

http://cdaac-www.cosmic.ucar.edu/cdaac/tar/rest.html

6.76.3 Programs

Convert_cosmic_gps_cdf

The data are distributed in netCDF file format. DART requires all observations to be in a proprietary format often called DART "obs_seq" format. The files in this directory (a combination of C shell scripts and a Fortran source executable) do this data conversion.

The shell_scripts directory contains several example scripts, including one which downloads the raw data files a day at a time (`download_script.csh`), and one which executes the conversion program (`convert_script.csh`). These scripts make 6 hour files by default, but have options for other times. Each profile is stored in a separate netcdf file and there are usually between 1000-3000 files/day, so the download process can be lengthy. You probably want to download as a separate preprocess step and do not use the script options to automatically delete the input files. Keep the files around until you are sure you are satisified with the output files and then delete them by hand.

The conversion executable `convert_cosmic_gps_cdf`, reads the namelist `&convert_cosmic_gps_nml` from the file `input.nml`.

The namelist lets you select from one of two different forward operators. The 'local' forward operator computes the expected observation value at a single point: the requested height at the tangent point of the ray between satellites. The 'non-local' operator computes values along the ray-path and does an integration to get the expected value. The length of the integration segments and height at which to end the integration are given in the namelist. In some experiments the difference between the two types of operators was negligible. This choice is made at the time of the conversion, and the type of operator is stored in the observation, so at runtime the corresponding forward operator will be used to compute the expected observation value.

The namelist also lets you specify at what heights you want observations to be extracted. The raw data is very dense in the vertical; using all values would not results in a set of independent observations. The current source code no longer does an intermediate interpolation; the original profiles appear to be smooth enough that this is not needed. The requested vertical output heights are interpolated directly from the full profile.

Convert_cosmic_ionosphere

Each profile is interpolated to a set of desired levels that are specified at run time. During the conversion process, each profile is checked for negative values of electron density above the minimum desired level. If negative values are found, the entire profile is discarded. If an observation sequence file already exists, the converter will simply add the new observations to it. Multiple profiles may be converted in a single execution, so it is easy to consolidate all the profiles for a single day into a single observation sequence file, for example. `convert_cosmic_ionosphere` reads the namelist `&convert_cosmic_ionosphere_nml` from the file `input.nml`. The original observation times are preserved in the conversion process. If it is desired to subset the observation sequence file such that observations too far away from desired assimilation times are rejected, a separate post-processing step using the *program obs_sequence_tool* is required. A script will be necessary to take a start date, an end date, an assimilation time step, and a desired time 'window' - and strip out the unwanted observations from a series of observation sequence files. There are multiple ways of specifying the observation error variance at run time. They are implemented in a routine named `electron_density_error()` and are selected by the namelist variable `observation_error_method`.

'constant'	a scalar value for all observations
'scaled'	the electron density is multiplied by a scalar value
'lookup'	a lookup table is read
'scaled_lookup'	the lookup table value is multiplied by a scalar value and the electron density value

I-Te Lee: " … the original idea for error of ionospheric observation is 1%. Thus, I put the code as "oerr = 0.01_r8 * obsval". Liu et. al and Yue et al investigated the Abel inversion error of COSMIC ionosphere profile, both of them figure out the large error would appear at the lower altitude and push model toward wrong direction at the lower ionosphere while assimilating these profiles. On the other hand, the Abel inversion error depends on the ionospheric electron density structure, which is a function of local time,

altitude and geomagnetic latitude. To simplify the procedure to define observation error of profiles, Xinan Yue help me to estimate an error matrix and saved in the file which named 'f3coerr.nc'. ... The number in the matrix is error percentage (%), which calculated by OSSE. Here are two reference papers. In the end, the observation error consists of instrumentation error (10%) and Abel error."

- X. Yue, W.S. Schreiner, J. Lei, S.V. Sokolovskiy, C. Rocken, D.C. Hunt, and Y.-H. Kuo (2010), Error analysis of Abel retrieved electron density profiles from radio occultation measurements. *Annales Geophysicae: Atmospheres, Hydrospheres and Space Sciences*. **28** No. 1, pp 217-222, doi:10.5194/angeo-28-217-2010
- J.Y. Liu, C.Y. Lin, C.H. Lin, H.F. Tsai, S.C. Solomon, Y.Y. Sun, I.T. Lee, W.S. Schreiner, and Y.H. Kuo (2010), Artificial plasma cave in the low-latitude ionosphere results from the radio occultation inversion of the FORMOSAT-3/COSMIC}, *Journal of Geophysical Research: Space Physics*. **115** No. A7, pp 2156-2202, doi:10.1029/2009JA015079

It is possible to create observation sequence files for perfect model experiments that have realistic observation sampling patterns and observation error variances that **do not have any actual electron densities**. The COSMIC data files are read, but the electron density information is not written. Keep in mind that some methods of specifying the observation error variance require knowledge of the observation value. If the observation value is bad or the entire profile is bad, no observation locations are created for the profile.

6.76.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&convert_cosmic_gps_nml
   obs_levels             = -1.0
   use_original_kuo_error = .false.
   local_operator         = .true.
   ray_ds                 = 5000.0
   ray_htop               = 15000.0
   gpsro_netcdf_file      = 'cosmic_gps_input.nc'
   gpsro_netcdf_filelist  = ''
   gpsro_out_file         = 'obs_seq.gpsro'
 /
```

Item	Type	Description
obs_levels	integer(200)	A series of heights, in kilometers, where observations from this profile should be interpolated. (Note that the other distances and heights in the namelist are specified in meters.) The values should be listed in increasing height order.
use_original_kuo_error	logical	If .true. use the observation error variances for a refractivity observation that come from a Kuo paper and were implied to be used for the CONUS domain. If .false. use observation error variances similar to what is used in GSI.
local_operator	logical	If .true. compute the observation using a method which assumes all effects occur at the tangent point. If .false. integrate along the tangent line and do ray-path reconstruction.
ray_ds	real(r8)	For the non-local operator only, the delta stepsize, in meters, to use for the along-path integration in each direction out from the tangent point.
ray_htop	real(r8)	For the non-local operator only, stop the integration when one of the endpoints of the next integration step goes above this height. Specify in meters.
gpsro_netcdf_file	character(len=128)	The input filename when converting a single profile. Only one of the file or filelist items can have a valid value, so to use the single filename set the list name 'gpsro_netcdf_filelist' to the empty string (' ').
gpsro_netcdf_filelist	character(len=128)	To convert a series of profiles in a single execution create a text file which contains each input file, in ascii, one filename per line. Set this item to the name of that file, and set 'gpsro_netcdf_file' to the empty string (' ').
gpsro_out_file	character(len=128)	The output file to be created. To be compatible with earlier versions of this program, if this file already exists it will be read in and the new data will be appended to that file.

A more useful example follows:

```
&convert_cosmic_gps_nml
  gpsro_netcdf_file      = ''
  gpsro_netcdf_filelist  = 'flist'
  gpsro_out_file         = 'obs_seq.gpsro'
  local_operator         = .true.
  use_original_kuo_error = .false.
  ray_ds                 = 5000.0
  ray_htop               = 13000.1
  obs_levels =        0.2,  0.4,  0.6,  0.8,
                1.0,  1.2,  1.4,  1.6,  1.8,
                2.0,  2.2,  2.4,  2.6,  2.8,
                3.0,  3.2,  3.4,  3.6,  3.8,
                4.0,  4.2,  4.4,  4.6,  4.8,
                5.0,  5.2,  5.4,  5.6,  5.8,
                6.0,  6.2,  6.4,  6.6,  6.8,
                7.0,  7.2,  7.4,  7.6,  7.8,
                8.0,  8.2,  8.4,  8.6,  8.8,
                9.0,  9.2,  9.4,  9.6,  9.8,
               10.0, 10.2, 10.4, 10.6, 10.8,
               11.0, 11.2, 11.4, 11.6, 11.8,
               12.0, 12.2, 12.4, 12.6, 12.8,
               13.0, 13.2, 13.4, 13.6, 13.8,
               14.0, 14.2, 14.4, 14.6, 14.8,
               15.0, 15.2, 15.4, 15.6, 15.8,
               16.0, 16.2, 16.4, 16.6, 16.8,
               17.0, 17.2, 17.4, 17.6, 17.8,
               18.0, 19.0, 20.0, 21.0, 22.0,
               23.0, 24.0, 25.0, 26.0, 27.0,
```

(continues on next page)

(continued from previous page)

```
               28.0, 29.0, 30.0, 31.0, 32.0,
               33.0, 34.0, 35.0, 36.0, 37.0,
               38.0, 39.0, 40.0, 41.0, 42.0,
               43.0, 44.0, 45.0, 46.0, 47.0,
               48.0, 49.0, 50.0, 51.0, 52.0,
               53.0, 54.0, 55.0, 56.0, 57.0,
               58.0, 59.0, 60.0,
 /
```

```
&convert_cosmic_ionosphere_nml
  input_file               = ''
  input_file_list          = 'input_file_list.txt'
  output_file              = 'obs_seq.out'
  observation_error_file   = 'none'
  observation_error_method = 'scaled_lookup'
  locations_only           = .false.
  obs_error_factor         = 1.0
  verbose                  = 0
  obs_levels               = -1.0
 /
```

Item	Type	Description
input_file	character(len=256)	The input filename when converting a single profile. Only one of the `input_file` or `input_file_list` items can have a valid value, so to use a single filename set `input_file_list = ''`
input_file_list	character(len=256)	To convert a series of profiles in a single execution create a text file which contains one filename per line. Set this item to the name of that file, and set `input_file = ''`
output_file	character(len=256)	The output file to be created. If this file already exists the new data will be added to that file. DART observation sequences are linked lists. When the list is traversed, the observations are in ascending order. The order they appear in the file is completely irrelevant.
observation_error_file	character(len=256)	This specifies a lookup table. The table created by I-Te Lee and Xinan Yue is called `f3coerr.nc`.
observation_error_method	character(len=128)	There are multiple ways of specifying the observation error variance. This character string allows you to select the method. The selection is not case-sensitive. Allowable values are: 'constant', 'scaled', 'lookup', or 'scaled_lookup'. Anything else will result in an error. Look in the `electron_density_error()` routine for specifics.
locations_only	logical	If `locations_only = .true.` then the actual observation values are not written to the output observation sequence file. This is useful for designing an OSSE that has a realistic observation sampling pattern. Keep in mind that some methods of specifying the observation error variance require knowledge of the observation value. If the observation value is bad or the entire profile is bad, this profile is rejected - even if `locations_only = .true.`
obs_error_factor	real(r8)	This is the scalar that is used in several of the methods specifying the observation error variance.
verbose	integer	controls the amount of run-time output echoed to the screen. 0 is nearly silent, higher values write out more. The filenames of the profiles that are skipped are ALWAYS printed.
obs_levels	integer(200)	A series of heights, in kilometers, where observations from this profile should be interpolated. (Note that the other distances and heights in the namelist are specified in meters.) The values must be listed in increasing height order.

A more useful example follows:

```
&convert_cosmic_ionosphere_nml
   input_file              = ''
   input_file_list         = 'file_list.txt'
   output_file             = 'obs_seq.out'
   observation_error_file  = 'f3coeff.dat'
   observation_error_method = 'scaled'
   locations_only          = .false.
   obs_error_factor        = 0.01
   verbose                 = 1
   obs_levels = 160.0, 170.0, 180.0, 190.0, 200.0,
                210.0, 220.0, 230.0, 240.0, 250.0,
                260.0, 270.0, 280.0, 290.0, 300.0,
                310.0, 320.0, 330.0, 340.0, 350.0,
                360.0, 370.0, 380.0, 390.0, 400.0,
                410.0, 420.0, 430.0, 440.0, 450.0
 /
```

6.76.5 Workflow for batch conversions

If you are converting only a day or two of observations you can download the files by hand and call the converter programs from the command line. However if you are going convert many days/months/years of data you need an automated script, possibly submitted to a batch queue on a large machine. The following instructions describe shell scripts we provide as a guide in the `shell_scripts` directory. You will have to adapt them for your own system unless you are running on an NCAR superscomputer.

Making DART Observations from Radio Occultation atmPrf Profiles:

```
Description of the scripts provided to process the COSMIC and
CHAMP GPS radio occultation data.

Summary of workflow:
1) cd to the ../work directory and run ./quickbuild.sh to compile everything.
2) Edit ./gpsro_to_obsseq.csh once to set the directory where the DART
    code is installed, and your CDAAC web site user name and password.
3) Edit ./convert_many_gpsro.csh to set the days of data to download/convert/remove.
4) Run ./convert_many_gpsro.csh either on the command line or submit to a batch system.


More details:

1) quickbuild.sh:

Make sure your $DART/mkmf/mkmf.template is one that matches the
platform and compiler for your system.  It should be the same as
how you have it set to build the other DART executables.

Run quickbuild.sh and it should compile all the executables needed
to do the GPS conversion into DART obs_sequence files.


2) gpsro_to_obsseq.csh:

Edit gpsro_to_obsseq.csh once to set the DART_DIR to where you have
downloaded the DART distribution.  (There are a few additional options
in this script, but the distribution version should be good for most users.)
If you are downloading data from the CDAAC web site, set your
web site user name and password.  After this you should be able to
ignore this script.


3) convert_many_gpsro.csh:

A wrapper script that calls the converter script a day at a time.
Set the days of data you want to download/convert/remove.  See the
comments at the top of this script for the various options to set.
Rerun this script for all data you need.  This script depends on
the advance_time executable, which should automatically be built
in the ../work directory, but you may have to copy or link to a
```

(continues on next page)

(continued from previous page)

```
version from this dir.  you also need a minimal input.nml here:

&utilities_nml
 /

is all the contents it needs.


It can be risky to use the automatic delete/cleanup option - if there are
any errors in the script or conversion (file system full, bad file format,
etc) and the script doesn't exit, it can delete the input files before
the conversion has succeeded.  But if you have file quota concerns
this allows you to keep the total disk usage lower.
```

Making DART Observations from Ionospheric ionPrf Profiles:

```
0) run quickbuild.sh as described above

1) iono_to_obsseq.csh

set the start and stop days.  downloads from the CDAAC and
untars into 100s of files per day.  runs the converter to
create a single obs_seq.ion.YYYYMMDD file per day.

2) split_obs_seq.csh

split the daily files into X minute/hour files - set the
window times at the top of the file before running.
```

Notes on already converted observations on the NCAR supercomputers
GPS Radio Occultation Data:

```
See /glade/p/image/Observations/GPS

These are DART observation sequence files that contain
radio-occultation measurements from the COSMIC
(and other) satellites.

Uses temperature/moisture bending of the signals as they
pass through the atmosphere between GPS source satellites
```

(continues on next page)

(continued from previous page)

```
and low-earth-orbit receiving satellites to compute the
delay in the arrival of data. the files also contain the
bending angle data, but we are not using that currently.


the subdirectories include:

local -- original processed files, single obs at nadir
local-cosmic2013 -- reprocessed by CDAAC in 2013
local-test2013 -- 2013 data, denser in vertical, diff errors
local-complete2013 - all satellites available for that time,
 new errors (from lydia c), 2013 cosmic reprocessed data
nonlocal -- original processed files, ray-path integrated
rawdata -- netcdf data files downloaded from the CDAAC

local: the ob is at a single location (the tangent point
of the ray and earth) and the entire effect is assumed
to be impacting the state at that point.

non-local: computes the ob value by doing a line integral
along the ray path to accumulate the total effect.

(in our experiments we have compared both and did not see
a large difference between the two methods, and so have
mistly used the local version because it's faster to run.)


some directories contain only the gps obs and must be
merged (with the obs_sequence_tool) with the rest of
the conventional obs before assimilation.

some directories contain both the gps-only files and
the obs merged with NCEP and ACARS data.


if a directory exists but is empty, the files are
likely archived on the HPSS.  see the README files
in the next level directory down for more info on
where they might be.

nsc
jan 2016
```

Ionosphere Data:

```
See /glade/p/image/Observation/ionosphere

These are COSMIC 'ionPrf' ionospheric profile observations.
```

(continues on next page)

(continued from previous page)

```
They are downloaded from the CDAAC website as daily tar files
and unpacked into the 'raw' directory.  They distribute these
observations with one profile per netcdf file.  Each profile has
data at ~500-1000 different levels.

Our converter has a fixed number of levels in the namelist
and we interpolate between the two closest levels to get the
data for that level.  If you give the converter a list of
input netcdf files it will convert all of them into a
single output file.

The 'daily' directory is a collection of all the profiles for
that day.

The 'convert' directory has the executables and scripting
for breaking up the daily files into 10 minute files which
are put in the '10min' directory.  Change the 'split_obs_seq.csh'
script to change the width of this window, or the names of
the output files.

The 'verify.csh' script prints out any missing files, which
happens if there are no profiles in the given window.

Our convention is to make a 0 length file for missing intervals
and we expect the filter run script to look at the file size
and loop if there is a file but with no contents.  This will
allow us to distinguish between a time where we haven't converted
the observations and a time where there are no observations.
In that case the script should add time to the next model
advance request and loop to the next interval.
```

6.76.6 Modules used

convert_cosmic_gps_cdf and convert_cosmic_ionosphere use the same set of modules.

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/modules/assimilation/adaptive_inflate_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/io/direct_netcdf_mod.f90
assimilation_code/modules/io/io_filenames_mod.f90
assimilation_code/modules/io/state_structure_mod.f90
assimilation_code/modules/io/state_vector_io_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/distributed_state_mod.f90
```

(continues on next page)

(continued from previous page)

```
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/netcdf_utilities_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/null_win_mod.f90
assimilation_code/modules/utilities/options_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
models/template/model_mod.f90
models/utilities/default_model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/forward_operators/obs_def_utilities_mod.f90
observations/obs_converters/utilities/obs_utilities_mod.f90
```

6.76.7 Errors

The converters have a parameter declaring the maximum number of desired levels as 200. If more than 200 levels are entered as input (to `obs_levels`), a rather uninformative run-time error is generated:

```
ERROR FROM:
 routine: check_namelist_read
 message:  INVALID NAMELIST ENTRY:  / in namelist convert_cosmic_ionosphere_nml
```

Your error may be different if `obs_levels` is not the last namelist item before the slash '/'

6.76.8 Known Bugs

Some COSMIC files seem to have internal times which differ from the times encoded in the filenames by as much as 2-3 minutes. If it is important to get all the observations within a particular time window files with filenames from a few minutes before and after the window should be converted. Times really outside the window can be excluded in a separate step using the *program obs_sequence_tool*.

6.77 Oklahoma Mesonet MDF Data

6.77.1 Overview

Program to convert Oklahoma Mesonet MDF files into DART observation sequence files.

6.77.2 Data sources

The observation files can be obtained from the Oklahoma Mesonet archive using urls of the format: `http://www.mesonet.org/index.php/dataMdfMts/dataController/getFile/YYYYMMDDHHMM/mdf/TEXT` where YYYYMMDDHHMM is the date and time of the desired set of observations. Files are available every 5 minutes.

If you are located outside of Oklahoma or are going to use this for a non-research purpose see this web page for information about access: http://www.mesonet.org/index.php/site/about/data_access_and_pricing

Static fields are drawn from the station description file provided by the OK Mesonet. Update the local file from: http://www.mesonet.org/index.php/api/siteinfo/from_all_active_with_geo_fields/format/csv

6.77.3 Programs

The programs in the `DART/observations/ok_mesonet/` directory extract data from the distribution files and create DART observation sequence (obs_seq) files. Build them in the `work` directory by running the `./quickbuild.sh` script. In addition to the converters, the `advance_time` and `obs_sequence_tool` utilities will be built.

The converter is a preliminary version which has no namelist inputs. It has hard-coded input and output filenames. It always reads a data file named `okmeso_mdf.in` and creates an output file named `obs_seq.okmeso`. The converter also requires a text file with the location of all the observating stations, called `geoinfo.csv`.

The converter creates observations of the following types:

- LAND_SFC_ALTIMETER
- LAND_SFC_U_WIND_COMPONENT
- LAND_SFC_V_WIND_COMPONENT
- LAND_SFC_TEMPERATURE
- LAND_SFC_SPECIFIC_HUMIDITY
- LAND_SFC_DEWPOINT
- LAND_SFC_RELATIVE_HUMIDITY

Example data files are in the `data` directory. Example scripts for converting batches of these files are in the `shell_scripts` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the scripts saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

The expected usage pattern is that a script will copy, rename, or make a symbolic link from the actual input file (which often contains a timestamp in the name) to the fixed input name before conversion, and move the output file to an appropriate filename before the next invocation of the converter. If an existing observation sequence file of the same output name is found when the converter is run again, it will open that file and append the next set of observations to it.

6.78 VIIRS/AQUA Satellite Ocean Color

6.78.1 Overview

The OCEAN COLOR (NASA's ocean color web) supports the collection, processing, calibration, archive and distribution of ocean-related products from a number of missions which are supported within the framework and facilities of the NASA Ocean Data Processing System. This converter support ocean color data from two different missions:

1. AQUA/MODIS (or Moderate Resolution Imaging Spectroradiometer) is a key instrument aboard the Terra (EOS AM) and Aqua (EOS PM). Terra MODIS and Aqua MODIS view the entire Earth's surface every 2 days.

2. VIIRS (or Visible and Infrared Imager/Radiometer Suite) is a multi-disciplinary instrument that is being flown on the Joint Polar Satellite System. VIIRS is the successor to MODIS for Earth Science data product generation.

6.78.2 Data source

In order to have access to the data, you will need a log in on: https://earthdata.nasa.gov/

Once a username and a password have been created, the ocean color data can be downloaded from: data-access-webpage where a manual file search can be performed or using the provided script: `shell_scripts/get_ocdata.sh`

6.78.3 Programs and Scripts

The programs and scripts in the `observations/obs_converters/ocean_color` directory download the data and create DART observation sequence (obs_seq) files. All programs are built in the `ocean_color/work` directory and can be done by running `./quickbuild.sh` In addition to the main converter (i.e., `convert_sat_chl`), other programs such `advance_time` and `preprocess` utilities will be built.

Converter namelist `convert_sat_chl_nml`: This namelist is added to the rest of DART program namelists in file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'.

```
&convert_sat_chl_nml
   file_in         = '../data/V2020350.L3m_DAY_SNPP_CHL_chlor_a_4km.nc'
   file_out        = '../data/obs_seq_chl'
   chl_thresh      = 0.03
   subsample_intv  = 1
   special_mask    = .true.
   debug           = .false.
   /
```

Contents	Type	Description
file_in	character	Name of the input netcdf data file. Example of absolute path: `$DART/observations/obs_converters/ocean_color/data/V2020336.L3m_DAY_SNPP_CHL_4km.nc`
file_out	character	Partial filename for the output file. The date and time are appended to `file_out` to construct a unique filename reflecting the time of the observations in the file.
chl_thresh	real(r8)	When the observed chlorophyll values are small, a threshold value is used for the obs. Example: `chl_thresh = 0.03`
subsample_intv	integer	It is possible to 'thin' the observations. `subsample_intv` allows one to take every nth observation.
special_mask	logical	A simple procedure to ignore data in certain areas. Users can the code in convert_sat_chl.f90 to change the mask location
debug	logical	Print extra information during the `convert_sat_chl` execution.

It's widely known that chlorophyll follows a log-normal distribution. After reading the observed chlorophyll data from the input netcdf file, a **LOG10** transformation is applied such that the resulting distribution is normal. The idea behind this step is to prepare the data for the EnKF which assumes both the state and observations to be Gaussian.

The `get_ocdata.sh` script is placed inside the `shell_scripts` directory. Technically, this is the only the script that the user needs to run. Prior to running the script, one should edit some parameters such as: date range, the resolution of the data, the instrument, username and password to access the data site, frequency, domain area coordinates, etc. These parameters are hard coded in the script. The tasks that this script do are:

1. It Downloads ocean color data for the requested period and frequency one file at a time.
2. It retains the chlorophyll variable for the requested domain and gets rid of unnecessary information in the netcdf data files.
3. It modifies the netcdf files by adding a time dimension with the necessary attributes.
4. It runs the obs converter `convert_sat_chl` for each data file.

So, each data file will have a corresponding obs_seq file. The observation sequence files are named such that the observation time is part of the file name. For example: `obs_seq_chl.2020-12-15-00000`

This script is *NOT* intended to be turnkey script; it will certainly needs to be customized for your use. This script will make use of some DART utilities and namelists such as `advance_time` and `input.nml`. If an existing observation sequence file of the same output name is found when the converter is run again, it will replace that file.

6.79 QuikSCAT SeaWinds Data

6.79.1 Overview

NASA's QuikSCAT mission is described in http://winds.jpl.nasa.gov/missions/quikscat/. "QuikSCAT" refers to the satellite, "SeaWinds" refers to the instrument that provides near-surface wind speeds and directions over large bodies of water. QuikSCAT has an orbit of about 100 minutes, and the SeaWinds microwave radar covers a swath under the satellite. The swath is comprised of successive scans (or rows) and each scan has many wind-vector-cells (WVCs). For the purpose of this document, we will focus only the **Level 2B** product at 25km resolution. If you go to the official JPL data distribution site http://podaac.jpl.nasa.gov/DATA_CATALOG/quikscatinfo.html , we are using the product labelled **L2B OWV 25km Swath**. Each orbit consists of (potentially) 76 WVCs in each of 1624 rows or scans. The azimuthal diversity of the radar returns affects the error characteristics of the retrieved wind speeds and directions, as does rain, interference of land in the radar footprint, and very low wind speeds. Hence, not all wind retrievals are created equal.

The algorithm that converts the 'sigma naughts' (the measure of radar backscatter) into wind speeds and directions has multiple solutions. Each candidate solution is called an 'ambiguity', and there are several ways of choosing 'the best' ambiguity. Beauty is in the eye of the beholder. At present, the routine to convert the original L2B data files (one per orbit) in HDF format into the DART observation sequence file makes several assumptions:

1. All retrievals are labelled with a 10m height, in accordance with the retrieval algorithm.
2. Only the highest-ranked (by the MLE method) solution is desired.
3. Only the WVCs with a wvc_quality_flag of **zero** are desired.
4. The mission specification of a wind speed rms error of 2 ms (for winds less than 20 m/s) and 10% for windspeeds between 20 and 30 m/s can be extended to all winds with a qc flag of zero.
5. The mission specification of an error in direction of 20 degrees rms is applicable to all retrieved directions.
6. All retrievals with wind speeds less than 1.0 are not used.
7. The above error characterstics can be simplified when deriving the horizontal wind components (i.e. U,V). **Note :** this may or may not be a good assumption, and efforts to assimilate the speed and direction directly are under way.

6.79.2 Data sources

The NASA Jet Propulsion Laboratory (JPL) data repository has a collection of animations and data sets from this instrument. In keeping with NASA tradition, these data are in HDF format (specifically, HDF4), so if you want to read these files directly, you will need to install the HDF4 libraries (which can be downloaded from http://www.hdfgroup.org/products/hdf4/)

If you go to the official JPL data distribution site http://podaac.jpl.nasa.gov/DATA_CATALOG/quikscatinfo.html, we are using the product labelled **L2B OWV 25km Swath**. They are organized in folders by day ... with each orbit (each revolution) in one compressed file. There are 14 revolutions per day. The conversion to DART observation sequence format is done on each revolution, multiple revolutions may be combined 'after the fact' by any `obs_sequence_tool` in the `work` directory of any model.

6.79.3 Programs

There are several programs that are distributed from the JPL www-site, ftp://podaac.jpl.nasa.gov/pub/ocean_wind/quikscat/L2B/sw/; we specifically started from the Fortran file read_qscat2b.f and modified it to be called as a subroutine to make it more similar to the rest of the DART framework. The original `Makefile` and `read_qscat2b.f` are included in the DART distribution in the `DART/observations/quikscat` directory. You will have to modify the `Makefile` to build the executable.

convert_L2b.f90

`convert_L2b` is the executable that reads the HDF files distributed by JPL. `DART/observations/quikscat/work` has the expected `mkmf_convert_L2b` and `path_names_convert_L2b` files and compiles the executable in the typical DART fashion - with one exception. The location of the HDF (and possible dependencies) installation must be conveyed to the `mkmf` build mechanism. Since this information is not required by the rest of DART, it made sense (to me) to isolate it in the `mkmf_convert_L2b` script. **It will be necessary to modify the ``mkmf_convert_L2b`` script to be able to build ``convert_L2b``**. In particular, you will have to change the two lines specifying the location of the HDF (and probably the JPG) libraries. The rest of the script should require little, if any, modification.

set JPGDIR = */contrib/jpeg-6b_gnu-4.1.2-64* set HDFDIR = */contrib/hdf-4.2r4_gnu-4.1.2-64*

There are a lot of observations in every QuikSCAT orbit. Consequently, the observation sequence files are pretty large - particularly if you use the ASCII format. Using the binary format (i.e. *obs_sequence_nml:write_binary_obs_sequence = .true.*) will result in observation sequence files that are about *half* the size of the ASCII format.

Since there are about 14 QuikSCAT orbits per day, it may be useful to convert individual orbits to an observation sequence file and then concatenate multiple observation sequence files into one file per day. This may be trivially accomplished with the `obs_sequence_tool` program in any `model/xxxx/work` directory. Be sure to include the `'../../../obs_def/obs_def_QuikSCAT_mod.f90'` string in `input.nml&preprocess_nml:input_files` when you run `preprocess`.

Obs_to_table.f90, plot_wind_vectors.m

`DART/diagnostics/threed_sphere/obs_to_table.f90` is a potentially useful tool. You can run the observation sequence files through this filter to come up with a 'XYZ'-like file that can be readily plotted with `DART/diagnostics/matlab/plot_wind_vectors.m`.

6.79.4 Namelist

This namelist is read from the file `input.nml`. We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. The following values are the defaults for these namelist items.

```
&convert_L2b_nml
   l2b_file = '',
   datadir = '.',
   outputdir = '.',
   lon1 = 0.0,
   lon2 = 360.0,
   lat1 = -90.0,
   lat2 = 90.0,
   along_track_thin = 0,
   cross_track_thin = 0
 /
```

It is possible to restrict the output observation sequence to contain data from a region of interest throught the use of the namelist parameters. If you need a region that spans the Prime Meridian lon1 can be a larger number than lon2, for example, a region from 300 E to 40 E and 60 S to 30 S (some of the South Atlantic), would be *lon1 = 300, lon2 = 40, lat1 = -60, lat2 = -30*.

Contents	Type	Description
l2b_file	character(len=128)	name of the HDF file to read - NOT including the directory, e.g. QS_S2B44444.20080021548
datadir	character(len=128)	the directory containing the HDF files
outputdir	character(len=128)	the directory for the output observation sequence files.
lon1	real(r4)	the West-most longitude of interest in degrees. [0.0, 360]
lon2	real(r4)	the East-most longitude of interest in degrees. [0.0, 360]
lat1	real(r4)	the South-most latitude of interest in degrees. [-90.0, 90.0]
lat2	real(r8)	the North-most latitude of interest in degrees. [-90.0, 90.0]
along_track_thin	integer	provides ability to thin the data by keeping only every Nth row. e.g. 3 == keep every 3rd row.
cross_track_thin	integer	provides ability to thin the data by keeping only every Nth wind vector cell in a particular row. e.g. 5 == keep every 5th cell.

Future Plans

1. There is one bit of error-checking that did not survive the conversion from F77 to F90. I need to restore the check that the HDF file being read is a 'Level 2B' product.
2. There is a lot of error-checking that is not being done. I need to bulletproof the code more.
3. We need namelist options to select something other than the highest-ranked ambiguity.
4. We need namelist options to select more QC flags - not just the ones with the 'perfect' QC value of 0
5. Add an option to leave the observations as speed and direction instead of converting them to U,V components. This is a natural implementation of the instrument error characteristics. However, it would require writing a specialized forward operator in order to assimilate obs of this type (speed, direction), and there is still a numerical problem with trying to do the statistics required during the assimilation of a cyclic direction value.

6.80 Even Sphere

It is frequently useful to generate a series of synthetic observations located at roughly evenly-distributed locations[1] on a sphere.

There are three methods described here.

1. A Matlab script and the standard DART observation generation utilities.
2. A csh script with all the parts of 1. (Not available for all models).
3. A stand-alone *Fortran* program.

The Fortran program does not generate the nice plots that the Matlab process does, but it may be faster and easier to automate for generating a large number of obs.

6.80.1 Matlab Scripts Plus Standard DART Observation Executables

This involves multiple steps:

1. determine how many locations are needed
2. determine the vertical levels needed
3. run the MATLAB function *even_sphere.m* to create the text file containing the input for `create_obs_sequence`
4. run *program create_obs_sequence* to create an observation sequence (usually `set_def.out`, although it is possible to create `obs_seq.out` files directly if you don't really care about the observation values).
5. if desired, run *program create_fixed_network_seq* to create a longer observation sequence file.
6. run *program perfect_model_obs* to harvest the synthetic observations from a chosen model.

This directory contains a MATLAB function (`even_sphere.m`) that generates input for the *program create_obs_sequence* . After executing `create_obs_sequence`, the resulting observation sequence file will have a template for 'RADIOSONDE_TEMPERATURE','RADIOSONDE_U_WIND_COMPONENT',and 'RADIOSONDE_V_WIND_COMPONENT' observations at specified pressure levels and roughly evenly-distributed locations across the entire globe. Optionally, bogus observation values may also inserted; which may be useful in certain circumstances.

[1] A python example of the Golden Section spiral algorithm can be found in https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere See the contribution from Fab von Bellinghousen.

6.80.2 even_sphere.m

`even_sphere.m` has many optional arguments to tailor its behavior. It has exactly 1 required argument - the number of horizontal locations desired.

- It will create a text file `even_create_input` to be used as input to `create_obs_sequence`
- The choice of pressure levels is described *here*.
- The default number of pressure levels is 21. Argument `nlevels` specifies how many levels to use from the beginning of the levels list.
- The default observation error variances for each observation type are level-dependent and are consistent with `DART/observations/obs_converters/obs_error/ncep_obs_err_mod.f90`
- The default is to create 'empty' observation sequences - i.e. they have no actual observation values and are suitable to be used with `perfect_model_obs`
- The default date of the observations is 2017-12-25 00:00:00
- A plot of the locations will be created. The number of gridlines is configurable but defaults to 288 in longitude and 192 in latitude.
- **All** the defaults can be changed by specifying 'variable-value' pairs of options, as described below. Examples of some options are also available via the normal MATLAB *help* facility. (Documenting all of them in the *help* makes the help page too long.)

Note that the number of observations will be the number of locations * the number of vertical levels * the number of variables (i.e. 3) `even_sphere.m` also takes observation error variances and includes them in the observation sequences.

6.80.3 Optional Argument Variable-Value pairs

The optional variable-value pairs can appear in any order.

optional variable	example value	Description
'nlevels'	5	number of pressure levels to use. May be less than the length of the 'levels' array, but cannot be more.
levels	[1000 500 300 200 100]	pressure levels desired. see *Levels* section for discussion.
T_error_var	[1.44 0.64 0.81 1.44 0.64]	level-specific Temperature error variances. see *Levels* section for discussion.
W_error_var	[1.96 4.41 9.00 7.29 4.41]	level-specific error variances for both U, V wind components. see *Levels* section for discussion.
'YMD'	'2017-12-25'	Date required for `create_obs_sequence`. If `create_fixed_network_seq` is run, this time is replaced.
fill_obs	false	'true' inserts a bogus observation value of 1.0 and a bogus QC value of 0.'false' does not insert bogus values and essentially creates an empty obs sequence file (typically `set_def.out`)
'nlon'	288	number of longitude grid lines in plot
'nlat'	192	number of latitude grid lines in plot

6.80.4 Examples

1. 30 horizontal locations at 6 pressure levels:

```
nprofiles   = 30;
levels      = [1000  850  500  300  200  100];
T_error_var = [1.44 0.64 0.64 0.81 1.44 0.64];
W_error_var = [1.96 2.25 4.41 9.00 7.29 4.41];
even_sphere(nprofiles, 'levels', levels, ...
           'T_error_var', T_error_var, 'W_error_var', W_error_var)
```

2. 30 horizontal locations at 3 pressure levels. Note that the *nlevels* argument specifies that only the first 3 pressure levels are used even though there are 6 potential pressure levels. Similarly, only the matching error variances are used.

```
nprofiles   = 30;
nlevels     = 3 ;
levels      = [1000  850  500  300  200  100];
T_error_var = [1.44 0.64 0.64 0.81 1.44 0.64];
W_error_var = [1.96 2.25 4.41 9.00 7.29 4.41];
even_sphere(nprofiles, 'nlevels', nlevels, 'levels', levels, ...
           'T_error_var', T_error_var, 'W_error_var', W_error_var)
```

6.80.5 Levels

Attention: If you need realistic error variances attached to your observations, be careful to align your levels and variances.

The default levels that this program generates are the *mandatory pressure levels* defined in the AMS glossary. The corresponding error variances are from ncep_obs_err_mod. See *Obs Error*. Levels at the top can be excluded by setting *nprofiles* < 21 (size(levels)).

```
levels      = [1000  925  850  700  500  400  300   250  200  150  100   70   50   30  ␣
↪20   10    7    5    3    2    1];
T_error_var = [1.44 1.00 0.64 0.64 0.64 0.64 0.81  1.44 1.44 1.00 0.64 0.64 0.81 1.00 1.
↪69 2.25 2.25 2.25 2.25 2.25 2.25];
W_error_var = [1.96 2.25 2.25 2.56 4.41 6.76 9.00 10.24 7.29 5.76 4.41 4.41 4.41 4.41 4.
↪41 4.41 4.41 4.41 4.41 4.41 4.41];
```

Here's an example of replacing the AMS levels with a set that has more levels near the surface and none above 150 hPa. Note that the error variances should change to be consistent with the levels.

```
levels      = [1000  950  900  850  800  750  700  650  600  550  500  400  300  200  ␣
↪150];
T_error_var = [1.44 1.21 0.81 0.64 0.64 0.64 0.64 0.64 0.64 0.64 0.64 0.64 0.81 1.44 1.
↪00];
W_error_var = [1.96 2.25 2.25 2.25 2.56 2.56 2.56 3.24 3.61 4.00 4.41 6.76 9.00 7.29 5.
↪76];
```

6.80.6 Running Matlab in Batch Mode

If you would prefer to run `even_sphere.m` in batch mode (i.e. from within a shell script), here is an example syntax that worked for me. The script ran in the same directory as `even_sphere.m`. There are many ways to construct the input, naturally - but you don't have to explicitly edit `even_sphere.m` this way.

```
#!/bin/csh

\rm -rf matlab_input.m

cat >> matlab_input.m << EndOfInput

   nprofiles   = 30;
   levels      = [1000  850  500  300  200  100];
   T_error_var = [1.44 0.64 0.64 0.81 1.44 0.64];
   W_error_var = [1.96 2.25 4.41 9.00 7.29 4.41];
   even_sphere(nprofiles, 'levels', levels, ...
               'T_error_var', T_error_var, 'W_error_var', W_error_var)
   fname = sprintf('even_sphere_%d_profiles',nprofiles);
   orient landscape
   print(fname,'-dpdf')

EndOfInput

matlab -nosplash -nodesktop -r "try; cd $PWD; matlab_input; catch; end; exit";
```

6.80.7 Automation Scripts

Here there are also scripts (`run_fixed_network_*.csh`) which use the output from `create_obs_sequence` and the *program create_fixed_network_seq* to generate a series of observation sequence files.

run_fixed_network_seq.csh

Calls `create_fixed_network_seq` to create a separate file for each time period. By default, it makes 2 files/day, 12 hours apart, single time per file. The intervals and dates can be changed by editing the script. It assumes that `create_fixed_network` has any model-specific files it needs in this directory. It requires a `set_def.out` file (usually created by `create_obs_sequence`).

run_fixed_network_daily.csh

Calls `create_fixed_network_seq` to create a separate file for each time period. By default it makes 1 file/day, single time (noon) per file. The dates and time can be changed by editing the script. It assumes that `create_fixed_network` has any model-specific files it needs in this directory. It requires a `set_def.out` file (usually created by `create_obs_sequence`).

The process, end to end:

MATLAB:

Set the number of profiles, the levels, etc. and run `even_sphere.m` in MATLAB. It creates the necessary text file `even_create_input` for the next step. It will also make a plot - which you can save.

DART:

Then you have a choice about building and running the `create_obs_sequence` and `create_fixed_network_seq` programs:

A. building them in the `models/template/work` directory

B. using the ones which were built in `models/your_model/work` directory by `quickbuild.sh`.

Choice A uses programs which have no model specific file dependencies, but may involve more separate steps than B.

A

1. Build the programs in `template/work`
2. Link (or copy) these files to the directory in which you want to create obs_seq files.

```
./even_create_sequence
./run_fixed_network_{seq or daily}.csh
models/template/work/create_fixed_network_seq
models/template/work/create_obs_sequence
models/template/work/input.nml
```

3. In your obs_seq directory, run create_obs_sequence, which creates a `set_def.out` file.

```
./create_obs_sequence < even_create_input > /dev/null
```

4. Edit and run your choice of `run_fixed_network_*.csh` for the desired dates. These call create_fixed_network_seq, which creates an `obs_seq.in` file for each specified date.

B

This choice may involve fewer steps, *if* there is a model specific script which combines the steps in A). See the cam-fv example (models/cam-fv/shell_scripts/synth_obs_locs_to_seqs.csh). If there is *not* a script like that for your model, you can follow the steps in A), substituting your model name for the "template" in the pathnames. NOTE: you may need to link any additional input files which your model requires into the directory where you will run the programs. These typically contain grid information and are found in `your_model/work`. For example, *cam-fv* needs a `caminput.nc` and `cam_phis.nc`.

6.80.8 Fortran program for generating obs directly

cd into the work directory and run `quickbuild.sh`.

This builds the `create_even_sphere` executable. Edit the `input.nml` to set the number of obs to generate and the date in the namelist. Run the program and the output file will be generated.

DETAILS of generating points evenly-distributed on a sphere

This is the algorithm that's being used[Page 209, 1]:

```
N     := the number of profiles you want
dlong := pi*(3-sqrt(5))  /* ~2.39996323 */
dy    := 2.0/N
phi   := 0
y     := 1 - dy/2
```

(continues on next page)

(continued from previous page)

```
for k := 0 .. N-1
    r       := sqrt(1-y*y)
    node[k] := (cos(phi)*r, sin(phi)*r, y)
    y       := y - dy
    phi     := phi + dlong
```

For the geometric and visually minded:

1. Picture a unit sphere in cartesion space (x,y,z).
2. Choose a value -1 < y < 1, which defines an x-z plane. That plane intersects with the unit sphere to form a circle whose center is on the y axis. (The circle radius is small near y = +/-1 and is 1 at y=0.)
3. Choose an angle ("phi") and draw a ray from the center of the circle to a point on the circle using this angle relative to the x positive direction. Where the ray intersects the circle (and sphere) is one of the evenly distributed points on the sphere which we want.
4. Its x and z coordinates can then be combined with the already defined y coordinate to define the cartesian location of the point.
5. The choice of the y and angle for each point is where the magic enters the algorithm. They are derived from the Fibonacci or Golden Spiral formula (derived elsewhere).

6.81 Obs Error

This directory is where to add modules that compute/set the observational errors for different types of real-world observations.

For the 2 existing files, the data source is:

ECMWF errors: http://www.ecmwf.int/research/ifsdocs/CY25r1/Observations/Observations-03-3.html

NCEP errors: a 2005 version of the GFS observation error tables.

(Note that the return values from these modules should be the ERROR STANDARD DEVIATION. In the obs_seq files, the value stored with each observation will be the variance.)

Each center uses different errors, and these separate files make it easy to collect these values in one place, and switch them in and out depending on the needs of the user who is creating new obs_seq files for DART.

Anyone who wants to contribute another error module is more than welcome to add files here.

IMPORTANT: Each file should have the same module name; i.e. the source file names will differ but the module name inside the file must be the same across all modules in this directory.

All the subroutines must also have the same names and calling sequence. They must return appropriate values for each observation type that is required. If errors for a new observation type is added, it should be added to all the files in this directory.

This way the user can change between error values by editing the filename in the path_names_xxx files and recompiling without changing the code.

Thanks to Ryan Torn for the idea and initial contributions.

6.82 Radar observations

6.82.1 Overview

DART provides limited support for the conversion of radar observations to `obs_seq` format. As an end goal, you want to assimilate radar observations that:

- Have been quality controlled to remove non-meteorological scatterers and other artifacts
- Have horizontal resolution that has been reduced to approximately twice the expected horizontal grid spacing of your model. For example, if your model has 3 km grid spacing, you should reduce your radar observations to every 6 km, interpolated along the sweep plane.

Reflectivity observations are often partitioned into two types:

1. Regular reflectivity observations
2. Clear-air reflectivity observations where no radar echoes are observed.

Quality control is best done with raw data. You should have an ability to perform quality control before converting your observations to obs_seq format.

Synthetic radar observations

The `create_obs_radar_sequence` program generates one or more sets of synthetic WSR-88D (NEXRAD) radar observations. It can generate reflectivity and/or doppler radial velocity observations with clear-air or storm sweep patterns. These synthetic observations can be used for testing your assimilation setup or for conducting Observing System Simulation Experiments (OSSEs).

To build `create_obs_radar_sequence`, change directory into the `work` subdirectory, ensure `input.nml` is configured properly and run the build script:

```
cd work
./quickbuild.sh
```

Real radar observations

Once you have ensured that your data are quality controlled, use the Observation Processing And Wind Synthesis (OPAWS) utility convert your data to *obs_seq* format. The OPAWS utility reads specific types of files as input, such as DORADE sweep files and NCAR EOL Foray data.

OPAWS analyzes and grids data in either:

- two-dimensions (on the conical surface of each sweep), or
- three-dimensions (Cartesian).

If your raw data are not in such a format, additional utilities are available for conversion such as the RADX library which is part of the LIDAR/RADAR Open Software Environment.

6.82.2 Guidance for Weather Research and Forecasting (WRF) users

If you intend to assimilate radar observations into WRF, you'll need to make some code modifications to allow for forward operator calculations. For reflectivity, most of the available microphysics schemes have built-in capability to output reflectivity, assuming a 10 cm wavelength. If you are not using an S-band radar, be aware that attenuation is not accounted for in the built-in reflectivity operator.

For radial velocity, you will also need to generate a new diagnostic field: terminal fall velocity. There is very limited support for fall velocity in WRF, although it is partially supported in the Thompson microphysics scheme.

Note: You will still need to modify WRF code to get this diagnostic output to history files.

With these two fields available in your WRF history files, you can add them to your DART *wrf_state_variables* list.

You should also use a special localization radius for radar observations, typically 12-24 km. If you leave range-folding in your radar observations, you will need to build the special version of DART that unfolds the velocity observations on-the-fly.

With all of those configurations in place, you will be ready to assimilate radar observations using WRF and DART.

For more information, see the WRF tests directory in `DART/models/wrf/regression/Radar/` for pointers to data to run a radar test case.

6.83 PROGRAM `snow_to_obs`

6.83.1 MODIS snowcover fraction observation converter

Overview

There are several satellite sources for snow observations. Generally the data is distributed in HDF-EOS format. The converter code in this directory DOES NOT READ HDF FILES as input. It expects the files to have been preprocessed to contain text, one line per observation, with northern hemisphere data only.

6.83.2 Data sources

not sure.

6.83.3 Programs

The `snow_to_obs.f90` file is the source for the main converter program.

To compile and test, go into the work subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

This converter creates observations of the "MODIS_SNOWCOVER_FRAC" type.

There is another program in this directory called `snow_to_obs_netcdf.f90` which is a prototype for reading netcdf files that contain some metadata and presumably have been converted from the original HDF. THIS HAS NOT BEEN TESTED but if you have such data, please contact dart@ucar.edu for more assistance. If you write something that reads the HDF-EOS MODIS files directly, please, please contact us! Thanks.

6.83.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&snow_to_obs_nml
  longrid         = 360,
  latgrid         = 90,
  year            = 2000,
  doy             = 1,
  snow_input_file = 'snowdata.input',
  missing_value   = -20.0,
  debug           = .false.
/
```

Item	Type	Description
longrid	integer	The number of divisions in the longitude dimension.
latgrid	integer	The number of divisions in the latitude dimension. This converter assumes the data is for the northern hemisphere only. A namelist item could be added to select northern verses southern hemisphere if needed.
year	integer	The year number of the data.
doy	integer	The day number in the year. Valid range 1 to 365 in a non-leap year, 1 to 366 in a leap year.
snow_input_file	character(len=128)	The name of the input file.
missing_value	real(r8)	The value used to mark missing data.
debug	logical	If set to .true. the converter will print out more information as it does the conversion.

Known Bugs

This program is hardcoded to read only northern hemisphere data. It should handle global values.

Future Plans

This program should use the HDF-EOS libraries to read the native MODIS granule files. Right now the ascii intermediate files contain no metadata, so if the namelist values don't match the actual division of the globe, bad things will happen.

6.84 PROGRAM `text_to_obs`

6.84.1 Text file to DART converter

Overview

If you have observations in spreadsheet or column format, in text, with a single line per observation, then the files this directory are a template for how to convert these observations into a format suitable for DART use.

The workflow is usually:

- read in the needed information about each observation - location, time, data value, observation type - from a data source (usually a file)
- call a series of DART library routines to construct a derived type that contains all the information about a single observation
- call another set of DART library routines to put it into a time-sorted series
- repeat the last 2 steps until all observations are processed
- finally, call a write subroutine that writes out the entire series to a file in a format that DART can read in

It is not recommended that you try to mimic the ascii file format by other means; the format is subject to change and the library routines will continue to be supported even if the physical format changes.

If your input data is in some kind of format like netCDF or HDF, then one of the other converters (e.g. the MADIS ones for netCDF) might be a better starting place for adapting code.

6.84.2 Data sources

This part is up to you. For each observation you will need a location, a data value, a type, a time, and some kind of error estimate. The error estimate can be hardcoded in the converter if they are not available in the input data. See below for more details on selecting an appropriate error value.

6.84.3 Programs

The `text_to_obs.f90` file is the source for the main converter program. Look at the source code where it reads the example data file. You will almost certainly need to change the "read" statement to match your data format. The example code reads each text line into a character buffer and then reads from that buffer to parse up the data items.

To compile and test, go into the work subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

To change the observation types, look in the `DART/observations/forward_operators` directory. If you can find an "obs_def_XXX_mod.f90" file with an appropriate set of observation types, change the 'use' lines in the converter source to include those types. Then add that filename in the `input.nml` namelist file to the &preprocess_nml namelist, the 'input_files' variable. Multiple files can be listed. Then run quickbuild.sh again. It remakes the table of supported observation types before trying to recompile the source code.

An example script for converting batches of files is in the `shell_scripts` directory. A tiny example data file is in the `data` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the script saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

6.84.4 Decisions you might need to make

See the discussion in the *Creating an obs_seq file from real observations* page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.85 Total Precipitable Water Observations

6.85.1 Overview

Several satellites contain instruments that return observations of integrated Total Precipitable Water (TPW). There are two MODIS Spectroradiometers, one aboard the TERRA satellite, and the other aboard the AQUA satellite. There is also an AMSR-E instrument on the AQUA satellite.

These instruments produce a variety of data products which are generally distributed in HDF format using the HDF-EOS libraries. The converter code in this directory IS NOT USING THESE FILES AS INPUT. The code is expecting to read ASCII TEXT files, which contain one line per observation, with the latitude, longitude, TPW data value, and the observation time. The Fortran read line is:

```
read(iunit, '(f11.6, f13.5, f10.4, 4x, i4, 4i3, f7.3)') &
          lat, lon, tpw, iyear, imonth, iday, ihour, imin, seconds
```

No program to convert between the HDF and text files is currently provided. Contact dart@ucar.edu for more information if you are interested in using this converter.

6.85.2 Data sources

This converter reads files produced as part of a data research effort. Contact dart@ucar.edu for more information if you are interested in this data.

Alternatively, if you can read HDF-EOS files and output a text line per observation in the format listed above, then you can use this converter on TPW data from any MODIS file.

6.85.3 Programs

The programs in the `DART/observations/tpw` directory extract data from the distribution text files and create DART observation sequence (obs_seq) files. Build them in the `work` directory by running the `./quickbuild.sh` script. In addition to the converters, several other general observation sequence file utilities will be built.

Generally the input data comes in daily files, with the string YYYYMMDD (year, month, day) as part of the name. This converter has the option to loop over multiple days within the same month and create an output file per day.

Like many kinds of satellite data, the TWP data is dense and generally needs to be subsampled or averaged (super-ob'd) before being used for data assimilation. This converter will average in both space and time. There are 4 namelist items (see the namelist section below) which set the centers and widths of time bins for each day. All observations within a single time bin are eligible to be averaged together. The next available observation in the bin is selected and any other remaining observations in that bin that are within delta latitude and delta longitude of it are averaged in both time and space. Then all observations which were averaged are removed from the bin, so each observation is only averaged into one output observation. Observations that are within delta longitude of the prime meridian are handled correctly by averaging observations on both sides of the boundary.

It is possible to restrict the output observation sequence to contain data from a region of interest using namelist settings. If your region spans the Prime Meridian min_lon can be a larger number than max_lon. For example, a region from 300 E to 40 E and 60 S to 30 S (some of the South Atlantic), specify *min_lon = 300, max_lon = 40, min_lat = -60, max_lat = -30*. So 'min_lon' sets the western boundary, 'max_lon' the eastern.

The specific type of observation created in the output observation sequence file can be select by namelist. "MODIS_TOTAL_PRECIPITABLE_WATER" is the most general term, or a more satellite-specific name can be chosen. The choice of which observations to assimilate or evaluate are made using this name. The observation-space diagnostics also aggregate statistics based on this name.

6.85.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&convert_tpw_nml
   start_year           = 2008
   start_month          = 1
   start_day            = 1
   total_days           = 31
   max_obs              = 150000
   time_bin_start       = 0.0
   time_bin_interval    = 0.50
   time_bin_half_width = 0.25
   time_bin_end         = 24.0
   delta_lat_box        = 1.0
   delta_lon_box        = 1.0
   min_lon              =   0.0
   max_lon              = 360.0
   min_lat              = -90.0
   max_lat              =  90.0
   ObsBase              = '../data'
   InfilePrefix         = 'datafile.'
   InfileSuffix         = '.txt'
   OutfilePrefix        = 'obs_seq.'
   OutfileSuffix        = ''
   observation_name     = 'MODIS_TOTAL_PRECIPITABLE_WATER'
 /
```

Item	Type	Description
start_year	integer	The year for the first day to be converted. (The converter will optionally loop over multiple days in the same month.)
start_month	integer	The month number for the first day to be converted. (The converter will optionally loop over multiple days in the same month.)
start_day	integer	The day number for the first day to be converted. (The converter will optionally loop over multiple days in the same month.)
total_days	integer	The number of days to be converted. (The converter will optionally loop over multiple days in the same month.) The observations for each day will be created in a separate output file which will include the YYYYMMDD date as part of the output filename.
max_obs	integer	The largest number of obs in the output file. If you get an error, increase this number and run again.
time_bin_start	real(r8)	The next four namelist values define a series of time intervals that define time bins which are used for averaging. The input data from the satellite is very dense and generally the data values need to be subsetted in some way before assimilating. All observations in the same time bin are eligible to be averaged in space if they are within the latitude/longitude box. The input files are distributed as daily files, so use care when defining the first and last bins of the day. The units are in hours. This item defines the midpoint of the first bin.
time_bin_interval	real(r8)	Increment added the time_bin_start to compute the center of the next time bin. The units are in hours.
time_bin_half_width	real(r8)	The amount of time added to and subtracted from the time bin center to define the full bin. The units are in hours.
time_bin_end	real(r8)	The center of the last bin of the day. The units are in hours.
delta_lat_box	real(r8)	For all observations in the same time bin, the next available observation is selected. All other observations in that bin that are within delta latitude or longitude of it are averaged together and a single observation is output. Observations which are averaged with others are removed from the bin and so only contribute to the output data once. The units are de-

6.85.5 Known Bugs

The input files are daily; be cautious of time bin boundaries at the start and end of the day.

6.85.6 Future Plans

- This program should use the HDF-EOS libraries to read the native MODIS granule files.
- This program could loop over arbitrary numbers of days by using the time manager calendar functions to increment the bins across month and year boundaries; it could also use the schedule module to define the bins.

6.86 PROGRAM `tc_to_obs`

6.87 Tropical Cyclone ATCF File to DART Converter

6.87.1 Overview

Tropical Cyclone data created by the 'Automated Tropical Cyclone Forecast (ATCF) System' can be converted into DART observations of the storm center location, minimum sea level pressure, and maximum wind speed. Several of the options can be customized at runtime by setting values in a Fortran namelist. See the namelist section below for more details. In the current release of DART only the *WRF* has forward operator code to generate expected obs values for these vortex observations.

This webpage documents many things about the ATCF system and the various file formats that are used for storm track data and other characteristics.

The converter in this directory is only configured to read the packed "b-deck" format (as described on the webpage referenced above). There are sections in the fortran code which can be filled in to read other format variants. This should mostly be a matter of changing the read format string to match the data in the file.

6.87.2 Data sources

A collection of past storm ATCF information can be found here. For each observation you will need a location, a data value, a type, a time, and some kind of error estimate. The error estimates will need to be hardcoded or computed in the converter since they are not available in the input data. See below for more details on selecting an appropriate error value.

6.87.3 Programs

The `tc_to_obs.f90` file is the source for the main converter program. Look at the source code where it reads the example data file. Given the variety of formatting details in different files, you may quite possibly need to change the "read" statement to match your data format. There is a 'select case' section which is intended to let you add more formats and select them at runtime via namelist.

To compile and test, go into the work subdirectory and run the `quickbuild.sh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

This converter creates observation types defined in the `DART/observations/forward_operators/obs_def_vortex_mod.f90` file. This file must be listed in the `input.nml` namelist file, in the `&preprocess_nml` namelist, in the 'input_files' variable, for any programs which are going to process these observations. If you have to change the `&preprocess_nml` namelist you will have to run `quickbuild.sh` again to build and execute the `preprocess` program before compiling other executables. It remakes the table of supported observation types before trying to recompile other source code.

There is an example b-deck data file in the `data` directory. This format is what is supported in the code as distributed. There are other variants of this format which have more spaces so the columns line up, and variants which have many more fields than what is read here.

6.87.4 Specifying expected error

The ATCF files DO NOT include any estimated error values. The source code currently has hardcoded values for location, sea level pressure, and max wind errors. These may need to be adjusted as needed if they do not give the expected results.

6.87.5 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&tc_to_obs_nml
   input_atcf_file         = 'input.txt'
   fileformat              = 'b-deck'
   obs_out_file            = 'obs_seq.out'
   append_to_existing_file = .false.
   debug                   = .false.
 /
```

Item	Type	Description
input_atcf_file	character(len=256)	Name of the input ascii text file in ATCF format.
file_format	character(len=128)	Currently only supports 'b-deck' but if other format strings are added, can switch at runtime between reading different varieties of ATCF file formats.
obs_out_file	character(len=256)	Name of the output observation sequence file to create.
append_to_existing_file	logical	If .false., this program will overwrite an existing file. If .true. and if a file already exists with the same name, the newly converted observations will be appended to that file. Useful if you have multiple small input files that you want to concatenate into a single output file. However, there is no code to check for duplicated observations. If this is .true. and you run the converter twice you will get duplicate observations in the file which is bad. (It will affect the quality of your assimilation results.) Use with care. You can concatenate multiple obs sequence files as a postprocessing step with the *program obs_sequence_tool* which comes with DART and is built by the quickbuild.sh script in the TC converter work directory.
debug	logical	Set to .true. to print out more details during the conversion process.

6.88 PROGRAM `littler_tf_dart`

6.88.1 Overview

Programs to convert littler data files into DART observation sequence files, and vice versa. The capability of the program is limited to wind and temperature from radiosondes.

The littler data files do not contain observation errors. The observation errors are in a separate file called `obserr.txt`. The littler file generated here has to be preprocessed by the program `3dvar_obs.exe` before beeing ingested in the WRF 3D-Var system.

6.88.2 Modules used

```
types_mod
obs_sequence_mod
obs_def_mod
obs_kind_mod
location/threed_sphere/location_mod
time_manager_mod
utilities_mod
```

6.88.3 Modules indirectly used

```
assim_model_mod
models/wrf/model_mod
models/wrf/module_map_utils
random_seq_mod
```

6.88.4 Namelist

The program does not have its own namelist. However, an `input.nml` file is required for the modules used by the program.

6.88.5 Files

- input namelist ; `input.nml`
- Input - output observation files; `obs_seq.out` and `little-r.dat`
- Input - output littler observation error files ; `obserr.txt`

File formats

If there are no observation error at a particular pressure level, the default value of -1 is written in `obserr.txt`.

6.88.6 References

- 3DVAR GROUP PAGE

6.88.7 Private components

call set_str_date(timestring, dart_time)

```
type(time_type),   intent(in)  ::  dart_time
character(len=20), intent(out) ::  timestring
```

Given a dart_time (seconds, days), returns date as bbbbbbyyyymmddhhmmss, where b is a blank space.

call set_dart_time(tstring, dart_time)

```
character(len=20), intent(in)  ::  tstring
type(time_type),   intent(out) ::  dart_time
```

Given a date as bbbbbbyyyymmddhhmmss, where b is a blank space, returns the dart_time (seconds, days).

call StoreObsErr(obs_err_var, pres, plevel, nlev, obs_err_std)

```
integer,  intent(in)    ::  nlev, pres
real(r8), intent(in)    ::  obs_err_var
integer,  intent(in)    ::  plevel(nlev)
real(r8), intent(inout) ::  obs_err_std(nlev)
```

If the incoming pres corresponds exactly to a pressure level in plevel, then transfers the incoming obs_err_var into the array obs_err_std at the corresponding level.

level_index = GetClosestLevel(ilev, vlev, nlev)

```
integer,  intent(in) ::  nlev, ilev
integer,  intent(in) ::  vlev(nlev)
```

Returns the index of the closest level in vlev to the incoming ilev.

call READ_OBSERR(filein, platform, sensor_name, err, nlevels)

```
CHARACTER (LEN=80), intent(in)  ::  filein
CHARACTER (LEN=80), intent(in)  ::  platform
CHARACTER (LEN=80), intent(in   ::  sensor_name
INTEGER,            intent(in)  ::  nlevels
REAL(r8),           intent(out) ::  err(nlevels)
```

Read observational error on pressure levels (in hPa) from the incoming filein and store the result in the array err. It is assumed that filein has the same format as WRF 3D-Var `obserr.txt` file. It reads observational error for a specific platform (e.g. RAOBS) and a specific sensor (e.g. WIND SENSOR ERRORS).

f_obstype = obstype(line)

```
CHARACTER (LEN= 80), intent(in) ::  line
```

Read in a line the string present after keyword 'BOGUS', which should be the sensor name.

f_sensor = sensor(line)

```
CHARACTER (LEN= 80), intent(in) ::  line
```

Read in a line the string present after numbers, which should be the platform name.

val = intplin(x,xx,yy)

```
INTEGER,  DIMENSION (:), intent(in) ::  xx
REAL(r8), DIMENSION (:), intent(in) ::  yy
REAL(r8),                intent(in) ::  x
```

Do a linear interpolation.

val = intplog(x,xx,yy)

```
INTEGER,  DIMENSION (:), intent(in) ::  xx
REAL(r8), DIMENSION (:), intent(in) ::  yy
REAL(r8),                intent(in) ::  x
```

Do a log-linear interpolation.

index = locate(x,xx)

```
INTEGER, DIMENSION (:), intent(in) ::  xx
REAL(r8),               intent(in) ::  x
```

Return the index in xx such that xx(index) < x < xx(index+1).

6.89 PROGRAM rad_3dvar_to_dart

6.89.1 Overview

Programs to convert MM5 3D-VAR 2.0 Radar data files into DART observation sequence files. The capability of the program is limited to DOPPLER_RADIAL_VELOCITY and RADAR_REFLECTIVITY.

6.89.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&rad_3dvar_to_dart_nml
   var_file = 'qc_radr_3dvar_2002083100.dat',
   obs_seq_out_file_name = 'obs_seq.out',
   calendar_type = 3
/
```

Item	Type	Description
var_file	character(len=129)	This is the name of the file containing MM5 3D-VAR 2.0 Radar observations.
obs_seq_out_file_name	character(len=129)	File name for output observation sequence file.
calendar_type	integer	Calendar type. We recommend using 3 (GREGORIAN).

6.89.3 Modules directly used

```
types_mod
obs_sequence_mod
obs_def_mod
obs_def/obs_def_radar_mod
obs_kind_mod
location/threed_sphere/location_mod
time_manager_mod
utilities_mod
```

6.89.4 Modules indirectly used

```
assim_model_mod
models/wrf/model_mod
models/wrf/module_map_utils
random_seq_mod
```

6.89.5 Files

- input namelist ; `input.nml`
- Input observation file; `qc_radr_3dvar_2002083100.dat`
- Output observation file; `obs_seq.out`

File formats

`input.nml` and `qc_radr_3dvar_2002083100.dat` are ASCII files. `obs_seq.out` is either ASCII or binary, depending on the logical write_binary_obs_sequence, which is the namelist entry for obs_sequence_mod.

6.89.6 References

- 3DVAR GROUP PAGE

6.90 3DVAR/4DVAR Observation Converters

6.90.1 Overview

The programs in the `obs_converters/var` directory help convert data which is formatted for input into the 3DVAR/4DVAR programs into DART obs_seq observation files.

The directory contains conversion programs for various obs formats related to 3D-Var, WRF-Var, and MM5:

- *PROGRAM littler_tf_dart* to and back from little-r format, temperature and winds only.
- *PROGRAM rad_3dvar_to_dart* the radar 3d-var obs only to dart format.
- `gts_to_dart.f90` from GTS to dart format.

You need to add some WRF-Var source files to the 3DVAR_OBSPROC directory, and then you can go into the work directory and run the 'quickbuild.sh' script. The required WRF-Var source files are listed in `3DVAR_OBSPROC/README`.

The little-r converter may need changes to the code to convert from the original quality control flags into QC flags compatible with DART. (in DART, 0 is good data.)

The GTS converter does not support SATEM thickness data but there are versions around which do; write dart@ucar.edu if you are interested in more about this.

And a final disclaimer: Whether these work with the latest 3D-Var format is untested at this point. Please contact the DART Development group if you are interested in using these tools.

6.91 Checking your initial assimilation

You may require several attempts to get your assimilation configured correctly. The next section, *Computing filter increments*, describes how to take the difference between two assimilation stages to determine whether your initial assimilation worked as intented.

If your assimilation does not change anything in the model state, you may need to rerun `filter` multiple times to understand what is wrong.

Thus you should make `filter` very fast to run. You can do this by:

1. Making an observation sequence file containing a single observation.
2. Configuring your run so that filter does a single assimilation and exits without having to advance the ensemble of models or do other work.

6.91.1 Making an observation sequence file containing a single observation

You can use one of these methods to make an `obs_seq` with just a single observation:

1. Run `create_obs_sequence` to make a new, short, observation sequence file.
2. Use the `obs_sequence_tool` to cut an existing `obs_seq.out` file down to just a few obs by selecting only a subset of the types and setting a very short time window, such as a second or two when you know there are observations available.

These programs are described in the *Programs directory*.

6.91.2 Configuring your run so that filter does a single assimilation and exits

To configure `filter` to only do a single assimilation:

1. Edit the `&filter_nml` namelist in `input.nml` to set the `init_time_days` and `init_time_seconds` to match the observation time in your truncated observation sequence file. This overrides any times in the input files and ensures that `filter` will only assimilate and not try to advance the model.
2. Make sure the truncated observation sequence file contains only a single observation or observations close enough together in time to fit into a single assimilation window.

6.92 Computing filter increments

Note: This document is written as if your experiment was run with `single_file_out = .true.`. The potential permutations of filenames output by filter is enormous, so it isn't feasible to write documentation for all possible cases.

After *filter* executes without error and produces an `obs_seq.final` file, a `preassim.nc` file, and an `analysis.nc` file, the first questions to ask are:

1. Is the model state output from `filter` different from the input?
2. Were any observations successfully assimilated?

You can check check if the output model state data was changed by the assimilation by using the `ncdiff` tool to create a file containing the difference of the `preassim.nc` and `analysis.nc` files. If you are running with `single_file_in = .true.` and `single_file_out = .true.` use `ncdiff` on the files output for the analysis and preassim stages:

```
$ ncdiff analysis.nc preassim.nc increments.nc
```

Otherwise, if you are running with `single_file_in = .false.` and `single_file_out = .false.`, use `ncdiff` on the ensemble mean files for the analysis and preassim stages:

```
$ ncdiff analysis_mean.nc preassim_mean.nc increments.nc
```

`ncdiff` generates a file, `increments.nc`, that contains the increments, or innovations, created by `filter`. You can view the increments using `ncview`:

```
$ ncview increments.nc
```

to examine the ensemble mean variables. If all values are 0, then the assimilation changed nothing in the state.

6.93 Computing filter increments using a complex model

The innovations to the model state are easy to derive. Use the NCO Operator *ncdiff* to difference the two DART diagnostic netCDF files to create the innovations. Be sure to check the *CopyMetaData* variable to figure out what *copy* is of interest. Then, use *ncview* to explore the innovations or the inflation values or ...

If the assimilation used state-space inflation, the inflation fields will be added as additional 'copies'. A sure sign of trouble is if the inflation fields grow without bound. As the observation network changes, expect the inflation values to change.

The only other thing I look for in state-space is that the increments are 'reasonable'. As the assimilation 'burns in', the increments are generally larger than increments from an assimilation that has been cycling for a long time. If the increments keep getting bigger, the ensemble is continually drifting away from the observation. Not good. In *ncview*, it is useful to navigate to the copy/level of interest and re-range the data to values appropriate to the current data and then hit the '>>' button to animate the image. It should be possible to get a sense of the magnitude of the innovations as a function of time.

6.93.1 Example from a model of intermediate complexity: the bgrid model

I ran a perfect model experiment with the bgrid model in the DART-default configuration and turned on some adaptive inflation for this example. To fully demonstrate the adaptive inflation, it is useful to have an observation network that changes through time. I created two observation sequence files: one that had a single 'RADIOSONDE_TEMPERATURE' observation at the surface with an observation error variance of 1.5 degrees Kelvin - repeated every 6 hours for 6 days (24 timesteps); and one that had 9 observations locations clustered in about the same location that repeated every 6 hours for 1.5 days (6 timesteps). I merged the two observation sequences into one using `obs_sequence_tool` and ran them through `perfect_model_obs` to derive the observation values and create an `obs_seq.out` file to run through `filter`.

Note: Other models may have their ensemble means and spreads and inflation values in separate files. *See the table of possible filenames.*

```
$ cd ${DARTROOT}/models/bgrid_solo/work
$ ncdiff analysis.nc preassim.nc Innov.nc
$ ncview preassim.nc &
$ ncview Innov.nc &
$ ncdump -v MemberMetadata preassim.nc
```

(continues on next page)

(continued from previous page)

```
netcdf preassim {
dimensions:
        metadatalength = 64 ;
        member = 20 ;
        time = UNLIMITED ; // (24 currently)
        NMLlinelen = 129 ;
        NMLnlines = 303 ;
        StateVariable = 28200 ;
        TmpI = 60 ;
        TmpJ = 30 ;
        lev = 5 ;
        VelI = 60 ;
        VelJ = 29 ;
variables:
        char MemberMetadata(member, metadatalength) ;
                MemberMetadata:long_name = "Metadata for each copy/member" ;
        ...
        double ps(time, member, TmpJ, TmpI) ;
                ps:long_name = "surface pressure" ;
                ps:units = "Pa" ;
                ps:units_long_name = "pascals" ;
        double t(time, member, lev, TmpJ, TmpI) ;
                t:long_name = "temperature" ;
                t:units = "degrees Kelvin" ;
        double u(time, member, lev, VelJ, VelI) ;
                u:long_name = "zonal wind component" ;
                u:units = "m/s" ;
        double v(time, member, lev, VelJ, VelI) ;
                v:long_name = "meridional wind component" ;
                v:units = "m/s" ;
        double ps_mean(time, TmpJ, TmpI) ;        The ensemble mean   is now a separate␣
↪variable.
        double t_mean(time, lev, TmpJ, TmpI) ;    The ensemble spread is now a separate␣
↪variable.
        double u_mean(time, lev, VelJ, VelI) ;    If I was using inflation, they would␣
↪also be separate variables.
        double v_mean(time, lev, VelJ, VelI) ;
        double ps_sd(time, TmpJ, TmpI) ;
        double t_sd(time, lev, TmpJ, TmpI) ;
        double u_sd(time, lev, VelJ, VelI) ;
        double v_sd(time, lev, VelJ, VelI) ;

data:
  MemberMetadata =
  "ensemble member      1 ",
  "ensemble member      2 ",
  "ensemble member      3 ",
  "ensemble member      4 ",
  "ensemble member      5 ",
  "ensemble member      6 ",
  "ensemble member      7 ",
  "ensemble member      8 ",
```

(continues on next page)

(continued from previous page)

```
  "ensemble member      9 ",
  "ensemble member     10 ",
  "ensemble member     11 ",
  "ensemble member     12 ",
  "ensemble member     13 ",
  "ensemble member     14 ",
  "ensemble member     15 ",
  "ensemble member     16 ",
  "ensemble member     17 ",
  "ensemble member     18 ",
  "ensemble member     19 ",
  "ensemble member     20 " ;
}
```

This is an exploration of the `preassim.nc` file. Note that I selected the '**t**' field, turned the coastlines 'off' under the 'Opts' button, used the 'Repl' instead of 'Bi-lin' (to more faithfully represent the model resolution), *navigated to copy 23 of 24 (in this case, the* inflation mean *)* select the **inflation mean variable of your choice** and advanced to the last timestep. The image plot is pretty boring, but does indicate that the inflation values are restricted to where I put the observations. Right-clicking on the 'Range' button automatically re-ranges the colorbar to the min/max of the current data. Clicking on any location generates a time series figure.

This is an exploration of the `Innov.nc` file as created by *ncdiff*. Note that the titles are somewhat misleading because they reflect information from the first file given to *ncdiff*. This time I left the rendering as 'Bi-lin' (which obfuscates the model resolution), *navigated to copy 1 of 24 (in this case, the* ensemble mean *)* selected the **t_mean** variable and advanced to the 6th timestep. Right-click on the 'Range' button to reset the colorbar. The image plot confirms that the innovations are restricted to a local region. Clicking on any location generates a time series.

This is fundamentally the same as the previous panel except that I have now selected the '**u**' **u_mean** variable. Despite the fact the observations were only of '**t**', the assimilation has generated (rightly so) increments to the '**u**' state variable.

6.94 DART missing data value

If all the prior and posterior mean values are -888888.0 (which is the DART "missing data" value), those observations were not assimilated.

Note: Some observations have precomputed values and the posterior values for these will always be -888888.0, no matter if the observation was assimilated or not.

If it is not already set, edit the `&filter_nml` name list in `input.nml` to set `num_output_obs_members` to be the same as the ensemble size.

This will give you all the forward operator values for all the ensemble members. You can determine if all ensemble members are failing in the same way, or if only a few are problematic.

6.95 DART quality control field

DART has a quality control (QC) field in the *obs_seq.final* file to report on the status of the assimilation of the variable. The most common reason for exploring the DART QC value is to help determine if the observation was assimilated (or evaluated) - or if the observation was rejected or ...

To learn more about how to intepret the QC field as well as other values in an observation sequence file, see *Detailed structure of an obs_seq file*. The 'DART QC' field is usually the second of the 2 "quality control" copies.

A list of all the DART QC values can be found in the QC table in *MODULE quality_control_mod*.

- If the DART QC values are 4, the forward operators have failed. Look at the *model_interpolate()* routine in your model_mod.f90 file, or the forward operator code in *observations/forward_operators/obs_def_xxx_mod.f90* for your observation type. A successful forward operator must return a valid obs_val and an *istatus = 0*. If the forward operator code returns different istatus values for different error types, you can set *&filter_nml::output_forward_op_errors = .true.* and rerun *filter* to see exactly what error istatus codes are being set. See *MODULE filter_mod* for more information on how to use the 'output_forward_op_errors' option. Negative istatus values are reserved for the system, *istatus = 0* is success, and any positive value indicates a failed forward operator. The code is free to use different positive values to signal different types of errors.
- If the DART QC values are 5, those observation types were intentionally ignored because they were not listed in the &obs_kind_nml namelist, in the 'assimilate_these_obs_types' stringlist.
- If the DART QC values are 6, the data quality control that came with the original observation data indicates this is a bad quality observation and it was skipped for this reason.
- If the DART QC values are 7, the observation value is too far away from the ensemble mean. Set *&filter_nml::outlier_threshold = -1* to ignore this for now and rerun. In general, this is not the optimal strategy as the number of observations inconsistent with the ensemble is a very powerful indicator of the success or failure of the assimilation.
- If the DART QC values are 8, it was not possible to convert the observation to the required vertical coordinate system.

If the prior and posterior values in the `obs_seq.final` are not -888888.0 but are identical, your obs are being assimilated but are having no impact.

The most common reasons assimilated obs have no impact on the model state include:

- **Zero spread in ensemble members** Your initial ensemble members must have different values for each state item. If all members have identical values, the observations cannot make a change. To diagnose this condition, look at the prior ensemble spread. This is either in `preassim.nc` or `preassim_sd.nc`, depending on your model. If all the values are 0, this is your problem. One way to generate an ensemble with some spread is to set *&filter_nml::perturb_from_single_instance = .false.,* (which will still require a single filter initial condition file) but then the *filter* code will add random gaussian perturbations to each state vector item to generate an initial ensemble with spread. The magnitude of the gaussian noise added is controlled by the *&filter_nml::perturbation_amplitude*. It is also possible to write your own perturbation routine in your `model_mod.f90` code.
- **Cutoff value too small** If the localization radius is too small, the observation may not be 'close enough' to the model grid to be able to impact the model. Check the localization radius (*&assim_tools_nml::cutoff*). Set it to a very large number (e.g. 100000) and rerun. If there is now an impact, the cutoff was restricting the items in the state vector so your obs had no impact before. Cutoff values are dependent on the location type being used. It is specified in radians for the threed_sphere locations module (what most large models use), or in simple distance (along a unit circle) if using a low order model (lorenz, ikeda, etc).
- **Obs error values too large (less likely)** If the observation error is very large, it will have no impact on the model state. This is less likely a cause than other possibilities.

- **No correlation (unlikely)** If there is no correlation between the distribution of the forward observation values and the state vector values, the increments will be very tiny. However there are generally still tiny increments applied, so this is also a low likelyhood case.
- **Errors in forward operator location computations, or get_close_obs()** If there is an error in the `model_mod.f90` code in either *get_state_meta_data()*, *model_interpolate()*, or the vertical conversion code in *get_close_obs()*, it is possible for the forward operators to appear to be working correctly, but the distances computed for the separation between the obs and the state vector values can be incorrect. The most frequent problem is that the wrong locations are being passed back from *get_state_meta_data()*. This can result in the increments being applied in the wrong locations or not at all. This is usually one of the things to test carefully when developing a new model interface, and usually why we recommend starting with a single observation at a known location.
- **Incorrect vertical conversion** If the model is using 3d coordinates and needs the capability to convert between pressure, height, and/or model level, the conversion may be incorrect. The state vector locations can appear to be too high or too low to be impacted by an observation. Some models have a height limit built into their model_mod code to avoid trying to assimilate observations at the model top. The observations cannot make meaningful changes to the model state there and trying to assimilate them can lead to problems with the inflation. If the code in the model_mod is excluding observations incorrectly, or you are testing with observations at the model top, this can result in no impact on the model state.

6.96 Examining the obs_seq.final file

1. If you are testing with a single observation, just look in the file. If this file is in binary format, edit the `&obs_sequence_nml` namelist in *input.nml* so the output observation sequence file will be written in ASCII:

   ```
   &obs_sequence_nml
     write_binary_obs_sequence = .false.
   /
   ```

   Then rerun *filter* to regenerate an *obs_seq.final* file in ASCII. For an explanation of the contents of your *obs_seq.final* file, see *Detailed structure of an obs_seq file*.

2. If you are using many observations, run the *obs_diag* program appropriate for your model. The *MATLAB® observation space diagnostics* will help to summarize your output and to explore what is going on.

If there are no changes in the model state after assimilation and a visual examination of *obs_seq.final* was not informative, convert the *obs_seq.final* file to netCDF with *obs_seq_to_netcdf* and either use the Matlab tools distributed with DART or something of your own. Actually, *obs_seq_to_netcdf* works on all observation sequence files, not just *obs_seq.final* files.

6.97 MATLAB® observation space diagnostics

The observation-space functions are in the `$DARTROOT/diagnostics/matlab` directory. Once you have processed the `obs_seq.final` files into a single `obs_diag_output.nc`, you can use that as input to your own plotting routines or use the following DART MATLAB® routines:

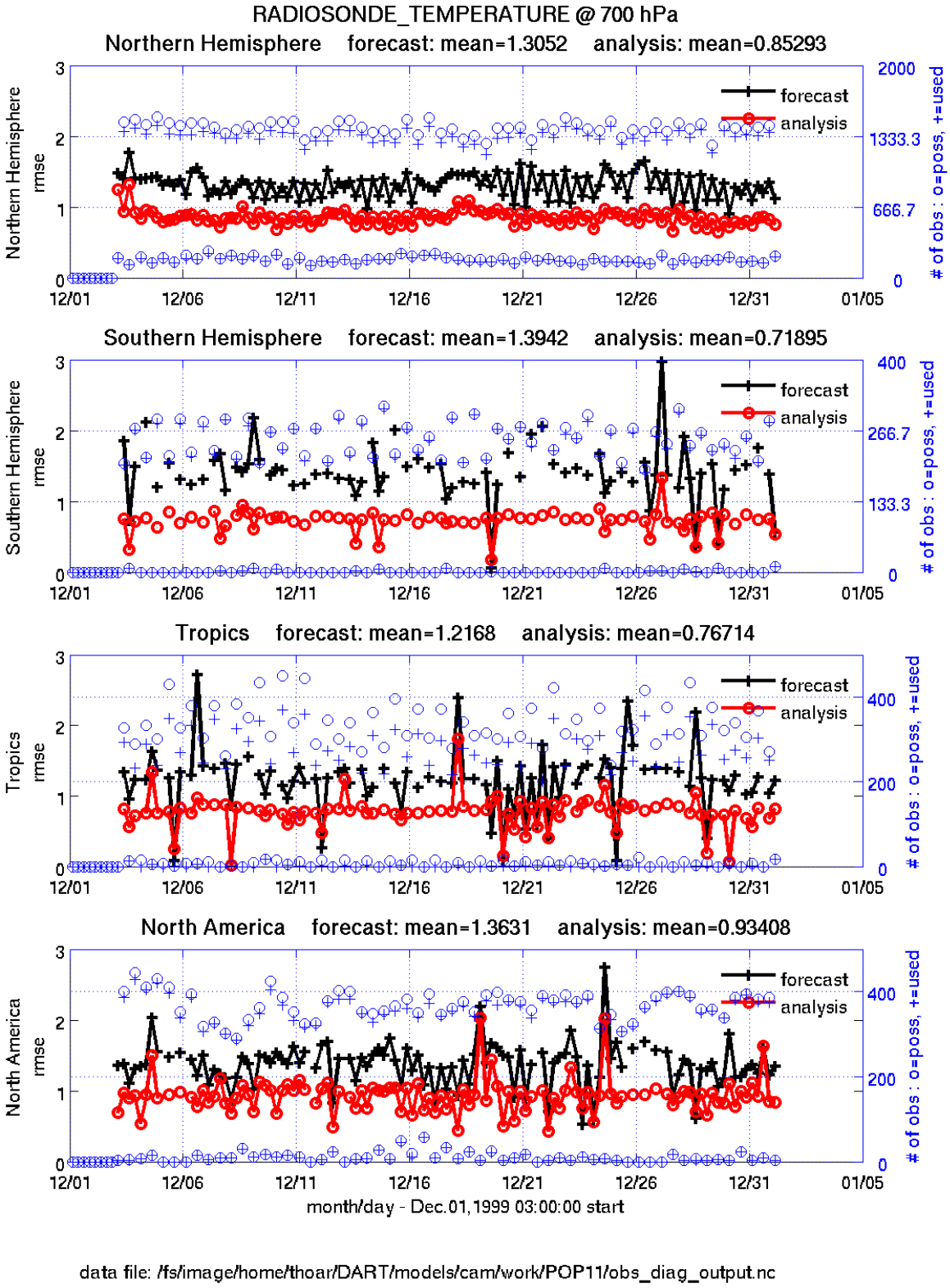
plot_evolution.m plots the temporal evolution of any of the quantities above for each variable for specified levels. The number of observations possible and used are plotted on the same axis.

```
fname      = 'POP11/obs_diag_output.nc';        % netcdf file produced by 'obs_diag'
copystring = 'rmse';                            % 'copy' string == quantity of interest
```

(continues on next page)

(continued from previous page)

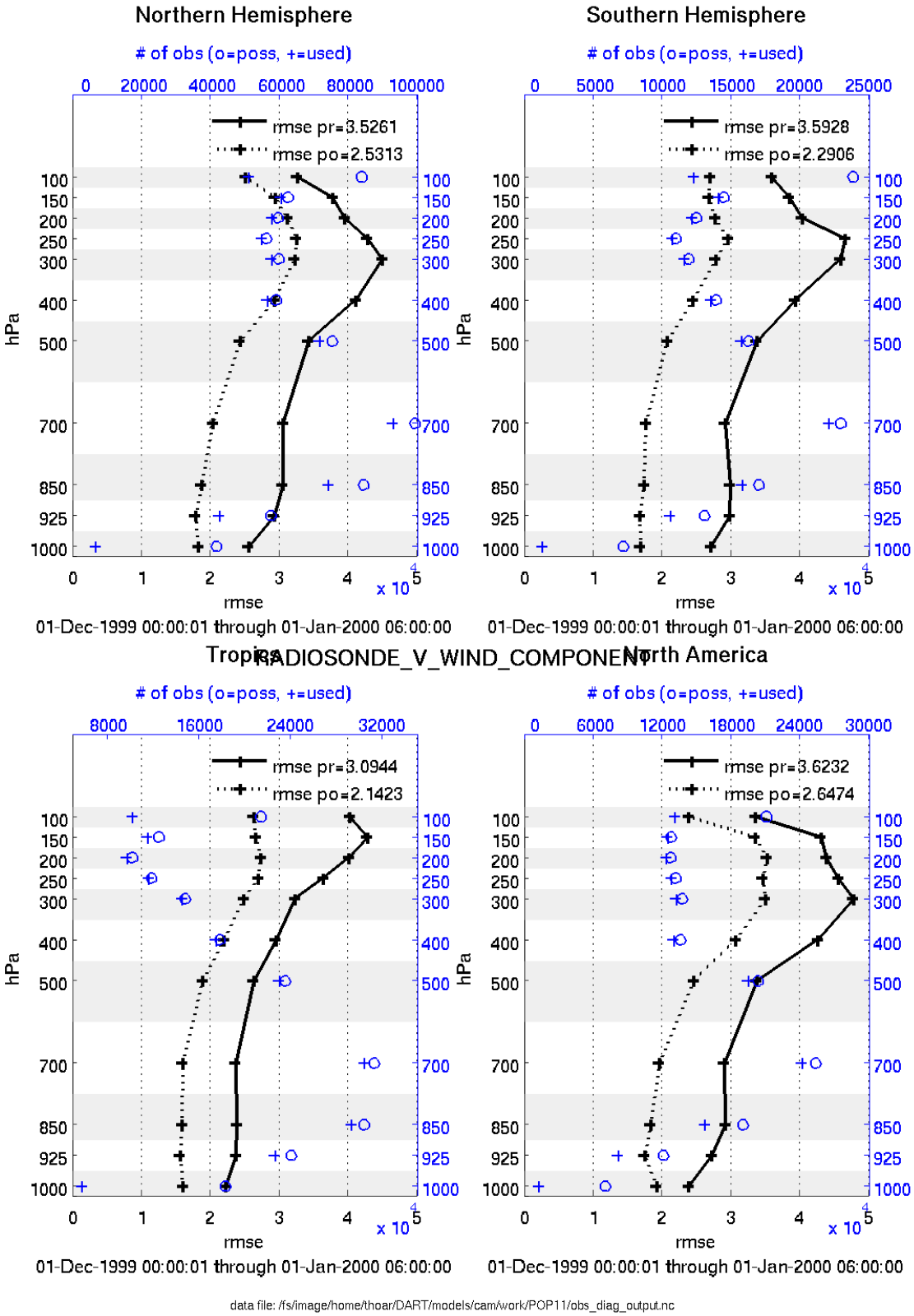
```
plotdat    = plot_evolution(fname,copystring);  % -- OR --
plotdat    = plot_evolution(fname,copystring,'obsname','RADIOSONDE_TEMPERATURE');
```



plot_profile.m plots the spatial and temporal average of any specified quantity as a function of height. The number of

observations possible and used are plotted on the same axis.

```
fname      = 'POP11/obs_diag_output.nc';        % netcdf file produced by 'obs_diag'
copystring = 'rmse';                            % 'copy' string == quantity of interest
plotdat    = plot_profile(fname,copystring);
```



plot_rmse_xxx_evolution.m same as `plot_evolution.m` but will overlay **rmse** on the same axis.

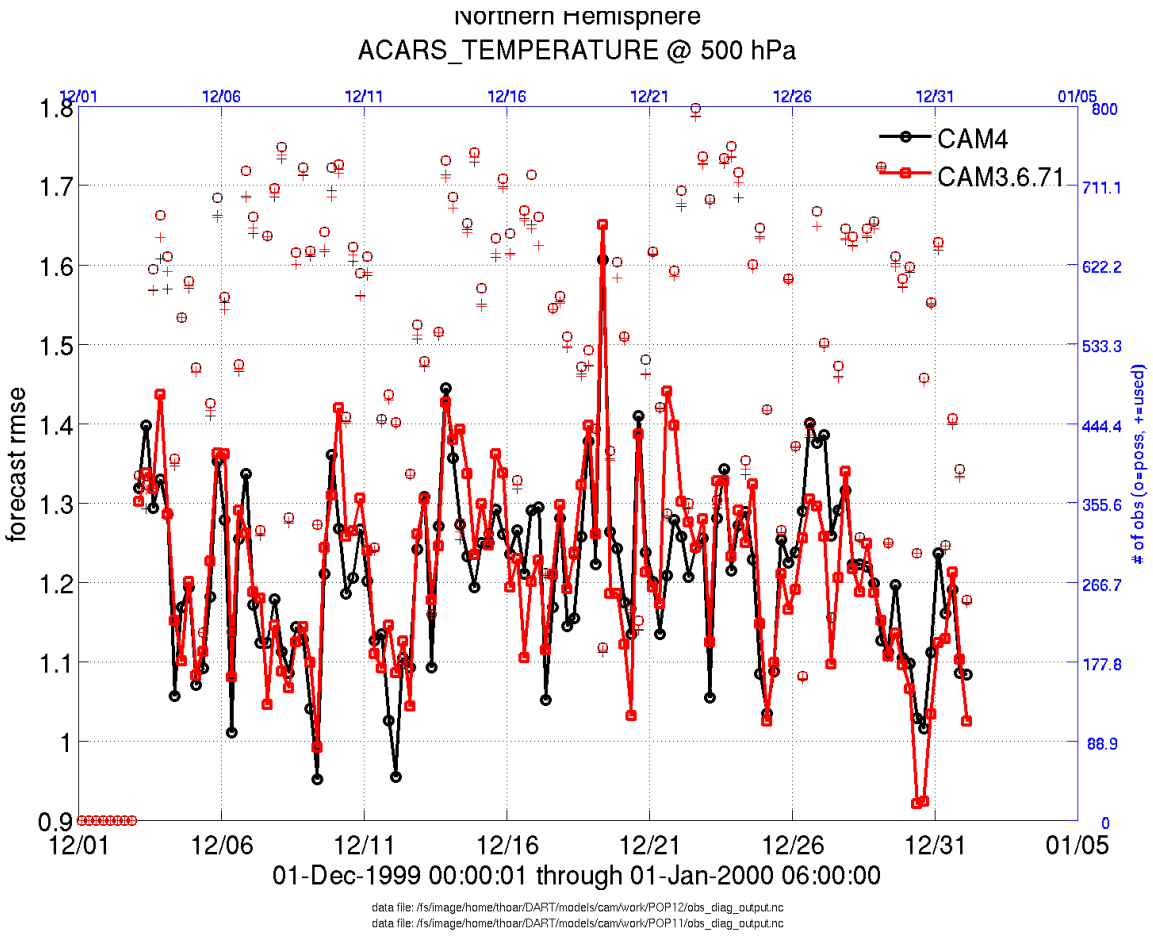
plot_rmse_xxx_profile.m same as `plot_profile.m` with an overlay of **rmse**.

plot_bias_xxx_profile.m same as `plot_profile.m` with an overlay of **bias**.

---

two_experiments_evolution.m same as `plot_evolution.m` but will overlay multiple (more than two, actually) experiments (i.e. multiple `obs_diag_output.nc` files) on the same axis. A separate figure is created for each region in the `obs_diag_output.nc` file.

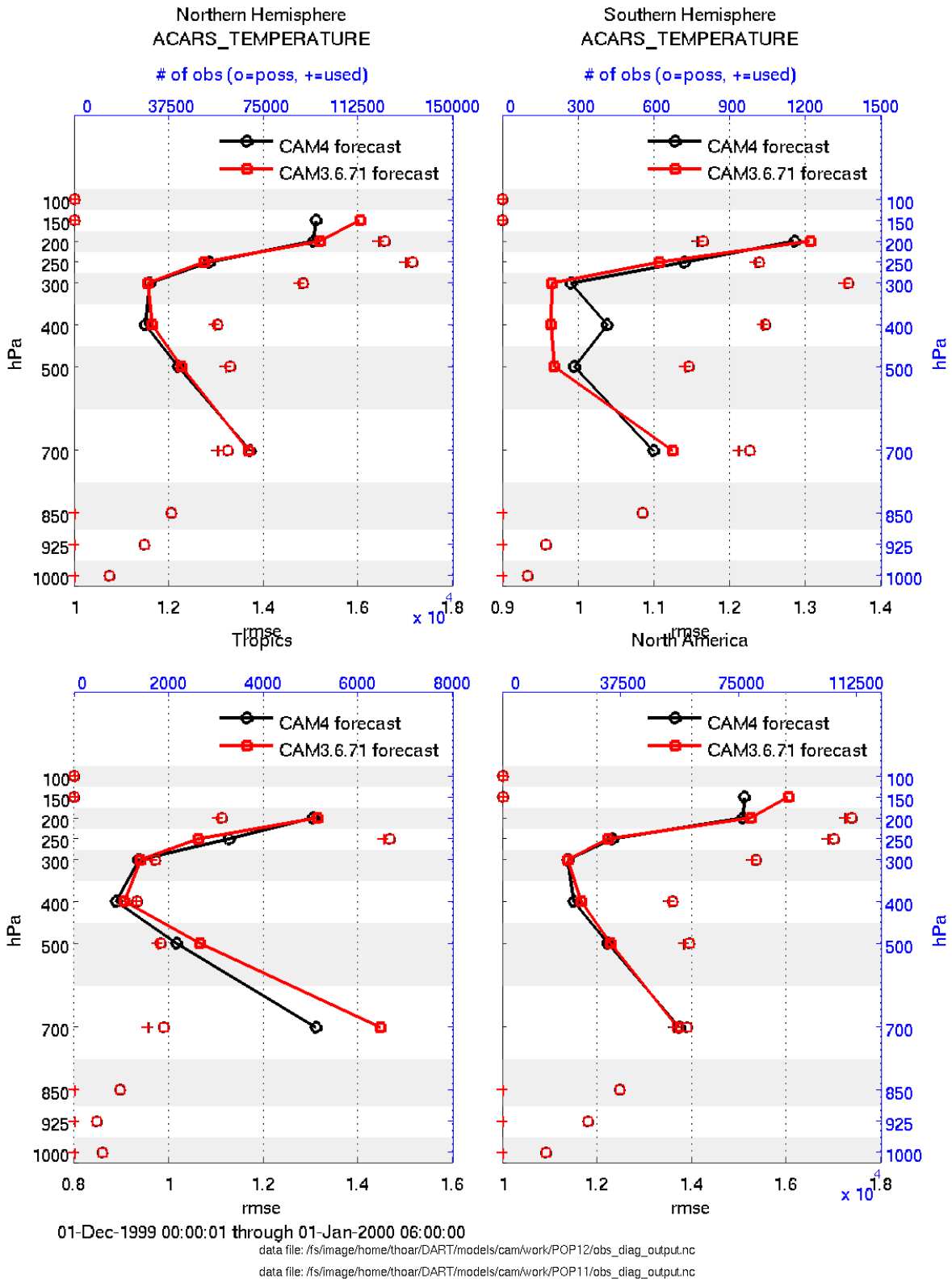
```
files    = {'POP12/obs_diag_output.nc','POP11/obs_diag_output.nc'};
titles   = {'CAM4','CAM3.6.71'};
varnames = {'ACARS_TEMPERATURE'};
qtty     = 'rmse';
prpo     = 'prior';
levelind = 5;
two_experiments_evolution(files, titles,{'ACARS_TEMPERATURE'}, qtty, prpo, levelind)
```



---

two_experiments_profile.m same as `plot_profile.m` but will overlay multiple (more than two, actually) experiments (i.e. multiple `obs_diag_output.nc` files) on the same axis. If the `obs_diag_output.nc` file was created with multiple regions, there are multiple axes on a single figure.

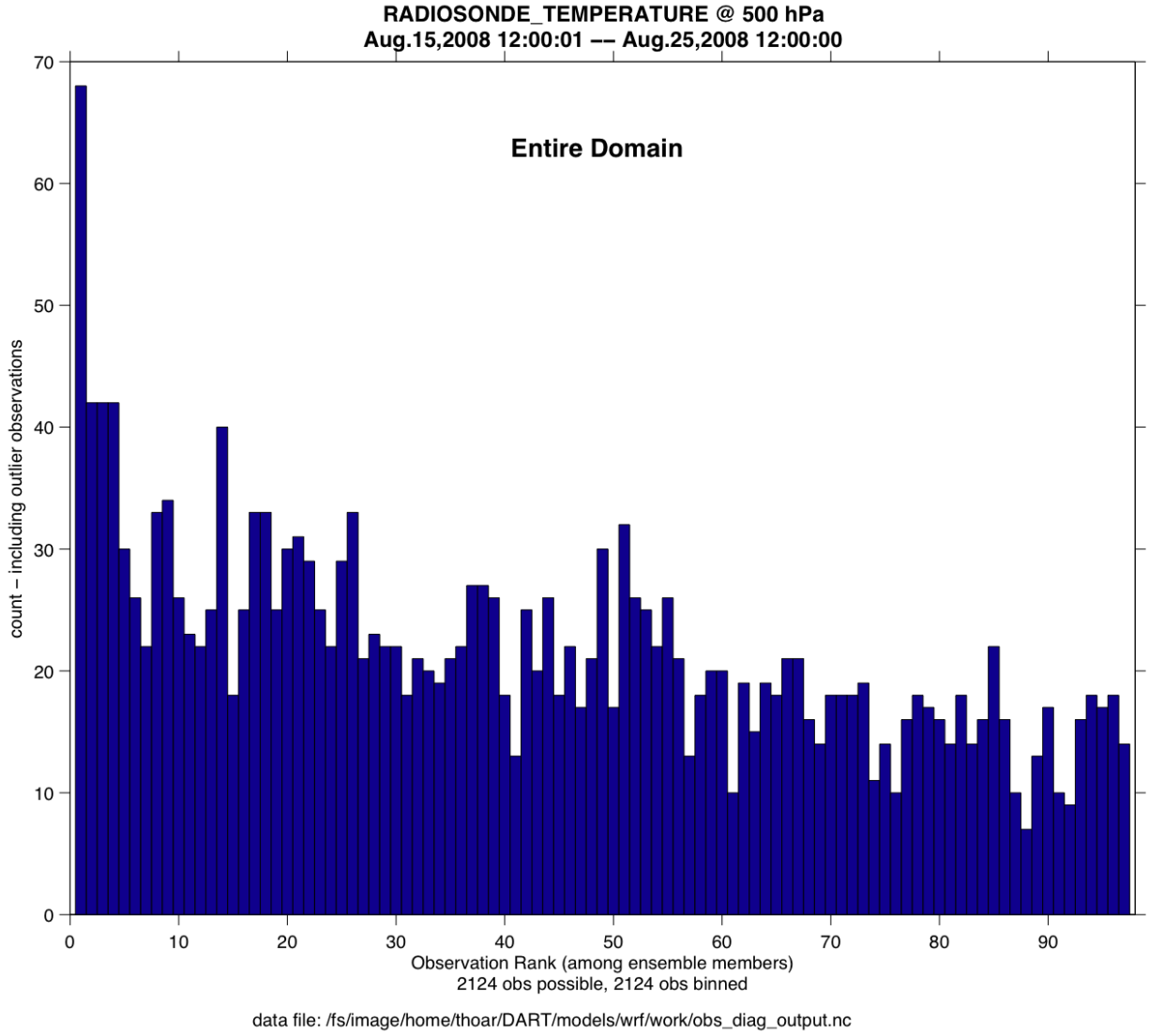
```
files    = {'POP12/obs_diag_output.nc','POP11/obs_diag_output.nc'};
titles   = {'CAM4','CAM3.6.71'};
varnames = {'ACARS_TEMPERATURE'};
qtty     = 'rmse';
prpo     = 'prior';
two_experiments_profile(files, titles, varnames, qtty, prpo)
```



plot_rank_histogram.m will create rank histograms for any variable that has that information present in

obs_diag_output.nc.

```
fname     = 'obs_diag_output.nc'; % netcdf file produced by 'obs_diag'
timeindex = 3;                    % plot the histogram for the third timestep
plotdat   = plot_rank_histogram(fname, timeindex, 'RADIOSONDE_TEMPERATURE');
```



You may also convert observation sequence files to netCDF by using *PROGRAM obs_seq_to_netcdf*. All of the following routines will work on observation sequences files AFTER an assimilation (i.e. `obs_seq.final` files that have been converted to netCDF), and some of them will work on `obs_seq.out`-type files that have been converted.

read_obs_netcdf.m reads a particular variable and copy from a netCDF-format observation sequence file and returns a single structure with useful bits for plotting/exploring. This routine is the back-end for `plot_obs_netcdf.m`.

```
fname         = 'obs_sequence_001.nc';
ObsTypeString = 'RADIOSONDE_U_WIND_COMPONENT';   % or 'ALL' ...
region        = [0 360 -90 90 -Inf Inf];
CopyString    = 'NCEP BUFR observation';
```

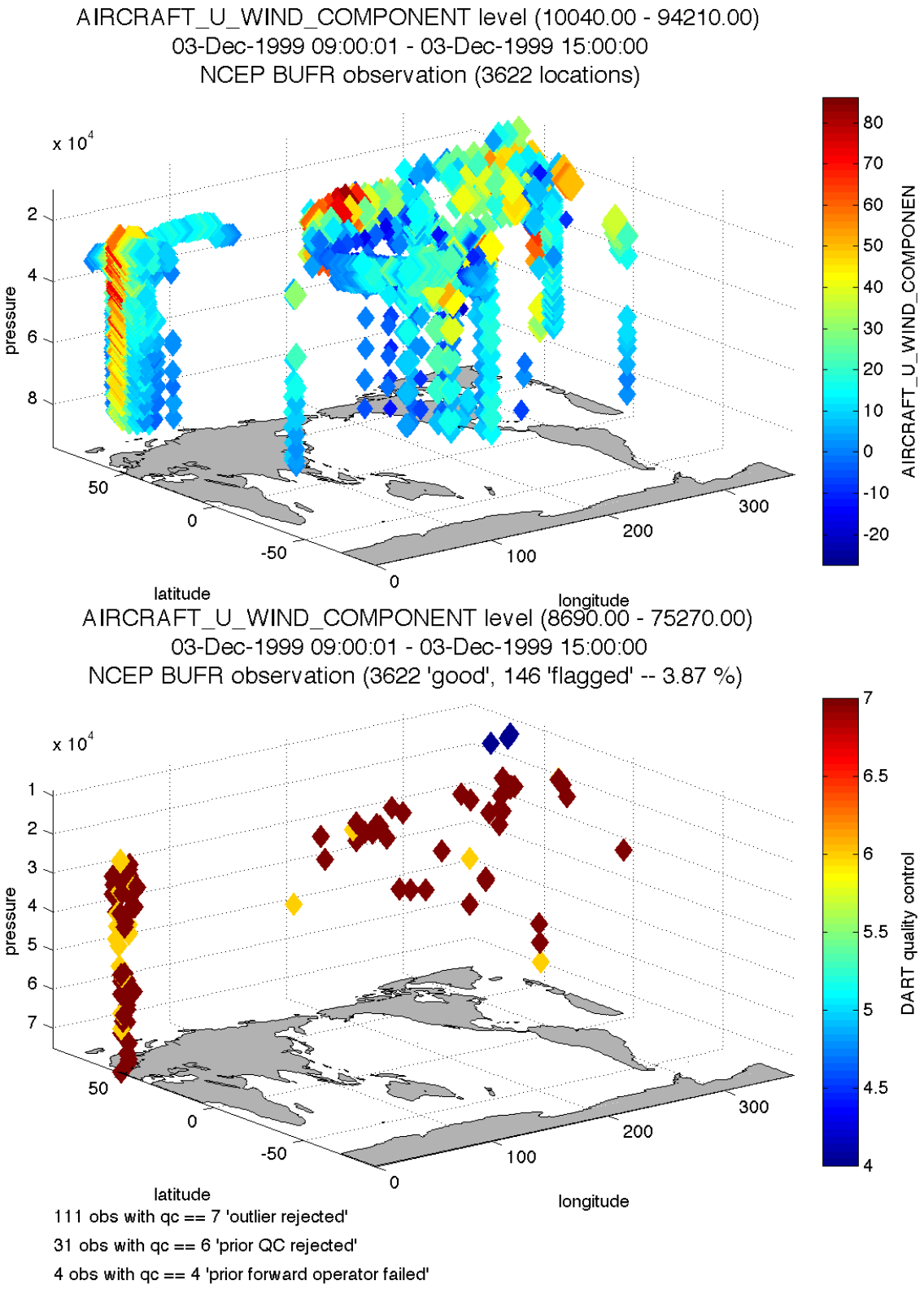
(continues on next page)

(continued from previous page)

```
QCString      = 'DART quality control';
verbose       = 1;   % anything > 0 == 'true'
obs = read_obs_netcdf(fname, ObsTypeString, region, CopyString, QCString, verbose);
```

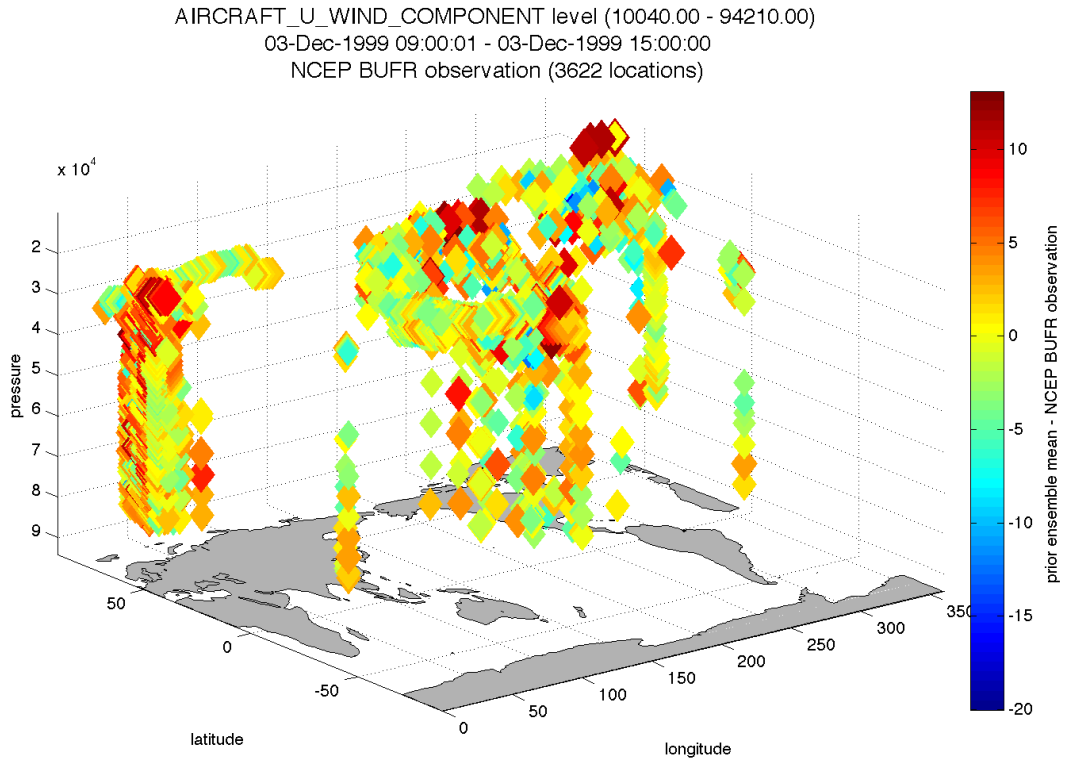
plot_obs_netcdf.m creates a 3D scatterplot of the observation locations, color-coded to the observation values. A second axis will also plot the QC values if desired.

```
fname         = 'POP11/obs_epoch_011.nc';
region        = [0 360 -90 90 -Inf Inf];
ObsTypeString = 'AIRCRAFT_U_WIND_COMPONENT';
CopyString    = 'NCEP BUFR observation';
QCString      = 'DART quality control';
maxgoodQC     = 2;
verbose       = 1;   % > 0 means 'print summary to command window'
twoup         = 1;   % > 0 means 'use same Figure for QC plot'
bob = plot_obs_netcdf(fname, ObsTypeString, region, CopyString, ...
                  QCString, maxgoodQC, verbose, twoup);
```



plot_obs_netcdf_diffs.m creates a 3D scatterplot of the difference between two 'copies' of an observation.

```
fname         = 'POP11/obs_epoch_011.nc';
region        = [0 360 -90 90 -Inf Inf];
ObsTypeString = 'AIRCRAFT_U_WIND_COMPONENT';
CopyString1   = 'NCEP BUFR observation';
CopyString2   = 'prior ensemble mean';
QCString      = 'DART quality control';
maxQC         = 2;
verbose       = 1;   % > 0 means 'print summary to command window'
twoup         = 0;   % > 0 means 'use same Figure for QC plot'
bob = plot_obs_netcdf_diffs(fname, ObsTypeString, region, CopyString1, CopyString2, ...
                            QCString, maxQC, verbose, twoup);
```



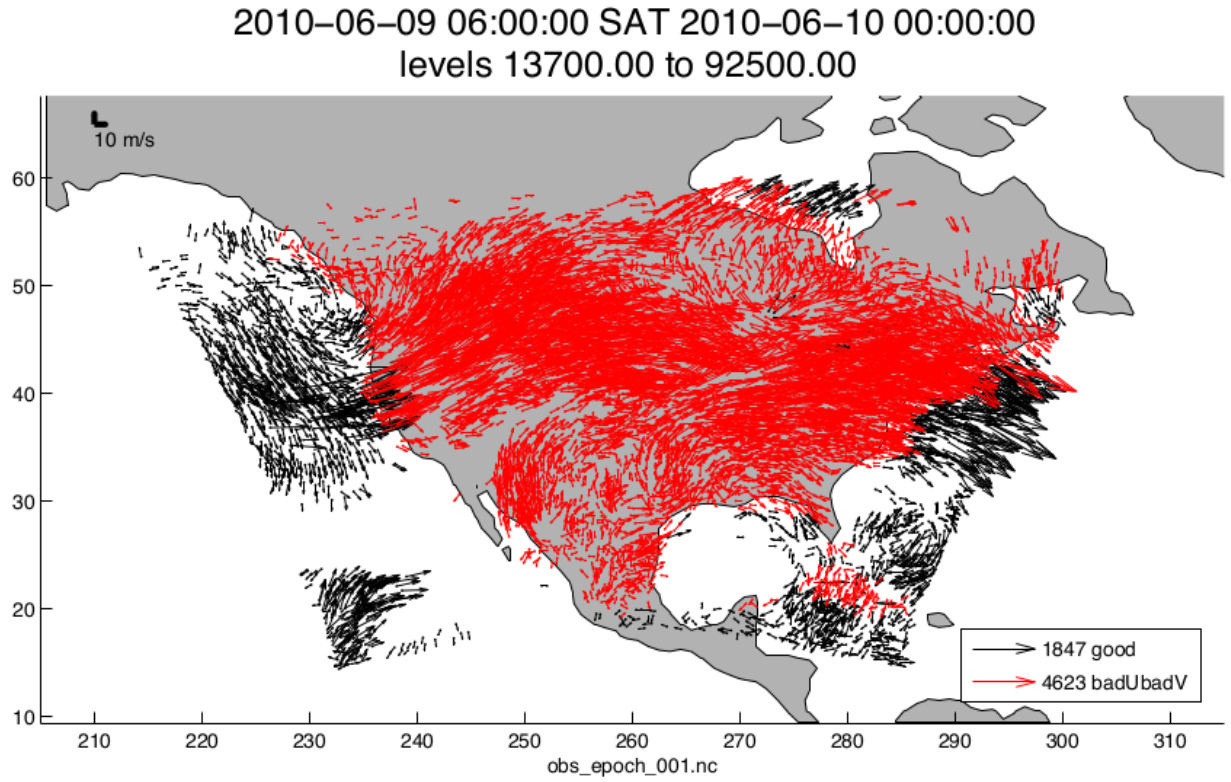
plot_wind_vectors.m creates a 2D 'quiver' plot of a wind field. This function is in the `matlab/private` directory - but if you want to use it, you can move it out. I find it has very little practical value.

```
fname       = 'obs_epoch_001.nc';
platform    = 'SAT';    % usually 'RADIOSONDE', 'SAT', 'METAR', ...
CopyString  = 'NCEP BUFR observation';
QCString    = 'DART quality control';
region      = [210 310 12 65 -Inf Inf];
scalefactor = 5;     % reference arrow magnitude
```

(continues on next page)

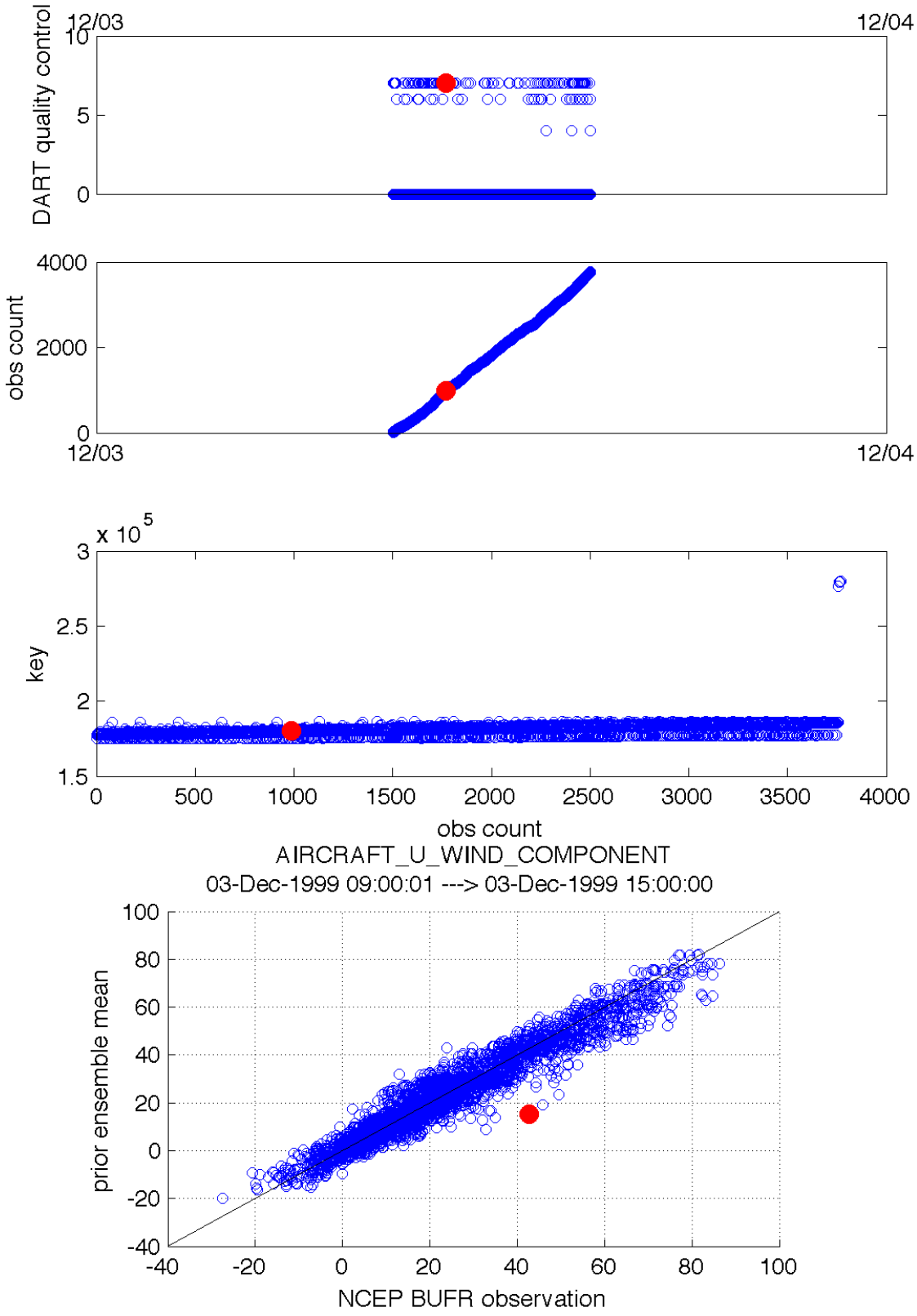
(continued from previous page)

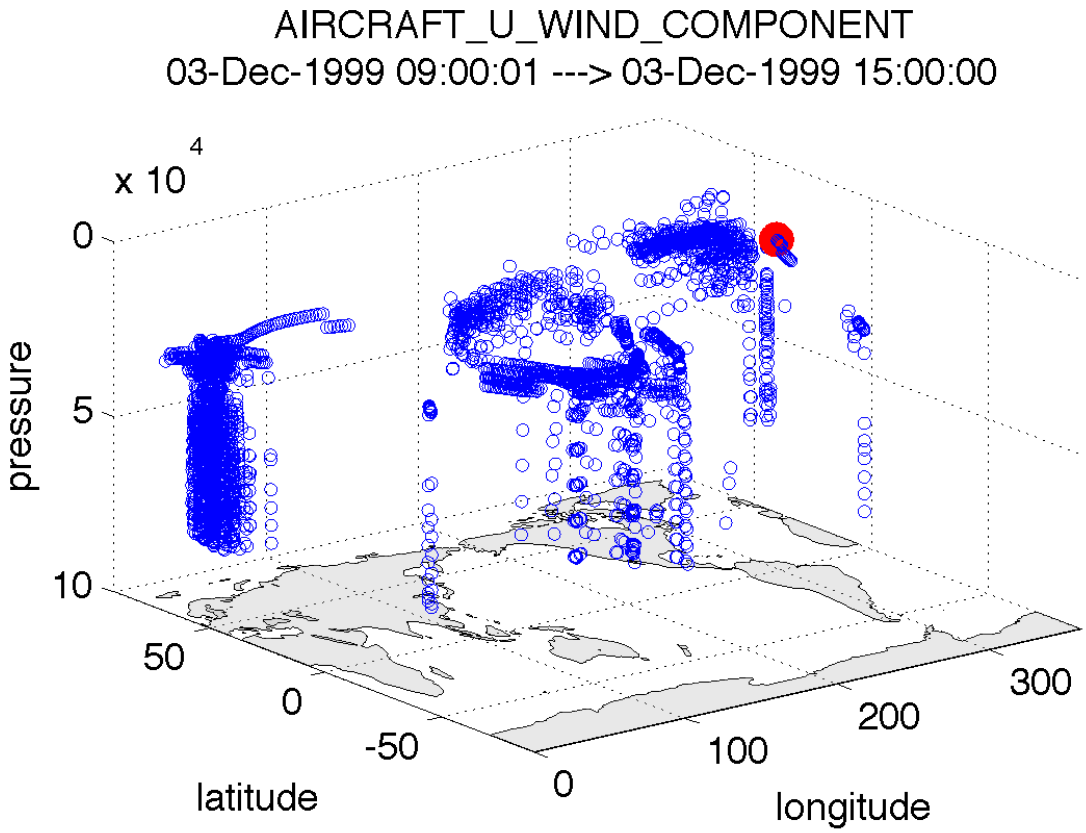
```
bob = plot_wind_vectors(fname, platform, CopyString, QCString, ...
                        'region', region, 'scalefactor', scalefactor);
```

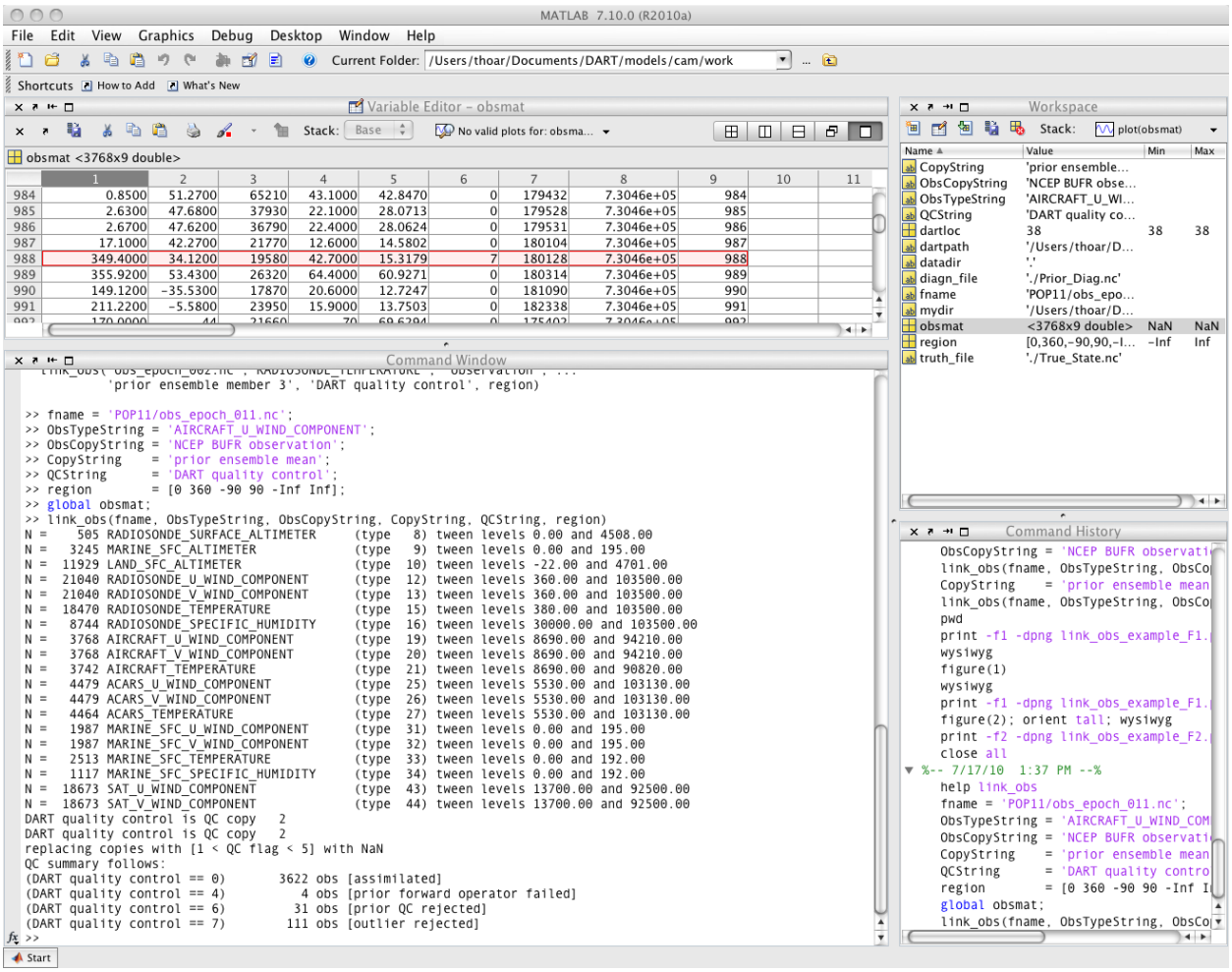


link_obs.m creates multiple figures that have linked attributes. This is my favorite function. Click on the little paintbrush icon in any of the figure frames and select some observations with "DART quality control == 7" in one window, and those same observations are highlighted in all the other windows (for example). The 3D scatterplot can be rotated around with the mouse to really pinpoint exactly where the observations are getting rejected, for example. If the data browser (the spreadsheet-like panel) is open, the selected observations get highlighted there too.

```
fname         = 'obs_epoch_001.nc';
ObsTypeString = 'RADIOSONDE_TEMPERATURE';
ObsCopyString = 'NCEP BUFR observation';
CopyString    = 'prior ensemble mean';
QCString      = 'DART quality control';
region        = [220 300 20 60 -Inf Inf];
global obsmat;
link_obs(fname, ObsTypeString, ObsCopyString, CopyString, QCString, region)
```


DART quality control
12/03
12/04
10
5
0
obs count
4000
2000
0
12/03
12/04
x 10^5
key
3
2.5
2
1.5
0
500
1000
1500
2000
2500
3000
3500
4000
obs count
AIRCRAFT_U_WIND_COMPONENT
03-Dec-1999 09:00:01 ---> 03-Dec-1999 15:00:00
prior ensemble mean
100
80
60
40
20
0
-20
-40
-40
-20
0
20
40
60
80
100
NCEP BUFR observation


AIRCRAFT_U_WIND_COMPONENT
03-Dec-1999 09:00:01 ---> 03-Dec-1999 15:00:00
x 10^4
0
5
10
pressure
50
0
-50
latitude
0
100
200
300
longitude



6.98 DART Tutorial

The DART Tutorial is intended to aid in the understanding of ensemble data assimilation theory and consists of step-by-step concepts and companion exercises with DART.

Before beginning the DART Tutorial, make sure you are familiar with the prerequisite statistical concepts by reading *Conditional probability and Bayes' theorem*.

The diagnostics in the tutorial use Matlab®. To learn how to configure your environment to use Matlab and the DART diagnostics, see the documentation for Configuring Matlab® for netCDF & DART.

- **Section 1:** Filtering For a One Variable System
- **Section 2:** The DART Directory Tree
- **Section 3:** DART Runtime Control and Documentation
- **Section 4:** How should observations of a state variable impact an unobserved state variable? Multivariate assimilation.
- **Section 5:** Comprehensive Filtering Theory: Non-Identity Observations and the Joint Phase Space
- **Section 6:** Other Updates for An Observed Variable

- **Section 7:** Some Additional Low-Order Models
- **Section 8:** Dealing with Sampling Error
- **Section 9:** More on Dealing with Error; Inflation
- **Section 10:** Regression and Nonlinear Effects
- **Section 11:** Creating DART Executables
- **Section 12:** Adaptive Inflation
- **Section 13:** Hierarchical Group Filters and Localization
- **Section 14:** Observation Quality Control
- **Section 15:** DART Experiments: Control and Design
- **Section 16:** Diagnostic Output
- **Section 17:** Creating Observation Sequences
- **Section 18:** Lost in Phase Space: The Challenge of Not Knowing the Truth
- **Section 19:** DART-Compliant Models and Making Models Compliant: Coming Soon
- **Section 20:** Model Parameter Estimation
- **Section 21:** Observation Types and Observing System Design
- **Section 22:** Parallel Algorithm Implementation: Coming Soon
- **Section 23:** Location Module Design
- **Section 24:** Fixed Lag Smoother (not available yet)
- **Section 25:** A Simple 1D Advection Model: Tracer Data Assimilation

6.99 Conditional probability and Bayes' theorem

This section introduces two prerequisite concepts for understanding data assimilation theory: conditional probability and Bayes' theorem.

6.99.1 Conditional probability

Most real-world events involve uncertainty because the occurence of a specific outcome isn't guaranteed. You can sense that in situations in which these are possible outcomes:

- your flight departs on time
- you keep your New Year's resolution
- your car needs repairs in the next 6 months

there is a chance that the opposite outcome might occur. Describing such situations accurately requires making probabilistic statements.

In mathematical notation, the probability of an event, A, is denoted by $P(A)$. If the event A means that your flight departs on time, you can write:

$$P(A) = likely$$

since most flights do actually depart on time.

Events usually occur in conjuction with other events, so it is useful to assign conditional probabilities, or the probability that an outcome occurs if another event also occurs.

If the event B is that a blizzard approaches the airport an hour before your scheduled departure you can write a conditional probability as $P(A|B)$, or the probability that A occurs, given that B also occurs. In this case, you can assign the probability that your flight departs on time given that a blizzard approaches the airport an hour before your scheduled departure as:

$$P(A|B) = unlikely$$

since it is unlikely that your flight departs on time in a blizzard. These examples use informal, subjective probabilities. But the mathematical notation can also be used to assign formal, quantitative probabilities as well.

6.99.2 Bayes' theorem

Imagine you are in a house and the carbon monoxide detector has set off its alarm. Carbon monoxide is colorless and odorless, so you evacuate the house, but you don't know whether there are actually significant concentrations of carbon monoxide inside or if your detector is faulty.

In the United States, 100,000 carbon monoxide exposure events occur in houses annually and the manufacturer of your detector claims that its detectors have a 0.1% error rate. Bayes' theorem allows you to calculate the quantitative probability of whether or not there is a carbon monoxide exposure event in the house, given that the carbon monoxide detector has set off its alarm.

Probability theory allows you to keep track of specific conditions and events. The names of the relevant terms, and what they represent in this example are:

- the *prior*, $P(A)$ - the probability of a carbon monoxide exposure event in your house
- the *likelihood*, $P(B|A)$ - the probability your detector sets off its alarm given that there is a carbon monoxide exposure event in your house
- the *normalization*, $P(B)$ - the probablity your detector sets off its alarm
- the *posterior*, $P(A|B)$ - the probability of a carbon monoxide exposure event in your house given that your detector sets of its alarm

If this is your first experience with probability theory, you may be unaccustomed to the terminology and level of nuance that the theory affords. Take your time to think through each of the probabilities and conditions. Notice, for example, the difference between $P(B|A)$ and $P(A|B)$.

Bayes' theorem allows you to calculate the probability you want to know, the posterior, $P(A|B)$. The theorem is:

$$posterior = \frac{likelihood \times prior}{normalization}$$

or:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

To compute the right hand side of the equation you'll need to estimate the prior, the likelihood, and the normalization.

Prior

You can estimate the probability of a carbon monoxide exposure event in your house, $P(A)$, by dividing the number of carbon monoxide exposure events that occur annually in houses by the total number of houses in the United States, which is 140 million houses:

$$P(A) = 100,000 \div 140,000,000 = 7.1 \times 10^{-4}$$

Likelihood

You can estimate the probability your detector sets off its alarm given that there is a carbon monoxide exposure event in your house, $P(B|A)$, since you know the error rate of the detector, 0.1%:

$$P(B|A) = 1 - 0.001 = 0.999$$

Normalization

Estimating the probablity your detector sets off its alarm, $P(B)$, requires estimating two cases: the probability of a false alarm, $P(B^-)$, and the probability of a true alarm, $P(B^+)$.

The probability of a false alarm is the portion of the population that does not experience a carbon monoxide exposure event times the error rate of the detector:

$$P(B^-) = \frac{(140,000,000 - 100,000)}{140,000,000} \times 0.001 = 9.9 \times 10^{-4}$$

The probability of a true alarm is the portion of the population that experiences a carbon monoxide exposure event times the rate that the detector will correctly set off its alarm:

$$P(B^+) = \frac{100,000}{140,000,000} \times (1 - 0.001) = 7.1 \times 10^{-4}$$

$P(B)$ is the sum of $P(B^-)$ and $P(B^+)$:

$$P(B) = 9.9 \times 10^{-4} + 7.1 \times 10^{-4} = 1.7 \times 10^{-3}$$

Posterior

You now have all of the necessary probabilities to estimate the probability of a carbon monoxide exposure event in your house given that your detector sets off its alarm, $P(A|B)$:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{0.999 \times 7.1 \times 10^{-4}}{1.7 \times 10^{-3}} = 0.42$$

Thus, the posterior probability is 0.42.

6.99.3 Bayesian inference

One of the primary benefits of Bayes' theorem is that it can be applied multiple times to update a probability when new information is available. This process is best illustrated by continuing the example.

While standing outside, you call the fire department. A fire engine arrives and firefighters enter the house with a carbon monoxide meter. This meter is more accurate than the one installed in the house. It has an error rate of 0.01%.

The meter detects dangerous levels carbon monoxide in the house. You know intuitively that it is now highly probable that there are dangerous levels of carbon monoxide in the house. Bayes' theorem provides a rigorous framework to support your intuition.

You can apply Bayes' theorem again to update your estimate of the probability of a carbon monoxide exposure event in the house. This updating process is called Bayesian inference.

When applying Bayes' theorem a second time, the process is the same but the probabilities involved are different.

Prior

In the first part of the example, you estimated the prior by dividing the number of carbon monoxide exposure events that occur annually in houses by the total number of houses in the United States. That was the correct approach at first. But now your prior is the posterior from the first part:

$$P(A) = 0.42$$

since that is the probability of a carbon monoxide exposure event in your house.

Likelihood

Since the firefighters' carbon monoxide meter has a lower error rate than the detector installed in the house, $P(B|A)$ is also different:

$$P(B|A) = 1 - 0.0001 = 0.9999$$

Normalization

The probablity that the meter detects carbon monoxide is still comprised of two parts, the probability of a false detection, $P(B^-)$, and the probability of a true detection, $P(B^+)$. But since the error rate of the firefighters' meter is lower and your detector has also set off its alarm, the normalization is different.

The probability of a false detection is the probability that there isn't a carbon monoxide exposure event in the house times the error rate of the meter:

$$P(B^-) = (1 - 0.42) \times 0.0001 = 5.8 \times 10^{-5}$$

The probability of a true detection is the probability that there is a carbon monoxide exposure event in the house times the rate that the meter will correctly detect it:

$$P(B^+) = 0.42 \times 0.9999 = 0.42$$

$P(B)$ is the sum of $P(B^-)$ and $P(B^+)$:

$$P(B) = 5.8 \times 10^{-5} + 0.42 = 0.42$$

Posterior

You have all of the necessary probabilities to estimate the probability of a carbon monoxide exposure event in your house given that both your detector set off its alarm and the firefighters' meter also detected carbon monoxide, $P(A|B)$:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{0.9999 \times 0.42}{0.42} = 0.9999$$

Thus, the second posterior probability is 0.9999. This makes sense intuitively: it is extremely likely that there is a carbon monoxide exposure event in the house if both your alarm and the firefighters' meter detect carbon monoxide.

It also demonstrates the ability of Bayes' theorem to update the probability of an event when new information becomes available.

With these concepts you can now begin the *DART Tutorial*.

6.100 DART_LAB Tutorial

6.100.1 Overview

The files in this directory contain PDF tutorial materials on DART, and Matlab exercises. See below for links to the PDF files and a list of the corresponding matlab scripts.

This tutorial begins at a more introductory level than the materials in the tutorial directory, and includes hands-on exercises at several points. In a workshop setting, these materials and exercises took about 1.5 days to complete.

6.100.2 DART tutorial presentations

Here are the PDF files for the presentation part of the tutorial:

- `Section 1:` The basics in 1D.
- `Section 2:` How should observations of a state variable impact an unobserved state variable? Multivariate assimilation.
- `Section 3:` Sampling error and localization.
- `Section 4:` The Ensemble Kalman Filter (Perturbed Observations).
- `Section 5:` Adaptive Inflation.

6.100.3 Matlab hands-on exercises

In the `matlab` subdirectory are a set of Matlab scripts and GUI (graphical user interface) programs which are exercises that go with the tutorial. Each is interactive with settings that can be changed and rerun to explore various options. A valid Matlab license is needed to run these scripts.

The exercises use the following functions:

- gaussian_product
- oned_model
- oned_ensemble
- run_lorenz_63
- run_lorenz_96
- twod_ensemble

To run these, cd into the DART_LAB/matlab directory, start matlab, and type the names at the prompt.

6.101 CLM5-DART Tutorial

6.101.1 Introduction

This tutorial describes how to set up a simple assimilation using CLM5-DART. The expectation is that the user is very familiar with running CLM off-line and thus the tutorial focuses upon the specific aspects of integrating CLM with DART. Once completed, the goal is that the user should be familar enough with the concepts of CLM5-DART in order to design an assimilation for their own research interests.

This tutorial was assembled to be compatible with CLM5.0.34 as part of CESM (release-cesm2.2.01) and the latest release on the `main` branch of the the DART repository. Other combinations of CLM and DART (prior to DART version 9.13.0) may not be compatible with this tutorial.

It is not recommended to use this tutorial without prior experience with running and understanding CLM. If new to CLM we recommend you attend a CLM tutorial or review the CLM documentation including the Technical Description, User Guide, and use the CESM User Forum for CLM-specific assistance.

If you are new to DART, we recommend that you become familiar by first working through the *DART Tutorial* and then understanding the *DART getting started* documentation.

This CLM5-DART tutorial is based on a simple example in which we describe how to:

1) set up the software

2) review concepts of assimilation

3) provide a specific example of how CLM and DART interact in an assimilation framework

If you wish to move beyond this tutorial and address a unique/new research question be prepared to invest a significant amount of time to modify the CLM-DART setup. In general, a CLM-DART simulation that is running without errors, does not mean it has been optimized for performance. Optimal performance requires the evaluation of assimilation diagnostics (e.g. observation acceptance rate, RMSE and bias between modeled and observed properties) and making additional modifications to the setup scripts. Examples of setup modifications include changes to localization, inflation, observations that are assimilated, updated model state variables, assimilation time window, assimilation frequency, and the model and observation grid resolution.

We encourage users to complete the tutorial and then modify CLM-DART to pursue their own research questions. We have comprehensive and searchable *documentation*, with trained and experienced staff that can help troubleshoot issues (dart@ucar.edu).

6.101.2 Tutorial Overview

The tutorial is a simplified global simulation that assimilates daily observations for up to a maximum of 10 days. A total of 5 observations are distributed across the globe that includes observations of leaf area, biomass, snow cover, and soil temperature. The observations are 'synthetic' meaning they have been generated from the output of a separate CLM simulation with random error added. Synthetic observations are generally easier to assimilate than real-world observations because they remove systemic differences between the model and observations. This assimilation uses a 5-member ensemble (i.e. 5 CLM simulations run simultaneously). An ensemble is required within Ensemble Kalman Filter (EnKF) data assimilation to calculate the modeled covariance amongst the state variables in CLM. The covariance amongst the CLM state variables dictates how the CLM model state is updated during the assimilation step given a set of observations. The beginning of the assimilation starts from near present day (January-2011) and is initalized in 'hybrid' mode from a set of CLM restart files generated from a previous CLM 5-member ensemble simulation. The atmospheric forcing used for the assimilation comes from the Community Atmospheric Model (CAM) reanalysis (ds199.1 and ds345.0). This reanalysis atmospheric data includes 80 total ensemble members in which the across-member variation represents atmospheric uncertainty. We use 5 different ensemble members from the CAM6 reanalysis to generate spread within this CLM5-DART tutorial simulation.

Note: This CLM5-DART tutorial has been simplified to reduce run-time and computational expense. For research applications an assimilation is typically run for many model months/years. In addition a 5-member ensemble is generally too small to properly sample model uncertainty and we recommend 80, but no less than 40 ensemble members for research applications.

The goal of this tutorial is to demonstrate how CLM5 and DART interact to perform data assimilation to provide an observation-constrained simulation of CLM. This requires setting numerous namelist/input values that control the configuration of CLM and DART setup.

After running this tutorial, you should be able to understand the major steps involved in setting up your own data assimilation experiments. However, you will need to do additional work to customize the CLM5-DART system for your particular research needs. For example, this tutorial provides both the initial conditions and observation sequence files to simplify the process and are specific to the tutorial, whereas you will need to generate custom initial conditions and observation sequence files for your own work.

Important: We have provided tutorial instructions for the NCAR supercomputer Cheyenne, however, if using your own machine you will need to customize the setup scripts in order to properly compile DART (see Step 4: Compiling DART). These system-specific setup steps may take a good deal of effort, especially if you are unfamiliar with details such as compilers, MPI, NetCDF libraries, batch submission systems etc. To perform this tutorial we assume the user is comfortable with LINUX operating systems as well as using text editors (e.g. vi, nedit, emacs) to edit the CLM5-DART setup scripts and namelist files etc.

Other required files to run the tutorial include the meteorology (Step 5), reference case (Step 6), and observation files (Step 7). These are all readily available if you are using Cheyenne. If you are using your own machine you need use the following links to download these files directly:

1. CAM6 Reanalysis Meteorology, Year 2011, ensemble members 1-5 only: `f.e21.FHIST_BGC.f09_025.CAM6assim.011.cpl_000{1-5}.ha2x3h.2011.nc`
2. Reference Case and Observations

6.101.3 Step 1: Download CLM5

CLM is continually being updated by the model developer and user community consisting of both NCAR and university scientists and researchers. In contrast, DART is maintained by a relatively small group that supports numerous earth system models (20+) including CLM. Therefore the DART team focuses on only supporting official released versions of CLM. This documentation and scripting was tested using the CESM tag `release-cesm2.2.0` following the download instructions here.

Although the DART code may work with more recent versions of CESM (CLM) we recommend checking out `release-cesm2.2.0`.

```
git clone https://github.com/escomp/cesm.git cesm_dart
cd cesm_dart
git checkout release-cesm2.2.0
./manage_externals/checkout_externals
```

6.101.4 Adding CLM5 SourceMods

Some minor modifications have to be made to the CLM5 source code in order to be run with DART. Most importantly, these include skipping several balance checks in CLM5 for the time step immediately after the assimilation update step. These sourcecode modifications are brought in through the SourceMod mechanism in CLM where modifications overwrite the template sourcecode during the compilation step. The SourceMods are located as tar files here. For this tutorial retrieve the most recent tar file `DART_SourceMods_cesm2_2_0_2021_07_02.tar` and untar it on your local machine as:

```
wget https://www.image.ucar.edu/pub/DART/CESM/DART_SourceMods_cesm2_2_0_2021_07_02.tar
tar -xvf DART_SourceMods_cesm2_2_0_2021_07_02.tar
```

For more information on the SourceMods see the main *CLM-DART documentation.*

6.101.5 Compiling CLM5

Compiling CLM5 on the NCAR machine Cheyenne is straightforward because the run and build environment settings are already defined within the `config_machines.xml` file located within the CESM installation: `<cesmroot>/cime/config/cesm/machines`. If you are using your own machine please follow the porting instructions located here. When performing a CLM5-DART assimilation run, the compiling step for CLM5 occurs within the `CLM5_setup_assimilation` script described later within this tutorial.

6.101.6 Step 2: Download DART

The tutorial material is available within the most recent release of the DART repository on the main branch.

```
cd /glade/work/$USER/
git clone https://github.com/NCAR/DART.git
cd DART
```

6.101.7 Step 3: Navigating DART Scripts

Below is a table of the key CLM5-DART setup scripts that include the settings required to perform an assimilation run. All scripts are located at path `DART/models/clm/shell_scripts/cesm2_2/` with the exception of `input.nml` which is located at `DART/models/clm/work/`.

You will be asked to locate, edit, and execute these scripts during the tutorial and they are critical to customizing the assimilation run. Given their importance, we introduce them right away.

For additional description of the CLM5-DART scripts and concepts please visit the main *CLM-DART documentation* In general, feel free to supplement this tutorial with the main CLM5-DART documentation. In some cases it will provide more detailed information than in this tutorial. If a concept is unclear we recommend using the search bar at the top of this page.

Key CLM5-DART scripts	Description
CLM5_setup_assimilation.csh	Main setup script that creates a CLM5-DART assimilation case. This script sets up `case`, `run` and `archive` directories and stages the necessary files to the `run` directory to create a `hybrid` run. It uses traditional commands `create_newcase`, `case.setup`, `preview_namelists` and `case.build` included as part of the CESM package.
DART_params.csh	This companion script to `CLM5_setup_assimilation` that defines important case settings for CLM and DART. The majority of case setting edits occur within this script.
CESM_DART_config	Once the case is created, this script turns 'on' assimilation by providing links between the CLM and DART code. Converts a 'free' run into an assimilation run.
assimilate.csh	This script is executed during the assimilation case run-time. It orchestrates communication between CLM and DART. After the CLM forecast step is complete, it calls upon the DART executables to relay the CLM files (restart, history) to DART and executes the `filter` step to update the CLM state variable. These updated files are then reinserted into the restart file for the next CLM forecast step.
input.nml	Contains DART specific namelist settings such as DA type, inflation, localization, outlier threshold etc.

6.101.8 Step 4: Compiling DART

Similar to CLM, it is necessary to compile the DART code before an assimilation can be performed. The DART code includes a variety of build template scripts that provide the appropriate compiler and library settings depending upon your system environment. This is an example of the default system environment for Cheyenne (e.g. `module list`), which was used to perform this tutorial:

```
Currently Loaded Modules:
  1) ncarenv/1.3   2) intel/19.0.5   3) ncarcompilers/0.5.0   4) mpt/2.22   5) netcdf/4.
↪7.4
```

Please note in this example we used the `intel` fortran compiler with `netcdf` libraries to support the netcdf file format and the `mpt` libraries to support the `mpi` tasks.

Below are instructions on how to modify the DART template script `mkmf_template` to properly compile DART on Cheyenne:

```
cd DART/build_templates
cp mkmf.template.intel.linux mkmf.template
```

Confirm the `mkmf_template` has the following settings:

```
MPIFC = mpif90
MPILD = mpif90
FC = ifort
LD = ifort
...
...
INCS = -I$(NETCDF)/include
LIBS = -L$(NETCDF)/lib -lnetcdff -lnetcdf
FFLAGS  = -O -assume buffered_io $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

Next we will test to make sure the DART scripts can be run correctly, by compiling and executing the `preprocess` script. The `preprocess` script must be run **before** the core DART code is compiled because it writes the source code that supports the observations. This provides the necessary support for the specific observations that we wish to assimilate into CLM. For more information see the *preprocess documentation.*

First make sure the list of `obs_def` and `obs_quantity` module source codes are contained in the `&preprocess_nml` namelist within the `input.nml`.

```
cd DART/models/clm/work
vi input.nml
```

Note: We use the **vi editor** within the tutorial instructions, but we recommend that you use the text editor you are most comfortable with. To close the vi editor follow these instructions from stackoverflow.

This example uses namelist setting that specifically loads `obs_def` and `obs_quantity` commonly used for land DA, including models like CLM. Confirm the settings are as follows:

```
&preprocess_nml
   input_obs_qty_mod_file  = '../../../assimilation_code/modules/observations/DEFAULT_
↪obs_kind_mod.F90'
   output_obs_qty_mod_file = '../../../assimilation_code/modules/observations/obs_kind_
↪mod.f90'
   input_obs_def_mod_file  = '../../../observations/forward_operators/DEFAULT_obs_def_
↪mod.F90'
   output_obs_def_mod_file = '../../../observations/forward_operators/obs_def_mod.f90'
   obs_type_files          = '../../../observations/forward_operators/obs_def_land_mod.
↪f90',
                             '../../../observations/forward_operators/obs_def_tower_mod.
↪f90',
                             '../../../observations/forward_operators/obs_def_COSMOS_mod.
↪f90'
   quantity_files          = '../../../assimilation_code/modules/observations/land_
↪quantities_mod.f90',
                             '../../../assimilation_code/modules/observations/space_
↪quantities_mod.f90'
                             '../../../assimilation_code/modules/observations/atmosphere_
↪quantities_mod.f90'
   /
```

Next run quickbuild.sh to build and run preprocess and build the dart exectuables:

```
./quickbuild.sh
```

Confirm the new source code has been generated for `DART/observations/forward_operators/obs_def_mod.f90` and `DART/assimilation_code/modules/observations/obs_kind_mod.f90`

6.101.9 Step 5: Setting up the atmospheric forcing

A requirement for Ensemble Kalman Filter (EnKF) type DA approaches is to generate multiple model simulations (i.e. a model ensemble) that quantifies 1) state variable uncertainty and 2) correlation between state variables. Given the sensitivity of CLM to atmospheric conditions an established method to generate multi-instance CLM simulations is through weather reanalysis data generated from a CAM-DART assimilation. These CAM-DART reanalyses are available from 1997-2010 ds199.1, and 2011-2020 ds345.0.

For this tutorial we will use the January 2011 CAM6 reanalysis (ds345.0) only. To make sure the scripts can locate the weather data first make sure the `DART_params.csh` variable `dartroot` is set to the path of your DART installation. For example, if you have a Cheyenne account and you followed the DART cloning instructions in Step 2 above your `dartroot` variable will be: `/<your Cheyenne work directory>/DART`. Make sure you update the default `dartroot` as shown below.

```
setenv dartroot          /glade/work/${USER}/DART
```

Next confirm within the `CLM5_setup_assimilation` script that the path (`${SOURCEDIR}/${STREAMFILE_*}`) to all four of your atmospheric stream file templates (e.g. `datm.streams.txt.CPLHISTForcing.Solar*`) is correct. In particular make sure the `SOURCEDIR` variable is set correctly below:

```
set STREAMFILE_SOLAR        = datm.streams.txt.CPLHISTForcing.Solar_single_year
set STREAMFILE_STATE1HR     = datm.streams.txt.CPLHISTForcing.State1hr_single_
↪year
set STREAMFILE_STATE3HR     = datm.streams.txt.CPLHISTForcing.State3hr_single_
↪year
set STREAMFILE_NONSOLARFLUX = datm.streams.txt.CPLHISTForcing.nonSolarFlux_
↪single_year
...
...
# Create stream files for each ensemble member
set SOURCEDIR = ${dartroot}/models/clm/shell_scripts/cesm2_2
${COPY} ${SOURCEDIR}/${STREAMFILE_SOLAR}         user_${FILE1} || exit 5
${COPY} ${SOURCEDIR}/${STREAMFILE_STATE1HR}      user_${FILE2} || exit 5
${COPY} ${SOURCEDIR}/${STREAMFILE_STATE3HR}      user_${FILE3} || exit 5
${COPY} ${SOURCEDIR}/${STREAMFILE_NONSOLARFLUX}  user_${FILE4} || exit 5
```

Next, edit each of your atmospheric stream file templates to make sure the `filePath` within `domainInfo` and `fieldInfo` below is set correctly to reference the CAM6 reanalysis file. The example below is for `datm.streams.txt.CPLHISTForcing.nonSolarFlux_single_year`. Repeat this for all four of the template stream files including for `Solar`, `State1hr` and `State3hr`.

```
<domainInfo>
     <variableNames>
        time          time
        doma_lon      lon
        doma_lat      lat
        doma_area     area
        doma_mask     mask
     </variableNames>
     <filePath>
        /glade/collections/rda/data/ds345.0/cpl_unzipped/NINST
     </filePath>
     <fileNames>
        f.e21.FHIST_BGC.f09_025.CAM6assim.011.cpl_NINST.ha2x3h.RUNYEAR.nc
```

(continues on next page)

(continued from previous page)

```
    </fileNames>
 </domainInfo>
 ...
 ...
 ...
 <fieldInfo>
    <variableNames>
       a2x3h_Faxa_rainc     rainc
       a2x3h_Faxa_rainl     rainl
       a2x3h_Faxa_snowc     snowc
       a2x3h_Faxa_snowl     snowl
       a2x3h_Faxa_lwdn      lwdn
    </variableNames>
    <filePath>
         /glade/collections/rda/data/ds345.0/cpl_unzipped/NINST
    </filePath>
    <offset>
       1800
    </offset>
    <fileNames>
       f.e21.FHIST_BGC.f09_025.CAM6assim.011.cpl_NINST.ha2x3h.RUNYEAR.nc
    </fileNames>
 </fieldInfo>
```

Selected variables within atmospheric stream file	Description
filePath	Directory of CAM6 reanalysis file. For tutorial, this only includes year 2011, with ensemble members 1-5. During execution of `CLM5_setup_assimilation` the text `NINST` is replaced with ensemble member number `0001-0005`. The ensemble member number is set through the `num_instances` variable located in `DART_params.csh`.
fileNames	The CAM6 reanalysis file name. For the tutorial, this only includes year 2011, with ensemble members 1-5. The `NINST` variable is replaced in the same way as described above for `filepath`. For this tutorial the `RUNYEAR` variable will be replaced by `2011`. The `RUNYEAR` variable is set through `stream_year_first` located within `DART_params.csh`.
variableNames	Meteorology variables within CAM6 reanalysis. First column is variable name within netCDF reanalysis file, whereas the second column is the meteorology variable name recognized by CLM.

Finally, edit the `DART_params.csh` file such that the `RUNYEAR` and `NINST` variables within the atmospheric stream templates are replaced with the appropriate year and ensemble member. To do this confirm the settings within `DART_params.csh` are as follows:

```
setenv num_instances  5

 ..
 ..
```

(continues on next page)

(continued from previous page)

```
setenv stream_year_align 2011
setenv stream_year_first 2011
setenv stream_year_last  2011
```

6.101.10 Step 6: Setting up the initial conditions for land earth system properties

The initial conditions for the assimilation run are prescribed (all state variables from the top of vegetation canopy to subsurface bedrock) by a previous 5-member ensemble run (Case: `clm5.0.06_f09_80`) that used the same CAM6 reanalysis to generate initial spread between ensemble members. This is sometimes referred to as an ensemble 'spinup'. This ensemble spinup was run for 10 years to generate sufficient spread amongst ensemble members for this tutorial.

Note: The proper ensemble spinup time depends upon the specific research application. In general, the goal is to allow the differences in meterological forcing to induce changes within the CLM variables that you plan to adjust during the DART update step. CLM variables that have relatively quick response to atmospheric forcing (e.g. leaf area, shallow-depth soil variables) require less spinup time. However, other CLM variables take longer to equilibrate to atmospheric forcing (e.g. biomass, soil carbon).

This initial ensemble spinup was run with resolution `f09_09_mg17` (0.9x1.25 grid resolution) with compset `2000_DATM%GSWP3v1_CLM50%BGC-CROP_SICE_SOCN_MOSART_SGLC_SWAV` (CESM run with only land and river components active). The starting point of the assimilation is run in CLM 'hybrid' mode which allows the starting date of the assimilaton to be different than the reference case, and loosens the requirements of the system state. The tradeoff is that restarting in hybrid mode does not provide bit-by-bit reproducible simulations.

For the tutorial, set the `DART_parms.csh` variables such that the end of the ensemble spinup (at time 1-1-2011) are used as the initial conditions for the assimilation:

```
setenv refcase      clm5.0.06_f09_80
setenv refyear      2011
setenv refmon       01
setenv refday       01
setenv reftod       00000
...
...
setenv stagedir /glade/p/cisl/dares/RDA_strawman/CESM_ensembles/CLM/CLM5BGC-Crop/ctsm_$
↪{reftimestamp}
...
...
setenv start_year    2011
setenv start_month   01
setenv start_day     01
setenv start_tod     00000
```

Important variables to set initial conditions	Description
refcase	The reference casename from the spinup ensemble that serves as the starting conditions for the assimilation.
refyear, refmon, refday reftod	The year, month, day and time of day of the reference case that the assimilation will start from.
stagedir	The directory location of the reference case files.

6.101.11 Step 7: Setting up the observations to be assimilated

In 'Step 4: Compiling DART' we have already completed an important step by executing `preprocess` which generates source code (`obs_def_mod.f90`, `obs_kind_mod.f90`) that supports the assimilation of observations used for this tutorial. In this step, we compile these observation definitions in to the DART executables. The observations are read into the assimilation through an observation sequence file whose format is described *here.*

First confirm that the `baseobsdir` variable within `DART_params.csh` is pointed to the directory where the observation sequence files are located. In Cheyenne they are located in the directory as:

```
setenv baseobsdir              /glade/p/cisl/dares/Observations/land
```

In this tutorial we have several observation types that are to be assimilated, including `SOIL_TEMPERATURE`, `MODIS_SNOWCOVER_FRAC`, `MODIS_LEAF_AREA_INDEX` and `BIOMASS`. To enable the assimilation of these observations types they must be included within the `&obs_kind_nml` within the `input.nml` file as:

```
&obs_kind_nml
  assimilate_these_obs_types = 'SOIL_TEMPERATURE',
                               'MODIS_SNOWCOVER_FRAC',
                               'MODIS_LEAF_AREA_INDEX',
                               'BIOMASS',
  evaluate_these_obs_types   = 'null'
  /
```

Below is an example of a single observation (leaf area index) within an observation sequence file used within this tutorial (`obs_seq.2011-01-02-00000`):

```
OBS            3
6.00864688253571
5.44649167346675
0.000000000000000E+000

obdef
loc3d
   5.235987755982989         0.000000000000000        -888888.0000000000      -2
kind
         23
   0     149750
0.200000000000000
```

Below is the same portion of the file as above, but with the variable names:

```
<Observation sequence number>
<Observation Value>
<True Observation Value>
<Observation Quality Control>

obdef
loc3d
   <longitude>    <latitude>     <vertical level>     <vertical code>
kind
  <observation quantity number>
     <seconds>    <days>
<Observation error variance>
```

Observation Sequence File Variable	Description
observation sequence number	The chronological order of the observation within the observation sequence file. This determines the order in which the observation is assimilated by DART for a given time step.
observation value	The actual observation value that the DART `filter` step uses to update the CLM model. This is derived from the true observation value generated from CLM model output with uncertainty added.
true observation value	The observation generated from CLM output. In this case the observation was generated as part of a perfect model experiment (OSSE; Observing System Simulation Experiment), thus the 'true' value is known.
observation quality control	The quality control value provided from the data provider. This can be used as a filter in which to exclude low quality observations from the assimilation.
longitude, latitude	Horizontal spatial location of the observation in radians
level, vertical level type code	Vertical observation location in units defined by vertical level type
observation type number	The DART observation type assigned to the obervation type (e.g. `MODIS_LEAF_AREA_INDEX (23) –> QTY_LEAF_AREA_INDEX`)
second, days	Time of the observations in reference to Jan 1, 1601
observation error variance	Uncertainty of the observation Value

Now that we have set both the path to the observation sequence files, and the types of observations to be assimilated, confirm the quality control settings within the `&quality_control_nml` of the `input.nml` file are as follows:

```
&quality_control_nml
   input_qc_threshold = 1.0
   outlier_threshold  = 3.0
   /
```

Quality Control Namelist	Description
input_qc_threshold	The quality control value that is provided from the observation product. Any value above this threshold will cause the observation to be rejected and ignored during the assimilation step.
outlier threshold	The observation is rejected if: (prior mean - observation) > (expected difference x outlier threshold). The prior mean is is calculated from the CLM model ensemble mean, and the expected difference is the square root of the sum of the square uncertainty of the prior mean and observation uncertainty.

These quality control settings do not play a role in this tutorial because we are using synthetic observations which are, by design, very close to the model output. Thus, in this tutorial example, systematic biases between the model and observations are removed. However, in the case of real observations, it is common for large systemic differences to occur between the model and observations either because 1) structural/parametric error exists within the model or 2) model or observation uncertainty is underestimated. In these cases it is beneficial to reject observations to promote a stable simulation and prevent the model from entering into unrealistic state space.

Note: This tutorial already provides properly formatted synthetic observations for the user, however, when using 'real' observations for research applications DART provides *observation converters.* Observation converters are scripts that convert the various data product formats into the observation sequence file format required by the DART code. Observations converters most relevant for land DA and the CLM model include those for *leaf area, flux data, snow,*

and *soil moisture here* and *here.* Even if an observation converter is not available for a particular data product, it is generally straightforward to modify them for your specific application.

6.101.12 Step 8: Setting up the DART and CLM states

Defining the DART state space is a critical part of the assimilation setup process. This serves two purposes, first, it defines which model variables are used in the forward operator. The forward operator is defined as any operation that converts from model space to observation space to create the 'expected observation'. The mismatch between the true and expected observation forms the foundation of the model update in the DART `filter` step.

In this tutorial, observations of `SOIL_TEMPERATURE`, `MODIS_SNOWCOVER_FRAC`, `MODIS_LEAF_AREA_INDEX`, and `BIOMASS` are supported by specific clm variables. See the table below which defines the dependency of each DART **observation type** upon specific DART **quantities** required for the forward operator. We also include the CLM variables that serve as the DART quantities for this tutorial:

DART Observation Type	DART Observation Quantities	CLM variables
SOIL_TEMPERATURE	QTY_SOIL_TEMPERATURE QTY_TEMPERATURE	TSOI T_SOISNO
MODIS_SNOWCOVER_FRAC	QTY_SNOWCOVER_FRAC	frac_sno
MODIS_LEAF_AREA_INDEX	QTY_LEAF_AREA_INDEX	TLAI
BIOMASS	QTY_LEAF_CARBON QTY_LIVE_STEM_CARBON QTY_DEAD_STEM_CARBON	leafc livestemc deadstemc

Note: For this tutorial example most of the **observation types** rely on a single **quantity** (and CLM variable) to calculate the expected observation. For these the CLM variable is spatially interpolated to best match the location of the observation. The `BIOMASS` observation type is an exception in which 3 **quantities** are required to calculate the expected observation. In that case the sum of the CLM variables of leaf, live stem and structural (dead) carbon represents the biomass observation.

Second, the DART state space also defines which portion of the CLM model state is updated by DART. In DA terminology, limiting the influence of the observations to a subset of the CLM model state is known as 'localization' which is discussed more fully in Step 9. In theory the complete CLM model state may be updated based on the relationship with the observations. In practice, a smaller subset of model state variables, that have a close physical relationship with the observations, are included in the DART state space. In this tutorial, for example, we limit the update to CLM variables most closely related to biomass, leaf area, soil temperature and snow. Modify the `&model_nml` within `input.nml` as below:

```
&model_nml
  ...
  ...
  clm_variables  = 'leafc',       'QTY_LEAF_CARBON',           '0.0', 'NA', 'restart' ,
↪'UPDATE',
                   'frac_sno',    'QTY_SNOWCOVER_FRAC',        '0.0', '1.', 'restart' ,
↪'NO_COPY_BACK',
                   'SNOW_DEPTH',  'QTY_SNOW_THICKNESS',        '0.0', 'NA', 'restart' ,
↪'NO_COPY_BACK',
                   'H2OSOI_LIQ',  'QTY_SOIL_LIQUID_WATER',     '0.0', 'NA', 'restart' ,
↪'UPDATE',
                   'H2OSOI_ICE',  'QTY_SOIL_ICE',              '0.0', 'NA', 'restart' ,
↪'UPDATE',
```

(continues on next page)

(continued from previous page)

```
                  'T_SOISNO',    'QTY_TEMPERATURE',            '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'livestemc',   'QTY_LIVE_STEM_CARBON',       '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'deadstemc',   'QTY_DEAD_STEM_CARBON',       '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'TLAI',        'QTY_LEAF_AREA_INDEX',        '0.0', 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                  'TSOI',        'QTY_SOIL_TEMPERATURE',       'NA' , 'NA', 'history' ,
↪'NO_COPY_BACK'
 /
```

The table below provides a description for each of the columns for `clm_variables` within `&model_nml`.

Column	Description
1	The CLM variable name as it appears in the CLM netCDF file.
2	The corresponding DART QUANTITY.
3	Minimum value of the posterior. If set to 'NA' there is no minimum value. The DART diagnostic files will not reflect this value, but the file used to restart CLM will.
4	Maximum value of the posterior. If set to 'NA' there is no maximum value. The DART diagnostic files will not reflect this value, but the file used to restart CLM will.
5	Specifies which file should be used to obtain the variable. `'restart'` => clm_restart_filename `'history'` => clm_history_filename `'vector'` => clm_vector_history_filename
6	Should `filter` update the variable in the specified file. `'UPDATE'` => the variable is updated. `'NO_COPY_BACK'` => the variable remains unchanged.

There are **important** distinctions about the `clm_variables` as described above. **First**, any clm variable whether it is a `restart`, `history` or `vector` file can be used as a forward operator to calculate the expected observation. Also if the 6th column is defined as `UPDATE`, then that variable is updated during the `filter` step regardless of the CLM variable

type. **However**, in order for the update step to have a permanent effect upon the evolution of the CLM model state, the update must be applied to a prognostic variable in CLM – which is always the `restart` file. Updates to `restart` file variables alters the file thus changing the initial conditions for the next time step. The CLM `history` and `vector` files, on the other hand, are diagnostic variables with no impact on the evolution of the model state.

A **second** important distinction amongst `clm_variables` is that the `restart` file state variables are automatically generated after each CLM simulation time step, thus are readily available to include within the DART state. In contrast, the `history` or `vector` file variables must be manually generated through the `user_nl_clm` file within CLM. This is generated within the portion of the `CLM5_setup_assimilation` script as shown below. Modify this portion of the `CLM5_setup_assimilation` script so that it appears as follows:

```
echo "hist_empty_htapes = .true."                                          >> ${fname}
echo "hist_fincl1 = 'NEP','H2OSOI','TSOI','EFLX_LH_TOT','TLAI'"            >> ${fname}
echo "hist_fincl2 = 'NEP','FSH','EFLX_LH_TOT_R','GPP'"                     >> ${fname}
echo "hist_fincl3 = 'NEE','H2OSNO','TLAI','TWS','SOILC_vr','LEAFN'"        >> ${fname}
echo "hist_nhtfrq = -$stop_n,1,-$stop_n"                                   >> ${fname}
echo "hist_mfilt  = 1,$h1nsteps,1"                                         >> ${fname}
echo "hist_avgflag_pertape = 'A','A','I'"                                  >> ${fname}
echo "hist_dov2xy = .true.,.true.,.false."                                 >> ${fname}
echo "hist_type1d_pertape = ' ',' ',' '"                                   >> ${fname}
```

The `hist_fincl` setting generates history files (`fincl1->h0`; `fincl2->h1`; `fincl3->h2`) for each of the clm variables as defined above. The `hist_dov2xy` setting determines whether the history file is output in structured gridded format (`.true.`) or in unstructured, vector history format (`.false.`). Most of the history files variables in this example are provided just for illustration, however, the tutorial requires that the `TLAI` variable is output in vector history format.

The `restart`, `history` and `vector` files define domains 1, 2 and 3 respectively within DART. The `restart` domain (domain 1) must always be defined, however domains 2 and 3 are optional. In this tutorial example all 3 domains are required, where domain 2 corresponds to the `h0` history file, and domain 3 corresponds with the `h2` history files.

6.101.13 Step 9: Set the spatial localization

Localization is the term used to restrict the portion of the state to regions related to the observation. Step 8 is a type of localization in that it restricts the state update to a subset of CLM variables. Here, we further restrict the influence of the observation to the state space most nearly physically collocated with the observation. The spatial localization is set through the the `assim_tools_nml`, `cov_cutoff_nml` and `location_nml` settings witin `input.nml` as:

```
# cutoff of 0.03 (radians) is about 200km
&assim_tools_nml
   filter_kind                     = 1
   cutoff                          = 0.05
```

```
&cov_cutoff_nml
  select_localization = 1
  /
```

```
&location_nml
   horiz_dist_only             = .true.
```

Localization namelist variable	Description
`cutoff`	Value (radians) of the half-width of the localization radius. At 2* `cutoff` distance between observation and model state, the observation has no impact on state.
`select_localization`	Defines a function that determines the decreasing impact an observation has on the model state. Value of 1 is the Gaspari-Cohn function.
`horiz_dist_only`	If `.true.` localization applied only horizontally. If `.false.` localization also applied in vertical.

In some research applications (not this tutorial) it may also be important to localize in the vertical direction. For land modeling this could be important for soil carbon or soil moisture variables which typically only have observations near the land surface, whereas the model state is distributed in layers well below the surface. For vertical localization the `horiz_dist_only` must be set to `.false.` For more information on localization see *assim_tools_mod.*

6.101.14 Step 10: Set the Inflation

Generating and maintaining ensemble spread during the assimilation allows for the covariance to be calculated between model state variables (that we want to adjust) and the expected observation. The strength of the covariance determines the model update. For CLM-DART assimilations the ensemble spread is generated through a boundary condition: the atmospheric forcing as described in Step 5. However, because the number of ensemble members is limited and boundary condition uncertainty is only one source of model uncertainty, the true ensemble spread is undersampled. To help compensate for this we employ **inflation** during the assimilation which changes the spread of the ensemble without changing the ensemble mean. The **inflation** algorithm computes the ensemble mean and standard deviation for each variable in the state vector in turn, and then moves the member's values away from the mean in such a way that the mean remains unchanged.

Although **inflation** was originally designed to account for ensemble sampling errors, it has also been demonstrated to help address systemic errors between models and observations as well. More information on inflation can be found *here.*

In this tutorial we implement a time and space varying inflation (inflation flavor 5: enhanced spatial-varying; inverse gamma) such that the inflation becomes an added state property which is updated during each assimilation step similar to CLM state variables. The inflation state properties include both a mean and standard deviation. The mean value determines how much spread is added across the ensemble (spread is generated when mean > 1). The standard deviation defines the certainty of the mean inflation value, thus a small value indicates high certainty and slow evolution of the mean with time. Conversely a high standard deviation indicates low certainty and faster evolution of the inflation mean with time.

Modify the inflation settings within `input.nml` for the `&filter_nml` and the `&fill_inflation_restart_nml` as follows:

Note: The `&filter_nml` has two columns, where column 1 is for prior inflation and column 2 is for posterior inflation. We only use prior inflation for this tutorial, thus inf_flavor=0 (no inflation) for column 2.

```
&filter_nml

  inf_flavor                  = 5,                       0
  inf_initial_from_restart    = .true.,                 .false.
  inf_sd_initial_from_restart = .true.,                 .false.
  inf_deterministic           = .true.,                  .true.
  inf_initial                 = 1.0,                     1.0
```

(continues on next page)

(continued from previous page)

```
  inf_lower_bound             = 0.0,                    1.0
  inf_upper_bound             = 20.0,                  20.0
  inf_damping                 = 0.9,                    0.9
  inf_sd_initial              = 0.6,                    0.6
  inf_sd_lower_bound          = 0.6,                    0.6
  inf_sd_max_change           = 1.05,                   1.05
```

```
&fill_inflation_restart_nml
   write_prior_inf    = .true.
   prior_inf_mean     = 1.00
   prior_inf_sd       = 0.6
```

Inflation namelist variable	Description
inf_flavor	The inflation algorithm type as described below: • 0: No inflation (Prior and/or Posterior) and all other inflation variables are ignored • 2: Spatially-varying state space inflation (gaussian) • 3: Spatially-uniform state space inflation (gaussian) • 4: Relaxation To Prior Spread (Posterior inflation only) • 5: Enhanced Spatially-varying state space inflation (inverse gamma)
inf_initial_from_restart	If .true. will read inflation settings from file named input_{prior,post}inf_mean.nc. If .false. will take inflation settings from &filter_nml.
inf_sd_initial_from_restart	If .true. will read inflation settings from file named input_{prior,post}inf_sd.nc. If .false. will take inflation settings from &filter_nml.
inf_deterministic	If .true. inflation is determinstic, if .false. inflation is stochastic
inf_initial	Initial value of inflation if not read from restart file
inf_lower_bound	Lower bound of inflation mean value
inf_upper_bound	Upper bound of inflation mean value
inf_damping	Damping factor for inflation mean values. The difference between the current inflation value and 1.0 is multiplied by this factor and added to 1.0 to provide the next inflation mean. An inf_damping = 0 turns inflation off, and inf_damping =1 turns damping off.
inf_sd_initial	Initial value of inflation standard deviation if not read from restart file. If 0, do not update the inflation values, so they are time-constant. If positive, the inflation values will adapt through time.
inf_sd_lower_bound	Lower bound for inflation standard deviation. If using a negative value for inf_sd_initial this should also be negative to preserve the setting.
inf_sd_max_change	For inf_flavor 5 (enhanced inflation), controls the maximum change of the inflation standard deviation when adapting for the next assimilation cycle. The value should be between 1.0 and 2.0. 1.0 prevents any changes, while 2.0 allows 100% change. For the enhanced inflation option, if the standard deviation initial value is equal to the standard deviation lower bound, the standard deviation will not adapt in time.
write_prior_inf	Setting this to .TRUE. enables fill_inflation_restart and writes a spatially uniform prior inflation mean and standard deviation files:input_priorinf_mean.nc input_priorinf_sd.nc for the first time-step only.
prior_inf_mean	Initial value of prior inflation mean when write_prior_inf = .TRUE.
prior_inf_sd	Initial value of prior inflation standard deviation when write_prior_inf = .TRUE.

It is also important to confirm that the domains defined in Step 8 (`restart`, `history`, `vector`) are the same as what is defined in the `&fill_inflation_restart_nml` and `&filter_nml` namelist settings. Confirm the input and output file list account for all 3 domains as:

```
&filter_nml
   input_state_file_list    = 'restart_files.txt',
                              'history_files.txt',
                              'vector_files.txt'
   output_state_file_list   = 'restart_files.txt',
                              'history_files.txt',
                              'vector_files.txt'
```

```
&fill_inflation_restart_nml
  input_state_files = 'clm_restart.nc','clm_history.nc','clm_vector_history.nc'
  single_file       = .false.
```

Important: The `input_state_file_list`, `output_state_file_list` and `input_state_files` must match the domains that were defined in Step 8.

The `assimilate.csh` script assigns the CLM file that defines each domain. In this tutorial the `restart`, `history` and `vector` domains are defined by the `.r.`, `.h0.` and `.h2.` files respectively. ***We show the portions of the assimilate.csh script below for illustration purposes only. Do not modify these lines for the tutorial.***

The domains are set within the `Block 4: DART INFLATION` portion of the script as:

```
set     LND_RESTART_FILENAME = ${CASE}.clm2_0001.r.${LND_DATE_EXT}.nc
set     LND_HISTORY_FILENAME = ${CASE}.clm2_0001.h0.${LND_DATE_EXT}.nc
set LND_VEC_HISTORY_FILENAME = ${CASE}.clm2_0001.h2.${LND_DATE_EXT}.nc
```

and set again during the `Block 5: REQUIRED DART namelist settings` in prepration for the `filter` step as:

```
ls -1         clm2_*.r.${LND_DATE_EXT}.nc  >! restart_files.txt
ls -1 ${CASE}.clm2_*.h0.${LND_DATE_EXT}.nc >! history_files.txt
ls -1 ${CASE}.clm2_*.h2.${LND_DATE_EXT}.nc >! vector_files.txt
```

6.101.15 Step 11: Complete the Assimilation Setup

A few setup steps remain before the assimilation case can be executed. First, the complete list of DART executables must be generated. At this point you should have already customized your `mkmf.template` and tested your local build environment in Step 4. In this step, you must compile the rest of the required DART scripts to perform the assimilation as follows:

```
cd DART/models/clm/work/
./quickbuild.sh
```

After completion the following DART executables should be available within your `work` folder.

```
preprocess
advance_time
clm_to_dart
create_fixed_network_seq
```

(continues on next page)

(continued from previous page)

```
create_obs_sequence
dart_to_clm
fill_inflation_restart
obs_diag
obs_seq_to_netcdf
obs_sequence_tool
filter
perfect_model_obs
model_mod_check
```

Next modify the `DART_params.csh` settings such that the directories match your personal environment.

Modify the `cesmtag` and `CASE` variable:

```
setenv cesmtag        <your cesm installation folder>
setenv resolution     f09_f09_mg17
setenv compset        2000_DATM%GSWP3v1_CLM50%BGC-CROP_SICE_SOCN_MOSART_SGLC_SWAV

 ..
 ..

if (${num_instances} == 1) then
  setenv CASE clm5_f09_pmo_SIF
else
  setenv CASE <your tutorial case name>
endif
```

Modify the `SourceModDir` to match the directory that you set in Step 1:

```
setenv use_SourceMods TRUE
setenv SourceModDir   <your SourceMods directory>
```

Confirm the following variables are set to match your personal environment, especially `cesmroot`, `caseroot`, `cime_output_root`, `dartroot` and `project`.

```
setenv cesmdata         /glade/p/cesmdata/cseg/inputdata
setenv cesmroot         /glade/work/${USER}/CESM/${cesmtag}
setenv caseroot         /glade/work/${USER}/cases/${cesmtag}/${CASE}
setenv cime_output_root /glade/scratch/${USER}/${cesmtag}/${CASE}
setenv rundir           ${cime_output_root}/run
setenv exeroot          ${cime_output_root}/bld
setenv archdir          ${cime_output_root}/archive
..
..
setenv dartroot         /glade/work/${USER}/git/DART_public
setenv baseobsdir       /glade/p/cisl/dares/Observations/land
..
..
setenv project      <insert project number>
setenv machine      cheyenne
```

6.101.16 Step 12: Execute the Assimilation Run

Set up the assimilation case by executing `CLM5_setup_assimilation`

```
cd DART/models/clm/shell_scripts/cesm2_2/
./CLM5_setup_assimilation
```

It takes approximately 7-10 minutes for the script to create the assimilation case which includes compiling the CESM executable. The script is submitted to a login node where it performs low-intensive tasks including the execution of `case_setup`, and `preview_namelist` and stages the appropriate files in the `rundir`. However, compiling CESM is more resource intensive, thus the `case_build` command is automatically submitted using `qcmd` which starts a non-interactive job on a single batch node in the Cheyenne "regular" queue for a default time of 1 hour.

Caution: Once the setup is complete the script will output steps (1-8) displaying 'Check the case'. These steps are good for general reference, however, **for the tutorial ignore these steps and continue to follow the instructions below.**

After the case is created, by default, it is set up as a 'free' or 'open-loop' run. This means if the case is submitted as is, it will perform a normal CLM simulation without using any DART assimilation. In order to enable DART do the following:

```
cd <caseroot>
./CESM_DART_config
```

Caution: After the script is executed and DART is enabled 'Check the DART configuration:' will be displayed followed by suggested steps (1-5). As before these steps are good for general reference, however, **for the tutorial ignore these steps and continue to follow the instructions below.**

Now that DART is enabled, confirm, and if necessary, modify the run-time settings to perform daily assimilations for 5 total days. Use the scripts `xmlquery` to view the current settings (e.g. `./xmlquery STOP_OPTION`), or you can view all the environment run settings within `env_run.xml`. Use the `xmlchange` command to change the current setting (e.g. `./xmlchange STOP_OPTION=nhours`). Make sure the run-time settings are as follows:

```
DATA_ASSIMILATION_LND=TRUE
DATA_ASSIMILATION_SCRIPT= <dartroot>/models/clm/shell_scripts/cesm2_2/assimilate.csh
STOP_OPTION=nhours
STOP_N=24
DATA_ASSIMILATION_CYCLES=5
RESUBMIT=0
CONTINUE_RUN=FALSE
```

Run Time Assimilation Case Settings	Description
DATA_ASSIMILATION_LND	If .TRUE. data assimilation is enabled. If .FALSE. no data assimilation is performed, and simulation is performed as a normal CLM run.
DATA_ASSIMILATION_SCRIPT	Location of script assimilate.csh that performs the assimilation. Controls execution of DART scripts and passes files between DART and CLM code.
STOP_OPTION	Unit of time that controls duration of each assimilation cycle (see STOP_N)
STOP_N	Duration of each assimilation cycle in units of time defined by STOP_OPTION
DATA_ASSIMILATION_CYCLES	The number of assimilation cycles performed for a single job submission
RESUBMIT	The number of times each job submission is repeated as defined by STOP_OPTION, STOP_N, and DATA_ASSIMILATION_CYCLES
CONTINUE_RUN	If .FALSE. this is the startup of the assimilation run beginning from the refcase. If .TRUE. this is a continuation of a previous assimilation run.

Before submission review your input.nml within your case folder to confirm the settings reflect those from previous steps of the tutorial. Next submit the assimilation run:

```
> cd <caseroot>
> ./case.submit
```

Check the status of the job on Cheyenne using PBS commands to determine if job is queued (Q), running (R) or completed.

```
qstat -u <your username>
```

The job requires approximately 5-10 minutes of runtime to complete the requested 5 assimilation cycles.

Tip: The DART code provides the script stage_cesm_files within <caseroot> to restart an assimilation case. This script re-starts the assimilation at a prior point in time by re-staging the proper restart files to the <rundir> and edits the rpointer files to reference the re-staged files. This script comes in handy if an assimilation run fails or if the user modifies the input.nml settings and does not want to re-create the assimilation case from scratch using CLM5_setup_assimilation.

Also the stage_dart_files script is available if the user makes changes to the DART source code after an assimilation case has been created. Changes to the DART source code requires the executables to be re-compiled within <dartroot>/models/clm/work. Executing stage_dart_files transfers the DART executables to <exeroot> making them available when the case is submitted.

6.101.17 Step 13: Diagnose the Assimilation Run

Once the job has completed it is important to confirm it ran as expected without any errors. To confirm this view the CaseStatus files:

```
cd <caseroot>
cat CaseStatus
```

A successful assimilation run will look like the following at the end of the file with case.run success at the end:

```
2022-01-14 14:21:11: case.submit starting
---------------------------------------------------
2022-01-14 14:21:18: case.submit success case.run:2465146.chadmin1.ib0.cheyenne.ucar.edu
```

(continues on next page)

(continued from previous page)

```
---------------------------------------------------
2022-01-14 14:21:28: case.run starting
---------------------------------------------------
2022-01-14 14:21:33: model execution starting
---------------------------------------------------
2022-01-14 14:23:58: model execution success
---------------------------------------------------
2022-01-14 14:23:58: case.run success
---------------------------------------------------
```

A failed run will provide an error message and a log file either from CESM or DART that hopefully provides more details of the error. This will look like this:

```
---------------------------------------------------
2022-01-14 14:24:57: case.run starting
---------------------------------------------------
2022-01-14 14:24:58: model execution starting
---------------------------------------------------
2022-01-14 14:25:08: model execution error
ERROR: Command: 'mpiexec_mpt -p "%g:"  -np 360  omplace -tm open64
/glade/scratch/bmraczka/cesm2.2.0/clm5_SWE0_MissingVal/bld/cesm.exe
>> cesm.log.$LID 2>&1 ' failed with error '' from dir
'/glade/scratch/bmraczka/cesm2.2.0/clm5_SWE0_MissingVal/run'
---------------------------------------------------
2022-01-14 14:25:08: case.run error
ERROR: RUN FAIL: Command 'mpiexec_mpt -p "%g:"  -np 360  omplace -tm open64
/glade/scratch/bmraczka/cesm2.2.0/clm5_SWE0_MissingVal/bld/cesm.exe   >>
cesm.log.$LID 2>&1 ' failed See log file for details:
/glade/scratch/bmraczka/cesm2.2.0/clm5_SWE0_MissingVal/run/cesm.log.2465146.chadmin1.ib0.
↪cheyenne.ucar.edu.220114-142457
```

If the case ran successfully proceed to the next step in the tutorial, **but if the case did not run successfully** locate the log file details which describe the error and resolve the issue. Contact dart@ucar.edu if necessary.

Just because an assimilation ran successfully (without errors) does not mean it ran with good performance. A simple, first check after any assimilation is to make sure:

1) Observations have been accepted

2) The CLM posterior member values are updated from their prior values

A quick way to confirm observation acceptance and the posteriors have been updated is through the `obs_seq.final` file located in your case run folder. Below we provide an example of a successful update (`clm_obs_seq.2011-01-02-00000.final`) which is derived from the same leaf area observation in `obs_seq.2011-01-02-00000` as described in Step 7.

```
OBS            3
  6.00864688253571
  5.44649167346675
  5.45489142211957
  5.45808308572015
  3.479253215076174E-002
  3.469211455745640E-002
  5.41512442472872
```

(continues on next page)

(continued from previous page)

```
  5.41843086313655
  5.44649167346675
  5.44970758027390
  5.50877061923831
  5.51180677764943
  5.46367193547511
  5.46683825691274
  5.44039845768897
  5.44363195062815
  0.000000000000000E+000
  0.000000000000000E+000
          2           4          -1
obdef
loc3d
    5.235987755982989         0.000000000000000        -888888.0000000000     -2
kind
         23
    0     149750
 0.200000000000000
```

Below we provide the variable names for the `clm_obs_seq.2011-01-02-00000.final` example from above.

```
  <observation sequence number>
  <observation value>
  <true observation value>
  <prior ensemble mean>
  <posterior ensemble mean>
  <prior ensemble spread>
  <posterior ensemble spread>
  <prior member 1>
  <posterior member 1>
  <prior member 2>
  <posterior member 2>
  <prior member 3>
  <posterior member 3>
  <prior member 4>
  <posterior member 4>
  <prior member 5>
  <posterior member 5>
  <data product QC>
  <DART quality control>

obdef
loc3d
    <longitude>    <latitude>   <vertical level>   <vertical code>
kind
   <observation quantity number>
   <seconds>     <days>
   <observation error variance>
```

In the example above, the observation has been **accepted** denoted by a `DART quality control value = 0`. If the `DART quality control value =7` this indicates the observation has fallen outside the `outlier_threshold` value and is rejected. For more details on the DART quality control variables read the *documentation.*

Now, compare your `clm_obs_seq.final` file to the example provided above:

```
cd <rundir>
less clm_obs_seq.2011-01-02-00000.final
```

First, confirm that the observation (and other observations) was accepted.

Second, confirm that the `posterior member` values have been updated from their respective `prior member` values. Your simulation has run successfully if the `posterior member` and `posterior ensemble mean` values have moved closer to the *observation value* as compared to the prior values.

Do not expect your own `clm_obs_seq.final` file to be bit-by-bit identical (i.e. identical to the 14th decimal place) to the example given above. Slight changes in compiler and run time environment are known to cause small changes when running CLM-DART. Furthermore, CLM is set up to run in `hybrid` mode, which unlike `branch` mode does not provide bit-by-bit reproducibility.

This tutorial has been purposely designed such that all observations are accepted and the posteriors have been updated. In research applications, however, the vast majority of observations may be rejected if there is large systemic biases between the model ensemble and the observations. In that case, it may take many assimilation time steps before the inflation creates a sufficient enough ensemble spread such that the observation falls within the outlier threshold and is accepted. In other cases, an observation may be accepted, but the posterior update is negligible. If you experience these issues, a helpful troubleshooting guide is located *here.*

6.101.18 Matlab Diagnostics

Once you have confirmed that the assimilation has been completed reasonably well as outlined by the steps above, the DART package includes a wide variety of Matlab diagnostic scripts that provide a more formal evaluation of assimilation performance. These diagnostics can provide clues to further maximize performance through adjustments of the DART settings (localization, inflation, etc.). The full suite of diagnostic scripts can be found at this path in your DART installation (`DART/diagnostics/matlab`) with supporting documentation found *here.*

Note: Additional scripts that are designed for CLM output visualization can be found here (`DART/models/clm/matlab`). The `clm_get_var.m` and `clm_plot_var.m` scripts are designed to re-constitute a vector-based file (e.g. restart.nc) into gridded averages to allow viewing of spatial maps. These scripts are helpful to view the model update by DART (innovations). An example of how to implement these scripts can be found here (`DART/models/clm/matlab/README.txt`).

Here we provide instructions to execute two highly recommended matlab scripts. First, the `plot_rmse_xxx_evolution.m` script provides a time series of assimilation statistics of 1) observation acceptance, 2) RMSE between the observations and the expected observation (derived from the CLM state), and 3) a third statistic of your choosing (we recommended 'total spread').

The observation acceptance statistic compares the number of observations assimilated versus the number of observations available for your domain. In general, it is desirable to assimilate the majority of observations that are available. The RMSE statistic quantifies the mismatch between the observations and the CLM state. A successful assimilation reduces the RMSE, thus reducing the mismatch between the observations and the CLM state. Finally the total spread provides contributions from the ensemble spread and observation error variance. This value should be comparable to the RMSE.

To execute `plot_rmse_xxx_evolution.m` do the following:

```
cd DART/models/clm/work
```

Confirm the DART executables used for the matlab diagnostics exist. These should have been compiled during Step 11 of this tutorial. The important DART executables for the diagnostics are `obs_diag` and `obs_seq_to_netcdf`. If

they do not exist, perform the `./quickbuild.sh` command to create them.

Next generate a text file that includes all the `clm_obs_seq*.final` files that were created from the tutorial simulation

```
ls <rundir>/*final > obs_seq_files_tutorial.txt
```

Next edit the `&obs_diag_nml` namelist within the `input.nml`. to assign the *obs_seqence_list* to the text file containing the names of the `clm_obs_seq*.final` files associated with the tutorial. Next, specify how the observations are displayed by defining the `bin` settings, which for this tutorial are set such that every day of observations are displayed individually. Because the tutorial is a global run we define the `lonlim` and `latlim` setting to include the entire globe. For more information about the `obs_diag` namelist settings go *here.*

```
&obs_diag_nml
  obs_sequence_name = ''
  obs_sequence_list = 'obs_seq_files_tutorial.txt'
  first_bin_center =  2011, 1,  2, 0, 0, 0
  last_bin_center  =  2011, 1,  6, 0, 0, 0
  bin_separation   =     0, 0,  1, 0, 0, 0
  bin_width        =     0, 0,  1, 0, 0, 0
  time_to_skip     =     0, 0,  0, 0, 0, 0
  max_num_bins     = 1000
  trusted_obs      = 'null'
  Nregions   = 1
  lonlim1    =     0.0,
  lonlim2    =   360.0,
  latlim1    =   -90.0,
  latlim2    =    90.0,
  reg_names  = 'tutorial',
  hlevel_edges =  0.0, 1.0, 2.0, 5.0, 10.0, 40.0
  print_mismatched_locs = .false.
  create_rank_histogram = .true.
  outliers_in_histogram = .true.
  use_zero_error_obs    = .false.
  verbose               = .true.
  /
```

Next convert the information in the `clm_obs_seq*.final` files into a netcdf format (`obs_diag_output.nc`) by executing the `obs_diag` executable.

```
./obs_diag
```

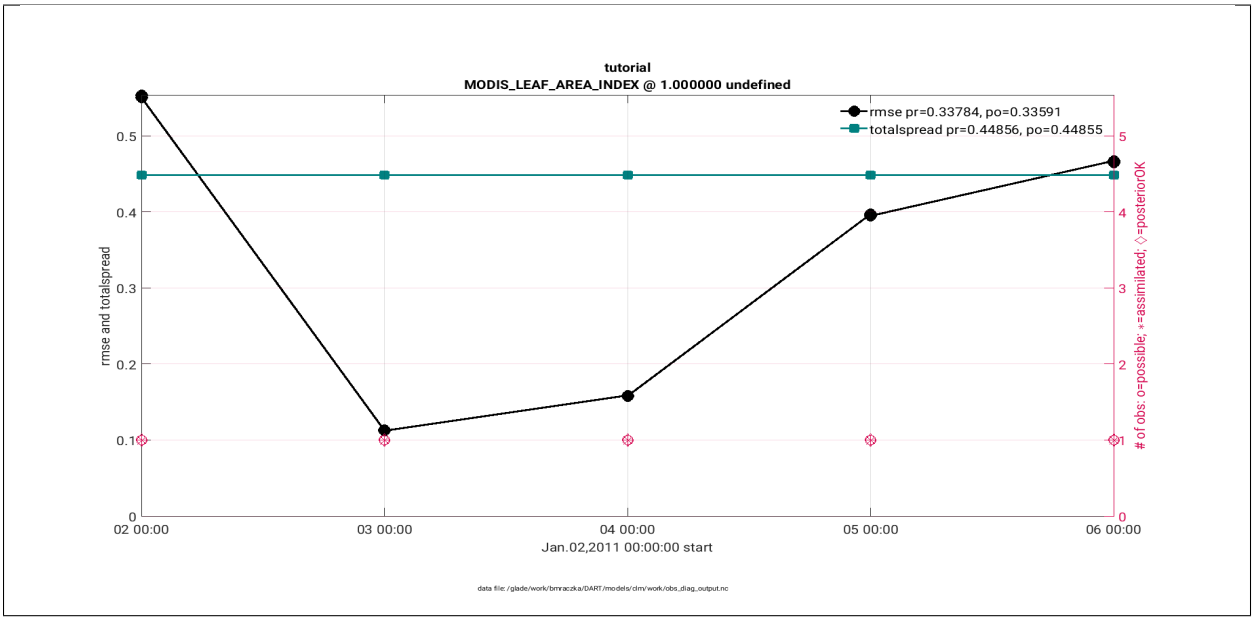
Next, use Matlab to create the `plot_rmse_xxx_evolution.m` figures. Note that this function automatically plots the RMSE, where the `copy` and the `obsname` variable are customizable.

```
cd DART/diagnostics/matlab
module load matlab
matlab -nodesktop
>> fname   = '<dartroot>/models/clm/work/obs_diag_output.nc';
>> copy    = 'totalspread';
>> obsname = 'MODIS_LEAF_AREA_INDEX';
>> plotdat = plot_rmse_xxx_evolution(fname, copy, 'obsname', obsname);
```

Tip: When remotley logged into Cheyenne there is a time delay when the Matlab figures are rendering, and also when interacting with the figures. For the purposes of this tutorial this delay is minimal. However, to improve responsiveness

for your own research you may find it convenient to port your diagnostic files (e.g. obs_diag_output.nc) and run the Matlab diagnostics on your local machine. This requires a compiled version of DART and a Matlab license for your local machine.

The finished figure should look like the following below. Click on it to enlarge. Notice that the figure has two axes; the left providing the RMSE and total spread statistics, whereas the right provides the observations available and observations assimilated for each time step.



Second, the `link_obs.m` provides several features, including a spatial map of the observation locations with color coded DART QC values. This allows the user to identify observation acceptance as a function of sub-regions.

Next edit both the `&obs_seq_to_netcdf_nml` and `&schedule_nml` namelist sections within `input.nml`.

For this tutorial we plot a list of the `clm_obs_seq*.final` files as shown below, which includes the global domain. We include all observations within a single `bin`. For more information about these settings go *here.*

```
&obs_seq_to_netcdf_nml
  obs_sequence_name = ''
  obs_sequence_list = 'obs_seq_files_tutorial.txt'
  append_to_netcdf  = .false.
  lonlim1    =    0.0
  lonlim2    =  360.0
  latlim1    =  -90.0
  latlim2    =   90.0
  verbose    = .false.
  /

&schedule_nml
  calendar        = 'Gregorian'
  first_bin_start =  1601,  1,  1,  0,  0,  0
  first_bin_end   =  2999,  1,  1,  0,  0,  0
  last_bin_end    =  2999,  1,  1,  0,  0,  0
  bin_interval_days    = 1000000
```

(continues on next page)

(continued from previous page)

```
  bin_interval_seconds = 0
  max_num_bins         = 1000
  print_table          = .true.
  /
```

Next execute the `obs_seq_to_netcdf` to convert the observation information into a netcdf readable file (`obs_epoch_001.nc`) for `link_obs.m`

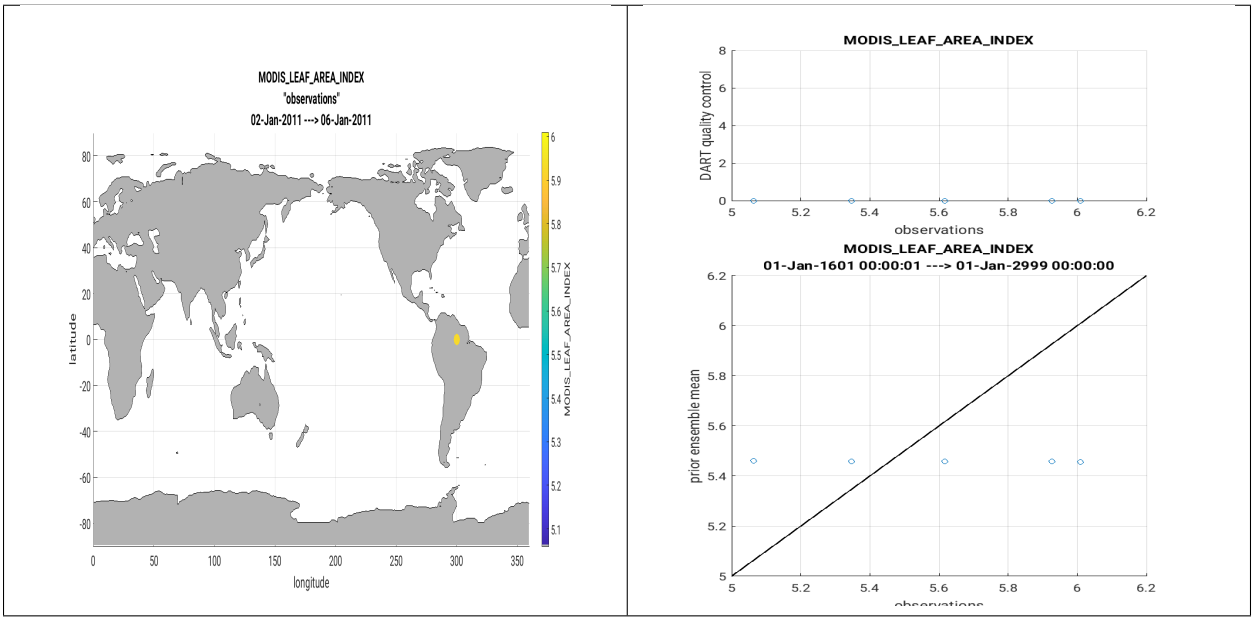
```
./obs_seq_to_netcdf
```

Next, use Matlab to create the `link_obs.m` figures.

```
cd DART/diagnostics/matlab
module load matlab
matlab -nodesktop
>> fname         = '<dartroot>/models/clm/work/obs_epoch_001.nc';
>> ObsTypeString = 'MODIS_LEAF_AREA_INDEX';
>> ObsCopyString = 'observations';
>> CopyString    = 'prior ensemble mean';
>> QCString      = 'DART quality control';
>> region        = [0 360 -90 90 -Inf Inf];
>> global obsmat;
>> link_obs(fname, ObsTypeString, ObsCopyString, CopyString, QCString, region)
```

The `link_obs.m` script creates 3 separate figures including a 1) 3D geographic scatterplot, 2) observation diagnostic plot as a function of time, and 3) 2D scatterplot that typically compares the 'prior/posterior expected obsevations' against the 'actual observation'. Read the commented section within the `link_obs.m` script for more information.

The completed `link_obs.m` figures are shown below for the 3D geographic scatterplot (left) and the 2d scatterplot (right). Click to enlarge. Note that the 2D scatterplot compares the prior expected observations vs. the actual observations. The 1:1 fit for this plot is poor, but should be slightly improved if you compare the posterior observations vs. the actual observations.



If you have completed all these steps (1-13) **Congratulations!** – you are well on your way to designing CLM5-DART assimilations for your own research.

6.102 WRF/DART Tutorial Materials for the Manhattan Release.

6.102.1 Introduction

This document will describe how to get started with your own Weather Research and Forecasting (WRF) data assimilation experiments using DART and only covers only the WRF-specific aspects of integrating with DART. It is not wise to try to run WRF/DART if you have no experience with WRF and/or no experience with DART.

This tutorial was assembled to be compatible with ~WRF V3.9.1 and the DART Manhattan release. Other releases of WRF may or may not be backwards or forwards compatible with this tutorial.

You must already be comfortable running the WRF system (WPS, real_em build of WRF). If not, work through the WRF model tutorial first before trying to link WRF and DART together. Check the WRF user guide or the WRFHELP forum for WRF-specific assistance.

If you are new to DART, we recommend that you become familiar with DART by working through the *DART Tutorial* and then understanding the *DART getting started* documentation.

before attempting the WRF/DART tutorial as you will find many helpful resources for learning the base DART configuration.

We do not claim that this is a "turnkey" or "black box" system. Be mentally prepared to invest a reasonable amount of time on the learning curve. There are many outstanding research issues which have no easy answers. This is not a one week/grad student/naive user system. Even after you get the code up and running, you have to be able to interpret the results, which requires developing specific skills. There are a lot of ways to alter how the system works – localization, inflation, which variables and observations are assimilated, the assimilation window time, the model resolution, etc, etc. This is both good and bad - you have many ways of improving your results, but you have to take care on how you leave all the settings of these inputs. Getting a set of scripts that runs doesn't mean the system is running well, or producing useful results. So - if you're still reading: Let the adventure begin!

This tutorial introduces a "canned" WRF/DART experiment involving an ensemble of 50 members that will be initialized from GFS initial conditions at 2017/04/27 00:00 UTC using a domain of the continental United States. The data included in the tutorial lasts until 2017/04/30 18:00 UTC. During this period, there was a strong rain and wind event that affected a large portion of the United States, causing record rains, localized flooding, and numerous tornadoes. For more information on the physical account of this case, see weather.gov.

By default, the tutorial case will only cover 12 hours of this event starting at 2017/04/27 00:00 UTC. The WRF model will be "spun-up" for six hours to generate a prior distribution. An assimilation of PREPBUFR observations will then be performed at 06:00 UTC, at which time analysis files will be generated to begin a new ensemble forecast. The WRF model will be advanced for 6 hours and a final assimilation cycle will be performed at 12:00 UTC. This process could then continue in order to investigate the strong rain and wind event. For what it's worth, on NCAR's *Cheyenne* under the default test configuration for this case, it can take an hour to complete a forecast/assimilation cycle. Since the tutorial runs for two cycles, it can take twice as long.

The goals of this tutorial are to demonstrate how WRF/DART works. After running this tutorial, you will be able to understand the major steps involved in setting up your own data assimilation (DA) experiments. However, you will need to do additional work before you can expect to have a fully functional WRF/DART system, as some of the steps involved in this tutorial (in particular, the perturbation bank and the observation sequence files) are provided for you in order to simplify the process. Furthermore, if you are not running on the UCAR/NCAR Cheyenne supercomputing system, you will likely need to customize the assimilation scripts to match the details of your particular system.

Important: We have provided instructions for the NCAR supercomputer Cheyenne, so you may need to tailor these

instructions to your system if you are not using Cheyenne. These system-specific setup steps may take a good deal of effort, especially if you are unfamiliar with details such as MPI, NetCDF, etc. Furthermore, even after you get the code up and running, you will need to properly interpret your results.

6.102.2 Step 1: Setup

There are several dependencies for the executables and scripting components. On Cheyennne, users have reported success building WRF, WPS, WRFDA, and DART with the default module environment including Intel compilers, MPT, and netCDF4. In addition, you'll need to load the nco and ncl modules to run the set of scripts that accompany the tutorial.

There are multiple phases for the setup: building the DART executables, getting the initial WRF boundary conditions etc., building (or using existing) WRF executables, and configuring and staging the scripting needed to perform an experiment.

Build the DART executables.

If you have not already, see *Getting Started* to download the DART software package. Set an environment variable *DART_DIR* to point to your base DART directory. How to do this will depend on which shell you are using.

shell	command
tcsh	`setenv DART_DIR <path_to_your_dart_installation>`
bash	`export DART_DIR=<path_to_your_dart_installation>`

In either case, you will replace <path_to_your_dart_installation> with the actual path to your DART installation. If you are using another shell, refer to your shell-specific documentation on how to set an environment variable.

Building the DART executables for the tutorial follows the same process as building any of the DART executables. Configure the `mkmf.template` file for your system, configure the `input.nml` for the model you want to compile, and run `quickbuild.sh` (which is not necessarily quick, but it is quicker than doing it by hand) to compile all the programs you might need for an experiment with that model.

1. It is assumed you have successfully configured the `$DART_DIR/build_templates/mkmf.template` file for your system. If not, you will need to do so now. See the *Getting Started* for more detail, if necessary.

2. [OPTIONAL] Modify the DART code to use 32bit reals. Most WRF/DART users run both the WRF model and the DART assimilation code using 32bit reals. This is not the default for the DART code. Make this single code change before building the DART executables to compile all reals as 32bit reals.

   Edit `$DART_DIR/assimilation_code/modules/utilities/types_mod.f90` with your favorite editor. Change

   ```
   ! real precision:
   ! TO RUN WITH REDUCED PRECISION REALS (and use correspondingly less memory)
   ! comment OUT the r8 definition below and use the second one:
   integer, parameter :: r4 = SELECTED_REAL_KIND(6,30)
   integer, parameter :: r8 = SELECTED_REAL_KIND(12)   ! 8 byte reals
   !integer, parameter :: r8 = r4                      ! alias r8 to r4
   ```

   to

```
! real precision:
! TO RUN WITH REDUCED PRECISION REALS (and use correspondingly less memory)
! comment OUT the r8 definition below and use the second one:
integer, parameter :: r4 = SELECTED_REAL_KIND(6,30)
! integer, parameter :: r8 = SELECTED_REAL_KIND(12)   ! 8 byte reals
integer, parameter :: r8 = r4                        ! alias r8 to r4
```

3. Copy the tutorial DART namelist from `$DART_DIR/models/wrf/tutorial/template/input.nml.template` to `$DART_DIR/models/wrf/work/input.nml`.

```
cd $DART_DIR/models/wrf
cp tutorial/template/input.nml.template work/input.nml
```

4. Build the WRF/DART executables:

```
cd $DART_DIR/models/wrf/work
./quickbuild.sh
```

   Many executables are built, the following executables are needed for the tutorial and will be copied to the right place by the *setup.csh* script in a subsequent step:

```
advance_time
fill_inflation_restart
filter
obs_diag
obs_seq_to_netcdf
obs_sequence_tool
pert_wrf_bc
wrf_dart_obs_preprocess
```

Preparing the experiment directory.

Approximately 100Gb of space is needed to run the tutorial. Create a "work" directory someplace with a lot of free space. The rest of the instructions assume you have an environment variable called *BASE_DIR* that points to this directory.

shell	command
tcsh	`setenv BASE_DIR <path_to_your_working_directory>`
bash	`export BASE_DIR=<path_to_your_working_directory>`

1. The WRF boundary conditions and perturbations required to make a viable ensemble are available in a 15 GB tar file. Put this file in your `$BASE_DIR`. Since this is a large file, we suggest using 'wget' to download the file directly to your local system:

```
cd $BASE_DIR
wget http://www.image.ucar.edu/wrfdart/tutorial/wrf_dart_tutorial_23May2018_v3.tar.
↪gz
tar -xzvf wrf_dart_tutorial_23May2018_v3.tar.gz
```

   After untarring the file you should see the following directories: *icbc, output, perts,* and *template*. The directory names (case sensitive) are important, as the scripts rely on these local paths and file names.

2. You will need template WRF namelists from the `$DART_DIR/models/wrf/tutorial/template` directory:

```
cp $DART_DIR/models/wrf/tutorial/template/namelist.input.meso   $BASE_DIR/template/.
cp $DART_DIR/models/wrf/tutorial/template/namelist.wps.template $BASE_DIR/template/.
```

3. You will also need the scripting to run a WRF/DART experiment. Copy the contents of `$DART_DIR/models/wrf/shell_scripts` to the `$BASE_DIR/scripts` directory.

```
mkdir $BASE_DIR/scripts
cp -R $DART_DIR/models/wrf/shell_scripts/* $BASE_DIR/scripts
```

Build or locate WRF executables.

The WRFDA package is needed to generate a set of perturbed initial ensemble member files and also to generate perturbed boundary condition files. Since the tutorial provides a perturbation bank for a specific case, it is not required to actually *run da_wrfvar.exe* but it needs to be in the `WRF_RUN` directory for the tutorial.

Build (or locate an appropriate build of) WRF, WPS and WRFDA.

WRF and WRFDA should be built with the "dmpar" option, while WPS can be built "serial"ly. See the WRF/WRFDA documentation for more information about building these packages.

Note: For consistency and to avoid errors, you should build WRF, WPS, WRFDA, and DART with the same compiler you use for NetCDF. Likewise MPI should use the same compiler. You will need the location of the WRF and WRFDA builds to customize the *params.csh* script in the next step.

Configure `$BASE_DIR/scripts/param.csh` with proper paths, info, etc.

This is a script that sets variables which will be read by other WRF/DART scripts. There are some specific parameters for either the Cheyenne supercomputing system using the PBS queueing system or the (decommissioned) Yellowstone system which used the *LSF* queueing system. If you are not using Cheyenne, you may still want to use this script to set your queueing-system specific parameters.

Important:

All variables that are marked `'set this appropriately #%%%#'` need to be set. This list is intended to provide some guidance on what needs to be set, but it is not an exhaustive list.

Script variable	Description
module load mpt	The Environment Modules MPI compiler to use (here the HPE MPI) compiler). Note that on Cheyenne the default compiler is Intel.
module load nco	The nco package.
module load ncl/6.6.2	The ncl package.
BASE_DIR	The directory containing icbc, output, perts, etc.
DART_DIR	The DART directory.
WRF_DM_SRC_DIR	The directory of the WRF dmpar installation.
WPS_SRC_DIR	The directory of the WPS installation.
VAR_SRC_DIR	The directory of the WRFDA installation.
GEO_FILES_DIR	The root directory of the WPS_GEOG files. NOTE: on Cheyenne these are available in the /glade/u/home/wrfhelp/WPS_GEOG directory
GRIB_DATA_DIR	The root directory of the GRIB data input into ungrib.exe. For this tutorial the grib files are included, so use ${ICBC_DIR}/grib_data
GRIB_SRC	The type of GRIB data (e.g. <Vtable.TYPE>) to use with ungrib.exe to copy the appropriate Vtable file. For the tutorial, the value should be 'GFS'.
COMPUTER_CHARGE_ACCOUNT	The project account for supercomputing charges. See your supercomputing project administrator for more information.
EMAIL	The e-mail address used by the queueing system to send job summary information.

Run the *setup.csh* script to create the proper directory structure and move executables to proper locations.

```
cd $BASE_DIR/scripts
./setup.csh param.csh
```

So far, your `$BASE_DIR` should contain the following directories:

```
icbc
obs_diag
obsproc
output
perts
post
rundir
scripts
template
```

Your `$BASE_DIR/rundir` directory should contain the following:

executables:

- advance_time,
- fill_inflation_restart,
- filter,
- obs_diag,
- obs_seq_to_netcdf,
- obs_sequence_tool,
- `pert_wrf_bc` (no helper page),
- wrf_dart_obs_preprocess

directories:

- `WRFIN` (empty)
- `WRFOUT` (empty)
- `WRF_RUN` (wrf executables and support files)

scripts:

- *add_bank_perts.ncl*
- *new_advance_model.csh*

support data:

- *sampling_error_correction_table.nc*

Check to make sure your `$BASE_DIR/rundir/WRF_RUN` directory contains:

```
da_wrfvar.exe
wrf.exe
real.exe
be.dat
contents of your WRF build run/ directory (support data files for WRF)
```

Note: Be aware that the *setup.csh* script is designed to remove `$BASE_DIR/rundir/WRF_RUN/namelist.input`. Subsequent scripting will modify `$BASE_DIR/template/namlist.input.meso` to create the `namelist.input` for the experiment.

For this tutorial, we are providing you with a specified WRF domain. To make your own, you would need to define your own wps namelist and use WPS to make your own geogrid files. See the WRF site for help with building and running those tools as needed. You would also need to get the appropriate grib files to generate initial and boundary condition files for the full period you plan to cycle. In this tutorial we have provided you with geogrid files, a small set of grib files, and a namelist to generate series of analyses for several days covering a North American region.

Let's now look inside the `$BASE_DIR/scripts` directory. You should find the following scripts:

Script name	Description
add_bank_perts.ncl	Adds perturbations to each member.
assim_advance.csh	Advances 1 WRF ensemble member to the next analysis time.
assimilate.csh	Runs filter … i.e. the assimilation.
diagnostics_obs.csh	Computes observation-space diagnostics and the model-space mean analysis increment.
driver.csh	Primary script for running the cycled analysis system.
first_advance.csh	Advances 1 WRF ensemble member (on the first time).
gen_pert_bank.csh	Saves the perturbations generated by WRFDA CV3.
gen_retro_icbc.csh	Generates the wrfinput and wrfbdy files.
init_ensemble_var.csh	Creates the perturbed initial conditions from the WRF-VAR system.
mean_increment.ncl	Computes the mean state-space increment, which can be used for plotting.
new_advance_model.csh	advances the WRF model after running DART in a cycling context.
param.csh	Contains most of the key settings to run the WRF/DART system.
prep_ic.csh	Prepares the initial conditions for a single ensemble member.
real.csh	Runs the WRF real.exe program.
setup.csh	Creates the proper directory structure and place executables/scripts in proper locations.

You will need to edit the following scripts to provide the paths to where you are running the experiment, to connect up files, and to set desired dates. Search for the string `'set this appropriately #%%%#'` for locations that you need

to edit.

```
cd $BASE_DIR/scripts
grep -r 'set this appropriately #%%%#' .
```

Other than *param.csh*, which was covered above, make the following changes:

File name	Variable / value	Change description
driver.csh	datefnl = 2017042712	Change to the final target date; here the final date is already set correctly for this tutorial.
gen_retro_icbc.csh	datefnl = 2017043000	This is the final date to create WRF initial/boundary conditions for. This is set to the last date that files are included in the tutorial.
gen_retro_icbc.csh	paramsh = <full path to param.csh>	The full path to param.csh. Change this on the line after the comment. While these two files are in the same directory here, in general it is helpful to have one param.csh for each experiment.
gen_pert_bank.csh	All changes	As the tutorial includes a perturbation bank, you will not need to run this script for the tutorial, so you will not need to change these values. However, you should set appropriate values when you are ready to generate your own perturbation bank.

Next, move to the `$BASE_DIR/perts` directory. Here you will find 100 perturbation files, called a "perturbation bank." For your own case, you would need to create a perturbation bank of your own. A brief description for running the script is available inside the comments of that file. However, again, for this tutorial, this step has already been run for you. The `$BASE_DIR/icbc` directory contains a *geo_em_d01.nc* file (geo information for our test domain), and grib files that will be used to generate the initial and boundary condition files. The `$BASE_DIR/template` directory should contain namelists for WRF, WPS, and filter, along with a wrfinput file that matches what will be the analysis domain. Finally, the `$BASE_DIR/output` directory contains observations within each directory name. Template files will be placed here once created (done below), and as we get into the cycling the output will go in these directories.

6.102.3 Step 2: Initial conditions

To get an initial set of ensemble files, depending on the size of your ensemble and data available to you, you might have options to initialize the ensemble from, say, a global ensemble set of states. Here, we develop a set of flow dependent errors by starting with random perturbations and conducting a short forecast. We will use the WRFDA random CV option 3 to provide an initial set of random errors, and since this is already available in the perturbation bank developed in the setup, we can simply add these to a deterministic GFS state. Further, lateral boundary uncertainty will come from adding a random perturbation to the forecast (target) lateral boundary state, such that after the integration the lateral boundaries have random errors.

First, we need to generate a set of GFS states and boundary conditions that will be used in the cycling. Use `$BASE_DIR/scripts/gen_retro_icbc.csh` to create this set of files, which will be added to a subdirectory corresponding to the date of the run in the `$BASE_DIR/output` directory. Make sure *gen_retro_icbc.csh* has the appropriate path to your *param.csh* script. If the *param.csh* script also has the correct edits for paths and you have the executables placed in the rundir, etc., then running *gen_retro_icbc.csh* should execute a series of operations to extract the grib data, run metgrid, and then twice execute *real.exe* to generate a pair of WRF files and a boundary file for each analysis time.

```
cd $BASE_DIR/scripts
./gen_retro_icbc.csh
```

Note: Ignore any `rm: No match` errors, as the script attempts to delete output files if they already exist, and they will not for the first run.

Once the script completes, inside your `$BASE_DIR/output/2017042700` directory you should see these files:

```
wrfbdy_d01_152057_21600_mean
wrfinput_d01_152057_0_mean
wrfinput_d01_152057_21600_mean
```

These filenames include the Gregorian dates for these files, which is used by the dart software for time schedules. Similar files (with different dates) should appear in all of the date directories between the *datea* and *datef* dates set in the *gen_retro_icbc.csh* script. All directories with later dates will also have an observation sequence file *obs_seq.out* that contains observations to be assimilated at that time.

Next, we will execute the script to generate an initial ensemble of states for the first analysis. For this we run the script *init_ensemble_var.csh*, which takes two arguments: a date string and the location of the *param.csh* script.

```
cd $BASE_DIR/scripts
./init_ensemble_var.csh 2017042700 param.csh
```

This script generates 50 small scripts and submits them to the batch system. It assumes a PBS batch system and the 'qsub' command for submitting jobs. If you have a different batch system, edit this script and look near the end. You will need to modify the lines staring with #PBS and change 'qsub' to the right command for your system. You might also want to modify this script to test running a single member first — just in case you have some debugging to do.

When complete for the full ensemble, you should find 50 new files in the directory `output/2017042700/PRIORS` with names like *prior_d01.0001*, *prior_d01.0002*, etc… You may receive an e-mail to helpfully inform you when each ensemble member has finished.

6.102.4 Step 3: Prepare observations [OPTIONAL]

For the tutorial exercise, observation sequence files are provided to enable you to quickly get started running a test WRF/DART system. If you want to run with the example observations, you can skip to Step 4.

However, observation processing is critical to the success of running DART and was covered in *getting started*. In brief, to add your own observations to WRF/DART you will need to understand the relationship between observation definitions and observation sequences, observation types and observation quantities, and understand how observation converters extract observations from their native formats into the DART specific format.

The observation sequence files that are provided in this tutorial come from NCEP BUFR observations from the GDAS system. These observations contain a wide array of observation types from many platforms within a single file.

If you wanted to generate your own observation sequence files from PREPBUFR for an experiment with WRF/DART, you should follow the guidance on the prepbufr page to build the bufr conversion programs, get observation files for the dates you plan to build an analysis for, and run the codes to generate an observation sequence file.

For completeness, we list here how you could generate these observation sequence files yourself.

Important: the following steps are **not necessary** for the tutorial as the processed PREPBUFR observation sequence files have already been provided for you. However, these steps are provided in order to help users get started with these observations quickly for their own experiments.

To (again, *optionally*) reproduce the observation sequence files in the *output* directories, you would do the following:

- Go into your DART prep_bufr observation converter directory and install the PREPBUFR utilities as follows:

```
cd $DART_DIR/observations/obs_converters/NCEP/prep_bufr
./install.sh
```

You may need to edit the *install.sh* script to match your compiler and system settings.

- Go to the `$DART_DIR/observations/obs_converters/NCEP/prep_bufr/work/` directory and run *quickbuild.sh* to build the DART PREPBUFR-to-intermediate-file observation processor:

```
cd $DART_DIR/observations/obs_converters/NCEP/prep_bufr/work
./quickbuild.sh
```

- Download the PREPBUFR observations for your desired time. Go to the NCAR/UCAR Research Data Archive page for the NCEP/NCAR Global Reanalysis Products. Register on the site, click on the "Data Access" tab, and follow either the instructions for external users or NCAR internal users.
- The downloaded *.tar* file will often be COS-blocked. If so, the file will appear corrupted if you attempt to untar it without converting the data. See the NCAR COS-block page for more information on how to strip the COS-blocking off of your downloaded file.
- Untar the data in your desired directory.
- In the `$DART_DIR/observations/obs_converters/NCEP/prep_bufr/work` directory, edit the *input.nml* file. This file will control what observations will be used for your experiment, so the namelist options are worth investigating a bit here. For example, you could use the following:

```
&prep_bufr_nml
   obs_window    = 1.0
   obs_window_cw = 1.5
   otype_use     = 120.0, 130.0, 131.0, 132.0, 133.0, 180.0
                   181.0, 182.0, 220.0, 221.0, 230.0, 231.0
                   232.0, 233.0, 242.0, 243.0, 245.0, 246.0
                   252.0, 253.0, 255.0, 280.0, 281.0, 282.0
   qctype_use    = 0,1,2,3,15
   /
```

This defines an observation time window of +/- 1.0 hours, while cloud motion vectors will be used over a window of +/- 1.5 hours. This will use observation types sounding temps (120), aircraft temps (130,131), dropsonde temps (132), mdcars aircraft temps, marine temp (180), land humidity (181), ship humidity (182), rawinsonde U,V (220), pibal U,V (221), Aircraft U,V (230,231,232), cloudsat winds (242,243,245), GOES water vapor (246), sat winds (252,253,255), and ship obs (280, 281, 282). Additionally, it will include observations with specified qc types only. See the prepbufr page for more available namelist controls.

- Within the `$DART_DIR/observations/obs_converters/NCEP/prep_bufr/work` directory, edit the *prepbufr.csh* file and change *BUFR_dir*, *BUFR_idir*, *BUFR_odir*, and *BUFR_in* to match the locations and format of the data you downloaded. A little trial and error might be necessary to get these set correctly.
- Copy over the executables from `../exe`, and run the *prepbufr.csh* script for a single day at a time:

```
cd $DART_DIR/observations/obs_converters/NCEP/prep_bufr/work
cp ../exe/\*.x .
./prepbufr.csh \<year\> \<month\> \<day\>
```

- Your PREPBUFR files have now been converted to an intermediate ASCII format. There is another observation converter to take the observations from this format and write them into the native DART format. Edit the *input.nml* namelist file in the *DART_DIR/observations/obs_converters/NCEP/ascii_to_obs/work* directory. Here is a basic example:

```
&ncepobs_nml
   year       = 2017,
   month      = 4,
```

(continues on next page)

(continued from previous page)

```
   day        = 27,
   tot_days   = 3,
   max_num    = 800000,
   select_obs = 0,
   ObsBase = '<path to observations>/temp_obs.',
   daily_file = .false.,
   lat1       = 15.0,
   lat2       = 60.0,
   lon1       = 270.0,
   lon2       = 330.0
   /
```

Choosing "select_obs = 0" will select all the observations in the ASCII file. Set "ObsBase" to the directory you output the files from during the last step. If you wish to choose specific observations from the ASCII intermediate file or control other program behavior, there are many namelist options documented on the create_real_obs page.

- It is now time to build *ascii_to_obs* programs. Run the following:

```
cd $DART_DIR/observations/obs_converters/NCEP/ascii_to_obs/work
./quickbuild.sh
```

- Run the *create_real_obs* program to create the DART observation sequence files:

```
cd $DART_DIR/observations/obs_converters/NCEP/ascii_to_obs/work
./create_real_obs
```

- The program *create_real_obs* will create observation sequence files with one file for each six hour window. For a cycled experiment, the typical approach is to put a single set of observations, associated with a single analysis step, into a separate directory. For example, within the `output` directory, we would create directories like `2017042700`, `2017042706`, `2017042712`, etc. for 6-hourly cycling. Place the observation files in the appropriate directory to match the contents in the files (e.g. *obs_seq2017042706*) and rename as simply *obs_seq.out* (e.g. `output/2017042706/obs_seq.out`).
- It is helpful to also run the wrf_dart_obs_preprocess program, which can strip away observations not in the model domain, perform superobservations of dense observations, increase observation errors near the lateral boundaries, check for surface observations far from the model terrain height, and other helpful pre-processing steps. These collectively improve system performance and simplify interpreting the observation space diagnostics. There are a number of namelist options to consider, and you must provide a *wrfinput* file for the program to access the analysis domain information.

6.102.5 Step 4: Creating the first set of adaptive inflation files

In this section we describe how to create initial adaptive inflation files. These will be used by DART to control how the ensemble is inflated during the first assimilation cycle.

It is convenient to create initial inflation files before you start an experiment. The initial inflation files may be created with *fill_inflation_restart*, which was built by the *quickbuild.sh* step. A pair of inflation files is needed for each WRF domain.

Within the `$BASE_DIR/rundir` directory, the *input.nml* file has some settings that control the behavior of *fill_inflation_restart*. Within this file there is the section:

```
&fill_inflation_restart_nml
   write_prior_inf = .true.
```

(continues on next page)

(continued from previous page)

```
   prior_inf_mean  = 1.00
   prior_inf_sd    = 0.6

   write_post_inf  = .false.
   post_inf_mean   = 1.00
   post_inf_sd     = 0.6

   input_state_files = 'wrfinput_d01'
   single_file       = .false.
   verbose           = .false.
   /
```

These settings write a prior inflation file with a inflation mean of 1.0 and a prior inflation standard deviation of 0.6. These are reasonable defaults to use. The *input_state_files* variable controls which file to use as a template. You can either modify this namelist value to point to one of the *wrfinput_d01_XXX* files under `$BASE_DIR/output/<DATE>`, for any given date, or you can copy one of the files to this directory. The actual contents of the file referenced by *input_state_files* do not matter, as this is only used as a template for the *fill_inflation_restart* program to write the default inflation values. Note that the number of files specified by *input_state_files* must match the number of domains specified in *model_nml:num_domains*, i.e. the program needs one template for each domain. This is a comma-separated list of strings in single 'quotes'.

After running the program, the inflation files must then be moved to the directory expected by the *driver.csh* script.

Run the following commands with the dates for this particular tutorial:

```
cd $BASE_DIR/rundir
cp ../output/2017042700/wrfinput_d01_152057_0_mean ./wrfinput_d01
./fill_inflation_restart
mkdir ../output/2017042700/Inflation_input
mv input_priorinf_*.nc ../output/2017042700/Inflation_input/
```

Once these files are in the right place, the scripting should take care of renaming the output from the previous cycle as the input for the next cycle.

6.102.6 Step 5: Cycled analysis system

While the DART system provides executables to perform individual tasks necessary for ensemble data assimilation, for large models such as WRF that are run on a supercomputer queueing system, an additional layer of scripts is necessary to glue all of the pieces together. A set of scripts is provided with the tutorial tarball to provide you a starting point for your own WRF/DART system. You will need to edit these scripts, perhaps extensively, to run them within your particular computing environment. If you will run on NCAR's Cheyenne environment, fewer edits may be needed, but you should familiarize yourself with running jobs on Cheyenne if necessary. A single forecast/assimilation cycle of this tutorial can take an hour on Cheyenne - longer if debug options are enabled or the shared nodes are busy - shorter if more cores or a higher optimization level is acceptable.

In this tutorial, we have previously edited the *param.csh* and other scripts. Throughout the WRF/DART scripts, there are many options to adjust cycling frequency, domains, ensemble size, etc., which are available when adapting this set of scripts for your own research. To become more famililar with this set of scripts and to eventually make these scripts your own, we advise commenting out all the places the script submits jobs while debugging, placing an 'exit' in the script at each job submission step. This way you will be able to understand how all of the pieces work together.

However, for this tutorial, we will only show you how the major components work. The next step in our process is the main *driver.csh* script, which expects a starting date (YYYYMMDDHH) and the full path of the resource file as

command line arguments. In this example (which uses csh/tcsh syntax), we are also capturing the run-time output into a file named *run.out* and the entire command will be running in the background:

```
cd $BASE_DIR/scripts
./driver.csh 2017042706 param.csh >& run.out &
```

driver.csh will - check that the input files are present (wrfinput files, wrfbdy, observation sequence, and DART restart files), - create a job script to run *filter* in `$BASE_DIR/rundir`, - monitor that expected output from *filter* is created, - submit jobs to advance the ensemble to the next analysis time, - (simultaneously with the ensemble advance) compute assimilation diagnostics - archive and clean up - and continue to cycle until the final analysis time has been reached.

6.102.7 Step 6: Check your results

Once you have run the analysis system, it is time to check if things ran well or if there are problems that need to be addressed. DART provides analysis system diagnostics in both state and observation space.

Check to see if the analysis system actually changed the state. You should find a file in the *$BASE_DIR/output/* directory called *analysis_increment.nc* which is the change in the ensemble mean state from the background to the analysis after running *filter*. Use a tool, such as *ncview*, to look at this file. You should see spatial patterns that look something like the meteorology of the day. These should be places where the background (short ensemble forecast) was adjusted based on the set of observations provided. Please become familiar with the *Diagnostics Section* of the DART Documentation.

The *driver.csh* script also ran the *diagnostics_obs.csh* which runs the obs_diag program to investigate the observation space analysis statistics. You'll find the results of this in `$BASE_DIR/output/<DATE>/obs_diag_output.nc`. There are many Matlab scripts in the `$DART_DIR/diagnostics/matlab` directory that help explore the effectiveness of the assimilation. Look for their examples in the *Observation-Space Diagnostics* section.

The additional files enable plotting the time series of recently assimilated observations once multiple cycles have been run. Be sure to check that a high percentage (> 90%) of available observations were assimilated. Low assimilation rates typically point to a problem with the background analysis, observation quality, and/or observation error specification which are important to address before using system results for science.

Additional statistics can be evaluated using the converted final observation sequence file in netcdf format from the obs_seq_to_netcdf tool. This file has a name like *obs_epoch_029.nc*, where the number in the file is largest in the most recent set of observations processed. There are Matlab tools to explore where and why the observations were rejected. *plot_obs_netcdf.m* and *link_obs.m* are particularly useful.

If you encounter difficulties setting up, running, or evaluating the system performance, please consider using the GitHub Issue facility or feel free to contact us at dart(at)ucar(dot)edu.

6.102.8 Agenda from the 22 Jan 2014 tutorial

- Introduction (Anderson) - DART Lab materials
- WRF/DART basic building blocks (Romine) -slides (some material is outdated)
- Computing environment support (Collins) -slides
- WRF/DART application examples (Romine) -slides (some material is outdated)
- Observation processing (Collins) -slides
- DART diagnostics (Hoar) - *observation diagnostics*

6.102.9 More Resources

- Check or Submit DART Issues
- DAReS website
- Register for DART
- Preparing MATLAB to use with DART.
- WRF model users page
- Need help? e-mail dart (at) ucar (dot) edu

6.103 Supported Models

DART supported models:

- *9-variable*
- *AM2*
- *bgrid_solo*
- *Atmospheric Models in CESM*
- *Community Earth System Model*
- *CICE*
- *CLM*
- *CM1*
- *COAMPS Nest*
- *COAMPS*
- *ECHAM*
- *FESOM*
- *GITM*
- *Ikeda*
- *LMDZ*
- *Lorenz 05*
- *Lorenz 63*
- *Lorenz 84*
- *Lorenz 96*
- *Lorenz 96 2-scale*
- *Lorenz 96 Tracer Advection*
- *Forced Lorenz 96*
- *MITgcm_ocean*
- *MPAS_ATM*
- *MPAS OCN*

- *NCOMMAS*
- *NOAH, NOAH-MP*
- *null_model*
- *PBL_1D*
- *pe2lyr*
- *POP*
- *ROMS*
- *ROSE*
- *Simple advection*
- *SQG*
- *TIEGCM*
- *WRF-Hydro*
- *WRF*

6.103.1 Hints for porting a new model to DART:

Copy the contents of the `DART/models/template` directory into a `DART/models/xxx` directory for your new model.

If your model is closer to the simpler examples (e.g. lorenz), the existing model_mod.f90 is a good place to start. If your model is a full 3d geophysical one (e.g. like cam, pop, etc) then rename full_model_mod.f90 to model_mod.f90 and start there.

Set `LOCATION` in quickbuild.sh to `LOCATION=threed_sphere` for full 3d geophysical models.

Try `./quickbuild.sh` and everything should compile at this point.

The required subroutines are these:

```
public :: get_model_size, &
          get_state_meta_data,  &
          model_interpolate, &
          shortest_time_between_assimilations, &
          static_init_model, &
          init_conditions,    &
          adv_1step, &
          nc_write_model_atts, &
          pert_model_copies, &
          nc_write_model_vars, &
          init_time, &
          get_close_obs, &
          get_close_state, &
          end_model, &
          convert_vertical_obs, &
          convert_vertical_state, &
          read_model_time, &
          write_model_time
```

If needed, model_mod can contain additional subroutines that are used for any model-specific utility programs. No routines other than these will be called by programs in the DART distribution.

Edit the model_mod and fill in these routines:

1. `static_init_model()` - make it read in any grid information and the number of variables that will be in the state vector. Fill in the model_size variable. Now `get_model_size()` and `get_model_time_step()` from the template should be ok as-is.
2. `get_state_meta_data()` - given an index number into the state vector return the location and kind.
3. `model_interpolate()` - given a location (lon/lat/vert in 3d, x in 1d) and a state QTY_xxx kind, return the interpolated value the field has at that location. this is probably one of the routines that will take the most code to write.

For now, ignore these routines:

```
nc_write_model_vars()
get_close_obs()
get_close_state()
end_model()
convert_vertical_obs()
convert_vertical_state()
read_model_time()
write_model_time()
```

If you have data in a dart initial condition/restart file, then you can ignore these routines:

```
shortest_time_between_assimilations()
init_conditions()
```

Otherwise, have them return an initial time and an initial default ensemble state.

If your model is NOT subroutine callable, you can ignore this routine:

```
adv_1step()
```

Otherwise have it call the interface to your model and add the files necessary to build your model to all the *work/path_names_** files. Add any needed model source files to a src/ directory.

If you want to let filter add gaussian noise to a single state vector to generate an ensemble, you can ignore this routine:

```
pert_model_copies()
```

Otherwise fill in code that does whatever perturbation makes sense to have an initial ensemble of states. in some cases that means adding a different range of values to each different field in the state vector.

At this point you should have enough code to start testing with the `model_mod_check` program. It is a stand-alone utility that calls many of the model_mod interface routines and should be easier to debug than some of the other DART programs.

Once you have that program working you should have enough code to test and run simple experiments.

The general flow is:

1. `./create_obs_sequence` - make a file with a single observation in it
2. `./perfect_model_obs` - should interpolate a value for the obs
3. generate an ensemble of states, or set 'perturb_from_single_instance' to .true.
4. run `./filter` with the single observation
5. Look at the preassim.nc and analysis.nc files Diff them with `ncdiff`:

```
ncdiff analysis.nc preassim.nc Innov.nc
```

plot it, with `ncview` if possible:

```
ncview Innov.nc
```

The difference between the two is the impact of that single observation see if it's at the right location and if the differences seem reasonable

If your model data cannot be output in NetCDF file format, or cannot be directly converted to NetCDF file format with the ncgen program, there are 2 additional steps:

- `model_to_dart` - read your native format and output data in NetCDF format
- `dart_to_model` - write the updated data back to the native file format

More details on each of these 5 steps follows.

Running `model_to_dart` if needed

If your model data is not stored in NetCDF file format, a program to convert your data from the model to NetCDF is needed. It needs to read your model data in whatever format it uses and create NetCDF variables with the field names, and appropriate dimensions if these are multi-dimensional fields (e.g. 2d or 3d). If the data is ASCII, the generic NetCDF utility ncgen may be helpful.

Running `create_obs_sequence`

You can make a synthetic observation (or a series of them) with this interactive program and use them for testing. Before running make sure the observation types you want to use are in the input.nml file in the &obs_kind_nml section, either in the assimilate or evaluate lists.

Run the program. Give the total number of obs you want to create (start with 1). Answer 0 to number of data items and 0 to number of quality control items. Answer 0 when it says enter -1 to quit. You will be prompted for an observation number to select what type of observation you are going to test.

Give it a location that should be inside your domain, someplace where you can compute (by hand) what the correct value should be. When it asks for time, give it a time that is the same as the time on your model data.

When it asks for error variance, at this point it doesn't matter. give it something like 10% of the expected data value. Later on this is going to matter a lot, but for testing the interpolation of a single synthetic obs, this will do.

For an output filename, it suggests 'set_def.out' but in this case tell it 'obs_seq.in'.

Running `perfect_model_obs`

Make sure the NetCDF file with your input data matches the input name in the input.nml file, the &perfect_model_obs_nml namelist. Make sure the input obs_sequence is still set to 'obs_seq.in'. run perfect_model_obs. Something bad will happen, most likely. Fix it.

Eventually it will run and you will get an 'obs_seq.out' file. For these tests, make sure &obs_sequence_nml : write_binary_obs_sequence = .false. in the input.nml file. The sequence files will be short and in ascii. You can check to see what the interpolated value is. if it's right, congratulations. If not, debug the interpolation code in the model_mod.f90 file.

Using a single input state

In the &filter_nml namelist, set 'perturb_from_single_instance' to .true. this tells filter that you have not generated N initial conditions, that you are only going to supply one and it needs to perturb that one to generate an initial ensemble. Make sure the 'input_state_files' matches the name of the single state vector file you have. You can use the 'obs_seq.out' file from the perfect_model run because now it has data for that observation. Later on you will need to decide on how to generate a real set of initial states, and then you will set 'perturb_from_single_instance' back to .false. and supply N files instead of one. You may need to set the &ensemble_manager_nml : perturbation_amplitude down to something smaller than 0.2 for these tests - 0.00001 is a good first guess for adding small perturbations to a state.

Running `filter`

Set the ens_size to something small for testing - between 4 and 10 is usually a good range. Make sure your observation type is in the 'assimilate_these_obs_types' list and not in the evaluate list. run filter. Find bugs and fix them until the output 'obs_seq.final' seems to have reasonable values. Running filter will generate NetCDF diagnostic files. The most useful for diagnosis will be comparing preassim.nc and analysis.nc.

Diagnostics

Run 'ncdiff analysis.nc preassim.nc differences.nc' and use your favorite netcdf plotting tool to see if there are any differences between the 2 files. For modules using a regular lat/lon grid 'ncview' is a quick way to scan files. For something on an irregular grid a more complicated tool will have to be used. If the files are identical the assimilation didn't do anything. Check to see if there is a non-zero DART quality control value in the obs_seq.final file. Check to see if there are errors in the dart_log.out file. Figure out why there's no change. If there is a difference, it should be at the location of the observation and extend out from it for a short distance. If it isn't in the right location, look at your get_state_meta_data() code. If it doesn't have a reasonable value, look at your model_interpolate() code.

Running `dart_to_model` if needed

After you have run filter, the files named in the 'output_state_files' namelist item will contain the changed values. If your model is reading NetCDF format it can ingest these directly. If not, an additional step is needed to copy over the updated values for the next model run.

6.104 9-variable

6.104.1 Overview

The 9-variable model is described in Lorenz (1980).[1] Lorenz developed this primitive-equation model using shallow-water equations as a starting point and manipulating the divergence equations so that the model exhibits quasi-geostrophic behavior and transient gravity waves that dissipate with time. Gent and McWilliams (1982)[2] explore the behavior of this model extensively. For an introduction to shallow-water equations, we recommend consulting the relevant section of a meteorology textbook such as section 4.5 of Holton and Hakim (2013).[3]

The model's three X variables are at 0, 1/9, and 2/9, three Y variables are at 3/9, 4/9 and 5/9, and three Z variables are at 6/9, 7/9, and 8/9 on a cyclic [0, 1] domain.

[1] Lorenz, Edward N., 1980: Attractor Sets and Quasi-Geostrophic Equilibrium. *Journal of the Atmospheric Sciences*, **37**, 1685-1699. doi:10.1175/1520-0469(1980)037<1685:ASAQGE>2.0.CO;2

[2] Gent, Peter R., and James C. McWilliams, 1982: Intermediate Model Solutions to the Lorenz Equations: Strange Attractors and Other Phenomena. *Journal of the Atmospheric Sciences*, **39**, 3-13. doi:10.1175/1520-0469(1982)039<0003:IMSTTL>2.0.CO;2

[3] Holton, James R., and Gregory J. Hakim, 2013: *An Introduction to Dynamic Meteorology – Fifth Edition.* Academic Press, 532 pp.

In the 9-variable model, DART advances the model, gets the model state and metadata describing this state. The model can be configured by altering the `&model_nml` *namelist* in the `input.nml` file. The details of the `&model_nml` namelist are always model-specific (there are no generic namelist values). The model time step defaults to 1 hour (3600 seconds) but is settable by altering the namelist.

The 9-variable model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced in Sections 7 and 10 of the *DART_tutorial* and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

6.104.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   g                = 8.0,
   deltat           = 0.0833333333333333,
   time_step_days   = 0,
   time_step_seconds = 3600
/
```

Description of each namelist entry

Item	Type	Description
g	real(r8)	Model parameter, see comp_dt in code for equations.
delta_t	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	real(r8)	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	real(r8)	Number of seconds for dimensional timestep, mapped to delta_t.

6.104.3 References

6.105 AM2

Attention: `AM2` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `AM2` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.105.1 Overview

AM2 is an atmospheric model developed as part of a coupled atmosphere-ocean general circulation system developed at NOAA's Geophysical Fluid Dynamics Laboratory.

If you are interested in running DART with this model please contact the DART group at dart@ucar.edu for more information.

6.106 bgrid_solo

6.106.1 Overview

DART interface module for the dynamical core of the GFDL AM2 Bgrid model. This model is subroutine callable from DART and can be run in a similar fashion to low-order models that produce diagnostic output files with multiple assimilation times per file.

The Bgrid model was originally configured as a comprehensive atmospheric model as described in Anderson et al. (2004).[1]

All of that code remains in the directories under the `DART/models/bgrid_solo` directory, however, much of the capability has been disabled by code modification. What is left is a dry dynamical core for a model with no diurnal cycle at equinox with forcing described in Held and Suarez (1994).[2]

The default settings are for a model with a 60x30 horizontal grid and 5 vertical levels. This is close to the smallest version that has somewhat realistic baroclinic instability resulting in mid-latitude 'storm tracks'. The model resolution can be changed with the entries in the `bgrid_cold_start_nml` namelist described in the *Namelist* section. It may be necessary to change the model time step to maintain stability for larger model grids. The model state variables are the gridded surface pressure, temperature, and u and v wind components.

The `bgrid_solo` directory has a `work/workshop_setup.csh` script that compiles and runs an example. This example is intended to demonstrate that the same process used for a low-order model may be used for a much more complex model and generates output for state-space or observation-space diagnostics.

Some examples of ways in which this model can be configured and modified to test DART assimilation capabilities are documented in Anderson et al. (2005).[3]

Several programs that generate interesting observation sequences are available in the `DART/models/bgrid_solo` directory. These programs take interactive user input and create a text file that can be piped into program `create_obs_sequence` to create obs_sequence files. These can serve as examples for users who are interested in designing their own custom obs_sequence files.

Program `column_rand` creates an obs_sequence with randomly located columns of observations (essentially synthetic radiosondes) that observe surface pressure along with temperature and wind components at all model levels.

Program `id_set_def_stdin` generates an obs_sequence file that observes every state variable with error variance of 10000 for surface pressure and 1.0 for temperature and wind components.

Program `ps_id_stdin` generates an obs_sequence that observes every surface pressure variable for the default model size (30x60) with an error variance of 100.

Program `ps_rand_local` generates a set of randomly located surface pressure observations with an interactively specified error variance. It also allows the observations to be confined to a rectangular subdomain.

[1] Anderson, J. L. and Coauthors, 2004: The new GFDL global atmosphere and land model AM2-LM2: Evaluation with prescribed SST simulations. *Journal of Climate*, **17**, 4641-4673. doi:10.1175/JCLI-3223.1

[2] Held, I. M., and M. J. Suarez, 1994: A proposal for the intercomparison of the dynamical cores of atmospheric general circulation models, *Bulletin of the American Meteorological Society*, **75(10)**, 1825-1830. doi:10.1175/1520-0477(1994)075<1825:APFTIO>2.0.CO;2

[3] Anderson, J. L., Wyman, B., Zhang, S. & Hoar, T., 2005: Assimilation of surface pressure observations using an ensemble filter in an idealized global atmospheric prediction system, *Journal of the Atmospheric Sciences*, **62**, 2925-2938. doi:10.1175/JAS3510.1

6.106.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   current_time =  0, 0, 0, 0
   override = .false.,
   dt_atmos = 3600,
   days     = 10,
   hours    = 0,
   minutes  = 0,
   seconds  = 0,
   noise_sd = 0.0,
   dt_bias  = -1,
   state_variables = 'ps', 'QTY_SURFACE_PRESSURE',
                     't',  'QTY_TEMPERATURE',
                     'u',  'QTY_U_WIND_COMPONENT',
                     'v',  'QTY_V_WIND_COMPONENT',
   template_file = 'perfect_input.nc'
/
# only used if initial conditions file not specified in run
&bgrid_cold_start_nml
   nlon = 60,
   nlat = 30,
   nlev = 5,
   equal_vert_spacing = .true.
/
# Values in hs_forcing_nml are described in Held and Suarez (1994)
&hs_forcing_nml
   delh      =  60.,
   t_zero    = 315.,
   t_strat   = 200.,
   delv      =  10.,
   eps       =   0.,
   ka        = -40.,
   ks        =  -4.,
   kf        =  -1.,
   sigma_b   =  .7,
   do_conserve_energy = .false.
/
&bgrid_core_driver_nml
   damp_coeff_wind   = 0.10,
   damp_coeff_temp   = 0.10,
   damp_coeff_tracer = 0.10,
   advec_order_wind   = 4,
       advec_order_temp   = 2,
       advec_order_tracer = 2,
       num_sponge_levels = 1,
       sponge_coeff_wind   = 1.00,
       sponge_coeff_temp   = 1.00,
       sponge_coeff_tracer = 1.00,
```

(continues on next page)

(continued from previous page)

```
       num_fill_pass = 2,
       decomp = 0,0,
       num_adjust_dt = 3,
       num_advec_dt  = 3,
       halo = 1,
       do_conserve_energy = .false.
/
&bgrid_integrals_nml
   file_name  = 'dynam_integral.out',
   time_units = 'days',
   output_interval = 1.00
/
```

Description of each namelist entry

The following values are specified in `model_nml`.

Item	Type	Description
current_time	integer(4)	Specifies the initial time of the Bgrid model internal clock. The four integer values are the day, hour, minutes, and seconds. The default version of the Bgrid model has neither a diurnal or seasonal cycle, so these can all be set to 0, the default value.
override	logical	If true, then the initial model date is taken from namelist entry current_time, even if an atmos_model.res file is found in directory INPUT. For most DART applications, atmospheric restart values are coming from DART files and no INPUT directory is used.
dt_atmos	integer	Model timestep in seconds.
noise_sd	real(r8)	Standard deviation of random perturbations to the time tendency of temperature applied at each timestep. Each gridpoint value of the computed temperature tendency is multiplied by 1+N(0, noise_sd) before the updated values of temperature are computed.
dt_bias	integer	Allows a simple mechanism to simulate model error. If dt_bias is non-zero, the assimilation programs believe that each model advance changes the time by dt_bias. However, internally the bgrid model is moving things forward by dt_atmos. By running perfect_model_obs with one time step for the internal bgrid clock (for instance dt_atmos = 3600, dt_bias = 3600), and filter with another (dt_atmos = 3000, and dt_bias = 3600) model error is simulated.
state_variables	character(len=129)	Strings that identify the bgrid_solo variables that should be part of the DART state vector. The first column is the netCDF variable name, the second column is the corresponding DART quantity.
template_file	character(len=256)	This is the name of the file that specifies the resolution of the variables DART uses to create the DART state vector. If *template_file* = *"null"* the *&bgrid_cold_start_nml* namelist variables are used to specify the resolution. The actual input filenames for *filter* and *perfect_model_obs* come from their respective namelists. The resolutions in the file specified in *template_file* must match the resolutions of the variables in the input filenames. To start an experiment with a new model resolution, set template_file to "null" and set the resolutions in bgrid_cold_start_nml.

The following values are specified in `bgrid_cold_start_nml`.

Item	Type	Description
nlon	integer	The number of longitudes on the model grid.
nlat	integer	The number of latitudes on the model grid.
nlev	integer	The number of model levels.
equal_vertical_spacing	logical	Model levels are equally spaced in pressure if true.

The Held-Suarez forcing details can be modified with the `hs_forcing_nml` namelist using the documentation in Held and Suarez (1994).

Model dynamics can be adjusted with the bgrid_core_driver_nml following the documentation in the references and internal documentation in the bgrid code.

6.106.3 References

6.107 Atmospheric Models in CESM

6.107.1 Overview

The larger context of the Community Earth System Model and DART interactions is described in the CESM readme This document focuses on the several atmospheric models that have been developed or adapted to run in the CESM environment. They are named according to their dynamical core ("dycore"). As of 2021 these include Finite Volume Community Atmosphere Model (CAM-FV), Spectral Element (CAM-SE), and MPAS. The DART system has supported data assimilation into CAM-FV continuously for many years. An interface to CAM-SE was added to DART in 2022. An interface to MPAS is being developed (contact us about the current status).

The flexibility of the DART environment has led to its use by graduate students, post-graduates, and scientists at universities and research labs to conduct data assimilation research. Others are using the products of data assimilation (analyses), which were produced here at NCAR using CESM+DART, to conduct related research. The latest reanalysis is described in the DART 1 degree reanalysis wiki

The variety of research can be sampled on the DART Publications page.

Terminology

The atmospheric component used in CESM is built with two independent main characteristics. CESM labels these as:

resolution

signifies both the horizontal resolution of the grid (not the vertical) **and** the dynamical core run on the specified grid. The dynamical core refers to the fluid dynamical equations run on the specified grid. Examples of resolution (short) names are f19_f19 (~2 degree Finite Volume dycore) or ne30np4_gx1v6 (~1 degree Spectral Element dycore).

compset

refers to the vertical grid **and** the parameterizations – the formulation of the subgridscale physics – as well as the combination of active, data, or stub model components. These parameterizations consist of the equations describing physical processes such as convection, radiation, and chemistry.

- The vertical grid is determined by the needs of the chosen parameterizations, thus the vertical spacing and the top level of the model domain vary with those choices.
- The combinations of parameterizations and vertical grids are named: CAM3.5, CAM5, CAM#, … WACCM, WACCM#, WACCM-X, CAM-Chem.

- The compset is specified as described in the CESM readme.

ensemble, multi-instance, and multidriver

These are essentially synonyms referring to multiple, closely related models or model states. "Ensemble" is DART's vocabulary, while "multi-instance" is CESM's original term for an ensemble. "Multidriver" is replacing "multi-instance", and refers to the CESM module which coordinates the running of all of the model components. Similarly, DART ensemble "members" are the same as CESM "instances".

CASE, CASEROOT

Running a DART setup script creates a CESM "CASE" (the name of your experiment) in the "CASEROOT" directory (from where jobs will be controlled and launched), both of which are defined in the setup script. There will also be a run directory named $CASEROOT in your scratch space (usually), where the fortran executables can also be found ($scratch/$CASEROOT/bld).

Setup Scripts describes how to specify these and other choices in the assimilation setup scripts and namelists.

6.107.2 CAM-FV

Here are some highlighted features of this DART interface to CAM-FV.

- Assimilate within the CESM software framework by using the multidriver capability of CESM2 (and later). This enables assimilation of suitable observations into a variety of CESM components and leverages CESM's build, run, and archiving capabilities.
- Use any horizontal and vertical resolution of CAM-FV.
- Assimilate a variety of observations. To date the observations successfully assimilated include:
  - NCEP reanalysis BUFR obs (T,U,V,Q),
  - Global Positioning System radio occultation observations (refractivity and electron density),
  - AIRS retrievals (T and Q),
  - MOPITT (carbon monoxide, when a chemistry model is incorporated into CAM-FV),
  - OCO2 (carbon dioxide),
  - Aura MLS (T),
  - SABER (T),
  - GNSS (total electron content, "TEC"),
  - The development of the ability to assimilate RTTOV radiances is nearly complete (2021: contact us for the current status).
  - Research has also explored assimilating surface observations, cloud liquid water, and aerosols.

  The Aura MLS, SABER, and GNSS observations have been assimilated into WACCM and WACCM-X; "high top" versions of CAM-FV.
- Specify, via namelist entries, the CAM (initial file) variables which will be directly affected by the observations, that is, the state vector.
- Generate analyses on the CAM grid which have only CAM model error in them, rather than another model's.

Reanalyses

There have been two large-scale reanalysis efforts using CAM-FV and DART. The **CAM6 Data Assimilation Research Testbed (DART) Reanalysis** is archived in the NCAR Research Data Archive DS345.0 . (See the 1 degree reanalysis wiki). It contains just under 120Tb (yes Tb) of data:

> These CAM6+DART Reanalysis data products are designed to facilitate a broad variety of research using NCAR's CESM2 models, ranging from model evaluation to (ensemble) hindcasting (initial conditions), data assimilation experiments, and sensitivity studies. They come from an 80 member ensemble reanalysis of the global troposphere and stratosphere using CAM6-FV from CESM2.1. The data products represent the actual states of the atmosphere from 2011-2019 at a ~1 degree horizontal resolution and up to 6 hourly frequency. Each ensemble member is an equally likely description of the atmosphere, and is also consistent with dynamics and physics of CAM6-FV.

An earlier, more limited dataset can be found in the **Ensemble of Atmospheric Forcing Files from a CAM4-FV Reanalysis** is archived in the NCAR Research Data Archive DS199.1 . It contains about 1.5Tb of data:

> This dataset contains files that are an ensemble of 'coupler history' files from an 80-member reanalysis performed with the Data Assimilation Research Testbed (DART) using the Community Atmosphere Model Version 4 with the finite volume core (CAM4-FV) at 1.9 degree by 2.5 degree resolution. The observations assimilated include all those used in the NCEP/NCAR reanalysis (temperature and wind components from radiosondes, aircraft, and satellite drift winds) plus radio occultation observations from the COSMIC satellites starting in late 2006. These files are intended to be used as 'DATM stream files' for CESM component sets that require a data atmosphere. Some example stream text files are included in the RDA to illustrate how to use these data.

6.107.3 Observations

The CAM6+DART Reanalysis used "observation sequence files" which contain the types of observations in the table below ("T" = temperature, "U" = zonal wind, "V" = meridional wind, "Q" = specific humidity, "refractivity" = the bending of light by density variations). These files are available on NCAR's glade file system: /glade/p/cisl/dares/Observations/NCEP+ACARS+GPS+AIRS/Thinned_x9x10. Versions of these files, which also have the results of the reanalysis in them, are available from the RDA ds345.0 linked above.

NCEP
NCEP's PREPBUFR files (prepqm) in NCAR's Research Data Archive: (https://rda.ucar.edu/datasets/ds090.0/)

COSMIC
This site (http://www.cosmic.ucar.edu/) provides atmospheric refractivity from a variety of satellites (including COSMIC), which receive Global Positioning System radio occultation signals.

AIRS
Retrievals from infrared soundings from the AQUA satellite They are thinned by a factor of 90 to make their density comparable to the radiosonde network.

Observation or Retrieval	Platform	Distribution	Data Source
T, U, V, Q	Radiosondes from balloons	mostly land	NCEP
T, U, V	ACARS commercial aircraft	mostly North America	NCEP
T, U, V	AIRCRAFT commercial aircraft	mostly non-North America	NCEP
U, V	Cloud drift winds from GOES satellites	midlatitudes and tropics	NCEP
index of refraction	Global Positioning System receivers	global	COSMIC
T, Q	AQUA satellite; AIRS instrument	global	AIRS
altimeter	Radiosondes, bouys	global surface	NCEP

Sample sets of observations, which can be used with CAM+DART assimilations, can be found at http://www.image.ucar.edu/pub/DART/Obs_sets/ of which the NCEP BUFR observations are the most widely used.

6.108 The CAM-FV DART Interface

The 19 public interface subroutines in `model_mod.f90` are standardized for all DART compliant models. These interfaces allow DART to get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for a variety of variables required by observational operators. Your choices for how the assimilation (not the hindcast) will happen are defined in the `cam-fv/work/input.nml` file. In that file, the `model_nml` namelist lets you control the interaction with CAM-FV. The CAM-FV, which DART will interact with, is defined by the setup scripts, as described next.

6.108.1 Setup Scripts

Unlike pre-Manhattan versions of DART-CAM, CESM (CAM) runs using its normal scripts, then stops and calls a DART script, which does the desired assimilation tasks, then returns to the CESM run script for the next model advance. See the CESM interface documentation in the CESM readme for more general information about running DART with CESM. Due to the complexity of the CESM software environment, the versions of CESM which can be used for assimilation are more restricted than previously. Each supported CESM version has similar, but unique, sets of setup scripts and CESM *SourceMods*. Those generally do not affect the `cam-fv/model_mod.f90` interface.

The primary purpose of a setup script is to *set up* a CESM "CASE" (compset, resolution, etc.), which can be used by DART. The ability to *use* DART programs is then set up by a second script; `DART_config`, which was created by the setup script. Here is an outline of the scripts, which are currently (2021) in shell_scripts. They are roughly in order of complexity, which is the order in which you might want to use them. The indenting shows which scripts are used by, or associated with, another script.

```
cesm2_1/                                Directory containing scripts developed for CESM2_1
    spinup_single                       Setup a single instance (member) CAM-FV case to␣
↪advance a model state
                                        some months to a desired date.
    setup_hybrid                        Basic script to set up an assimilation case.
       DART_config.template             Modified to create the script which modifies a␣
↪CESM CASE to do assimilation.
          no_assimilate.csh.template    Modified to create a script which does no␣
↪assimilation,
                                        but prepares files for the next model advance.
          assimilate.csh.template       Modified to create the assimilate.csh script
    compress.csh                        Example of compressing assimilation output for␣
↪efficient archiving.
                                        Can be called by assimilate.csh
    mv_to_campaign.csh                  Example of how to use globus to move files to a␣
↪remote archive.
    setup_advanced                      Like setup_hybrid, but more model and assimilation␣
↪features can be modified.
                                        It modifies DART_config.template like setup_hybrid␣
↪does.
    setup_single_from_ens               Set up a single-instance run using initial␣
↪conditions taken from
                                        a single instance of a multi-instance CAM hindcast.
↪  Useful for debugging.
```

(continues on next page)

(continued from previous page)

```
   standalone.pbs                    Batch job tests of assimilation with no model␣
↪advances.
      test_assimilate.csh            A simpler (earlier) form of assimilate.csh.
 cesm2_0/                            Similar contents to cesm2_1, plus the following.
   obs_seq_tool_series.csh           Script to process a series of obs_seq.final files,
                                     to change any of the properties available to obs_
↪sequence_tool.
   spinup_single_sst.25              Same as cesm2_1/spinup_single, but uses a high␣
↪resolution SST dataset.
 synth_obs_locs_to_seqs.csh          Take text output from, e.g. even_sphere.m, and␣
↪create obs_seq.in files
                                     for use in perfect_model_obs.
```

The scripts in cesm#_# will handle, for that CESM version;

- all CAM-FV "physics" variants and vertical resolutions. For example, CAM5.5, CAM6, ..., WACCM4, WACCM6, WACCM-X, ..., CAM-Chem.
- all horizontal resolutions of CAM-FV; 1.9x2.5 (f19xf19), 0.9x1.25 (f09xf09), ...

Physics variants of other dycores are handled in other "model" interfaces, such as models/cam-se.

SourceMods

The most recent SourceMods for the CAM6+DART interface can be fetched from the github reanalysis repository. Change to the cesm2_1_forcing_rean branch, which includes a SourceMods tar file. Unpack that file into the location you specify in the setup script, before building the CASE.

6.108.2 Namelists

DART assembles the namelists for all of the relevant modules into a single namelist file; `models/cam-fv/input.nml`. This section focuses on `model_nml`, but others are referenced, as needed. Namelists start with an ampersand & and terminate with a slash /. Character strings that contain a / must be enclosed in quotes to prevent them from prematurely terminating the namelist. Text outside of the &.../ pairs is ignored.

Here's a list of the model_nml variables and default values. More detailed descriptions follow in a table and subsections.

```
&model_nml
   cam_template_filename              = 'caminput.nc'
   cam_phis_filename                  = 'cam_phis.nc'
   vertical_localization_coord        = 'PRESSURE'
   use_log_vertical_scale             = .false.
   no_normalization_of_scale_heights  = .true.
   no_obs_assim_above_level           = -1,
   model_damping_ends_at_level        = -1,
   state_variables                    = ''
   assimilation_period_days           = 0
   assimilation_period_seconds        = 21600
   suppress_grid_info_in_output       = .false.
   custom_routine_to_generate_ensemble = .true.
   fields_to_perturb                  = ''
   perturbation_amplitude             = 0.0_r8
   using_chemistry                    = .false.
```

(continues on next page)

(continued from previous page)

```
   use_variable_mean_mass              = .false.
   debug_level                         = 0
/
```

Item	Type	Description
cam_template_filename	character (len=128)	CAM initial file used to provide configuration information, such as the grid resolution, number of vertical levels, whether fields are staggered or not, etc. Created by the first hindcast.
cam_phis_filename	character (len=128)	CAM topography file. Reads the "PHIS" NetCDF variable from this file. Typically this is a CAM History file because this field is not normally found in a CAM initial file. Created by the first hindcast.
vertical_localization_coord	character (len=128)	The vertical coordinate to which all vertical locations are converted in model_mod. Valid options are "pressure", "height", "scaleheight" or "level".
use_log_vertical_scale	logical	Use the log of the vertical distances when interpolating. This is only used for locations having which_vert = VERTISPRESSURE. It should be .true. when vertical_localization_coord = "scaleheight" or "height".
no_normalization_of_scale_height	logical	If true (default), scale height is computed as the log of the pressure at the given location. Beware: unnormalized scale heights decrease upward, and may have values < 0. This works because only differences of scale height are used and find_enclosing_indices assigns the larger and smaller coordinate values correctly in the interpolation. If false, the scale height is computed as the log of the ratio of the surface pressure to the pressure aloft. In previous versions normalization was the default. It is slightly less efficient.
no_obs_assim_above_level	integer	Because the top of the model is highly damped it is recommended to NOT assimilate observations in the top model levels. The units here are CAM model level numbers. Set it to equal or below the lowest model level (the highest number) where damping is applied in the model. See *Diffusion*, below.
model_damping_ends_at_level	integer	Set this to the lowest model level (the highest number) where model damping is applied. Observations below the 'no_obs_assim_above_level' cutoff, but close enough to the model top to have an impact during the assimilation, will have their impacts decreased smoothly to 0 at this given model level. The assimilation should make no changes to the model state above the given level. See *Diffusion*, below.
state_variables	character (len=64) dimension(100)	Character string table that includes: 1. CAM initial file variable names of fields to be read into the state vector, 2. the corresponding DART QTY (quantity) 3. if a bounded quantity, the minimum and maximum valid values, 4. the string 'UPDATE' indicates that the updated values should be written back to the output file. 'NOUPDATE' will skip writing this field at the end of the assimilation. See *State Variables*, below.
assimilation_period_days	integer	With assimilation_period_seconds, sets the assimilation cycle length. They should match the model advance time. The CAM scripts distributed with DART set these to 0 days, 21600 seconds (6 hours). They also set the assimilation window width.
assimilation_period_seconds	integer	See assimilation_period_days
suppress_grid_info_in_output	logical	Filter can update fields in existing files or create diagnostic/output files from scratch. By default files created from scratch include a full set of CAM grid information to make the file fully self-contained and plottable. However, to save disk space the grid variables can be suppressed in files created by filter by setting this to true.
custom_routine_to_generate_ensemble	logical	Use the subroutines in model_mod.f90 to create an ensemble of initial conditions (with non-0 spread) from a single CAM initial file. This is useful when there is no existing ensemble of ICs. See *Perturbed*, below.
fields_to_perturb	character, (len=32) dimension(100)	If perturbing a single state to generate an ensemble, set 'custom_routine_to_generate_ensemble = .true.' and list here the DART QTYs of the field(s) to be perturbed.
perturbation_amplitude	real(r8), dimension(100)	For each field name in the 'fields_to_perturb' list, give the standard deviation of the gaussian noise to add to each field being perturbed.
using_chemistry	logical	If using CAM-CHEM, set this to .true.

6.108.3 Setup Variations

The default values in `cam-fv/shell_scripts/cesm#_#/setup*` and in the namelists in `cam-fv/work/input.nml` are (mostly) set up for a single assimilation cycle of CAM-fV, starting from a single model state, which must be perturbed into an ensemble. The following are suggestions for setting it up for other assimilations. Namelist variables listed here might be in any namelist within `input.nml`.

State Variables

This implementation of the DART interface module for the CAM and WACCM models uses the CAM initial files (**not** restart files) for transferring the model state to and from the `filter`.

The DART state vector should include all prognostic variables in the CAM initial files which cannot be calculated directly from other prognostic variables. In practice the state vector sometimes contains derived quantities to enable DART to compute forward operators (expected observation values) efficiently. The derived quantities are often overwritten when the model runs the next timestep, so the work DART does to update them is wasted work. The standard state vector contains the following fields, as entered into the `input.nml:model_nml` namelist.

```
state_variables  =
      'T',     'QTY_TEMPERATURE',          'NA', 'NA', 'UPDATE'
      'US',    'QTY_U_WIND_COMPONENT',     'NA', 'NA', 'UPDATE'
      'VS',    'QTY_V_WIND_COMPONENT',     'NA', 'NA', 'UPDATE'
      'Q',     'QTY_SPECIFIC_HUMIDITY',    'NA', 'NA', 'UPDATE'
      'CLDLIQ','QTY_CLOUD_LIQUID_WATER',   'NA', 'NA', 'UPDATE'
      'CLDICE','QTY_CLOUD_ICE',            'NA', 'NA', 'UPDATE'
      'PS',    'QTY_SURFACE_PRESSURE',     'NA', 'NA', 'UPDATE'
```

Any tracers or chemicals ("constituents" in CESM's vocabulary), which are needed for a given study and exist in the initial files, can be added to `state_variables`. See the list for CAM6, below. CAM6 variables which are *not* in the initial file can be added to it if they are in CAM's list of constituents (or "tracers"). Those variables are identified by a `&IC` suffix in the "MASTER FIELD LIST" in an "atm.log…" or "atm_0001.log…" file. Finally (you're deeply into the weeds here), variables can be added to the list of constituents using CAM's `cnst_add` function, which will not be described here. In all of these cases, minor modifications to `model_mod.f90` and CAM may be necessary.

Here is a list of CAM initial file variables, excluding the variables listed as parts of the most common state vector, above. Each would need to have a DART `*QTY*` associated with it.

Other moisture variables

- NUMICE "cloud ice number "
- NUMLIQ "cloud liquid number "
- NUMRAI "rain number "
- NUMSNO "snow number "
- RAINQM "rain amount "
- SNOWQM "snow amount "

Aerosols

- DMS "dimethyl sulfide "
- H2O2 "H_2O_2"
- H2SO4 "H_2SO_4"
- SO2 "SO_2"

- SOAG "secondary organic aerosols gas "

MAM4 modal aerosol scheme variables ("[ ]" means use a single digit.)

- bc_a[1,4] "black carbon, modes 1 and 4 "
- dst_a[1-3] "dust, modes 1 through 3"
- ncl_a[1-3] "sea salt (NaCl) , modes 1 through 3"
- num_a[1-4] "aerosol number density, modes 1 through 4"
- pom_a[1,4] "primary-organic aerosols, modes 1 and 4"
- soa_a[1,2] "secondary-organic aerosols, modes 1 and 2"
- so4_a[1-3] "sulfate (SO_4) modes 1 through 3"

Expected observation values on pressure, scale height, height or model levels can be requested from `model_interpolate`. Surface observations can not yet be interpolated, due to the difference between the model's lowest level (~7 hPa above the model surface) and the Earth's surface where the observations are made. Model_interpolate can be queried for any (non-surface) variable in the state vector (which are variables native to CAM) plus pressure on height levels.

The reasons initial files are used instead of restart files include:

1. The contents of the restart files vary depending on both the model release version and the physics packages selected.
2. There is no metadata describing the variables in the restart files. Some information can be tracked down in the `atm.log` file, but not all of it.
3. The restart files (for non-chemistry model versions) are much larger than the initial files (and we need to deal with an ensemble of them).
4. The temperature on the restart files is virtual equivalent potential temperature, which requires (at least) surface pressure, specific humidity, and sensible temperature to calculate.
5. CAM does not call the initialization routines when a hindcast is started in ''restart" mode, so fields which are not modified by DART may be inconsistent with fields which are.
6. If DART modifies the contents of the `.r.` restart file, it might also need to modify the contents of the `.rs.` restart file, which has similar characteristics (1-3 above) to the `.r.` file.
7. There is no need for exact restart performance because filter alters the model state, making exact restarts impossible.

Inflation

Assimilation using CAM and WACCM should generally use one of DART's adaptive inflation algorithms. As of 2021 these are `inf_flavor` = 2 (a widely used and tested option) and flavor 5 (similar to 2, but enhanced by the use of an gamma distribution instead of a normal distribution). "Prior" inflation is generally a better choice than "posterior", so set `input.nml:filter_nml:`

```
inf_initial_from_restart    = .true.,   .false.
inf_sd_initial_from_restart = .true.,   .false.
```

For the first cycle, if you have inflation restart files, you should stage those in the $RUNDIR where the other restart files will be staged, with names which include "dart.rh.cam_output_priorinf_mean" and "dart.rh.cam_output_priorinf_sd" in them, so that assimilate.csh will find them. If you don't have restart files, set `*initial_from_restart` to .false. and assimilate.csh will create inflation restart files using the values in `inf_initial` and `inf_sd_initial`. You will

need to run the assimilation for some days in order to allow the inflation values to equilibrate with the observation network and model ensemble spread.

Perturbed Ensemble

A multidriver configuration of CAM needs an ensemble of initial condition files for each active component in order to start a hindcast. The set of files must include, at a minimum, CAM initial files and CLM restart files. Usually CICE is also active, and other components may be, which need their own restart files. If there is no suitable initial ensemble for starting the ensemble hindcast, one can be generated from a single model state by linking it into suitably named files (see ../CESM/shell_scripts/link_ens_to_single.csh), running the first ensemble hindcast, and then telling DART to perturb each member before the first assimilation.

The default perturbation routine in filter adds gaussian noise equally to all fields in the state vector. For CAM it is preferable to use the perturbation mechanism in the cam-fv/model_mod.f90. This allows the exclusion of fields which are tricky to perturb, such as specific humidity. The mechanism is controlled by the input.nml:model_nml "perturb" variables. Typically, ensemble spread is generated from a single state by adding small perturbations to only the temperature field "T" and letting the model expand the perturbations to other fields and increase the sizes. For example,

```
filter_nml:
   single_file_in               = .false., (Even though your initial ensemble may be␣
↪linked to a single file)
   perturb_from_single_instance = .true.
   perturbation_amplitude         (ignored, because model_mod defines it)

model_nml:
   custom_routine_to_generate_ensemble = .true.
   fields_to_perturb                   = 'QTY_TEMPERATURE'
   perturbation_amplitude              = 0.1
```

Continuing after the first cycle

If your first hindcast+assimilation cycle uses an ensemble created from a single file, you will need to change to the 'continuing' mode, where CAM will not perform all of its startup procedures and DART will use the most recently created ensemble.

```
! model_nml:
   custom_routine_to_generate_ensemble = .true.
   fields_to_perturb                   = ''   (Turns off perturbations)
   perturbation_amplitude              = 0.1  (Ignored.  Can change to 0.0_r8 for␣
↪consistency)

! CESM's env_run.xml:
    <entry id="CONTINUE_RUN" value="TRUE">
```

Combining multiple cycles into one job

`Setup_hybrid` and `setup_pmo` are set up in the default cycling mode, where each submitted job performs one model advance and one assimilation, then resubmits the next cycle as a new job. For long series of cycles, this can result in a lot of time waiting in the queue for short jobs to run. Prevent this by using CESM's multicycling mode. To request 2 hours to run 8 assimilation cycles, in $CASEROOT run commands:

```
=  ./xmlchange DATA_ASSIMILATION_CYCLES=8
  ./xmlchange --subgroup case.run --id JOB_WALLCLOCK_TIME      --val 2:00:00
  ./xmlchange --subgroup case.run --id USER_REQUESTED_WALLTIME --val 2:00
```

Diffusion Near the Model Top

CAM applies extra diffusion to the top levels of the model. The number of levels is indirectly controlled by the CAM namelist variable `div24del2flag`. It's not productive to assimilate in those levels because of the distorting effects of the diffusion, so the cam-fv/model_mod namelist has variables to prevent assimilation there. `Model_damping_ends_at_level` can be set to the same value that is activated by div24del2flag, or larger. An alternative way to prevent assimilation in those layers is to exclude high observations using `no_obs_assim_above_level`. The CAM6 reanalysis mentioned above used this option, with no_obs_assim_above_level = 5. If model_damping_ends_at_level is turned on (has a value other than -1) it's still sensible to exclude high observations using no_obs_assim_above_level.

It's worth considering the vertical localization when setting the value of no_obs_assim_above_level. Observations at one level can affect model variables at other levels if the localization is broad enough. The effective vertical localization can be calculated by

```
cutoff * 2 * vert_normalization_{your_vert_coord}
```

where `cutoff` is the half-width (hence the 2) of the horizontal localization (radians) and `vert_normalization_...` is the conversion from radians to the vertical coordinate system you've chosen using `vertical_localization_coord`. The resulting number can be compared against CAM's vertical levels to decide which should be excluded.

Minimum Recommended Values to Control Assimilation Near the Model Top.

div24del2flag	Diffusion levels	model_damping_ends_at_level	no_obs_assim_above_level
CAM: 2	2	2	(2; depends on localization)
WACCM: 2	3	3	(3; depends on localization)
CAM 4, 24	3	3	(3; depends on localization)
WACCM: 4, 24	4	4	(4; depends on localization)

WACCM

WACCM[#][-X] has a much higher top than the CAM versions, which requires the use of scale height as the vertical coordinate, instead of pressure, during assimilation. Another impact of the high top is that the number of top model levels with extra diffusion in the FV version is different than in the low-topped CAM-FV, so the `div24del2flag` options lead to the larger minimum values listed in the table above.

You may need to experiment to find the best choices of DART namelist variables to use with WACCM, but a good place to start includes

```
use_log_vertical_scale          = .true.
use_variable_mean_mass          = .true.
vertical_localization_coord     = 'SCALEHEIGHT'
```

(continues on next page)

(continued from previous page)

```
vert_normalization_scale_height = 1.5
cutoff                          = 0.15
no_obs_assim_above_level        = 4,
```

In any case, make the following changes (or similar) to convert from a CAM setup to a WACCM setup in `setup_hybrid`:

```
setenv compset     FWHIST
setenv resolution  f19_f19
setenv refcase     {the CASE name of the initial condition file(s) (differs from this␣
↪assimilation)}
setenv refyear     {\                                                }
setenv refmon      { >{the date of the initial condition file(s)}
setenv refday      {/                                                }
```

If there are problems with instability in the WACCM foreasts, try changing some of the following parameters in either the setup script or input.nml.

- The default `div24del2flag` in WACCM is 4. Change it in the CAM namelist section of the setup script to

```
echo " div24del2flag         = 2 "                          >> ${fname}
```

- Set a larger `ATM_NCPL` in the setup script. The default for WACCM is 144 (per day). The default for WACCM-X is 288 (per day). It's safest to choose a value which will evenly divide an hour, (for WACCM: ATM_NCPL = 168 or 192 ... multiples of 24) but evenly dividing the hindcast period might work (for a 6 hour hindcast: ATM_NCPL = 148 or 152 ... multiples of 4). To convert an existing CASE, try changing the related namelist variables `$CASEROOT/user_nl_cpl:{component}_cpl_dt` (component $\neq$ "rof")

```
user_nl_cpl:
   atm_cpl_dt = 300
   glc_cpl_dt = 300
   ice_cpl_dt = 300
   lnd_cpl_dt = 300
   ocn_cpl_dt = 300
   wav_cpl_dt = 300
```

- Increase model_damping_ends_at_level in input.nml
- Set a larger nsplit and/or nspltvrm in the CAM namelist section of the setup script:

```
echo " nsplit          = 16 "                               >> ${fname}
echo " nspltvrm        =  4 "                               >> ${fname}
```

- Reduce `inf_damping` from the default value of `0.9` in `input.nml`:

```
inf_damping            = 0.6,                  0,
```

6.108.4 CAM-SE

DART requires more information than what is available in the default output files from CAM-SE. Set the following options in the CESM `user_nl_cam` namelist to have CESM generate the files required for DART.

```
inithist               = 'ENDOFRUN'
se_write_all_corners = .true.
```

6.108.5 Nitty gritty: Efficiency and Issues to Address

Warning: Experience on a variety of machines has shown that it is a very good idea to make sure your run-time environment has the following:

```
limit stacksize unlimited
limit datasize unlimited
```

It may be very beneficial to set MPI environment variables to larger values than the defaults in $CASEROOT/env_mach_specific.xml:

```
<environment_variables>
  <env name="MPI_COMM_MAX">16383</env>
  <env name="MPI_GROUP_MAX">1024</env>
```

Reduce total core hours and queue wait times by finding the minimum number of whole nodes on which CAM will run reliably. Use that number in the setup script for each member of the ensemble.

Reduce core hours wasted by the single tasked creation of the CESM namelists before each hindcast by:

- calling case.submit with the –skip-preview-namelists argument
- replacing the cime/src/drivers/mct/cime_config/buildnml with the one in the *SourceMods* tar file.
- **ISSUE: Improve this page**
  - Add links and references to this document.
  - Publications web page.
  - CAM-chem; link? More description?
- ISSUE?; `model_interpolate` assumes that obs with a vertical location have 2 horizontal locations too. The state vector may have fields for which this isn't true, but no obs we've seen so far violate this assumption. It would have to be a synthetic/perfect_model obs, like some sort of average or parameter value.
- ISSUE: the cam-se variable `max_neighbors` is set to 6, but could be set to 4 for non-refined grids. Is there a good mechanism for this? Is it worth the file space savings?
- ISSUE: the cam-se variables `x_planar` and `y_planar` could be reduced in rank, if no longer needed for testing and debugging.

6.108.6 References and Acknowledgements

- CESM homepage

Ave Arellano did the first work with CAM-Chem, assimilating MOPPITT CO observations into CAM-Chem. Jerome Barre and Benjamin Gaubert took up the development work from Ave, and prompted several additions to DART, as well as `model_mod.f90`.

Nick Pedatella developed the first vertical_localization_coord = 'SCALEHEIGHT''' capability to enable assimilation using WACCM(-X).

Rafael Montuoro designed the first multicoupler in CESM.

6.109 Community Earth System Model

6.109.1 Preliminaries

If you found your way to this file without reading the DART ''getting started" files or tutorial, please read those first. $DART/README.rst is a good place to find pointers to them. This document gives specific help in setting up a CESM+DART assimilation for the first time. Also see the ../{your_model(s)}/readme.html documentation about the code-level interfaces and namelist values.

6.109.2 CESM

CESM is a software framework for setting up and running a combination of models, each designed to represent a part of the Earth system. Each component model (CAM, CLM, POP, CICE, ...) contributes in one of 3 modes:

- *active*; the model state evolves in time based on its own calculations
- *data*; the model simply reads data from external sources and sends representations of it to the other components.
- *stub*; the model does not interact with the other components.

The components influence each other only by passing fluxes and interface field values through the coupler. The combination chosen for a given application is called a `compset`. A compset is assembled using CESM's `create_newcase` and `case.setup` scripts, which create a new "CASE" in the "CASEROOT" directory.

CESM uses the term 'fully coupled' to refer to a compset which has an active atmosphere and ocean, regardless of other components. We use that phrase in reference to CESM compsets only, not assimilations. In CESM an active atmosphere almost always implies an active land, but that is not necessary for it to be called 'fully coupled', and, by itself, is not 'fully coupled'.

Most nonCESM models are either called by DART (low order models), or are run by DART via a shell script command (e.g. WRF). In contrast, CESM runs its hindcast, and then tells DART to do the assimilation. The result is that assimilation setup scripts for CESM components focus on modifying the build of CESM to accommodate DART's needs, such as ensemble (`multi-instance`) hindcasts, stopping at the assimilation times to run filter, and restarting with the updated model state.

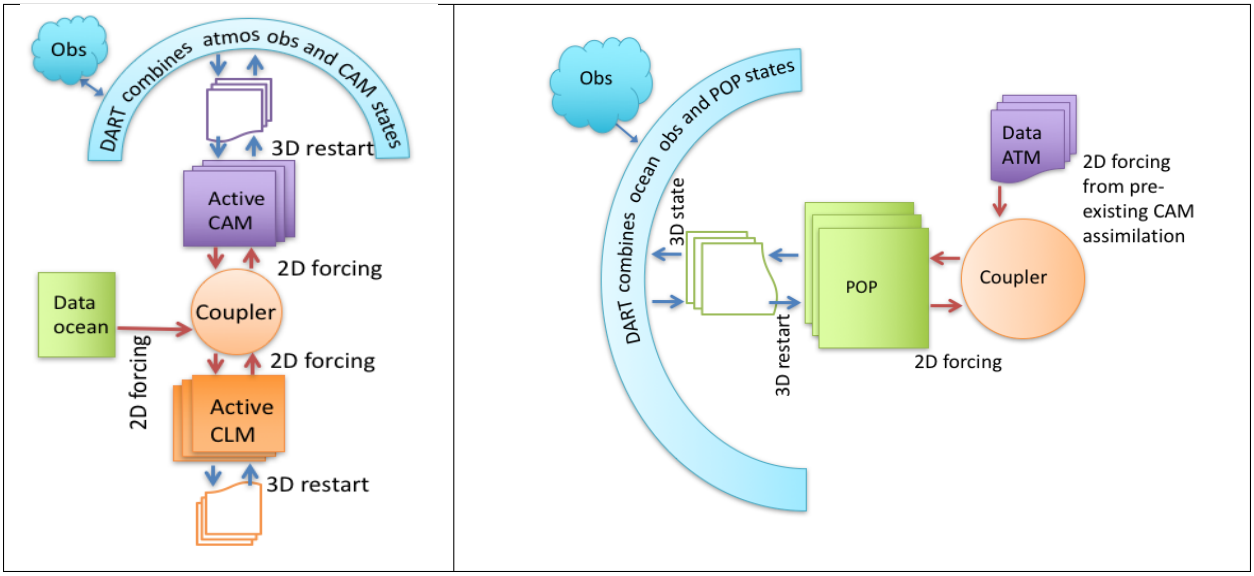
6.109.3 CESM+DART Overview

There are several modes of assimilating observations using CESM as the hindcast model. They share both the CESM and DART environments, but differ somewhat for practical and historical reasons.

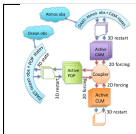
Single-component assimilation

The first, and simplest, consists of assimilating relevant observations into one active component. The other components may be active, data, or stub. Each of these assimilations is handled by one of the $DART/models interfaces (cam-fv, POP, clm, ..., see, for example the cam-fv readme

If you want to use a new combination of active and data components, you may need to (work with us to) modify the setup scripts or develop a new model interface. An example of single-component is when observations of the oceans are assimilated into the POP model state, while the atmospheric component is in data mode (it provides forcing of the ocean from CAM reanalysis files), the sea ice model may be active (but not be affected directly by observations) and the land model (CLM) may be a stub. A variation of this is used by CAM assimilations. A CAM hindcast usually uses an active land component (CLM) as well as an active atmospheric component. Atmospheric observations are assimilated only into the CAM state, while the land state is modified only through its interactions with CAM through the coupler. The sea surface temperature forcing of the atmosphere is provided by a data ocean component.



Multi-component assimilation (often called ''weakly coupled")



It's also possible to assimilate observations into multiple active components, but restricting the impact of observations to only ''their own" component. So in a ''coupled" CESM with active CAM and POP, atmospheric observations change only the CAM model state while oceanic observations change only the POP model state. This mode uses multiple *DART* model interfaces (cam-fv and POP in this example) to make a filter for each model. This mode uses scripts found in the CESM/shell_scripts directory. This does not require a models/CESM/model_mod.f90, since it uses a separate filter for each component (cam-fv, POP, ...).

Cross-component assimilation (often called ''strongly coupled")

	Work is underway to enable the assimilation of all observations into multiple active CESM components. So observations of the atmosphere would directly change the POP state variables and observations of the ocean would change the CAM state variables without interaction through the coupler. Some unresolved issues include defining the ''distance" between an observation in the atmosphere and a grid point in the ocean (for localization), and how frequently to assimilate in CAM versus POP. This mode will use code in this models/CESM directory.

Note: See *setup guidelines* for details about setting up assimilations using observations of multiple Earth system components.

6.109.4 SourceMods

Since the ability to use DART has not been completely integrated into CESM testing, it is necessary to use some CESM fortran subroutines which have been modified for use with DART. These must be provided to CESM through the SourceMods mechanism. SourceMods for selected versions of CESM are available as described in the readme.html pages of the component model interfaces ($DART/models/{cam-fv,clm,...}. This release of DART focuses on selected CESM versions from CESM2 (June, 2017) and later. Using this DART with other CESM versions will quite possibly fail, in which case existing SourceMods can often be used as a template for making SourceMods for a different CESM version. If you have nonDART CESM modifications, they must be merged with the DART modifications before building the case.

CESM2

CESM2 has several helpful features (compared to CESM1), from DART's perspective.

- ''Multi-driver" capability enables the efficient ensemble hindcasts which DART needs.
- Cycling capability, which enables multiple assimilation cycles in a single job. This reduces the frequency of waiting in the queue.
- Removal of the short term archiver from the run script so that the MPI run doesn't need to idle while the single task archiver runs.
- CESM's translation of the short term archiver to python, and control of it to an xml file ($CASEROOT/env_archive.xml), so that DART modifications to the short term archiver are more straight-forward.
- The creation of a new component class, ''External System Processing" (''esp"), of which DART is the first instance, integrates DART more fully into the CESM development, testing, and running environment. This is similar to the atm class, which has CAM-FV as an instance. This will help make DART available in the most recent tagged CESM versions which have the most recent CESM component versions.
- Reduced number of subroutines in DART's SourceMods.

These have been exploited most fully in the CAM interfaces to DART, since some other components' interfaces use older CESMs. The cam-fv/shell_scripts can be used as a template for updating other models' scripting. The multi-cycling capability, with the short term archiver running as a separate job at the end, results in assimilation jobs which rapidly fill the scratch space. Cam-fv's and POP's assimilate.csh scripts have code to remove older and unneeded CESM restart file sets during the run. All of DART's output, and user selected, restart file sets are preserved.

6.109.5 DART

DART's manhattan release includes the change to filter setting input and output filenames, instead of the user setting them in namelists. See stages. The assimilate.csh of the model interface can rename these files into the CESM file format:

$case.$component{_$instance}.$filetype.$date.nc.

DART's file names are used as new filetypes, just like CESM's existing filetypes; ''r", ''h0", ... For example, file `preassim_mean.nc` from a CAM assimilation case `Test0` may be renamed

Test0.cam.preassim_mean.2013-03-14-21600.nc

6.109.6 $DART/models/{cesm components} organization

PATHNAMES	NOTES
$DART/models/**cam-fv/**	An interface for the CAM-FV dynamical core (CAM-SE is available in $DART/models/cam-se)
...model_mod.*	The fortran interface between CAM-FV and DART
...work/	Build DART executables (filter, ...) here before running setup_*
...shell_scripts/	Setup and support scripts
.....cesm**2_0/**	Directory of scripts to setup and run in CESM2_0
.....cesm**2_1/**	Directory of scripts to setup and run in CESM2_1
...	
$DART/models/**POP/**	An interface for the POP ocean model (MOM may be interfaced next)
...model_mod.*	The fortran interface between POP and DART
...work/	Build DART executables (filter, ...) here before running setup_*
...shell_scripts/	Setup and support scripts
.....cesm**1_x/**	Directory of scripts for setting up and running POP in several versions of CESM1
.....cesm**2_0/**	Same for CESM2_0
...	
$DART/models/**clm/**	An interface for the Community Land Model (CTSM may be interfaced next)
...model_mod.*	The fortran interface between CLM and DART
...work/	Build DART executables (filter, ...) here before running setup_*
...shell_scripts/	Setup and support scripts
.....CESM1_2_1_setup_hybrid	Script to set up an ensemble assimilation case using CESM1_2_1
.....CESM1_2_1_setup_pmo	Script to set up a perfect model observation case using CESM1_2_1 (single member).
.....CESM_DART_config	Script to activate the assimilation within the CESM case.
.....*.csh	Helper scripts for setup and running.
.....user_datm.streams	CESM files which tell CLM which data atmosphere forcing to use.
...	
$DART/models/**CESM/**	An interface for multi-component assimilation (cross-component is being developed)
(...model_mod.*)	No fortran model interface. It uses the components' executables and namelists.
(...work/)	No executables; they are built in the component interfaces' work directories.
...shell_scripts/	Setup and support scripts
.....CESM1_1_1_setup_hybrid	Set up a multi-component assimilation using initial conditions from a single CESM B compset ca
.....CESM1_1_1_setup_initial	Same, but initial conditions from a different sources for each active component.
.....CESM1_1_1_setup_pmo	Set up a perfect model observation case (single member).
.....CESM_DART_config	Script to activate the assimilation within the CESM case.

contin

Table 2 – continued from previous page

PATHNAMES	NOTES
……assimilate.csh	Small script to successively call the {comp}_assimilate.csh scripts
……{comp}_assimilate.csh	assimilate.csh scripts for each active component (cam, pop, clm, …)
……run_perfect_model_obs.csh	Script to run a perfect model observation job.
…	

Warning: Experience on a variety of machines has shown that it is a very good idea to make sure your run-time environment has the following:

```
limit stacksize unlimited
 limit datasize unlimited
```

Tip: Also, large ensemble assimilations may run more efficiently by setting environment variables in $CASEROOT/env_mach_specific.xml to something larger than their defaults. The variable names in an Intel MPI environment (17.x.x in 2020) are:

```
   <environment_variables>
      <env name="MPI_GROUP_MAX">1024</env>
      <env name="MPI_COMM_MAX">16383</env>

and preventing unnecessary rebuilds of the component namelists
by submitting the job using

$ case.submit --skip-preview-namelist
```

6.110 CICE

6.110.1 Overview

The Community Ice CodE (CICE) is a sea ice model that was first developed by Elizabeth Hunke as the Los Alamos Sea Ice Model. Its code base and capabilities have grown as a result of continued development by the broader geosciences community, an effort organized by the CICE Consortium.

Dr. Cecilia Bitz implemented support for the CICE model (as part of CESM) in DART. The DART model interface was developed to work with CICE's dynamical core on an Arakawa B-grid.[1] When CICE is coupled to POP in CESM, the ocean and sea ice grids are identical.

According to the CICE manual:

> The spatial discretization is specialized for a generalized orthogonal B-grid as in Murray (1996)[2] or Smith et al. (1995).[3] The ice and snow area, volume and energy are given at the center of the cell, velocity is

[1] Arakawa, Akio and Vivian R. Lamb, 1977: Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model. *Methods in Computational Physics: Advances in Research and Applications*, **17**, 173–265, doi:10.1016/B978-0-12-460817-7.50009-4

[2] Murray, Ross J., 1996: Explicit Generation of Orthogonal Grids for Ocean Models. *Journal of Computational Physics*, **126**, 251–273, doi:10.1006/jcph.1996.0136

[3] Smith, Richard D., Samuel Kortas and Bertrand Meltz, 1995: Curvilinear Coordinates for Global Ocean Models. Technical Report LA-UR95-1146, Los Alamos National Laboratory.

defined at the corners, and the internal ice stress tensor takes four different values within a grid cell; bilinear approximations are used for the stress tensor and the ice velocity across the cell, as described in Hunke and Dukowicz (2002).[4] This tends to avoid the grid decoupling problems associated with the B-grid.

Hence, in the DART interface:

- `U`, `V` are at grid cell corners
- `T`, `h`, `hs`, and the various scalar quantities are at grid cell centers

CICE is under development to work with other grids, such as the unstructured grid in MPAS and the C-grid in MOM.

6.110.2 Namelist

```
&model_nml
   assimilation_period_days     = 1
   assimilation_period_seconds  = 0
   model_perturbation_amplitude = 0.00002
   update_dry_cell_walls        = .false.
   binary_grid_file_format      = 'big_endian'
   debug                        = 1
   model_state_variables        = 'aicen', 'QTY_SEAICE_CONCENTR',   'UPDATE',
                                  'vicen', 'QTY_SEAICE_VOLUME',     'UPDATE',
                                  ...
                                  'vsnon', 'QTY_SEAICE_SNOWVOLUME', 'UPDATE',
/
```

Description of each namelist entry

Item	Type	Description
time_step_days	integer	Number of days for dimensional timestep, mapped to deltat.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to deltat.
model_perturbation_amplitude	real(r8)	Perturbation amplitude
update_dry_cell_walls	logical	Currently does nothing. Additional code is needed to detect the cells which are wet but within 1 cell of the bottom/sides/etc.
binary_grid_file_format	character(64)	Byte sequence for the binary grid. Valid values are native, big_endian & little_endian.
debug	integer	When set to 0, debug statements are not printed. Higher numbers mean more debug reporting.
model_state_variables	character(*)	List of model state variables

[4] Hunke, Elizabeth C., and John K. Dukowicz, 2002: The Elastic–Viscous–Plastic Sea Ice Dynamics Model in General Orthogonal Curvilinear Coordinates on a Sphere—Incorporation of Metric Terms. *Monthly Weather Review*, **130**, 1848–1865, doi:10.1175/1520-0493(2002)130%3C1848:TEVPSI%3E2.0.CO;2

References

6.111 CLM

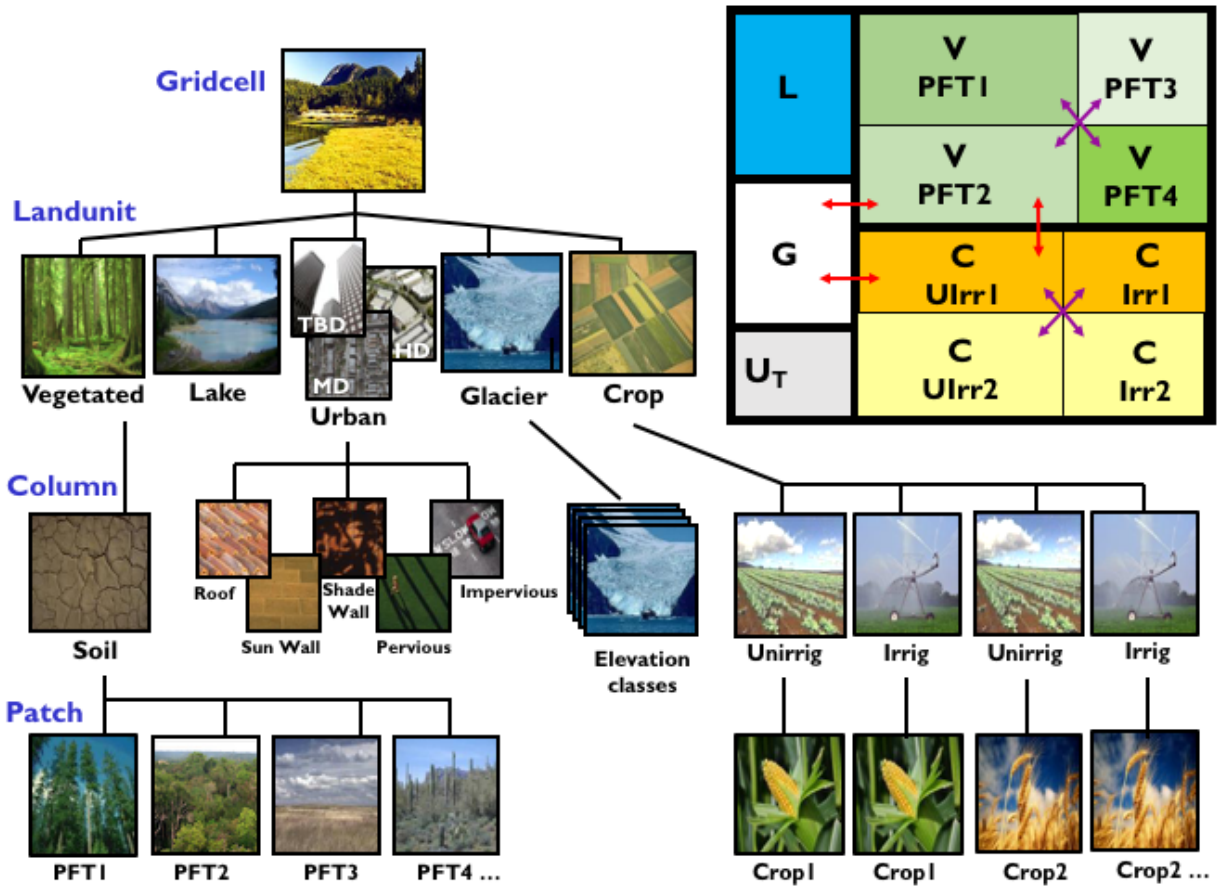


Figure from the CLM5.0 Tech Note. The image is a link to the document.

6.111.1 Overview

This is the DART interface to the CESM2 Community Land Model. Specifically, CESM **release-cesm2.2.0** using CLM **release-cesm2.2.01**

This document is most useful if the user has a prior understanding of running CESM and also running CLM-DART. For this reason **we strongly recommend the following two steps be completed before reviewing this document**:

First, we recommend that you become familiar with running a single or multi-instance experiment in CESM (i.e. a CLM 'free' run) before you try to run CLM-DART. The CLM-DART software uses language and concepts that should be familiar to CESM users. The CLM-DART capability is entirely dependent on the multi-instance capability of CESM, first supported in its entirety in CESM1.1.1. Consequently, this version or newer is required to run CLM/DART. The CTSM Documentation has reference material for CLM.

Second, we recommend the user complete the *CLM5-DART Tutorial.* This tutorial provides the user with 13 steps of hands-on experience for downloading, compiling, editing, executing,and diagnosing a simple CLM5-DART assimilation run. It will provide users with skills to modify the CLM-DART scripts for their own research applications.

DART uses the multi-instance capability of CESM, which means that DART is not responsible for advancing the model. This GREATLY simplifies the traditional DART workflow, but it means *CESM has to stop and write out a restart file every time an assimilation is required*. The multi-instance capability is relatively new to CESM and we are in close collaboration with the CESM developers to make using DART with CESM as easy as possible. See the *SourceMods* section for a desciption of the changes to the CLM source code that are useful in a data assimilation context.

CLM is a rapidly-moving target and DART is developed and maintained by a small group of people. Consequently, we have focused on supporting *released* versions of CLM. This documentation and scripting were tested using the CESM tag **release-cesm2.2.0** and CLM tag **release-cesm2.2.01** following the download instructions from https://github.com/ESCOMP/CESM .

CLM-DART has been used to assimilate snow data, soil moisture, leaf area index, biomass, solar-induced fluorescence and more. See the *References* section below for some examples. The list is by no means complete. Each experiment has extended the capabilities of the CLM-DART system. The observations supported by DART are contained in the *obs_def_land_mod.f90*, *obs_def_tower_mod.f90*, and *obs_def_COSMOS_mod.f90* modules - each of which may be easily extended. New observation types are usually easily supported. There are **many** examples of routines to convert from native data formats (netCDF, HDF, csv, etc.) to a DART observation sequence file.

6.111.2 Important Features

Land DA is extremely diverse. The support for Land DA as pertains to CLM-DART has some features that need to be described in some detail.

SourceMods

While we strive to keep DART requirements out of the model code, there are a few SourceMods needed to run DART for CLM from within CESM. Appropriate SourceMods for each CESM version are available as tar files at http://www.image.ucar.edu/pub/DART/CESM They will create a directory with the appropriate SourceMods structure. It is up to you to either use them 'as is' or put them under version control in your CESM source code installation. The DART scripting allows you to specify a directory containing the SourceMods - and then copies those SourceMods to the appropriate CASEROOT directory.

The `DART_params.csh` file has a large comment block in the section pertaining to SourceMods.

SourceMod File	Description
`biogeochem/CNBalanceCheckMod.F90`	Suppress balance checks for first restart step. This is **highly recommended**. Many people find it to suppress balance checks for ALL timesteps.
`cpl/mct/lnd_import_export.F90`	(deprecated) DS199.1 originally had some slightly negative downward radiations that needed to be corrected.
`biogeophys/SurfaceRadiationMod.F90`	Allows the use of 'PARVEG' in a history file. Normally, only 'PARVEGLN' is output.
`biogeophys/CanopyFluxesMod.F90`	Used to calculate SIF
`biogeophys/PhotosynthesisMod.F90`	Used to calculate SIF

CLM indeterminate values

CLM variables are encoded as rectangular arrays in the netCDF files. However, this means that some variables have space for layers that are unused. Anything with snow layers, for example. CLM has the *SNLSNO* variable to indicate which snow layers are active. The *unused* layers may not have the *_FillValue* value, but can have 'indeterminate' values. The *PROGRAM clm_to_dart* must be run to convert these indeterminate values to *_FillValue* to be interpreted correctly by DART. After the assimilation is complete, the *PROGRAM dart_to_clm* must be called to replace the *_FillValue* with whatever is originally in that slot. This approach preserves the 'indeterminate' value for *unused* snow layers and prevents DART from adjusting the value during the *filter* step. If the surface snow layer has a *trace* of snow this is considered an active snow layer, and we allow DART to adjust this value. See the *Discussion of Indeterminate Values* section of *PROGRAM clm_to_dart* for more details.

Model Interpolate - The Forward Operator

Since the subgridscale components of CLM have no explicit location associated with them, the location of every component in the gridcell is the same as the gridcell itself. The DART forward operators fundamentally rely on interpolating the model state to some arbitrary location. At present, the best we can do is to create an area-weighted average of all components in the gridcell. This is sub-optimal because it introduces representation mismatch between the grid cell and observation spatial resolution. A nice project would be to use a lookup table for the observation location to determine the dominant PFT (or relevant metadata) at that location and only average the PFTs specifically associated with the observation within the gridcell. This will allow the forward operator to be more accurate and might have a discernable impact on the regression relationship (i.e. ensemble covariance) between the variables in the DART state vector.

The *model_interpolate* function in DART achieves efficiency by interpolating all the ensemble members at the same time. This gives rise to some challenging problems when interpolating values for variables with with changing numbers of active layers. For example, some ensemble members may only have 2 active snow layers, some may have 3. This is an untenable situation when asked for the snow temperature or water content in layer 3, for example. Consequently - *model_interpolate* will fail and return an error code - the forward operator will fail - and the observation is rejected and the DART QC is marked as such. Be aware.

Localization

Localization is the term used to restrict the portion of the state to the portion believed to be related to the observation. Most often, this is a spatial argument but it does not need to be restricted to that. In some way, even the selection of the CLM variables to include in the DART state is a de-facto localization. Since CLM has such a rich description of land unit types: urban columns, glaciers, lakes, etc. it is also possible (and probably desirable) to explicitly declare some columns and/or PFTs to be unaffected by the assimilation - i.e., we declare that soil moisture observations should not impact urban columns or deep lakes or … The **get_close_state()** function employs a routine to explicitly declare what subgridscale components are allowed to be modified by the assimilation. This routine can easily be customized to suit your purpose. The code segment below should make this clear.

```
! Determine if state_index is a variable from a column (or whatever is of interest).
! Determine what dimension is of interest, need to know to index into
! cols1d_ityplun(ncolumn) array (for example).

RELATEDLOOP: do jdim = 1, get_num_dims(dom_id, var_id)

   dimension_name = get_dim_name(dom_id, var_id, jdim)
   select case ( trim(dimension_name) )
          case ("gridcell","lon","lat")
             related = .true.
          case ("lndgrid")
```

(continues on next page)

(continued from previous page)

```
            related = .true.
         case ("landunit")
            if ( land1d_ityplun(indices(jdim)) == ilun_vegetated_or_bare_soil ) related␣
↪= .true.
            if ( land1d_ityplun(indices(jdim)) == ilun_crop                   ) related␣
↪= .true.
         case ("column")
            if ( cols1d_ityplun(indices(jdim)) == icol_vegetated_or_bare_soil ) related␣
↪= .true.
            if ( cols1d_ityplun(indices(jdim)) == icol_crop                   ) related␣
↪= .true.
         case ("pft")
            related = .true.
         case default
   end select

   ! Since variables can use only one of these dimensions,
   ! there is no need to check the other dimensions.
   if (related) exit RELATEDLOOP

enddo RELATEDLOOP
```

Snow Data Assimilation

The *prognostic* variables for snow (i.e. the ones that impact the forecast) are the ones that have layers. The snow observations are typically without explicit depths and are essentially column-integrated quantities like snow water equivalent (SWE - CLM variable *H2OSNO*) or snow depth (CLM variable *SNOW_DEPTH*). These CLM *diagnostic* variables simplify the forward operator and have been part of the DART state. However, updating the prognostic variables (*T_SOISNO, H2OSOI_LIQ, H2OSOI_ICE, DZSNO, ZSNO, ZISNO*) through their ensemble covariance with the update to *H2OSNO*, for example will generally not result in a posterior SWE (calculated from the prognostic variables) that matches the posterior SWE in *H2OSNO*.

In order to address this challenge, a snow repartitition function has been created in `dart_to_clm` that redistributes the posterior SWE into the appropriate prognostic variables. This **guarantees** that the posterior SWE of the prognostic snow variables matches the posterior SWE in H2OSNO. **When snow related variables are being updated within an assimilation it is recommended to invoke this repartitioning function by setting the namelist option ``repartition_swe = 1 or 2`` within ``&dart_to_clm_nml``.** See the *PROGRAM dart_to_clm* for more details describing the repartitioning function including guidance on how to set up a case that repartitions snow. Note that we have not attempted to include any of the snow property variables most important to controlling albedo (eg. grain radius, carbon, dust) within the DART state. To what extent adjusting mass and dimensional properties of snow layers indirectly influences the albedo properties is an active scientific question. See the *PROGRAM dart_to_clm* for more details on how to implement `repartition_swe` if conserving albedo is important for your application.

The snow formulation in CLM is complex. Reducing the amount of snow through assimilation is well-defined. Creating snow when there is none is **a limited capability** in CLM-DART. If snow exists for a subset of ensemble members at a given location, then it is possible to adjust ensemble members with a value of zero to a non-zero value. On the other hand, **if all ensemble members do not have snow, or at least one member has a FillValue**, the statistical assumptions for ensemble data assimilation are not valid and the snow variables remain at zero. The best method would be to alter the amount of snow *from the forcing file* and let CLM manage the snow. This is beyond the scope of CLM-DART. We have thought that if one member does not have snow - maybe we should just use the values from some other member - but when does that stop being acceptable? 10 ensemble members? 20? The distributions become multimodal, and the logical end result is that you could wind up using 1 ensemble member to declare the snow for all the remaining

members. That seems like a bad idea.

Similar logic applies to the variables related to plant growth. If the LAI observations indicates there should be something growing and nothing has sprouted yet, DART does nothing to the variables.

6.111.3 Configuring an Experiment

Our notes on how to set up, configure, build, and run CESM for an assimilation experiment evolved into scripts. These scripts are not intended to be a 'black box'; you will have to read and understand them and modify them to your own purpose. They are heavily commented – in keeping with their origins as a set of notes. If you would like to offer suggestions on how to improve those notes - please send them to dart@ucar.edu - we'd love to hear them.

shell_scripts/cesm2_2	Description
simple.csh	The script to run first. This configures and builds a very simple single-instance CLM experiment. This is intended to check if CLM runs on your machine. It has nothing to do with DART. There are **no** DART requirements in it.
DART_params.csh	Parameter file for use when running CLM and DART. This file has all the configuration items needed and will be copied into the CASEROOT directory to be used during an experiment. Other setup scripts within this table require the parameter values defined in this file.
CLM5_spinup.csh	This script takes the single (spun-up) CLM state supplied with the compset and forecasts an ensemble of these. Each ensemble member uses a unique data atmosphere stream file. After some time, the ensemble of CLM states have enough diversity to be a useful initial ensemble for an assimilation experiment. In this configuration, no observations are required, and no DART exectuables are involved.
CLM5_hybrid_ensemble	Given an ensemble of CLM states, advance the ensemble using unique DATM stream files for each ensemble member. This starts from a CESM 'hybrid' run-type, so the initial staging of the ensemble is required (and performed by this script). In this configuration, no observations are required, and no DART exectuables are involved.
CLM5_setup_pmo	Take a single instance from a spun-up ensemble and advances CLM in 24-hour segments. If that works, the setup can be extended to run `perfect_model_obs` to harvest synthetic observations from the single instance, which is now the true state of the system. This requires the creation of a series of *obs_seq.in* files which can be created with *program create_obs_sequence* `CLM5_setup_pmo` creates a file called *CESM_instructions.txt* in the CASEROOT directory with instructions on how to extend the setup to run DART.
CLM5_setup_assimilation	Setup a single instance CLM experiment and can be used to perform an assimilation. CLM advances in 24-hour segments. If that works, the setup can be extended to run `filter`. Keep in mind that if all observations are denoted as 'evaluate_these_obs', this is equivalent to a free run with the added advantage that you can compare the observation-space diagnotics to a subsequent experiment that assimilates the observations. Each CLM instance uses a unique DATM forcing, `CLM5_setup_assimilation` creates a file called *CESM_instructions.txt* in the CASEROOT directory with instructions on how to extend the setup to run DART. The *user_nl_clm* namelists have been configured to output several history files - some in the common 'XY' (gridcell) format, some in the same format as the restart file (vector). This is to demonstrate methods for creating variables useful for forward operators.
CESM_DART_config	Add the bits to a CESM case with the bits and pieces required to run DART. When either `CLM5_setup_pmo` or `CLM5_setup_assimilation` gets executed, `CESM_DART_config` gets copied to the CESM CASEROOT directory and should be run there. It is designed such that you can execute it at any time during a CESM experiment. When you do execute it, it will build the DART executables if needed and copy them into the CESM EXEROOT directory, stage the run-time configurable `input.nml` in the CASEROOT directory, etc. `CESM_DART_config` creates a file called *DART_instructions.txt* in the CASEROOT directory with instructions on how to modify the experiment to run DART. `CESM_DART_config` also creates a file called *stage_dart_files* in the CASEROOT directory to assist in copying updated executables to the correct directory for the experiment.
perfect_model.csh	This script is invoked by CESM when specified by the DATA_ASSIMILATION_SCRIPT resource. This resource is specified by `CESM_DART_config`. `perfect_model.csh` locates the appropriate observation sequence file and links it to the expected name, removes the indeterminate values from the input files so the DART requirements are met, and then harvests the synthetic observations and moves them to the *pmo_output_baseobsdir* directory (as specified in *DART_params.csh*)
assimilate.csh	This script is invoked by CESM when specified by the DATA_ASSIMILATION_SCRIPT resource. This resource is specified by `CESM_DART_config`. `assimilate.csh` links the appropriate observation sequence file and and **copies** the CLM restart files and removes the indeterminate values. If inflation is specified, `assimilate.csh` reads the latest inflation values from the inflation pointer file. The assimilation is performed. If inflation was specified the inflation pointer files are updated in preparation for the next assimilation cycle. All output is tagged with the date-time-stamp of the model state for that cycle.

6.111.4 Declaring the Variables in the DART State

The DART state vector is constructed in a very flexible manner. A namelist is used to relate the netCDF variable name, the netCDF file type [restart, (XY) history, or vector history] with a DART QUANTITY. Including variables from an 'XY' CLM history file allows the inclusion of diagnostic variables that can speed up the forward observation operators if gridcell averages are appropriate.

It is also possible to read some variables from the restart file, and some from a 'vector-based' history file that has the same structure (gridcell/landunit/column/pft) as the restart file - but may be temporal averages instead of instantaneous quantities. Care must be taken to assign the proper DART QUANTITY to the variables. Any variable in the DART state is updated, but the forward operator looks for specific QUANTITIES. If you want to use the vector-based history file for the forward operator - make sure you declare it to be of the QUANTITY used by the forward operator code.

The namelist specification of what goes into the DART state vector includes the ability to specify if the quantity should have a lower bound, upper bound, or both, what file the variable should be read from, and if the variable should be modified by the assimilation or not. Make sure you read the *Inflation* section to fully understand what happens when you designate a varible 'NO_COPY_BACK'.

> **Attention:** It is important to know that the variables in the DART diagnostic files `preassim`, `postassim`, `analysis`, and `output` will contain the unbounded versions of ALL the variables specified in `clm_variables`. Only the files specified in the `filter_nml:output_state_file_list` will have the 'clamped' values.

The example `input.nml model_nml` demonstrates how to construct the DART state vector. The following table explains in detail each entry for `clm_variables`:

Column	Description
1	The CLM variable name as it appears in the CLM netCDF file.
2	The corresponding DART QUANTITY.
3	Minimum value of the posterior. If set to 'NA' there is no minimum value. The DART diagnostic files will not reflect this value, but the file used to restart CLM will.
4	Maximum value of the posterior. If set to 'NA' there is no maximum value. The DART diagnostic files will not reflect this value, but the file used to restart CLM will.
5	Specifies which file should be used to obtain the variable. `'restart'` => clm_restart_filename `'history'` => clm_history_filename `'vector'` => clm_vector_history_filename
6	Should `filter` update the variable in the specified file. `'UPDATE'` => the variable is updated. `'NO_COPY_BACK'` => the variable remains unchanged.

The following are only meant to be examples - they are not scientifically validated. Some of these that are UPDATED are probably diagnostic quantities, Some of these that should be updated may be marked NO_COPY_BACK. This list is by no means complete.

```
clm_variables  = 'leafc',       'QTY_LEAF_CARBON',            '0.0', 'NA', 'restart' ,
↪'UPDATE',
                 'frac_sno',    'QTY_SNOWCOVER_FRAC',         '0.0', '1.', 'restart' ,
↪'UPDATE',
                 'SNOW_DEPTH',  'QTY_SNOW_THICKNESS',         '0.0', 'NA', 'restart' ,
↪'NO_COPY_BACK',
                 'H2OSOI_LIQ',  'QTY_SOIL_LIQUID_WATER',      '0.0', 'NA', 'restart' ,
↪'UPDATE',
                 'H2OSOI_ICE',  'QTY_SOIL_ICE',               '0.0', 'NA', 'restart' ,
↪'UPDATE',
                 'T_SOISNO',    'QTY_TEMPERATURE',            '0.0', 'NA', 'restart' ,
↪'UPDATE',
                 'livestemc',   'QTY_LIVE_STEM_CARBON',       '0.0', 'NA', 'restart' ,
```

(continues on next page)

(continued from previous page)

```
↪'UPDATE',
                'deadstemc',   'QTY_DEAD_STEM_CARBON',       '0.0', 'NA', 'restart' ,
↪'UPDATE',
                'NEP',         'QTY_NET_CARBON_PRODUCTION',  'NA' , 'NA', 'history' ,
↪'NO_COPY_BACK',
                'H2OSOI',      'QTY_SOIL_MOISTURE',          '0.0', 'NA', 'history' ,
↪'NO_COPY_BACK',
                'SMINN_vr',    'QTY_SOIL_MINERAL_NITROGEN',  '0.0', 'NA', 'history' ,
↪'NO_COPY_BACK',
                'LITR1N_vr',   'QTY_NITROGEN',               '0.0', 'NA', 'history' ,
↪'NO_COPY_BACK',
                'TSOI',        'QTY_SOIL_TEMPERATURE',       'NA' , 'NA', 'history' ,
↪'NO_COPY_BACK',
                'FSDSVDLN',    'QTY_PAR_DIRECT',             '0.0', 'NA', 'history' ,
↪'NO_COPY_BACK',
                'FSDSVILN',    'QTY_PAR_DIFFUSE',            '0.0', 'NA', 'history' ,
↪'NO_COPY_BACK',
                'PARVEGLN',    'QTY_ABSORBED_PAR',           '0.0', 'NA', 'history' ,
↪'NO_COPY_BACK',
                'NEE',         'QTY_NET_CARBON_FLUX',        'NA' , 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                'H2OSNO',      'QTY_SNOW_WATER',             '0.0', 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                'TLAI',        'QTY_LEAF_AREA_INDEX',        '0.0', 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                'TWS',         'QTY_TOTAL_WATER_STORAGE',    'NA' , 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                'SOILC_vr',    'QTY_SOIL_CARBON',            '0.0', 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                'SOIL1N_vr',   'QTY_SOIL_NITROGEN',          '0.0', 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                'SMP',         'QTY_SOIL_MATRIC_POTENTIAL',  '0.0', 'NA', 'vector'  ,
↪'NO_COPY_BACK'
  /
```

Only the first variable for a DART QUANTITY in the clm_variables list will be used for the forward observation operator. The following is perfectly legal:

```
clm_variables = 'LAIP_VALUE', 'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'restart' , 'UPDATE',
                'tlai',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'restart' , 'UPDATE',
                'elai',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'restart' , 'UPDATE',
                'ELAI',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'history' , 'NO_COPY_
↪BACK',
                'LAISHA',     'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'history' , 'NO_COPY_
↪BACK',
                'LAISUN',     'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'history' , 'NO_COPY_
↪BACK',
                'TLAI',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'history' , 'NO_COPY_
↪BACK',
                'TLAI',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'vector'  , 'NO_COPY_
↪BACK'
  /
```

however, only **LAIP_VALUE** will be used to calculate the LAI when an observation of LAI is encountered. **All** (the other LAI) variables in the DART state will be modified by the assimilation based on the relationship of LAIP_VALUE and the observation. It is possible that several clm variables could serve as the input for the forward operator, however, in practice, the user should choose the variable that best matches the observation (temporal/spatial resolution, units etc), to help limit the complexity of the forward operator.

6.111.5 Inflation

Inflation has been shown to be quite useful in our experience of DA with CLM and DART. The model is strongly influenced by the atmospheric forcing and will cause the CLM ensemble to relax to a state consistent with the forcing when the assimilation stops.[5] Depending on the forecast length between assimilations, and sometimes just to restore the variance lost during an assimilation, inflation should be used.

The 'NO_COPY_BACK' designation has some side effects when it comes to state-space inflation (inf_flavor 2,4 or 5 - 'VARYING_SS_INFLATION','RELAXATION_TO_PRIOR_SPREAD', or 'ENHANCED_SS_INFLATION' - respectively). State-space inflation requires an inflation value for everything in the DART state. If the variable has been designated as 'NO_COPY_BACK' the DART write routine (when called from `filter`) simply skips the variable and nothing is written. This is a problem for inflation files that need to adapt.

The solution is to run *PROGRAM fill_inflation_restart* to create an initial inflation file with inflation values of 1.0 (i.e. no inflation). `fill_inflation_restart` has been specially designed to output inflation values for every variable in the DART state. The idea is to copy the *input* inflation file to the *output* inflation file name *before each assimilation cycle*. No new values will be written for the variables designated 'NO_COPY_BACK', the original values will persist.

It remains a scientific question as to whether or not this is the **right** thing to do! The 'NO_COPY_BACK' mechanism was initially intended to simply avoid writing variables that did not impact the next model forecast. Since inflation is a powerful mechanism to overcome observation-model bias, it might be perfectly warranted to 'UPDATE' these diagnostic variables. Be warned, if you do 'UPDATE' the diagnostic variables, you may want to create copies of the prior so you explore exactly what happens during an assimilation.

If the filter namelist specifies the use of inflation, the `assimilate.csh` script is configured to run `fill_inflation_restart` on the first assimilation cycle. The inflation filenames are put in a pointer file which is continually updated as the experiment progresses.

> **Attention:** It is recommended to apply no inflation during the first assimilation step. In other words within `input.nml` and namelist `&fill_inflation_restart_nml` set `prior_inf_mean = 1.00` and `post_inf_mean = 1.00`. Otherwise, a spatially uniform inflation will be applied to the entire spatial domain of the assimilation which can make CLM unstable. In general, inflation is intended to account for biases between the observation and model-estimated observation, as well as to restore ensemble spread after an observation has been assimilated.

6.111.6 Namelist

Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. These are the defaults:

```
&model_nml
  clm_restart_filename         = 'clm_restart.nc',
  clm_history_filename         = 'clm_history.nc',
  clm_vector_history_filename  = 'clm_vector_history.nc',
  output_state_vector          = .false.,
```

(continues on next page)

[5] Fox, A. M., Hoar, T. J., Anderson, J. L., Arellano, A. F., Smith, W. K., Litvak, M. E., et al., 2018: Evaluation of a data assimilation system for land surface models using CLM4.5. *Journal of Advances in Modeling Earth Systems*, **10**, 2471–2494, doi.org/10.1029/2018MS001362

(continued from previous page)

```
 assimilation_period_days     = 2,
 assimilation_period_seconds  = 0,
 model_perturbation_amplitude = 0.2,
 calendar                     = 'Gregorian',
 debug                        = 0
 clm_variables  = 'frac_sno',    'QTY_SNOWCOVER_FRAC',         'NA' , 'NA', 'restart' ,
↪'NO_COPY_BACK',
                  'H2OSNO',      'QTY_SNOW_WATER',             '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'H2OSOI_LIQ',  'QTY_SOIL_MOISTURE',          '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'H2OSOI_ICE',  'QTY_ICE',                    '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'T_SOISNO',    'QTY_SOIL_TEMPERATURE',       'NA' , 'NA', 'restart' ,
↪'UPDATE',
                  'SNOWDP',      'QTY_SNOW_THICKNESS',         'NA' , 'NA', 'restart' ,
↪'UPDATE',
                  'LAIP_VALUE',  'QTY_LEAF_AREA_INDEX',        'NA' , 'NA', 'restart' ,
↪'NO_COPY_BACK',
                  'cpool',       'QTY_CARBON',                 '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'frootc',      'QTY_ROOT_CARBON',            '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'leafc',       'QTY_LEAF_CARBON',            '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'leafn',       'QTY_LEAF_NITROGEN',          '0.0', 'NA', 'restart' ,
↪'UPDATE',
                  'NEP',         'QTY_NET_CARBON_PRODUCTION',  'NA' , 'NA', 'history' ,
↪'NO_COPY_BACK',
                  'TV',          'QTY_VEGETATION_TEMPERATURE', 'NA' , 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                  'RH2M_R',      'QTY_SPECIFIC_HUMIDITY',      'NA' , 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                  'PBOT',        'QTY_SURFACE_PRESSURE',       'NA' , 'NA', 'vector'  ,
↪'NO_COPY_BACK',
                  'TBOT',        'QTY_TEMPERATURE',            'NA' , 'NA', 'vector'  ,
↪'NO_COPY_BACK'
  /
```

Item	Type	Description
clm_restart_filename	character(len=256)	this is the filename of the CLM restart file. The DART scripts resolve linking the specific CLM restart file to this generic name. This file provides the elements used to make up the DART state vector. The variables are in their original landunit, column, and PFT-based representations.
clm_history_filename	character(len=256)	this is the filename of the CLM `.h0.` history file. The DART scripts resolve linking the specific CLM history file to this generic name. Some of the metadata needed for the DART/CLM interfaces is contained only in this history file, so it is needed for all DART routines.
clm_vector_history_filename	character(len=256)	this is the filename of a second CLM history file. The DART scripts resolve linking the specific CLM history file to this generic name. The default setup scripts actually create 3 separate CLM history files, the `.h2.` ones are linked to this filename. It is possible to create this history file at the same resolution as the restart file, which should make for better forward operators. It is only needed if some of the variables specified in `clm_variables` come from this file.
output_state_vector	logical	If .true. write state vector as a 1D array to the DART diagnostic output files. If .false. break state vector up into variables before writing to the output files.
assimilation_period_days, assimilation_period_seconds	integer	Combined, these specify the width of the assimilation window. The current model time is used as the center time of the assimilation window. All observations in the assimilation window are assimilated. BEWARE: if you put observations that occur before the beginning of the assimilation_period, DART will error out because it cannot move the model 'back in time' to process these observations.
model_perturbation_amplitude	real(r8)	Required by the DART interfaces, but not used by CLM.
calendar	character(len=32)	string specifying the calendar to use with DART. The CLM dates will be interpreted with this same calendar. For assimilations with real observations, this should be 'Gregorian'.
debug	integer	Set to 0 (zero) for minimal output. Successively higher values generate successively more output.

6.111.7 Modules used

```
default_model_mod
distributed_state_mod
ensemble_manager_mod
mpi_utilities_mod
netcdf_utilities_mod
obs_def_utilities_mod
obs_kind_mod
options_mod
state_structure_mod
threed_sphere/location_mod
time_manager_mod
types_mod
utilities_mod
```

6.111.8 Files

filename	purpose
input.nml	to read the model_mod namelist
clm_restart.nc	both read and modified by the CLM model_mod
clm_history.nc	read by the CLM model_mod for metadata and possible diagnostic variables.
clm_vector_history.nc	read by the CLM model_mod for possible diagnostic variables.
dart_log.out	the run-time diagnostic output
dart_log.nml	the record of all the namelists actually USED - contains the default values

6.111.9 Error codes and conditions

Routine	Message	Comment
nc_write_model_atts	Various netCDF-f90 interface error messages	From one of the netCDF calls in the named routine

6.111.10 Future plans:

1. Implement a lookup table that relates the observation location to a dominant PFT or COLUMN so the *model_interpolate* code can average quantities from similar PFTs or COLUMNs instead of everything in the entire grid cell.
2. Implement a fast way to get the quantities needed for the calculation of radiative transfer models - needs a whole column of CLM variables, redundant if multiple frequencies are used.
3. Figure out what to do when one or more of the ensemble members does not have snow/leaves/etc. when the observation indicates there should be. Ditto for removing snow/leaves/etc. when the observation indicates otherwise.
4. Right now, the soil moisture observation operator is used by the COSMOS code to calculate the expected neutron intensity counts. This is the right idea, however, the COSMOS forward operator uses m3/m3 and the CLM units are kg/m2. I have not checked to see if they are, in fact, identical. This brings up a bigger issue in that the soil moisture observation operator would also be used to calculate whatever a TDT probe or ??? would measure. What units are they in? Can one operator support both?

6.111.11 References

The CTSM Documentation is THE reference for CLM.

Attention: The remainder of the document describes the deprecated scripts in the *shell_scripts/cesm1_x* directory - for reference only. These scripts **will not work** with CESM2.

Script	Description
shell_scripts/ CESM1_1_1_setup_pmo	runs a single instance of CLM to harvest synthetic observations for an OSSE or "perfect model" experiment. It requires a single CLM state from a previous experiment and uses a specified DATM stream for forcing. This parallels an assimilation experiment in that in the multi-instance setting each CLM instance may use (should use?) a unique DATM forcing. This script has almost nothing to do with DART. There is one (trivial) section that records some configuration information in the DART setup script, but that's about it. This script should initially be run without DART to ensure a working CESM environment. As of (V7195) 3 October 2014, this script demonstrates how to create 'vector'-based CLM history files (which requires a bugfix) and has an option to use a bugfixed snow grain-size code. http://bugs.cgd.ucar.edu/show_bug.cgi?id=1730 http://bugs.cgd.ucar.edu/show_bug.cgi?id=1934
shell_scripts/ CESM1_2_1_setup_pmo	functionally identical to CESM1_1_1_setup_pmo but is appropriate for the the CESM 1_2_1 release, which supports both CLM 4 and CLM 4.5.
shell_scripts/ CESM1_1_1_setup_hybrid	runs a multi-instance CLM experiment and can be used to perform a free run or 'open loop' experiment. By default each CLM instance uses a unique DATM forcing. This script also has almost nothing to do with DART. There is one (trivial) section that records some configuration information in the DART setup script, but that's about it. This script should initially be run without DART to ensure a working CESM. As of (V7195) 3 October 2014, this script demonstrates how to create 'vector'-based CLM history files (which requires a bugfix) and has an option to use a bugfixed snow grain-size code. http://bugs.cgd.ucar.edu/show_bug.cgi?id=1730 http://bugs.cgd.ucar.edu/show_bug.cgi?id=1934
shell_scripts/ CESM1_2_1_setup_hybrid	functionally identical to CESM1_1_1_setup_hybrid but is appropriate for the the CESM 1_2_1 release, which supports both CLM 4 and CLM 4.5.
shell_scripts/ CESM_DART_config	creates a CESM case with the bits and pieces required to run DART. When either CESM1_?_1_setup_pmo or CESM1_?_1_setup_hybrid gets executed, CESM_DART_config gets copied to the CESM "CASEROOT" directory. It is designed such that you can execute it at any time during a CESM experiment. When you do execute it, it will build the DART executables and copy them into the CESM "bld" directory, stage the run-time configurable input.nml in the "CASEROOT" directory, etc. and also *modifies* the CESM case.run script to call the DART scripts for assimilation or to harvest synthetic observations.

In addition to the script above, there are a couple scripts that will either perform an assimilation (assimilate.csh) or harvest observations for a perfect model experiment (perfect_model.csh). These scripts are designed to work on several compute platforms although they require configuration, mainly to indicate the location of the DART observation sequence files on your system.

6.112 PROGRAM `clm_to_dart`

6.112.1 Overview

`clm_to_dart` is used to replace indeterminate values in the CLM restart file with the *_FillValue* value specified by the variable attribute. This is necessary to ensure that all ensemble members have the proper *_FillValue* applied to any variable that may be part of the DART state. Only CLM restart files have variables that require this treatment.

`clm_to_dart` reads every variable in the input file and determines if the variable has snow layers by looking for a *levtot* or *levsno* dimension in the variable. The *SNLSNO* variable is used to detemine which layers are unused. However, when there is only a trace of snow, *SNLSNO* may indicate there are no snow layers in use and yet still have some valid values in the snow layer closest to the ground. This situation is confirmed by checking for non-zero values of snowcover fraction (*frac_sno*) as it seems to be a conservative predictor of the presence of snow. Therefore, in general, the indeterminate value for unused snow layer is preserved during the assimilation step, **unless there is a trace of snow** in which case it may be adjusted.

See the *Discussion of Indeterminate Values* section for an example.

6.112.2 Usage

The issue arises because all CLM columns have a fixed number of snow layers but only some of the layers may actually contain snow. CLM internally tracks the number of active snow layers in the **SNLSNO** variable and the indeterminate values in the unused snow layers are of no consequence. To prevent these indeterminate values from being operated on by `filter`, DART requires these indeterminate values to be replaced by the DART **MISSING_R8** code, which is done automatically if the variable has a *_FillValue*.

`clm_to_dart` **overwrites** the input file. Consequently, the intended use is to **copy** the CLM restart file and feed the copy into `clm_to_dart`. This strategy allows the copies to be read into DART and updated with the posterior values. The companion *PROGRAM dart_to_clm* uses the *_FillValue* values as a mask to only update the original CLM restart file with valid posterior values. Where the DART posterior file has *_FillValue* values, the original CLM values are left unchanged.

The *SNLSNO* variable is required. Any file that does not have *SNLSNO* will be unchanged and the program will report the error and terminate. This is why only CLM restart files need the treatment. CLM history files do not have *SNLSNO*.

It is a useful exercise to dump the CLM restart file and the output of `clm_to_dart` and compare the results (for select variables). You have to use your own (low-resolution!) restart file.

```
cp T_SOISNO clm5_tag34.clm2.r.0003-01-01-00000.nc clm_restart.nc
ncdump -f F -v T_SOISNO clm_restart.nc > T_SOISNO_org.txt
./clm_to_dart
ncdump -f F -v T_SOISNO clm_restart.nc > T_SOISNO_new.txt
diff T_SOISNO_org.txt T_SOISNO_new.txt
```

6.112.3 Namelist

Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. These are the defaults:

```
&clm_to_dart_nml
  clm_restart_file  = 'clm_restart.nc'
  verbose           = 0
 /
```

Item	Type	Description
clm_restart_file	character(len=256)	Path name of the CLM restart file to be overwritten.
verbose	integer	Flag to control how much run-time output is created. 0 is very little output. 1 reports which variables are being updated. 2 reports all 2d variables with columns as one dimension. 3 reports all variables and columns which contain indeterminate values *Warning*, 3 can generate a lot of output.

6.112.4 Modules used

```
assimilation_code/modules/utilities/netcdf_utilities_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
```

6.112.5 Files

- clm_restart_file, specified by namelist;
- namelistfile; `input.nml`

Discussion of Indeterminate Values

To explore the most robust way to replace the indeterminate values with the DART-required *_FillValue*, a low-resolution run (10 deg x 15 deg) was performed using the **f10_f10_musgs** tag on the **release-clm5.0.34** branch. The `clm_to_dart_nml` verbosity was set to 3 and the `clm_to_dart` run-time output was captured and is summarized below. The output has been pruned to improve clarity and the column headers have been added. Only the output for the *T_SOISNO* variable is shown as it clearly illustrates the situation.

The **frac_sno** variable (snow cover fraction) is used to determine when there is a reliable trace amount of snow and the original values in the snow layer closest to the ground are left alone. This was determined to be a more useful indicator than **H2OSNO**, as demonstrated in the example below. Please note that if the **frac_sno** > 0 this indicates a trace of snow exists for that column/layer, and the indeterminate value for *T_SOISNO* is left alone. On the other hand, if **frac_sno** < 0 this indicates that particular column/layer is truly empty, and the indeterminate value will be overwritten with the *_FillValue* attribute value, such that DART will not adjust it.

Note: The program clm_to_dart identifies indeterminate values for any variable that contains snow layers. These variables are identified if they have dimensions such as *column*, *levtot* or *levsno*. These include variables related to water mass (e.g. **H2OSOI_ICE**, **H2OSOI_LIQ**), thickness (e.g. **DZSNO**, **ZSNO**, **ZISNO**) and other snow layer characteristics that influence albedo (e.g. **Snw_rds**, **Mss_bcpho**, **Mss_dst1**).

In the example below, the execution of `clm_to_dart` will overwrite the indeterminate values within the clm restart file with the *_FillValue* attribute value to prevent DART from performing an update to that value.

```
-------------------------------------
Starting ... at YYYY MM DD HH MM SS =
                2021  6 14 16 57  4
Program clm_to_dart
-------------------------------------

 set_nml_output Echo NML values to log file only
minval of SNLSNO is          -10
There are          520  variables in the file.
 inquire variable number          204 varname T_SOISNO dimensions are levtot column
 clm_to_dart  variable #          204  is "T_SOISNO" - replacing indeterminate values

VARIABLE  LEVEL  COLUMN   VALUE               H2OSNO                    frac_sno        ␣
↪          SNOW_DEPTH
T_SOISNO  12        488   273.14999999999998  0.0000000000000000        0.
↪000000000000000        0.0000000000000000
T_SOISNO  12        503   273.14999999999998  0.0000000000000000        0.
↪000000000000000        0.0000000000000000
T_SOISNO  12        522   273.14999999999998  0.0000000000000000        0.
↪000000000000000        0.0000000000000000
T_SOISNO  12        523   273.14999999999998  0.0000000000000000        0.
↪000000000000000        0.0000000000000000
T_SOISNO  12        524   273.14999999999998  0.0000000000000000        0.
↪000000000000000        0.0000000000000000
```

(continues on next page)

(continued from previous page)

```
T_SOISNO  12        525   273.14999999999998  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        526   273.14999999999998  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        527   273.14999999999998  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        528   273.14999999999998  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        533   273.04618842687802  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        538   273.14999999999998  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        574   272.77291836570703  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        601   271.54492290326198  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        603   271.54645412989498  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        607   271.57732177585501  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        628   272.98586559496499  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        762   272.45467200034398  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        763   271.34254388393202  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12        769   271.34932534583101  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2386   273.04040468859000  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2479   273.14991153505002  1.3694861766505699       0.
↪11795998521670301        4.6284512959401197E-002
T_SOISNO  12       2760   273.08583001204897  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2761   273.04455346669300  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2762   272.91332916458902  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2763   273.12159885062198  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2768   272.89811384494902  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2769   273.04499957659402  0.0000000000000000        0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2885   273.14979133425902  6.6918688785517395E-007   6.
↪6918687813277700E-008  0.63344272495211695
T_SOISNO  12       2887   273.14964165253900  2.2492550993511499E-020   0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2889   272.53294466428503  1.5189620271589999E-020   0.
↪0000000000000000        0.0000000000000000
T_SOISNO  12       2891   273.14945332076400  5.4222364686270699E-020   0.
↪0000000000000000        0.0000000000000000
```

(continues on next page)

(continued from previous page)

```
T_SOISNO  12       2893   273.14980341715801  1.4201683058442200E-020   0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       2895   273.14999999999998  3.5526247972183701E-020   0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       2896   273.14999404812397  9.6873949694950192E-006   9.
↪6873931487717592E-007  0.20872425049864601
T_SOISNO  12       2899   273.14970403389998  6.0580004576503196E-022   0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       2901   273.14957578176399  1.0670595573239900E-021   0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       2903   273.14947638500303  1.1110267474343600E-021   0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       2905   273.14935524945503  2.5542577505086200E-022   0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       2907   273.14958719516397  1.9081446552101299E-022   0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       2997   273.14999999999998  0.53787417421960904       1.
↪2912522851932499E-002   2.1771531678050301E-003
T_SOISNO  12       3146   273.14999999999998  0.0000000000000000        0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       3329   273.14999999999998  0.0000000000000000        0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       3409   273.14999999999998  0.0000000000000000        0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       3410   273.14999999999998  0.0000000000000000        0.
↪0000000000000000         0.0000000000000000
T_SOISNO  12       3411   273.14999999999998  1.0623461506454501E-002   1.
↪0618761027172401E-003  0.10534145620281500
T_SOISNO  12       3412   273.14999999999998  1.0717589342154700E-002   1.
↪0712805155617101E-003  0.10527378255254000
T_SOISNO  12       3413   273.14999999999998  1.1963762509646500E-002   1.
↪1957631326947600E-003  0.11023440903539100

---------------------------------------
Finished ... at YYYY MM DD HH MM SS =
                2021  6 14 16 57  4
Program clm_to_dart
---------------------------------------
```

6.112.6 References

- none, but https://escomp.github.io/ctsm-docs/versions/master/html/tech_note/Snow_Hydrology/CLM50_Tech_Note_Snow_Hydrology.html is very relevant.

6.113 PROGRAM dart_to_clm

6.113.1 Overview

dart_to_clm replaces the contents of a CLM restart file with the posterior values from DART. Only variables with posterior values are updated. The *_FillValue* in the DART posterior is used as a mask when replacing the value. The original CLM restart file values are maintained when the DART posterior value is *_FillValue*.

6.113.2 Usage

dart_to_clm overwrites the output file. See the companion *PROGRAM clm_to_dart* for a discussion on replacing indeterminate values in the CLM restart snow variables with *_FillValue* so that the DART netCDF read routines handle the special values correctly. Where the DART posterior file has *_FillValue* values, the original CLM restart file is left unchanged, preserving the original values.

dart_to_clm also **includes an option that repartitions the snow water equivalent (SWE)** within the CLM snow layer variables (repartition_swe). This algorithm functions as follows: if during the DART *filter* step an increment is applied to the CLM SWE variable (H2OSNO), this algorithm partitions the SWE increment amongst the active snow layers for each CLM column to maintain the same relative ice and liquid distribution amongst the CLM ice and liquid variables (H2OSOI_ICE and H2OSOI_LIQ). Then, based upon the change in snow layer ice mass, the algorithm updates the snow layer dimensions (SNOW_DEPTH, DZSNO, ZSNO and ZISNO) assuming the prior snow density remains the same.

This repartitioning approach has two main advantages over the traditional *filter* update applied by DART:

1. It is not required that a given snow layer exists amongst all CLM ensemble members in order for the snow layer to be adjusted. Because the CLM snow algorithm splits and aggregates snow layers based upon the vertical thickness in each layer, the uncertainty in the CAM reanalysis (ie. snowfall, melting of snow) may lead to CLM members with different number of snow layers. The default version of DART must ignore these snow layers and no update is applied. When the repartition_swe algorithm is turned on a snow layer which is missing in a subset of ensemble members can still be updated. The only requirement is that each ensemble member includes at least one active snow layer for a CLM column.

2. The repartitioning algorithm guarantees the snow layer adjustments (both mass and dimensions) are consistent with the update to the SWE clm variable (H2OSNO). In other words, the sum of the ice/liquid mass for all the vertical snow layers is identical to the increment of the total column ice/liquid mass. In the default implementation of DART there is no consistency between the total column update of snow and the update of the snow layer variables, because the snow layer adjustment is based upon the regression between the expected observation and each snow layer.

Whether to repartition the snow characteristics (mass, dimensions) across all active snow layers (repartition_swe = 1) or to the bottom snow layer only (repartition_swe = 2) depends upon the application. Repartitioning to all snow layers maintains the prior relative mass distribution of ice and liquid which could be important for certain water-limited regimes with influence upon the carbon cycle. On the other hand, repartitioning to all snow layers indirectly influences the albedo characteristics of the snowcover as snow variables such as black carbon and dust are concentrated/diluted depending upon the snow layer mass adjustment. Repartitioning the snow/ice to only the bottom snow layer avoids indirect changes to the snow albedo characteristics because the black carbon and dust of the topmost snow layer (exposed to sun) is conserved. This could be important for applications where snow albedo, suface energy balance, and surface temperature are critical. However, applying the mass adjustment to the bottom snow layer only, does not maintain the same relative mass distribution and given the proximity to the soil, may provide a more immediate impact on soil moisture conditions.

6.113.3 Namelist

Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. These are the defaults:

```
&dart_to_clm_nml
   dart_to_clm_input_file    = 'dart_posterior.nc'
   dart_to_clm_output_file   = 'clm_restart.nc'
   repartition_swe           = 0
   repartition_vhist_file    = 'clm_vector_history.nc'
   repartition_analysis_file = 'dart_posterior_vector.nc'
   verbose                   = 0
   /
```

Item	Type	Description
dart_to_clm_input_file	character(len=256)	Name of the DART posterior (the output of the filter program)
dart_to_clm_output_file	character(len=256)	Name of the CLM restart file to modify.
repartition_swe	integer	Flag to choose snow repartitioning algorithm 0 Snow repartitioning off. Uses default DART update 1 Snow repartitioned to all active snow layers 2 Snow repartitioned to bottom snow layer only
repartition_vhist_file	character(len=256)	Name of the CLM vector history file (prior SWE value) Only required when repartition_swe = 1,2
repartition_analysis_file	character(len=256)	Name of the DART analysis file (posterior SWE value) Only required when repartition_swe = 1,2
verbose	integer	Flag to control how much run-time output is created. 0 is very little output. 1 reports which CLM variables are being updated. 3 if repartition_swe = 1,2 reports on snow layer diagnostic variables during repartitioning process

6.113.4 Variable requirements when snow repartitioning is enabled

When swe_repartition = 1,2 the DART code requires CLM mass and dimensional snow layer variables output in a specific format described in the table below. A list of the CLM variables (with dimensions) are listed in the left most column of the table. The CLM dimensions important for snow variables are as follows. A 'column' is the land unit at which snow properties are defined. The 'levtot' is a vertical dimension of the total number of snow layers (1-12) and soil layers (13-37) within a column. The 'levsno' dimension is the total number of snow layers.

Variable (dimension)	File Type	Description
H2OSNO (column)	clm vector history	The prior snow water equivalent. Needs to be output in vector format within the CLM history file (h2). Must be included in DART state `clm_variables` (`&model_nml`)
H2OSNO (column)	DART analysis stage	The posterior snow water equivalent output from the *filter* analysis stage. This requires that 'analysis' stage be output within the `stages_to_write` (`&filter_nml`)
H2OSOI_ICE (column,levtot)	clm restart	The ice mass in each snow/soil layer. Must be included in DART state `clm_variables` (`&model_nml`)
H2OSOI_LIQ (column,levtot)	clm restart	The liquid mass in each snow/soil layer. Must be included in DART state `clm_variables` (`&model_nml`)
DZSNO (column,levsno)	clm restart	The snow layer thickness. Must be included in DART state `clm_variables` (`&model_nml`)
ZISNO (column,levsno)	clm restart	The top interface depth of each snow layer. Must be included in DART state `clm_variables` (`&model_nml`)
ZSNO (column,levsno)	clm restart	The middle depth of each snow layer. Must be in included in DART state `clm_variables` (`&model_nml`)
SNOW_DEPTH (column)	clm restart	The total snow depth (sum of all layers). Must be included in DART state `clm_variables` (`&model_nml`)

When adjusting snow layer variables within an assimilation, at a minimum, the `clm_variables` within `model_nml` must include the following:

```
clm_variables  = 'H2OSNO',      'QTY_SNOW_WATER',              '0.0', 'NA', 'vector'   ,
↪'NO_COPY_BACK',
                 'SNOW_DEPTH',  'QTY_SNOW_THICKNESS',        '0.0', 'NA', 'restart' ,
↪'UPDATE',
                 'H2OSOI_LIQ',  'QTY_SOIL_LIQUID_WATER',     '0.0', 'NA', 'restart' ,
↪'UPDATE',
                 'H2OSOI_ICE',  'QTY_SOIL_ICE',              '0.0', 'NA', 'restart' ,
↪'UPDATE',
                 'DZSNO',       'QTY_SNOW_THICKNESS',        '0.0', 'NA', 'restart' ,
↪'UPDATE',
                 'ZSNO',        'QTY_SNOW_THICKNESS',        'NA',  'NA', 'restart' ,
↪'UPDATE',
                 'ZISNO',       'QTY_SNOW_THICKNESS',        'NA',  'NA', 'restart' ,
↪'UPDATE',
                 /
```

Note: The H2OSOI_ICE and H2OSOI_LIQ variables include both snow layer and subsurface soil layers. **Only the snow and liquid mass within the snow layers are repartitioned,** whereas the subsurface layers are updated through the

default DART approach based upon the regression relationship between the subsurface layer property and an expected observation.

6.113.5 Modules used

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/location/utilities/location_io_mod.f90
assimilation_code/modules/assimilation/adaptive_inflate_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/io/direct_netcdf_mod.f90
assimilation_code/modules/io/io_filenames_mod.f90
assimilation_code/modules/io/state_structure_mod.f90
assimilation_code/modules/io/state_vector_io_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/distributed_state_mod.f90
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/netcdf_utilities_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/null_win_mod.f90
assimilation_code/modules/utilities/options_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
models/clm/model_mod.f90
models/utilities/default_model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/forward_operators/obs_def_utilities_mod.f90
```

6.113.6 Files

- `input.nml` is used for `dart_to_clm`
- `dart_posterior.nc` is one of the netCDF files output from the *filter* program.
- `clm_restart.nc` is the CLM generated netCDF file that is modified.
- `clm_vector_history.nc` is the CLM generated netCDF file that provides the prior estimate of snow water equivalent (SWE)
- `dart_posterior_vector.nc` is the DART generated analysis stage file that provide posterior estimate of SWE
- `dart_log.out` list directed output from the `dart_to_clm`.

6.113.7 References

none

6.114 CM1

6.114.1 Overview

Cloud Model 1 (CM1) version 18 (CM1r18) is compatible with the DART. CM1 is a non-hydrostatic numerical model in Cartesian 3D coordinates designed for the study of micro-to-mesoscale atmospheric phenomena in idealized to semi-idealized simulations.

The CM1 model was developed and is maintained by George Bryan at the National Center for Atmospheric Research (NCAR) Mesoscale and Microscale Meteorology Laboratory (MMM).

The model code is freely available from the CM1 website and must be downloaded and compiled outside of DART.

This model interface and scripting support were created by Luke Madaus. **Thanks Luke!**

6.114.2 namelist.input

Several modifications to the CM1 namelist `namelist.input` are required to produce model output compatible with DART. The values are described here and an example is shown below.

The `namelist.input` file is partitioned into several distinct namelists. These namelists are denoted `&param0`, `&param1`, `&param2`, ... `&param13`.

These namelists start with an ampersand & and terminate with a slash `/`. Thus, character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

Using CM1 output files as a prior ensemble state in DART requires each ensemble member to produce a restart file in netCDF format (which requires setting `restart_format=2` in the `&param9` namelist) and these restart files must only contain output at the analysis time (which requires setting `restart_filetype=2` in the `&param9` namelist).

Here is an example configuration of the `&param9` namelist in `namelist.input`:

```
&param9
  restart_format     = 2         restart needs to be netCDF
  restart_filetype   = 2         restart must be the analysis time - ONLY
  restart_file_theta = .true.    make sure theta is in restart file
  restart_use_theta  = .true.
/
```

Important: The only required state variable to be updated is potential temperature (`theta`). Thus two additional settings in the `&param9` namelist – `restart_file_theta = .true.` and `restart_use_theta = .true.` must be set to ensure `theta` is output the CM1 restart files.

Additional state variables that have been tested within DART include:

`ua, va, wa, prs, u0, v0, u10, v10, t2, th2, tsk, q2, psfc, qv, qc, qr, qi qs, qg, nc, nr, ns, ni, nh, ng`.

At present, observation times are evaluated relative to the date and time specified in the `&param11` namelist.

Observation locations are specified in meters relative to the domain origin as defined the `iorigin` setting of `&param2`.

6.114.3 About Testing CM1 and DART

There are two sets of scripts in the `shell_scripts` directory. Luke contributed a set written in python, and the DART team had a set written in csh. The csh scripts have not been tested in quite some time, so use with the understanding that they will need work. Those csh scripts and some unfinished python scripts reside in a `shell_scripts/unfinished` directory and should be used with the understanding that they require effort on the part of the user before the scripts will actually work.

6.114.4 Strategy and Instructions for Using the Python Scripts

A List of Prerequisites

1. CM1 is required to use netCDF restart files.
2. A collection of CM1 model states for initial conditions will be available.
3. There is a separate observation sequence file for each assimilation time.
4. The DART *input.nml* file has some required values as defined below.
5. Each time CM1 is advanced, it will start from the same filename, and the restart number in that filename will be 000001 - ALWAYS. That filename will be a link to the most current model state.

Testing a Cycling Experiment

The big picture: three scripts (`setup_filter.py`, `run_filter.py`, and `advance_ensemble.py`) are alternated to configure an experiment, perform an assimilation on a set of restart files, and make the ensemble forecast. Time management is controlled through command-line arguments.

It is required that you have generated the DART executables before you test. The term `{centraldir}` refers to a filesystem and directory that will be used to run the experiment, the working directory. `{centraldir}` should have a lot of capacity, as ensemble data assimilation will require lots of disk. The term `{dart_dir}` will refer to the location of the DART source code.

The data referenced in the directories (the initial ensemble, etc.) are provided as a compressed tar file cm1r18_3member_example_data.tar.gz.

You will have to download the tar file, uncompress it, and modify the scripts to use these directories instead of the example directories in the scripts. You will also have to compile your own cm1 executable.

1. Set some variables in both `shell_scripts/setup_filter.py` and `shell_scripts/advance_ensemble.py` as described below.
2. In the `{dart_dir}/models/cm1/shell_scripts` directory, run:

```
$ ./setup_filter.py -d YYYYmmDDHHMMSS -i
```

   where `YYYYmmDDHHMMSS` is the date and time of the first assimilation cycle (the `-i` option indicates this is the initial setup and extra work will be performed). This will create the working directory `{centraldir}`, link in required executables, copy in the initial conditions for each member from some predetermined location, copy in the observation sequence file for this assimilation time from some predetermined location, modify namelists, and build a queue submission script in the `{centraldir}`: `run_filter.py`.

3. Change into `{centraldir}` and verify the contents of `run_filter.py`. Ensure the assimilation settings in `input.nml` are correct. Once you are satisfied, submit `run_filter.py` to the queue to perform an assimilation.

4. After the assimilation job completes, check to be sure that the assimilation completed successfully, and the archived files requested in the `setup_filter.py` `files_to_archive` variable are in `{centraldir}/archive/YYYYmmDDHHMMSS`.

5. Change into `{dart_dir}/models/cm1/shell_scripts` and advance the ensemble to the next assimilation time by running:

```
$ ./advance_ensemble.py -d YYYYmmDDHHMMSS -l nnnn
```

   where `YYYYmmDDHHMMSS` is the date of the COMPLETED analysis (the start time for the model) and `nnnn` is the length of model integration in seconds (the forecast length). (The forecast length option is specified by 'hypen ell' - the lowercase letter L, not the number one.) `advance_ensemble.py` will submit jobs to the queue to advance the ensemble.

6. After all ensemble members have successfully completed, run:

```
$ ./setup_filter.py -d YYYYmmDDHHMMSS
```

   where $YYYYmmDDHHMMSS$ is the **new** current analysis time. Note the $-i$ flag is NOT used here, as we do not need to (should not need to!) re-initialize the entire directory structure.

7. Change into `{centraldir}` and run:

```
$ ``run_filter.py``
```

   to perform the assimilation.

8. Go back to step 4 and repeat steps 4-7 for each assimilation cycle until the end of the experiment.

Within the `setup_filter.py` and `advance_ensemble.py` scripts, the following variables need to be set between the "BEGIN USER-DEFINED VARIABLES" and "END USER-DEFINED VARIABLES" comment blocks:

`jobname`

A name for this experiment, will be included in the working directory path.

`ens_size`

Number of ensemble members.

`restart_filename`

The filename for each ensemble member's restart. Highly recommended to leave this as `cm1out_rst_000001.nc`

`window_mins`

The assimilation window width (in minutes) for each assimilation cycle.

`copy`

The copy command with desired flags for this system.

`link`

The link command with desired flags for this system.

`remove`

The remove command with desired flags for this system.

`files_to_archive`

A list of DART output files to archive for each assimilation cycle. Note that any inflation files generated are automatically carried over.

centraldir

Directory (which will be created if setup_filter.py is run in intialization mode) where the assimilation and model advances will take place. Should be on a system with enough space to allow for several assimilation cycles of archived output.

dart_dir

Path to the cm1 subdirectory of DART.

cm1_dir

Path to the cm1 model executable (*cm1.exe*)

icdir

Path to the ensemble of initial conditions. It is assumed that within this directory, each ensemble member has a subdirectory (*m1*, *m2*, *m3*, ...) that contains:

- a restart file for cm1 at the desired start time and having the filename defined in restart_filename above
- a namelist.input file compatible with the generation of that restart file.

obsdir

Path to a directory containing observation sequence files to be assimilated. It is assumed that the observation sequence files are named following the convention YYYYmmDDHHMMSS_obs_seq.prior, where the date of the analysis time whose observations are contained in that file is the first part of the file name.

setup_filter.py and advance_ensemble.py assume that mpi queue submissions are required to run cm1.exe and filter. These variables control how that is handled.

queue_system

The name of the queueing system

mpi_run_command

The command used in a submitted script to execute an mpi task in the queue, including any required flags

queue_sub_command

The command used to submit a script to the queue

job_sub_info

A dictionary of all flags required to execute a job in the queue, with the key being the flag and the value being the variable. e.g. {'-P' : 'PROJECT CODE HERE', '-W' : '00:20'}, etc.

6.114.5 Namelist

The &model_nml namelist is read from the input.nml file. Again, namelists start with an ampersand & and terminate with a slash /. Character strings that contain a / must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   assimilation_period_days     = 0
   assimilation_period_seconds  = 21600
   model_perturbation_amplitude = 0.2
   cm1_template_file            = 'null'
   calendar                     = 'Gregorian'
```

(continues on next page)

(continued from previous page)

```
   periodic_x                    = .true.
   periodic_y                    = .true.
   periodic_z                    = .false.
   debug                         = 0
   model_variables               = ' '
/
```

Description of each namelist entry

Item	Type	Description
assimilation_period_[days,seconds]	integer	This specifies the width of the assimilation window. The current model time is used as the center time of the assimilation window. All observations in the assimilation window are assimilated. BEWARE: if you put observations that occur before the beginning of the assimilation_period, DART will error out because it cannot move the model 'back in time' to process these observations.
model_perturbation_amplitude	real(r8)	unsupported
cm1_template_file	character(len=256)	filename used to read the variable sizes, location metadata, etc.
calendar	character(len=256)	Character string to specify the calendar in use. Usually 'Gregorian' (since that is what the observations use).
model_variables	character(:,5)	Strings that identify the CM1 variables, their DART quantity, the minimum & maximum possible values, and whether or not the posterior values should be written to the output file. The DART QUANTITY must be one found in the *DART/obs_kind/obs_kind_mod.f90* AFTER it gets built by *preprocess*. *model_variables(:,1)* specifies the CM1 variable name in the netCDF file. *model_variables(:,2)* the DART quantity for that variable. *model_variables(:,3)* minimum bound (if any) for that variable. *model_variables(:,4)* maximum bound (if any) for that variable. *model_variables(:,5)* if the variable should be updated in the restart file. The value may be "UPDATE" or anything else.
periodic_x	logical	a value of *.true.* means the 'X' dimension is periodic.
periodic_y	logical	a value of *.true.* means the 'Y' dimension is periodic.
periodic_z	logical	unsupported
debug	integer	switch to control the amount of run-time output is produced. Higher values produce more output. 0 produces the least.

Note: The values above are the default values. A more realistic example is shown below and closely matches the values in the default `input.nml`. The example input block is for a case run using the Morrison Microphysics scheme. Any changes in microphysics will require the user to update the hydrometeor state variables.

```
&model_nml
   assimilation_period_days     = 0
   assimilation_period_seconds  = 60
   cm1_template_file            = 'cm1out_rst_000001.nc'
   calendar                     = 'Gregorian'
   periodic_x                   = .true.
   periodic_y                   = .true.
   periodic_z                   = .false.
   debug                        = 0
   model_variables = 'ua'   , 'QTY_U_WIND_COMPONENT'      , 'NULL', 'NULL', 'UPDATE',
                     'va'   , 'QTY_V_WIND_COMPONENT'      , 'NULL', 'NULL', 'UPDATE',
                     'wa'   , 'QTY_VERTICAL_VELOCITY'     , 'NULL', 'NULL', 'UPDATE',
                     'theta', 'QTY_POTENTIAL_TEMPERATURE' , 0.0000, 'NULL', 'UPDATE',
                     'prs'  , 'QTY_PRESSURE'              , 'NULL', 'NULL', 'UPDATE',
                     'qv'   , 'QTY_VAPOR_MIXING_RATIO'      , 0.0000, 'NULL', 'UPDATE',
                     'qc'   , 'QTY_CLOUD_LIQUID_WATER'      , 0.0000, 'NULL', 'UPDATE',
                     'qr'   , 'QTY_RAINWATER_MIXING_RATIO'  , 0.0000, 'NULL', 'UPDATE',
                     'qi'   , 'QTY_CLOUD_ICE'               , 0.0000, 'NULL', 'UPDATE',
                     'qs'   , 'QTY_SNOW_MIXING_RATIO'       , 0.0000, 'NULL', 'UPDATE',
                     'qg'   , 'QTY_GRAUPEL_MIXING_RATIO'    , 0.0000, 'NULL', 'UPDATE',
                    'ncr'   , 'QTY_RAIN_NUMBER_CONCENTR'    , 0.0000, 'NULL', 'UPDATE',
                    'nci'   , 'QTY_ICE_NUMBER_CONCENTRATION', 0.0000, 'NULL', 'UPDATE',
                    'ncs'   , 'QTY_SNOW_NUMBER_CONCENTR'    , 0.0000, 'NULL', 'UPDATE',
                    'ncg'   , 'QTY_GRAUPEL_NUMBER_CONCENTR' , 0.0000, 'NULL', 'UPDATE',
                    'rho'   , 'QTY_DENSITY'                 , 0.0000, 'NULL', 'UPDATE',
                    'dbz'   , 'QTY_RADAR_REFLECTIVITY'      , 0.0000, 'NULL', 'UPDATE',


/
```

Note: From Jon Labriola on additional model variables that could be considered:

There are other model variables that can be included if you use a very specific forecast configuration. The following model variables output by CM1 when you run with a simulation with a surface model and set "output_sfcdiags = 1". These variables can be used to simulated surface observations including TEMPERATURE_2M, U_WIND_10, V_WIND_10, SPECIFIC_HUMIDITY_2M, and SURFACE_PRESSURE.

```
'u10'  , 'QTY_10M_U_WIND_COMPONENT'  , 'NULL', 'NULL', 'UPDATE',
'v10'  , 'QTY_10M_V_WIND_COMPONENT'  , 'NULL', 'NULL', 'UPDATE',
't2'   , 'QTY_2M_TEMPERATURE'        , 0.0000, 'NULL', 'UPDATE',
'th2'  , 'QTY_POTENTIAL_TEMPERATURE' , 0.0000, 'NULL', 'UPDATE',
'tsk'  , 'QTY_SURFACE_TEMPERATURE'   , 0.0000, 'NULL', 'UPDATE',
'q2'   , 'QTY_SPECIFIC_HUMIDITY'     , 0.0000, 'NULL', 'UPDATE',
'psfc' , 'QTY_SURFACE_PRESSURE'      , 0.0000, 'NULL', 'UPDATE',
```

If you want to assimilate pseudo "near-surface" observations but are not using a surface model in your forecast, I recommend defining a single radionsonde or dropsonde observation that is located near the surface (but at or above the lowest model level. The DART forward operator will perform 3D interpolation to obtain the near-surface observation.

Also be warned - from what I can tell DART forward operators are unable to calculate air temperature and specific humidity from CM1 model fields. To simulate these fields (e.g., DROPSONDE_TEMPERATURE, RADIOSONDE_SPECIFIC_HUMIDITY,...) you will have to manually go into the CM1 code (./src/writeout_nc.F and ./src/restart.F) and update the model to output 3D air temperature (air_temp) and specific humidity (spec_hum) fields. Then you can add the following fields to model_variables.

```
'air_temp'   , 'QTY_TEMPERATURE'              , 'NULL', 'NULL', 'UPDATE',
'spec_hum'   , 'QTY_SPECIFIC_HUMIDITY'        , 0.0000, 'NULL', 'UPDATE',
```

Some other areas of interest in input.nml will also be discussed

&assim_tools_nml

The CM1 DART code was updated so that horizontal and vertical localization radii can be defined for each assimilated observation type. The horizontal and vertical localization radius is first defined by the `cutoff` variable in the &assim_tools_nml code block of input.nml. However, you can use `special_localization_obs_types` and `special_localization_cutoffs` variables to define the observation type and corresponding half localization radius, respectively.

`horizontal localization radius = 2 * special_localization_cutoffs[ob_type]`

An example of the &assim_tools_nml using per-type radii is provided below.

```
&assim_tools_nml
  adaptive_localization_threshold = -1
  cutoff                          = 15000.0
  filter_kind                     = 1
  print_every_nth_obs             = 100
  rectangular_quadrature          = .true.
  sampling_error_correction       = .false.
  sort_obs_inc                    = .false.
  spread_restoration              = .false.
  gaussian_likelihood_tails       = .false.
  distribute_mean                 = .true.
  output_localization_diagnostics = .false.
  localization_diagnostics_file   = 'localization_diagnostics'
  special_localization_obs_types    = 'RADIOSONDE_U_WIND_COMPONENT','RADIOSONDE_V_WIND_
↪COMPONENT', 'RADAR_REFLECTIVITY', 'DOPPLER_RADIAL_VELOCITY'
  special_localization_cutoffs      = 50000., 50000., 6000., 6000.  ! Horizontal␣
↪Localization radii (100km, 100km, 12km, 12km)
/
```

&location_nml

You can also define the vertical localization radius for each observation type. `special_vert_normalization_obs_types` is where you define the observation type and `special_vert_normalization_heights` is where you define the normalization factor in the vertical. If you just one set one normalization for all observation types use `vert_normalization_height` (not typically recommended since horizontal localization radii often switch between obs types).

`vertical localization radius = 2 * special_localization_cutoffs[ob_type]*`
`special_vert_normalization_heights[ob_type]`

or if no obs types are defined...

`vertical localization radius = 2 * cutoff * vert_normalization_height`

You can also define periodic boundary conditions in the &location_nml code block. Do not change `nx`, `ny`, and `nz` they are defined search radii used by DART (not the domain size itself).

An example of the &location_nml is provided below.

```
&location_nml
   x_is_periodic       = .false.
   y_is_periodic       = .false.
   z_is_periodic       = .false.
   min_x_for_periodic  = 0.0
   max_x_for_periodic  = 200000.0
   min_y_for_periodic  = 0.0
   max_y_for_periodic  = 200000.0
   min_z_for_periodic  = 0.0
   max_z_for_periodic  = 1.0
   compare_to_correct  = .false.
   output_box_info     = .false.
   print_box_level     = 0
   nx                  = 10
   ny                  = 10
   nz                  = 10
   vert_normalization_height = 1
   special_vert_normalization_obs_types = 'RADIOSONDE_U_WIND_COMPONENT','RADIOSONDE_V_
↪WIND_COMPONENT', 'RADAR_REFLECTIVITY', 'DOPPLER_RADIAL_VELOCITY'
   special_vert_normalization_heights   = 0.04, 0.04, 0.33, 0.33
/
```

&obs_def_radar_mod_nml

The block defines how radar observations are assimilated e.g., what forward operator is used for reflectivity, radial velocity, microphysical information, observation cutoff values.

Rather than diagnosing radar reflectivity directly from model output I feed the DART system the 3D radar reflectivity variable output by cm1. If you decide to follow this same technique set `microphysics_type = 5`.

To calculate radar radial velocity you need to set `allow_dbztowt_conv=.true.` this will allow the DART to diagnose particle fall speed via reflecvtivity which is used for the radial velocity calculation.

A sample &obs_def_radar_mod_nml is provided below:

```
&obs_def_radar_mod_nml
   apply_ref_limit_to_obs      =   .false.,
   reflectivity_limit_obs      =     -10.0,
   lowest_reflectivity_obs     =     -10.0,
   apply_ref_limit_to_fwd_op   =   .false.,
   reflectivity_limit_fwd_op   =     -10.0,
   lowest_reflectivity_fwd_op  =     -10.0,
   max_radial_vel_obs          =   1000000,
   allow_wet_graupel           =   .false.,
   microphysics_type           =         5,
   allow_dbztowt_conv          =    .true.,
   dielectric_factor           =     0.224,
   n0_rain                     =     8.0e6,
```

(continues on next page)

(continued from previous page)

```
   n0_graupel                  =     4.0e6,
   n0_snow                     =     3.0e6,
   rho_rain                    =    1000.0,
   rho_graupel                 =     400.0,
   rho_snow                    =     100.0
   /
```

6.115 COAMPS Nest

Attention: `COAMPS_NEST` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `COAMPS_NEST` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.115.1 Overview

An updated version of the COAMPS model interfaces and scripts.

This interface was contributed by Alex Reinecke of the Naval Research Lab-Monterey.

The primary differences from the original COAMPS model code are:

- the ability to assimilate nested domains
- assimilates real observations
- a simplified way to specify the state vector
- I/O COAMPS data files
- extensive script updates to accommodate additional HPC environments

6.116 COAMPS

Attention: `COAMPS` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `COAMPS` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.116.1 Overview

DART interface module for the Coupled Ocean / Atmosphere Mesoscale Prediction (COAMPS ®) model. The 16 public interfaces listed here are standardized for all DART compliant models. These interfaces allow DART to advance the model, get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for a variety of variables required in observational operators.
The following model description is taken from the COAMPS overview web page:

> "The Coupled Ocean/Atmosphere Mesoscale Prediction System (COAMPS) has been developed by the Marine Meteorology Division (MMD) of the Naval Research Laboratory (NRL). The atmospheric com-

ponents of COAMPS, described below, are used operationally by the U.S. Navy for short-term numerical weather prediction for various regions around the world.

The atmospheric portion of COAMPS represents a complete three-dimensional data assimilation system comprised of data quality control, analysis, initialization, and forecast model components. Features include a globally relocatable grid, user-defined grid resolutions and dimensions, nested grids, an option for idealized or real-time simulations, and code that allows for portability between mainframes and workstations. The nonhydrostatic atmospheric model includes predictive equations for the momentum, the nondimensional pressure perturbation, the potential temperature, the turbulent kinetic energy, and the mixing ratios of water vapor, clouds, rain, ice, grauple, and snow, and contains advanced parameterizations for boundary layer processes, precipitation, and radiation.

The distributed version of the COAMPS code that can be downloaded from the web site has been designed to use the message-passing interface (MPI), OpenMP directives, and horizontal domain decomposition to achieve parallelism. The code is capable of executing efficiently across vector, parallel, or symmetric muti-processor (SMP) machines by simply changing run-time options."

6.116.2 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
obs_kind_mod
random_seq_mod
netcdf
typesizes
coamps_grid_mod
coamps_interp_mod
coamps_restart_mod
coamps_util_mod
```

6.116.3 Public interfaces

use model_mod, only :	get_model_size
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model
	adv_1step
	end_model
	init_time
	init_conditions

The last 4 interfaces are only required for low-order models where advancing the model can be done by a call to a subroutine. The COAMPS model only advances by executing the coamps program. Thus the last 4 interfaces only appear as stubs in this module.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector as an integer. This includes all nested domains.

`model_size`	The length of the model state vector.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Returns metadata about a given element, indexed by index_in, in the model state vector. The location defines where the state variable is located while the type of the variable (for instance temperature, or u wind component) is returned by var_type. The integer values used to indicate different variable types in var_type are themselves defined as public interfaces to model_mod if required.

index_in	Index of state vector element about which information is requested.
location	Returns location of indexed state variable. The location should use a location_mod that is appropriate for the model domain. For realistic atmospheric models, for instance, a three-dimensional spherical location module that can represent height in a variety of ways is provided.
var_type	Returns the type of the indexed state variable as an optional argument.

call model_interpolate(x, location, obs_kind, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                  intent(in)  ::  obs_kind
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given model state, returns the value of observation type interpolated to a given location by a method of the model's choosing. All observation kinds defined in obs_kind_mod are supported. In the case where the observational operator is not defined at the given location (e.g. the observation is below the model surface or outside the domain), obs_val is returned as -888888.0 and istatus = 1. Otherwise, istatus = 0. The interpolation is performed in the domain with the highest resolution containing the observation.

x	A model state vector.
location	Location to which to interpolate.
obs_kind	Integer indexing which type of observation is to be interpolated.
obs_val	The interpolated value from the model.
istatus	Integer flag indicating the result of the interpolation.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the model base time step as a time_type. For now this is set to 1 minute.

var	Smallest time step of model.

call static_init_model()

Used for runtime initialization of the model. This is the first call made to the model by any DART compliant assimilation routine. It reads the model namelist parameters, initializes the pressure levels for the state vector, and generates the location data for each member of the state.

ierr = nc_write_model_atts(ncFileId)

```
integer             ::  nc_write_model_atts
integer, intent(in) ::  ncFileId
```

Function to write model specific attributes to a netCDF file. At present, DART is using the NetCDF format to output diagnostic information. This is not a requirement, and models could choose to provide output in other formats. This function writes the metadata associated with the model to a NetCDF file opened to a file identified by ncFileID.

ncFileId	Integer file descriptor opened to NetCDF file.
ierr	Returned error code.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                             ::  nc_write_model_vars
integer,                intent(in) ::  ncFileID
real(r8), dimension(:), intent(in) ::  statevec
integer,                intent(in) ::  copyindex
integer,                intent(in) ::  timeindex
```

Writes a copy of the state variables to a NetCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state.

ncFileID	Integer file descriptor opened to NetCDF file.
statevec	State vector.
copyindex	Integer index to which copy is to be written.
timeindex	Integer index of which time in the file is being written.
ierr	Returned error code.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:),   intent(in)    ::  state
real(r8), dimension(:),   intent(out)   ::  pert_state
logical,                  intent(out)   ::  interf_provided
```

Given a model state, produces a perturbed model state. This is used to generate initial ensemble conditions perturbed around some control trajectory state when one is preparing to spin-up ensembles. In the COAMPS interface, this can be done three different ways:

- No perturbation
- Uniform perturbation - each element of the field has the same additive perturbation
- Individual perturbation - each element of the field has a different additive perturbation The perturbation magnitude and option are supplied out of the dynamic restart vector definition - this allows us to supply a variance appropriate for each type of variable at each level.

state	State vector to be perturbed.
pert_state	Perturbed state vector is returned.
interf_provided	Returns .true. for this model.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
```

(continues on next page)

(continued from previous page)

```
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Pass-through to the 3-D sphere locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in)  :: ens_mean
```

A local copy is available here for use during other computations in the model_mod code.

ens_mean	Ensemble mean state vector

call adv_1step(x, time)

```
real(r8), dimension(:),   intent(inout) ::  x
type(time_type),          intent(in)    ::  time
```

This operation is not defined for the COAMPS model. This interface is only required if `synchronous' model state advance is supported (the model is called directly as a Fortran90 subroutine from the assimilation programs). This is generally not the preferred method for large models and a stub for this interface is provided for the COAMPS model.

x	State vector of length model_size.
time	Gives time of the initial model state. Needed for models that have real time state requirements, for instance the computation of radiational parameters. Note that DART provides a time_manager_mod module that is used to support time computations throughout the facility.

call end_model( )

Called when use of a model is completed to clean up storage, etc. A stub is provided for the COAMPS model.

call init_time(i_time)

```
type(time_type),        intent(in)  ::  i_time
```

Returns the time at which the model will start if no input initial conditions are to be used. This is frequently used to spin-up models from rest, but is not meaningfully supported for the COAMPS model.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) ::  x
```

Returns default initial conditions for model; generally used for spinning up initial model states. For the COAMPS model just return 0's since initial state is always to be provided from input files.

x	Model state vector.

6.116.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
  cdtg = '2006072500',
  y_bound_skip = 3,
  x_bound_skip = 3,
  need_mean = .false.,
/
```

Item	Type	Description
cdtg	character(len=10)	Date/time group.
x_bound_skip, y_bound_skip	integer	Number of x and y boundary points to skip when perturbing the model state.
need_mean	logical	Does the forward operator computation need the ensemble mean?

6.116.5 Files

filename	purpose
input.nml	to read the model_mod namelist
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.116.6 References

The COAMPS registration web site is http://www.nrlmry.navy.mil/coamps-web/web/home and COAMPS is a registered trademark of the Naval Research Laboratory.

6.116.7 Private components

N/A

6.117 ECHAM

6.117.1 Overview

ECHAM is the atmospheric general circulation component of the Max Planck Institute Earth System Model (MPI-ESM). It was originally branched from the numerical weather prediction model developed by the European Centre for Medium-Range Weather Forecasts (ECMWF) in the late 1980's and is developed and supported by the Max Planck Institute for Meteorology in Hamburg, Germany. Thus the ECHAM acronym is comprised of EC from ECMWF, H for Hamburg and AM for atmospheric model.

There are several DART users who have working DART interface code to ECHAM. If you are interested in running DART with this model please contact the DART group at dart@ucar.edu for more information. We currently do not have a copy of the model_mod interface code nor any of the scripting required to run an assimilation, but we may be able to put you in contact with the right people to get it.

6.118 FESOM

The Finite Element Sea-ice Ocean Model (FESOM) is an unstructured mesh global ocean model using finite element methods to solve the hydro-static primitive equations with the Boussinesq approximation (Danilov et al., 2004[1]; Wang et al., 2008[2]). FESOM v1.4 is interfaced with DART by Aydoğdu et al. (2018a)[3] using a regional implementation in Turkish Straits System (Gürses et al. 2016[4], Aydoğdu et al. 2018b[5]).

[1] Danilov, S., Kivman, G., and Schröter, J.: A finite-element ocean model: principles and evaluation, Ocean Modell., 6, 125–150, 2004.

[2] Wang, Q., Danilov, S., and Schröter, J.: Finite element ocean circulation model based on triangular prismatic elements, with application in studying the effect of topography representation, J. Geophys. Res.-Oceans (1978–2012), 113, C05015, doi:10.1029/2007JC004482, 2008.

[3] Aydoğdu, A., Hoar, T. J., Vukicevic, T., Anderson, J. L., Pinardi, N., Karspeck, A., Hendricks, J., Collins, N., Macchia, F., and Özsoy, E.: OSSE for a sustainable marine observing network in the Sea of Marmara, Nonlin. Processes Geophys., 25, 537-551, doi:10.5194/npg-25-537-2018, 2018a.

[4] Gürses, Ö., Aydoğdu, A., Pinardi, N., and Özsoy, E.: A finite element modeling study of the Turkish Straits System, in: The Sea of Marmara – Marine Biodiversity, Fisheries, Conservations and Governance, edited by: Özsoy E., Çağatay, M. N., Balkis, N., and Öztürk, B., TUDAV Publication, 169–184, 2016.

[5] Aydoğdu, A., Pinardi, N., Özsoy, E., Danabasoglu, G., Gürses, Ö., and Karspeck, A.: Circulation of the Turkish Straits System under interannual atmospheric forcing, Ocean Sci., 14, 999-1019, doi:10.5194/os-14-999-2018, 2018b.

There is a recent version of the model called the Finite-volumE Sea ice–Ocean Model (FESOM2, Danilov et al. 2017[6]). A version for coastal applications FESOM-C v.2 (Androsov et al., 2019[7]) has also been published.

The FESOM V1.4 source code can be downloaded from https://fesom.de/models/fesom14

The FESOM/DART interfaces, diagnostics and support scripting were contributed by **Ali Aydoğdu**. Thanks Ali!

6.118.1 Overview

model_mod.f90

A module called *fesom_modules* is provided to pass the information from FESOM to DART. fesom_modules.f90 includes fortran routines adopted from FESOM v1.4 to read the mesh, set the variables and dimensions of the arrays. *fesom_modules* should have access to *nod2d.out*, *nod3d.out*, *elem2d.out*, *elem3d.out*, *aux3d.out*, *depth.out* and *m3d.ini* mesh files.

Forward operators use an interpolation using the closest model node in the horizontal, given that the application in Aydoğdu et al. (2018a) uses a very high-resolution mesh. In the vertical, a linear interpolation is performed between two enclosing model layers. Interpolation in model_interpolate routine can be improved, if needed.

Note that because the FESOM-native code explicitly types reals, the DART mechanism of being able to run in reduced precision by defining real(r8) to be the same as real(r4) via 'types_mod.f90' is not supported.

Workflow

1. *environment.load* Must be modified to contain the specifics of an experiment. This file is sourced by every other script below.
2. *experiment.launch* Takes the information from **environment.load** and creates runnable scripts from the template script files. This also initiates the first cycle of the experiment.

   2.1. *ensemble.sh*

   2.1.1. *initialize.template* (first cycle only)

   2.1.2. *advance_model.template* (job array to advance the ensemble)

   2.1.3. *check_ensemble.sh* (if all goes well, assimilate)

   2.1.3.1. *filter.template* (assimilate)

   2.1.3.2. *finalize.sh* if all goes well and experiment is not finished . . .
   continue to 2.1

[6] Danilov, S., Sidorenko, D., Wang, Q., and Jung, T.: The Finite-volumE Sea ice–Ocean Model (FESOM2), Geosci. Model Dev., 10, 765-789, doi:10.5194/gmd-10-765-2017, 2017.

[7] Androsov, A., Fofonova, V., Kuznetsov, I., Danilov, S., Rakowsky, N., Harig, S., Brix, H., and Wiltshire, K. H.: FESOM-C v.2: coastal dynamics on hybrid unstructured meshes, Geosci. Model Dev., 12, 1009-1028, doi:10.5194/gmd-12-1009-2019, 2019.

Shell Scripts

Shell scripts are written in bash for LSF queuing system. They should be modified to work with others such as SLURM. FESOM executables are called externally detached from DART therefore no need for an advance model.

Script	Queue	Definition
environment.load	serial	Includes environment variables, relevant directories, experiment specifications. This file is sourced by every other script below.
experiment.launch	serial	Main script which modifies `ensemble.sh` and calls `ensemble.${EXPINFO}.sh`. An experiment-specific summary which should be modified before launching the scripts.
ensemble.sh	serial	Calls and submits `initialize.template`, `advance_model.template` `check_ensemble.sh` one after the other.
initialize.template	serial	Called only once at the beginning of the experiment. Sets the experiment directory, copies initial ensemble, namelists.
advance_model.template	parallel	Submits a job array for all ensemble members.
check_ensemble.sh	serial	Checks if the forwarding for all members is finished. If so, first calls `filter.template` and then calls `finalize.sh` to conclude current assimilation cycle.
filter.template	parallel	Runs the filter to perform the assimilation.
finalize.sh	serial	Checks if the whole experiment is finished. If so, stops. Otherwise, resubmits `ensemble.${EXPINFO}.sh` for the next assimilation cycle.

Diagnostics

A toolbox for diagnostics is provided. Some are written for a specific regional application using Ferrybox observations of temperature and salinity. However, it shouldn't be difficult to add new tools following the present ones. A fortran toolbox post-processes the FESOM outputs and visualization is done using Generic Mapping Tools (GMT). DART post-processed netCDF outputs are visualized using FERRET. Please see the expanded description inside each source file.

Directory	code file	description
src/		
	fesom_post_main.F90	main fortran routine calling each tool selected in the namelist
	fesom_ocean_mod.F90	ocean diagnostic routines
	fesom_dart_mod.F90	DART diagnostic output routines
	fesom_forcing_mod.F90	forcing diagnostic routines
	fesom_observation_mod.F90	observation diagnostic routines
	gen_input.F90	routines for I/O (adapted from FESOM)
	gen_modules_clock.F90	routines for timing (adapted from FESOM)
	gen_modules_config.F90	routines for configuration (adapted from FESOM)
	mesh_read.F90	routines for reading the mesh (adapted from FESOM)
	Makefile	Makefile (adapted from FESOM) but reads DART environment
	oce_dens_press.F90	routines to compute density and pressure (adapted from FESOM)
	oce_mesh_setup.F90	routines for mesh setup (adapted from FESOM)
	oce_modules.F90	routines for ocean modules (adapted from FESOM)
	random_perturbation.F90	random perturbation to observation sampling
	utilities.F90	various utilities
script/		

continues on next page

Table 3 – continued from previous page

Directory	code file	description
	compute_ensemble_mean	computes ensemble mean and extracts a transect or level
	compute_increment	computes increment using DART diagnostic output
	compute_NR_diff	computes the difference between a nature run and the ensemble prior mean
	dart_obs_seq_diag	DART observation-space statistics from `obs_epoch.nc` and `obs_diag.nc`
	dart.postproc.env	DART environment variables
	fesom.postproc.env	FESOM environment variables
	observe_nature_run	creates synthetic observations from a nature run
	transect_daily_mean	extracts and plots a transect of an individual ensemble member
	zlevel_daily_mean	extracts and plots a level of an individual ensemble member
gmt/		
	plot_ensemble_mean.gmt	plots ensemble mean created by `compute_ensemble_mean`
	plot_increment.gmt	plots increment created by `compute_increment`
	plot_NR_diff.gmt	plots difference created by `compute_NR_diff`
	transect_daily_mean.gmt	plots transects created by `transect_daily_mean`
	zlevel_yearly_mean.gmt	plots levels created by `zlevel_daily_mean`
ferret/		
	frt.obs_diag_TeMPLaTe.jnl	plot DART diags created by `dart_obs_seq_diag`
	frt.obs_epoch_TeMPLaTe.jnl	plot DART diags created by `dart_obs_seq_diag`

6.118.2 References

6.119 GITM

Attention: `GITM` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `GITM` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

A preliminary Manhattan-compliant interface to GITM exists and has been used for science applications but has not yet been released for public use.

6.119.1 Overview

The Global Ionosphere Thermosphere Model (GITM) is a 3-dimensional spherical code that models the Earth's thermosphere and ionosphere system using a stretched grid in latitude and altitude.

The **GITM** interface for **Data Assimilation Research Testbed (DART)** is under development. If you wish to use GITM, you are urged to contact us. The original scripts were configured to run on the University of Michigan machine **NYX** using the Portable Batch System (**PBS**). We have attempted to extend the scripts to work with both PBS and LSF and are only partway through the process.

DART does not come with the GITM code. You need to get that on your own. The normal procedure of building GITM creates some resource files that are subsequently needed by DART - just to compile. These include:

1. `models/gitm/GITM2/src/ModConstants.f90`
2. `models/gitm/GITM2/src/ModEarth.f90`
3. `models/gitm/GITM2/src/ModKind.f90`
4. `models/gitm/GITM2/src/ModOrbital.f90`

5. models/gitm/GITM2/src/ModSize.f90
6. models/gitm/GITM2/src/ModTime.f90
7. models/gitm/GITM2/src/time_routines.f90

GITM uses binary files for their restart mechanisms, so no metadata is available to confirm the number and order of fields in the file. Care must be used to make sure the namelist-controlled set of variables to be included in the DART state vector is consistent with the restart files. Each variable must also correspond to a DART "KIND"; required for the DART interpolate routines.

For example, this configuration of `input.nml` is nowhere close to being correct:

```
&model_nml
   gitm_state_variables = 'Temperature',       'QTY_TEMPERATURE',
                          'eTemperature',      'QTY_TEMPERATURE_ELECTRON',
                          'ITemperature',      'QTY_TEMPERATURE_ION',
                          'iO_3P_NDensityS',   'QTY_DENSITY_NEUTRAL_O3P',
                          'iO2_NDensityS',     'QTY_DENSITY_NEUTRAL_O2',
                          'iN2_NDensityS',     'QTY_DENSITY_NEUTRAL_N2',
                               ...                  ...
/
```

These variables are then adjusted to be consistent with observations and stuffed back into the same netCDF restart files. Since DART is an ensemble algorithm, there are multiple restart files for a single restart time: one for each ensemble member. Creating the initial ensemble of states is an area of active research.

DART reads grid information for GITM from several sources. The `UAM.in` file specifies the number of latitudes/longitudes per block, and the number of blocks comes from the `GITM2/src/ModSize.f90` module. Internal to the DART code, the following variables exist:

Item	Type	Description
LON(:)	real(r8)	longitude array [0, 360)
LAT(:)	real(r8)	latitude array (-90,90)
ALT(:)	real(r8)	altitude array (0,~inf)
NgridLon	integer	the length of the longitude array
NgridLat	integer	the length of the latitude array
NgridAlt	integer	the length of the altitude array

6.119.2 Compiling

GITM has been sucessfully tested with DART using the `gfortran` compiler, version `4.2.3`. The DART components were built with the following `mkmf.template` settings.

```
FC = gfortran
LD = gfortran
NETCDF = /Users/thoar/GNU
INCS = -I${NETCDF}/include
LIBS = -L${NETCDF}/lib -lnetcdf -lcurl -lhdf5_hl -lhdf5 -lz  -lm
FFLAGS = -O0 -fbounds-check -frecord-marker=4 -ffpe-trap=invalid $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

6.119.3 Converting Between DART Files and GITM Restart Files

The binary GITM files contain no metadata, so care is needed when converting between DART state variables and GITM files.

There are two programs - both require the list of GITM variables to use in the DART state vector: the `&model_nml:gitm_state_variables` variable in the `input.nml` file.

`gitm_to_dart.f90`	to convert a set of GITM restart files (there is one restart file per block) *bxxxx.rst* into a DART-compatible file normally called *dart_ics* . We usually wind up linking to this static filename.
`dart_to_gitm.f90`	to insert the DART output into existing GITM restart files. There are two different types of DART output files, so there is a namelist option to specify if the DART file has two time records or just one. If there are two, the first one is the 'advance_to' time, followed by the 'valid_time' of the ensuing state. If there is just one, it is the 'valid_time' of the ensuing state. *dart_to_gitm* determines the GITM restart file name from the *input.nml model_nml:gitm_restart_dirname*. If the DART file contains an 'advance_to' time, *dart_to_gitm* creates a *DART_GITM_time_control.txt* file which can be used to control the length of the GITM integration.

6.119.4 Simple Test

The simplest way to test the converter is to compile GITM and run a single model state forward using `work/clean.sh`. To build GITM ... download GITM and unpack the code into `DART/models/gitm/GITM2` and run the following commands:

```
$ cd models/gitm/GITM2
$ ./Config.pl -install -compiler=ifortmpif90 -earth
$ make
$ cd ../work
$ ./clean.sh 1 1 0 150.0 170.0 1.0
```

6.119.5 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand & and terminating with a slash / for all our namelist input. Character strings that contain a / **must** be enclosed in quotes to prevent them from prematurely terminating the namelist.

This namelist is read from a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the model dynamical timestep.

Sample input.nml Configuration

```
# The list of variables to put into the state vector is here:
# The definitions for the DART kinds are in DART/observations/forward_operators/obs_
↪def*f90
# The order doesn't matter to DART. It may to you.

&model_nml
   gitm_restart_dirname         = 'advance_temp_e1/UA/restartOUT',
   assimilation_period_days     = 0,
```

(continues on next page)

(continued from previous page)

```
   assimilation_period_seconds  = 1800,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .false.,
   calendar                     = 'Gregorian',
   debug                        = 0,
   gitm_state_variables  = 'Temperature',            'QTY_TEMPERATURE',
                           'eTemperature',           'QTY_TEMPERATURE_ELECTRON',
                           'ITemperature',           'QTY_TEMPERATURE_ION',
                           'iO_3P_NDensityS',        'QTY_DENSITY_NEUTRAL_O3P',
                           'iO2_NDensityS',          'QTY_DENSITY_NEUTRAL_O2',
                           'iN2_NDensityS',          'QTY_DENSITY_NEUTRAL_N2',
                           'iN_4S_NDensityS',        'QTY_DENSITY_NEUTRAL_N4S',
                           'iNO_NDensityS',          'QTY_DENSITY_NEUTRAL_NO',
                           'iN_2D_NDensityS',        'QTY_DENSITY_NEUTRAL_N2D',
                           'iN_2P_NDensityS',        'QTY_DENSITY_NEUTRAL_N2P',
                           'iH_NDensityS',           'QTY_DENSITY_NEUTRAL_H',
                           'iHe_NDensityS',          'QTY_DENSITY_NEUTRAL_HE',
                           'iCO2_NDensityS',         'QTY_DENSITY_NEUTRAL_CO2',
                           'iO_1D_NDensityS',        'QTY_DENSITY_NEUTRAL_O1D',
                           'iO_4SP_IDensityS',       'QTY_DENSITY_ION_O4SP',
                           'iO2P_IDensityS',         'QTY_DENSITY_ION_O2P',
                           'iN2P_IDensityS',         'QTY_DENSITY_ION_N2P',
                           'iNP_IDensityS',          'QTY_DENSITY_ION_NP',
                           'iNOP_IDensityS',         'QTY_DENSITY_ION_NOP',
                           'iO_2DP_IDensityS',       'QTY_DENSITY_ION_O2DP',
                           'iO_2PP_IDensityS',       'QTY_DENSITY_ION_O2PP',
                           'iHP_IDensityS',          'QTY_DENSITY_ION_HP',
                           'iHeP_IDensityS',         'QTY_DENSITY_ION_HEP',
                           'ie_IDensityS',           'QTY_DENSITY_ION_E',
                           'U_Velocity_component',   'QTY_VELOCITY_U',
                           'V_Velocity_component',   'QTY_VELOCITY_V',
                           'W_Velocity_component',   'QTY_VELOCITY_W',
                           'U_IVelocity_component',  'QTY_VELOCITY_U_ION',
                           'V_IVelocity_component',  'QTY_VELOCITY_V_ION',
                           'W_IVelocity_component',  'QTY_VELOCITY_W_ION',
                           'iO_3P_VerticalVelocity', 'QTY_VELOCITY_VERTICAL_O3P',
                           'iO2_VerticalVelocity',   'QTY_VELOCITY_VERTICAL_O2',
                           'iN2_VerticalVelocity',   'QTY_VELOCITY_VERTICAL_N2',
                           'iN_4S_VerticalVelocity', 'QTY_VELOCITY_VERTICAL_N4S',
                           'iNO_VerticalVelocity',   'QTY_VELOCITY_VERTICAL_NO',
                           'f107',                   'QTY_1D_PARAMETER',
                           'Rho',                    'QTY_DENSITY',
    /
```

Description of Each Term in the Namelist

Item	Type	Description
gitm_restart_dirname	character(len=256)	The name of the directory containing the GITM restart files and run-time control information.
assimilation_period_days	integer	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer	In addition to `assimilation_period_days` the number of seconds to advance the model for each each assimilation.
model_perturbation_amplitude	real(r8)	Reserved for future use.
output_state_vector	logical	The switch to determine the form of the of the state vector in the output netCDF files. If `.true.` the state vector will be output exactly as DART uses it . . . one long array. If `.false.`, the state vector is parsed into prognostic variables and output that way – much easier to use with 'ncview', for example.
calendar	character(len=32)	Character string specifying the calendar being used by GITM.
debug	integer	The switch to specify the run-time verbosity. • `0` is as quiet as it gets • `> 1` provides more run-time messages • `> 5` provides ALL run-time messages
gitm_state_variables	character (len=NF90_MAX_NAME):: dimension(160)	The table that relates the GITM variables to use to build the DART state vector, and the corresponding DART kinds for those variables.

6.119.6 Files

filename	purpose
input.nml	to read the model_mod namelist
Several GITM source modules: ModConstants, ModSizeGitm, ModEarth …	provides grid dimensions, model state, and 'valid_time' of the model state
header.rst, bNNNN.rst	provides the 'valid_time' of the model state and the model state, respectively
true_state.nc	the time-history of the "true" model state from an OSSE
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.119.7 References

NASA's official *GITM* description can be found at their Community Coordinated Modeling Center website.

6.120 PROGRAM `netcdf_to_gitm_blocks`

Attention: `GITM` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `GITM` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

The Global Ionosphere Thermosphere Model (GITM) is a 3-dimensional spherical code that models the Earth's thermosphere and ionosphere system using a stretched grid in latitude and altitude. For a fuller description of using GITM within DART, please see the *GITM* documentation.

`netcdf_to_gitm_blocks` is the program that updates the GITM restart files (i.e. `b?????.rst`) with the information from a DART output/restart file (e.g. `perfect_ics, filter_ics, ...`).

The list of variables used to create the DART state vector are specified in the `input.nml` file.

Conditions required for successful execution of `netcdf_to_gitm_blocks`:

- a valid `input.nml` namelist file for DART
- a valid `UAM.in` control file for GITM
- a set of `b?????.rst` data files for GITM
- a `header.rst` file for GITM
- the DART/GITM interfaces must be compiled in a manner consistent with the GITM data and control files. The following GITM source files are required to build *any* DART interface:
  - models/gitm/GITM2/src/ModConstants.f90
  - models/gitm/GITM2/src/ModEarth.f90
  - models/gitm/GITM2/src/ModKind.f90
  - models/gitm/GITM2/src/ModOrbital.f90

- models/gitm/GITM2/src/ModSize.f90
- models/gitm/GITM2/src/ModTime.f90
- models/gitm/GITM2/src/time_routines.f90

Versions of these are included in the DART release. `ModSize.f90`, in particular, must match what was used to create the `b????.rst` files.

The individual model instances are run in unique directories. This is also where the converter routines `gitm_to_dart` and `netcdf_to_gitm_blocks` are run. This makes it easy to use a single 'static' name for the input and output filenames. `advance_model.csh` is responsibile for linking the appropriate files to these static filenames.

The simplest way to test the converter is to compile GITM and run a single model state forward using `work/clean.sh`. To build GITM ... download GITM and unpack the code into `DART/models/gitm/GITM2` and follow these instructions:

```
cd models/gitm/GITM2
./Config.pl -install -compiler=ifortmpif90 -earth
make
cd ../work
./clean.sh 1 1 0 150.0 170.0 1.0
```

And then manually run `netcdf_to_gitm_blocks` on the result.

6.120.1 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&netcdf_to_gitm_blocks_nml
   netcdf_to_gitm_blocks_output_file = 'dart_restart',
   advance_time_present     = .false.
   /

&model_nml
   gitm_restart_dirname         = 'advance_temp_e1/UA/restartOUT',
   assimilation_period_days     = 0,
   assimilation_period_seconds  = 1800,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .false.,
   calendar                     = 'Gregorian',
   debug                        = 0,
   gitm_state_variables = 'Temperature',            'QTY_TEMPERATURE',
                          'eTemperature',           'QTY_TEMPERATURE_ELECTRON',
                          'ITemperature',           'QTY_TEMPERATURE_ION',
                          'iO_3P_NDensityS',        'QTY_DENSITY_NEUTRAL_O3P',
   ...
```

Contents	Type	Description
netcdf_to_gitm_blocks_out	character(len=128)	The output file of the DART file containing the model state derived from the GITM restart files.
advance_time_present	logical	If you are manually converting a DART initial conditions or restart file this should be `.false.`; these files have a single timestamp describing the valid time of the model state. If `.true.`, TWO timestamps are expected in the DART file header and `DART_GITM_time_control.txt`) is created with the settings appropriate to advance GITM to the time requested by DART.

The full description of the `model_nml` namelist is documented in the gitm model_mod, but the most important variable for `netcdf_to_gitm_blocks` is repeated here.

Contents	Type	Description
gitm_restart_dirname	character(len=256)	The name of the directory containing the GITM restart files and runtime control information.
gitm_state_variables	character(len=32), dimension(2,80)	The list of variable names in the gitm restart file to use to create the DART state vector and their corresponding DART kind. The default list is specified in model_mod.nml

6.120.2 Modules used

```
obs_def_upper_atm_mod.f90
assim_model_mod.f90
types_mod.f90
location/threed_sphere/location_mod.f90
models/gitm/GITM2/src/ModConstants.f90
models/gitm/GITM2/src/ModEarth.f90
models/gitm/GITM2/src/ModKind.f90
models/gitm/GITM2/src/ModSize.f90
models/gitm/GITM2/src/ModTime.f90
models/gitm/GITM2/src/time_routines.f90
models/gitm/dart_gitm_mod.f90
models/gitm/netcdf_to_gitm_blocks.f90
models/gitm/model_mod.f90
null_mpi_utilities_mod.f90
obs_kind_mod.f90
random_seq_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.120.3 Files read

- gitm restart files: `b????.rst`
- gitm control files: `header.rst`
- gitm control files: `UAM.in.rst`
- DART namelist file: `input.nml`

6.120.4 Files written

- DART initial conditions/restart file; e.g. `dart_ics`

6.120.5 References

- The official `GITM` site is: can be found at ccmc.gsfc.nasa.gov/models/modelinfo.php?model=GITM

6.121 gitm_blocks_to_netcdf``

Attention: `GITM` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `GITM` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

The Global Ionosphere Thermosphere Model (GITM) is a 3-dimensional spherical code that models the Earth's thermosphere and ionosphere system using a stretched grid in latitude and altitude. For a fuller description of using GITM within DART, please see the *GITM* documentation.

`gitm_blocks_to_netcdf` is the program that reads GITM restart files (i.e. `b?????.rst`) and creates a DART output/restart file (e.g. `perfect_ics, filter_ics, ...`).

The list of variables used to create the DART state vector are specified in the `input.nml` file.

Conditions required for successful execution of `gitm_blocks_to_netcdf`:

- a valid `input.nml` namelist file for DART
- a valid `UAM.in` control file for GITM
- a set of `b?????.rst` data files for GITM
- a `header.rst` file for GITM
- the DART/GITM interfaces must be compiled in a manner consistent with the GITM data and control files. The following GITM source files are required to build *any* DART interface:
  - models/gitm/GITM2/src/ModConstants.f90
  - models/gitm/GITM2/src/ModEarth.f90
  - models/gitm/GITM2/src/ModKind.f90
  - models/gitm/GITM2/src/ModOrbital.f90
  - models/gitm/GITM2/src/ModSize.f90
  - models/gitm/GITM2/src/ModTime.f90

- models/gitm/GITM2/src/time_routines.f90

Versions of these are included in the DART release. `ModSize.f90`, in particular, must match what was used to create the `b????.rst` files.

The individual model instances are run in unique directories. This is also where the converter routines `gitm_blocks_to_netcdf` and `dart_to_gitm` are run. This makes it easy to use a single 'static' name for the input and output filenames. `advance_model.csh` is responsibile for linking the appropriate files to these static filenames.

The simplest way to test the converter is to compile GITM and run a single model state forward using `work/clean.sh`. To build GITM ... download GITM and unpack the code into `DART/models/gitm/GITM2` and follow these instructions:

```
cd models/gitm/GITM2
./Config.pl -install -compiler=ifortmpif90 -earth
make
cd ../work
./clean.sh 1 1 0 150.0 170.0 1.0
```

6.121.1 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&gitm_blocks_to_netcdf_nml
   gitm_blocks_to_netcdf_output_file = 'dart_ics',
   /

&model_nml
   gitm_restart_dirname          = 'advance_temp_e1/UA/restartOUT',
   assimilation_period_days      = 0,
   assimilation_period_seconds   = 1800,
   model_perturbation_amplitude = 0.2,
   output_state_vector           = .false.,
   calendar                      = 'Gregorian',
   debug                         = 0,
   gitm_state_variables = 'Temperature',            'QTY_TEMPERATURE',
                          'eTemperature',           'QTY_TEMPERATURE_ELECTRON',
                          'ITemperature',           'QTY_TEMPERATURE_ION',
                          'iO_3P_NDensityS',        'QTY_DENSITY_NEUTRAL_O3P',
   ...
```

Contents	Type	Description
gitm_blocks_to_netcdf_output_file	character(len=128)	The name of the DART file containing the model state derived from the GITM restart files.

The full description of the `model_nml` namelist is documented in the gitm model_mod, but the most important variable for `gitm_blocks_to_netcdf` is repeated here.

Contents	Type	Description
gitm_restart_dirname	character(len=256)	The name of the directory containing the GITM restart files and runtime control information.
gitm_state_variables	character(len=32), dimension(2,80)	The list of variable names in the gitm restart file to use to create the DART state vector and their corresponding DART kind. The default list is specified in model_mod.nml

6.121.2 Modules used

```
obs_def_upper_atm_mod.f90
assim_model_mod.f90
types_mod.f90
location/threed_sphere/location_mod.f90
models/gitm/GITM2/src/ModConstants.f90
models/gitm/GITM2/src/ModEarth.f90
models/gitm/GITM2/src/ModKind.f90
models/gitm/GITM2/src/ModSize.f90
models/gitm/GITM2/src/ModTime.f90
models/gitm/GITM2/src/time_routines.f90
models/gitm/dart_gitm_mod.f90
models/gitm/gitm_blocks_to_netcdf.f90
models/gitm/model_mod.f90
null_mpi_utilities_mod.f90
obs_kind_mod.f90
random_seq_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.121.3 Files read

- gitm restart files: `b????.rst`
- gitm control files: `header.rst`
- gitm control files: `UAM.in.rst`
- DART namelist file: `input.nml`

6.121.4 Files written

- DART initial conditions/restart file; e.g. `dart_ics`

6.121.5 References

- The official GITM site is: can be found at ccmc.gsfc.nasa.gov/models/modelinfo.php?model=GITM

6.122 Ikeda

6.122.1 Overview

DART interface module for the Ikeda model. The 16 public interfaces are standardized for all DART compliant models. These interfaces allow DART to advance the model, get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for model state variables.
The Ikeda model is a 2D chaotic map useful for visualization data assimilation updating directly in state space. There are three parameters: a, b, and mu. The state is 2D, x = [X Y]. The equations are:

```
X(i+1) = 1 + mu * ( X(i) * cos( t ) - Y(i) * sin( t ) )
Y(i+1) =     mu * ( X(i) * sin( t ) + Y(i) * cos( t ) ),
```

where

```
t = a - b / ( X(i)**2 + Y(i)**2 + 1 )
```

Note the system is time-discrete already, meaning there is no delta_t. The system stems from nonlinear optics (Ikeda 1979, Optics Communications). Interface written by **Greg Lawson, CalTech**. Thanks Greg!

> "The initial conditions were generated by observing state variable 1 with an enormous (~1,000,000.0) observation error variance. The observation was defined to be taken at day=0, seconds = 0. `create_fixed_network_sequence` was run to create a sequence with 3000 hourly observations starting at day=0, seconds =0. The initial conditions for filter can accomodate 100 ensemble members."

6.122.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   a  = 0.40,
   b  = 6.00,
   mu = 0.83,
   time_step_days      = 0,
   time_step_seconds   = 3600,
   output_state_vector = .true.
/
```

Item	Type	Description
a	real(r8)	Model parameter.
b	real(r8)	Model parameter.
mu	real(r8)	Model parameter.
time_step_days	integer	Model advance time in days.
time_step_seconds	integer	Model advance time in seconds.
output_state_vector	logical	If true, output the state vector data to the diagnostic files as a single 1D array. If false, break up output data into logical model variables.

6.122.3 Other modules used

```
types_mod
time_manager_mod
oned/location_mod
utilities_mod
```

6.122.4 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = *get_model_size( )*

```
integer :: get_model_size
```

Returns the length of the model state vector as an integer. This is fixed at 2 for this model.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Advances the model for a single time step. The time associated with the initial model state is also input although it is not used for the computation.

x	State vector of length model_size.
time	Unused in this model.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Returns the location of the given index, and a dummy integer as the var_type.

index_in	Index of state vector element about which information is requested.
location	Returns location of indexed state variable. The location should use a location_mod that is appropriate for the model domain. For realistic atmospheric models, for instance, a three-dimensional spherical location module that can represent height in a variety of ways is provided.
var_type	Returns the type of the indexed state variable as an optional argument.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

A NULL INTERFACE in this model. Always returns istatus = 0.

x	A model state vector.
location	Location to which to interpolate.
itype	Integer indexing which type of state variable is to be interpolated. Can be ignored for low order models with a single type of variable.
obs_val	The interpolated value from the model.
istatus	Quality control information about the observation of the model state.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the models base time step, or forecast length, as a time_type. This is settable in the namelist.

var	Smallest time step of model.

call static_init_model()

Reads the namelist, defines the 2 initial locations of the state variables, and sets the timestep.

call end_model()

A NULL INTERFACE in this model.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns a time of 0.

time	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Sets 2 initial locations close to the attractor.

x	Initial conditions for state vector.

ierr = nc_write_model_atts(ncFileID)

```
integer             :: nc_write_model_atts
integer, intent(in) :: ncFileID
```

Uses the default template code.

ncFileID	Integer file descriptor to previously-opened netCDF file.
ierr	Returns a 0 for successful completion.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                             :: nc_write_model_vars
integer,                intent(in) :: ncFileID
real(r8), dimension(:), intent(in) :: statevec
integer,                intent(in) :: copyindex
integer,                intent(in) :: timeindex
```

Uses the default template code.

ncFileID	file descriptor to previously-opened netCDF file.
statevec	A model state vector.
copyindex	Integer index of copy to be written.
timeindex	The timestep counter for the given state.
ierr	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state, produces a perturbed model state. This particular model does not implement an interface for this and so returns .false. for interf_provided.

state	State vector to be perturbed.
pert_state	Perturbed state vector: NOT returned.
interf_provided	Returned false; interface is not implemented.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 1-D locations module. See *get_close_maxdist_init()* for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 1-D locations module. See *get_close_obs_init()* for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Pass-through to the 1-D locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

A NULL INTERFACE in this model.

ens_mean	State vector containing the ensemble mean.

6.122.5 Files

filename	purpose
input.nml	to read the model_mod namelist
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.122.6 References

Ikeda 1979, Optics Communications

6.122.7 Private components

N/A

6.123 LMDZ

6.123.1 Overview

The Laboratoire de Météorologie Dynamique Zoom (LMDZ) model is a global atmospheric model developed by the Institut Pierre-Simon Laplace (IPSL) in France. It serves as the atmospheric component of the IPSL Integrated Climate Model.

The DART interface to LMDZ was primarily developed by **Tarkeshwar Singh** while he was at the Indian Institute of Technology at Delhi. He later moved to the Nansen Environmental and Remote Sensing Center in Bergen, Norway. A detailed description of the LMDZ DART implementation is published in Singh et al. (2015).[1]

Please email Tarkeshwar for documentation beyond what is contained within the repository.

Assimilation with LMDZ is supported in the Lanai release of DART. If you are interested in using LMDZ in the Manhattan version of DART, we encourage you to contact us. We would like to participate!

6.123.2 References

6.124 Lorenz 05

6.124.1 Naming History

In earlier versions of DART, this collection of models was referred to as Lorenz 04. Edward Lorenz provided James A. Hansen these model formulations before they had been published, since both Lorenz and Hansen were faculty members

[1] **Singh, Tarkeshwar, Rashmi Mitta, and H.C. Upadhyaya**, 2015: Ensemble Adjustment Kalman Filter Data Assimilation for a Global Atmospheric Model. *International Conference on Dynamic Data-Driven Environmental Systems Science*, 284-298, doi:10.1007/978-3-319-25138-7_26.

at MIT at the time. Hansen developed the DART model interface and incorporated it into the DART codebase in 2004. Thus, within DART, it was named Lorenz 04.

The collection of models was published a year later in Lorenz (2005),[1] thus, within the wider community, the models are typically referred to as Lorenz 05. To reflect this fact, the collection of models was renamed within DART from Lorenz 04 to Lorenz 05 during the Manhattan release.

6.124.2 Overview

Lorenz (2005) provides a fascinating account of the difficulties involved in designing simple models that exhibit chaotic behavior and realistically simulate aspects of atmospheric flow. It presents three models of increasing complexity:

- Model I is a single-scale model, similar to Lorenz (1996),[2] intended to represent the atmosphere at a specific height and latitude.
- Model II is also a single-scale model, similar to Model I, but with spatial continuity in the waves.
- Model III is a two-scale model. It is fundamentally different from the Lorenz 96 two-scale model because of the spatial continuity and the fact that both scales are projected onto a single variable of integration. The scale separation is achieved by a spatial filter and is therefore not perfect (i.e. there is leakage).

Model II and Model III are implemented in this DART model interface, and the user is free to choose Model II or III by editing the *namelist*. For users interested in Model I, please use Lorenz 96. The slow scale in Model III is Model II, and thus Model II is a deficient form of Model III.

The Lorenz 05 model has a `work/workshop_setup.csh` script that compiles and runs an example. This example may be used anywhere in the *DART tutorial* to explore multiscale dynamics and to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

Model Formulation

For Lorenz 05, DART to advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation for model state variables.

6.124.3 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size        = 960,
   forcing           = 15.00,
   delta_t           = 0.001,
   space_time_scale  = 10.00,
   coupling          = 3.00,
   K                 = 32,
   smooth_steps      = 12,
   time_step_days    = 0,
   time_step_seconds = 3600,
   model_number      = 3
/
```

[1] Lorenz, Edward N., 2005: Designing Chaotic Models. *Journal of the Atmospheric Sciences*, **62**, 1574-1587.
[2] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability.* **1**, ECMWF, Reading, Berkshire, UK, 1-18.

Description of each namelist entry

Contents	Type	Description
model_size	integer	Number of variables in model
forcing	real(r8)	Forcing, F, for model
delta_t	real(r8)	Non-dimensional timestep
space_time_scale	real(r8)	Determines temporal and spatial relationship between fast and slow variables (model III)
coupling	real(r8)	Linear coupling between fast and slow variables (model III)
K	integer	Determines the wavenumber of the slow variables (K=1, smooth_steps=0 reduces model II to Lorenz 96)
smooth_steps	integer	Determines filter length to separate fast and slow scales
time_step_days	integer	Arbitrary real time step days
time_step_seconds	integer	Arbitrary real time step seconds (could choose this for proper scaling)
model_number	integer	2 = single-scale, 3 = 2-scale. (This follows the notation in the paper.)

6.124.4 References

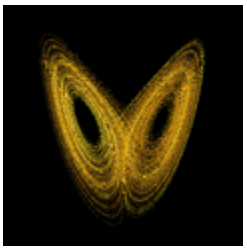
6.125 Lorenz 63

6.125.1 Overview

This 3-variable model was described in Lorenz (1963).[1] In Lorenz 63, DART advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation for model state variables. The distinctive part of the model interface is the *namelist*.

Lorenz 63 was developed as a simplified model to study convection rolls in the atmosphere. It is a deceptively simple model – its formulation is simpler than Lorenz's earlier atmospheric models – yet it demonstrates chaotic behavior. It has thus become a widely studied model.

Plotting the location of the x, y, z values as they progress through time traces out the classic 'butterfly' attractor plot which has become an iconic image of chaotic systems:



[1] Lorenz, Edward N., 1963: Deterministic Nonperiodic Flow. *Journal of the Atmospheric Sciences*, **20**, 130-141, doi:0.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2

The system of equations for Lorenz 63 is:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(r - z) - y$$
$$\frac{dz}{dt} = xy - bz$$

and, within DART, the constants have default values of:

$$\sigma = 10, r = 28, b = 8/3$$

that can be altered by editing the `&model_nml` *namelist* in the `input.nml` file.

This model is an interesting data assimilation test in that different ensemble members may bifurcate over to the other lobe of the attractor on different cycles. Also, as they diverge from each other they do not spread out uniformly in 3D space, but spread along the linear attractor lines.

The Lorenz 63 model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced at various points in the *DART tutorial* and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

`run_lorenz_63.m` is an excellent Matlab tool to explore the behavior of the Lorenz 63 model. It is part of the *DART_LAB Tutorial.*

6.125.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600
   solver = 'RK2'
/
```

Description of each namelist entry

Item	Type	Description
sigma	real(r8)	Model parameter.
r	real(r8)	Model parameter.
b	real(r8)	Model parameter.
deltat	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	integer	Number of days for dimensional timestep, mapped to deltat.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to deltat.
solver	character(8)	The name of the solver to use. 'RK2', the default, is a two-step Runge-Kutta used in the original Lorenz 63 paper. 'RK4' is the only other option which uses the four-step classic Runge-Kutta method.

6.125.3 References

6.126 Lorenz 84

6.126.1 Overview

This model was described in Lorenz (1984).[1] In Lorenz 84, DART advances the model, gets the model state and metadata describing this state, find states variables that are close to a given location, and does spatial interpolation for model state variables. The distinctive part of the model interfaces is the *namelist*.

The system of equations is:

$$\frac{dx}{dt} = -y^2 - z^2 - ax + aF \frac{dy}{dt} = xy - bxz - y + G \frac{dz}{dt} = bxy + xz - z$$

and, within DART, the model parameters have default values of:

$$a = \frac{1}{4}, b = 4, F = 8, G = \frac{5}{4}$$

that can be altered by editing the `&model_nml` *namelist* in the `input.nml` file.

The Lorenz 84 model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced specifically in Section 7 of the *DART tutorial* and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

The Lorenz 84 model may be used instead of the Lorenz 63 model in many sections of the Tutorial. It has a more complex attractor, is not as periodic as Lorenz 63 and may be more challenging for certain filter variants.

[1] Lorenz, Edward N., 1984: Irregularity: A Fundamental Property of the Atmosphere. *Tellus*, **36A**, 98-110, doi:10.1111/j.1600-0870.1984.tb00230.x

6.126.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
  a      = 0.25,
  b      = 4.00,
  f      = 8.00,
  g      = 1.25,
  deltat = 0.01,
  time_step_days    = 0,
  time_step_seconds = 3600
/
```

Description of each namelist entry

Item	Type	Description
a	real(r8)	Model parameter.
b	real(r8)	Model parameter.
f	real(r8)	Model parameter.
g	real(r8)	Model parameter.
deltat	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	integer	Number of days for dimensional timestep, mapped to deltat.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to deltat.

References

6.127 Lorenz 96

6.127.1 Overview

The Lorenz 96 model was first described by Edward Lorenz during a seminar at the European Centre for Medium-Range Weather Forecasts in the Autumn of 1995, the proceedings of which were published as Lorenz (1996)[1] the following year, hence the model is commonly referred to as Lorenz 96.

Lorenz and Emmanuel (1998)[2] describe the model as:

> … consisting of 40 ordinary differential equations, with the dependent variables representing values of some atmospheric quantity at 40 sites spaced equally about a latitude circle. The equations contain quadratic, linear, and constant terms representing advection, dissipation, and external forcing. Numerical integration indicates that small errors (differences between solutions) tend to double in about 2 days. Localized errors tend to spread eastward as they grow, encircling the globe after about 14 days.

[1] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability.* **1**, ECMWF, Reading, Berkshire, UK, 1-18.

[2] Lorenz, Edward N., and Kerry A. Emanuel, 1998: Optimal Sites for Supplementary Weather Observations: Simulations with a Small Model. *Journal of the Atmospheric Sciences*, **55**, 399-414, doi:10.1175/1520-0469(1998)055<0399:OSFSWO>2.0.CO;2

We have chosen a model with J variables, denoted by:

$$X_1, ..., X_j;$$

in most of our experiments we have let $J = 40$. The governing equations are:

$$dX_j/dt = (X_{j+1} - X_{j-2})X_{j-1} - X_j + F(1)$$

for:

$$j = 1, ..., J.$$

To make Eq. (1) meaningful for all values of j we define:

$$X_{-1} = X_{J-1}, X_0 = X_J, \& X_{J+1} = X_1,$$

so that the variables form a cyclic chain, and may be looked at as values of some unspecified scalar meteorological quantity, perhaps vorticity or temperature, at *J* equally spaced sites extending around a latitude circle. Nothing will simulate the atmosphere's latitudinal or vertical extent.

For Lorenz 96, DART advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation for model state variables.

The Lorenz 96 model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced at various points in the *DART tutorial* and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results! Be aware that the `input.nml` file is modified by the `workshop_setup.csh` script.

There are also some excellent Matlab tools to explore the behavior of the Lorenz 96 model, namely `run_lorenz_96.m` and `run_lorenz_96_inf.m`, both of which are part of the *DART_LAB Tutorial.*

6.127.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size         = 40,
   forcing            = 8.00,
   delta_t            = 0.05,
   time_step_days     = 0,
   time_step_seconds  = 3600
/
```

Description of each namelist entry

Item	Type	Description
model_size	integer	Number of variables in model.
forcing	real(r8)	Forcing, F, for model.
delta_t	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	integer	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to delta_t.

6.127.3 References

6.128 Lorenz 96 2-scale

6.128.1 Overview

The Lorenz 96 2-scale model was first described by Edward Lorenz during a seminar at the European Centre for Medium-Range Weather Forecasts in the Autumn of 1995, the proceedings of which were published as Lorenz (1996)[1] the following year, hence the model is commonly referred to as Lorenz 96.

The model state varies on two separate time scales, one for the X dimension and another in the Y dimension. It is constructed by coupling together two implementations of the Lorenz 96 single-scale model. The constant F term in Lorenz 96 single-scale model is replaced by a term that couples the two scales together.

Lorenz 96 2-scale is a widely studied model because the differing timescales can be viewed as an analog of processes that occur on different time and spatial scales in the atmosphere such as large-scale flow and localized convection. The *references* contain some of the earlier studies including Palmer (2001),[2] Smith (2001),[3] Orrell (2002),[4] Orrel (2003),[5] Vannitsem and Toth (2002),[6] Roulston and Smith (2003),[7] and Wilks (2005).[8]

The Lorenz 96 2-scale model has a `work/workshop_setup.csh` script that compiles and runs an example. This example may be explored in the *DART tutorial* and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

Development History

This DART model interface was developed by Josh Hacker as an adaptation of the Lorenz 96 implementation. The 2-scale model is the second model described in Lorenz (1996).

6.128.2 Quick Start

To run Lorenz 96 2-scale with its default settings:

1. Ensure you have the correct settings in mkmf.template in `<DARTROOT>/build_templates/mkmf.template`
2. Build the DART executables using the `quickbuild.sh` script in the `./work` directory.
3. Once the executables have been built, the two Perl scripts provided in the `./shell_scripts` directory, `spinup_model.pl` and `run_expt.pl`, can be used to spin up the model and run an experiment.

[1] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability*. **1**, ECMWF, Reading, Berkshire, UK, 1-18.

[2] Palmer, Timothy N., 2001: A nonlinear dynamical perspective on model error: A proposal for non-local stochastic-dynamic parametrization in weather and climate prediction models. *Quarterly Journal of the Royal Meteorological Society*, **127**, 279–304. https://doi.org/10.1002/qj.49712757202

[3] Smith, Leonard A., 2001: Disentangling uncertainty and error: On the predictability of nonlinear systems. *Nonlinear dynamics and statistics,* Alistair I. Mees, Editor, Birkhauser, Boston, USA, 31–64.

[4] Orrell, David, 2002: Role of the metric in forecast error growth: How chaotic is the weather? *Tellus*, **54A**, 350–362.

[5] Orrell, David, 2003: Model error and predictability over different timescales in the Lorenz '96 Systems. *Journal of the Atmospheric Sciences*, **60**, 2219–2228.

[6] Vannitsem, Stéphane and Zoltan Toth, 2002: Short-term dynamics of model errors. *Journal of the Atmospheric Sciences*, **59**, 2594–2604.

[7] Roulston, Mark S. and Leonard A. Smith, 2003: Combining dynamical and statistical ensembles. *Tellus*, **55A**, 16–30.

[8] Wilks, Daniel S., 2005: Effects of stochastic parametrizations in the Lorenz '96 system. *Quarterly Journal of the Royal Meteorological Society*. **131**. 389-407. https://doi.org/10.1256/qj.04.03

6.128.3 Namelist

The model also implements the variant of Smith (2001), which can be invoked by setting `local_y = .true.` in the `&model_nml` namelist in the `input.nml` file.

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size_x         = 36,
   y_per_x              = 10,
   forcing              = 15.00,
   delta_t              = 0.005,
   coupling_b           = 10.0,
   coupling_c           = 10.0,
   coupling_h           = 1.0,
   local_y              = .false.,
   time_step_days       = 0,
   time_step_seconds    = 3600
   template_file        = 'filter_input.nc'
/
```

Description of each namelist entry

Item	Type	Description
model_size_x	integer	Number of variables in x-dimension.
y_per_x	integer	Scaling factor for number of variables in y-dimension compared to x-dimension.
forcing	real(r8)	Forcing, F, for model.
delta_t	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
coupling_b	real(r8)	
coupling_c	real(r8)	
coupling_h	real(r8)	
local_y	boolean	
time_step_days	integer	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to delta_t.
template_file	character(len=256)	this in script

References

6.129 Lorenz 96 Tracer Advection

6.129.1 Overview

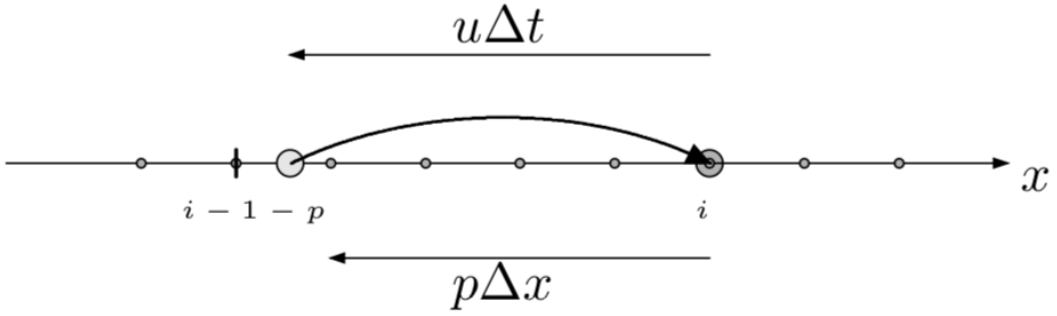
The Lorenz 96 Tracer Advection model combines the Lorenz 96 model with an upstream semi-Lagrangian method. This simulates the advection of tracer particles from any source(s) by the Lorenz 96 variables (here serving as wind).

The Lorenz 96 model was first described by Edward Lorenz during a seminar at the European Centre for Medium-Range Weather Forecasts in the Autumn of 1995, the proceedings of which were published as Lorenz (1996)[1] the following year, hence the model is commonly referred to as Lorenz 96.

Lorenz and Emmanuel (1998)[2] describe the model as:

> ... consisting of 40 ordinary differential equations, with the dependent variables representing values of some atmospheric quantity at 40 sites spaced equally about a latitude circle. The equations contain quadratic, linear, and constant terms representing advection, dissipation, and external forcing. Numerical integration indicates that small errors (differences between solutions) tend to double in about 2 days. Localized errors tend to spread eastward as they grow, encircling the globe after about 14 days.
>
> We have chosen a model with J variables, denoted by:
>
> $$X_1, ..., X_j;$$
>
> in most of our experiments we have let $J = 40$. The governing equations are:
>
> $$dX_j/dt = (X_{j+1} - X_{j-2})X_{j-1} - X_j + F(1)$$
>
> for:
>
> $$j = 1, ..., J.$$
>
> To make Eq. (1) meaningful for all values of *j* we define:
>
> $$X_{-1} = X_{J-1}, X_0 = X_J, \& X_{J+1} = X_1,$$
>
> so that the variables form a cyclic chain, and may be looked at as values of some unspecified scalar meteorological quantity, perhaps vorticity or temperature, at *J* equally spaced sites extending around a latitude circle. Nothing will simulate the atmosphere's latitudinal or vertical extent.

In this model we are using the Semi-Lagrangian Scheme to model how tracer particles get distributed upstream across the grids by the Lorenz 96 winds



The figure above describes the implementation of the Semi-Lagrangian scheme in a one dimensional array. The tracer particle in the figure lands on a predefined grid point at t^{n+1}. The trajectory of this tracer particle is then integrated backwards by one time step to time t^n, often landing between grid points. Then, due to advection without diffusion, the concentration of tracer at time

t^{n+1} is simply the concentration of tracer at time t^n, which

[1] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability.* **1**, ECMWF, Reading, Berkshire, UK, 1-18.

[2] Lorenz, Edward N., and Kerry A. Emanuel, 1998: Optimal Sites for Supplementary Weather Observations: Simulations with a Small Model. *Journal of the Atmospheric Sciences*, **55**, 399-414, doi:10.1175/1520-0469(1998)055<0399:OSFSWO>2.0.CO;2

can be determined by interpolating concentrations of the surrounding grids[3].

Once the coupled Lorenz 96 and semi-Lagrangian is run with a source of strength 100 units/s and location at grid point one (with exponential sinks present in all grid points), the time evolution is as depicted below:

For Lorenz 96 Tracer Advection, DART advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation for model state variables.

6.129.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size        = 120,
   forcing           = 8.00,
   delta_t           = 0.05,
   time_step_days    = 0,
   time_step_seconds = 3600
/
```

Description of each namelist entry

Item	Type	Description
model_size	integer	Total number of items in the state vector. The first third of the state vector describes winds, the second third describes tracer concentration, and the final third of the state vector describes the location strength of sources.
forcing	real(r8)	Forcing, F, for model.
delta_t	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	integer	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to delta_t.

[3] Cushman-Roisin, Benoit, and Jean-Marie Beckers. 2011. Introduction to Geophysical Fluid Dynamics: Volume 101: Physical and Numerical Aspects. 2nd ed. San Diego, CA: Academic Press.

6.129.3 References

6.130 Forced Lorenz 96

6.130.1 Overview

The *forced_lorenz_96* model implements the standard Lorenz (1996)[1] equations except that the forcing term, F, is added to the state vector and is assigned an independent value at each gridpoint. The result is a model that is twice as big as the standard L96 model. The forcing can be allowed to vary in time or can be held fixed so that the model looks like the standard L96 but with a state vector that includes the constant forcing term. An option is also included to add random noise to the forcing terms as part of the time tendency computation which can help in assimilation performance. If the random noise option is turned off (see namelist) the time tendency of the forcing terms is 0.

DART state vector composition:

state variables	forcing terms
traditional Lorenz_96 state	*"extended" state*
`indices 1 - 40`	`indices 41 - 80`

The *forced_lorenz_96* model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced in Section 20 of the DART_tutorial and is intended to provide insight into parameter estimation and model/assimilation behavior. Be aware that the `input.nml` file is modified by the `workshop_setup.csh` script.

6.130.2 Quick Start

To become familiar with the model, try this quick experiment.

1. compile everything in the `model/forced_lorenz_96/work` directory.

```
cd $DARTROOT/models/forced_lorenz_96/work
./quickbuild.ch
```

2. make sure the `input.nml` looks like the following (there is a lot that has been left out for clarity, these are the settings of interest for this example):

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   restart_in_file_name  = "perfect_ics",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   ...
/

&filter_nml
   async                 = 0,
   ens_size              = 80,
   start_from_restart    = .true.,
   output_restart        = .true.,
```

(continues on next page)

[1] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability*. **1**, ECMWF, Reading, Berkshire, UK, 1-18.

(continued from previous page)

```
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "filter_ics",
   restart_out_file_name    = "filter_restart",
   num_output_state_members = 80,
   num_output_obs_members   = 80,
   ...
/

&model_nml
   num_state_vars     = 40,
   forcing            = 8.00,
   delta_t            = 0.05,
   time_step_days     = 0,
   time_step_seconds  = 3600,
   reset_forcing      = .false.,
   random_forcing_amplitude = 0.10
/
```

3. Run `perfect_model_obs` to generate `true_state.nc` and `obs_seq.out`. The default `obs_seq.in` will cause the model to advance for 1000 time steps.

```
./perfect_model_obs
```

4. If you have *ncview*, explore the `true_state.nc`. Notice that the State Variable indices from 1-40 are the dynamical part of the model and 41-80 are the Forcing variables.

```
ncview true_state.nc
```

5. Run `filter` to generate `preassim.nc`, `analysis.nc` and `obs_seq.final`.

```
./filter
```

6. Launch Matlab and run `plot_ens_time_series`.

```
>> plot_ens_time_series
Input name of prior or posterior diagnostics file for preassim.nc:
preassim.nc
OPTIONAL: if you have the true state and want it superimposed,
provide the name of the input file. If not, enter a dummy filename.
Input name of True State file for true_state.nc:
true_state.nc
Using state state variable IDs 1 13 27
If these are OK, ;
If not, please enter array of state variable ID's
To choose from entire state enter A 25 50 75 (between 1 and 80)
To choose traditional model state enter S 1 23 40 (between 1 and 40)
To choose forcing estimates enter F 2 12 22 (between 1 and 40)
(no intervening syntax required)
A 20 30 40 60 70 80
```

Indices 20, 30, and 40 will be from the dynamical part of the lorenz_96 attractor, indices 60, 70, and 80 will be the corresponding Forcing values. Here are some images for just indices 20 and 60. Click on each image for a high-res version.

Repeat the experiment with *reset_forcing = .true.* when creating the true state and *reset_forcing = .false.* when assimilating. What happens?

6.130.3 Namelist

The model also implements the variant of Smith (2001), which can be invoked by setting `local_y = .true.` in the `&model_nml` namelist in the `input.nml` file.

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   num_state_vars     = 40,
   forcing            = 8.00,
   delta_t            = 0.05,
   time_step_days     = 0,
   time_step_seconds  = 3600,
   reset_forcing      = .false.,
   random_forcing_amplitude = 0.10
/
```

Description of each namelist entry

Item	Type	Description
num_state_vars	integer	Number of variables in model.
forcing	real(r8)	Forcing, F, for model.
delta_t	real(r8)	Non-dimensional timestep.
time_step_days	real(r8)	Base model time step maps to this much real time.
time_step_seconds	real(r8)	Base model time step maps to this.
reset_forcing	logical	If true, all forcing values are held fixed at the value specified for the forcing namelist.
random_forcing_amplitude	real(r8)	Standard deviation of the gaussian noise with zero mean that is added to each forcing value's time step.

6.130.4 References

6.131 MITgcm_ocean

6.131.1 Overview

The MIT ocean GCM version 'checkpoint59a' is the foundation of this directory. It was modified by Ibrahim Hoteit of Scripps for his use, and so it differs from the original distribution. The whole process hinges on setting the ocean model namelist values such that it is doing a cold start for every model advance.

The data assimilation period is controlled in the `input.nml&model_nml` namelist. In combination with the ocean model dynamics timestep `data&PARM03:deltaTClock` this determines the amount of time the model will advance for each assimilation cycle.

Observations

The forward operators for ocean observations are in `observations/forward_operators/obs_def_oceam_mod.f90`. *PROGRAM create_ocean_obs* is the program to create a DART observation sequence from a particular ASCII file.

Converting between DART and the model

The MITgcm_ocean model reads and writes Fortran direct-access big-endian data files. To convert MITgcm_ocean files into netcdf files for input to DART use the program `mit_to_dart`. To convert the netcdf output from DART to MITgcm_ocean files use the program `dart_to_mit`. When converting from DART to MIT and back, the following options can be set in the `&trans_mitdart_nml` namelist in `input.nml`.

```
&trans_mitdart_nml
  do_bgc = .false.          ! change to .true. if doing bio-geo-chemistry
  log_transform = .false.   ! change to .true. if using log_transform
/
```

Warning:

The `trans_mit_dart_mod` module has hardcoded record lengths, `recl3d` and `recl2d`. Be sure to check these are correct for the compiler you are using.

```
! set record lengths
recl3d = Nx*Ny*Nz*4
recl2d = Nx*Ny*4
```

Controlling the model advances

The assimilation period is specified by two namelist parameters in the `input.nml&model_nml` namelist: `assimilation_period_days` and `assimilation_period_seconds`. Normally, all observations within (+/-) HALF of the total assimilation period are used in the assimilation.

The time of the initial conditions is specified by two namelist parameters in the `input.nml&model_nml` namelist: `init_time_days` and `init_time_seconds`; depending on the settings of these parameters, the times may or may not come directly from the DART initial conditions files.

The ocean model **MUST always** start from the input datasets defined in the `data&PARM05` namelist. Apparently, this requires `data&PARM03:startTime` to be **0.0**. One of the DART support routines `trans_mitdart` converts the DART state vector to the files used in `data&PARM05` and creates new `data.cal&CAL_NML` and `data&PARM03` namelists with values appropriate to advance the model to the desired time.

The ocean model then advances till `data&PARM03:endTime` and writes out snapshot files. `trans_mitdart` converts the snapshot files to a DART-compatible file which is ingested by `filter`. `filter` also reads the observation sequence file to determine which observations are within the assimilation window, assimilates them, and writes out a set of restart files, one for each ensemble member. `filter` then waits for each instance of the ocean model (one instance for each ensemble member) to advance to `data&PARM03:endTime`. The whole process repeats until 1) there are no more observations to assimilate (i.e. the observation sequence file is exhausted) or 2) the time specified by `input.nml&filter_nml:last_obs_days,last_obs_seconds` has been reached.

Getting started

1. create an initial ensemble for DART.
2. create a TINY set of 'perfect' observations in the normal fashion: *program create_obs_sequence* and then *program create_fixed_network_seq* to create an empty observation sequence file (usually called `obs_seq.in`)
3. modify `data`, `data.cal`, and `input.nml` to control the experiment and populate the observation sequence file by running *program perfect_model_obs*
4. Now use the full ensemble of initial conditions from Step 1 and run *PROGRAM filter*

A perfectly sensible approach to get to know the system would be to try to

1. assimilate data for the first assimilation period and stop. Do not advance the model at all. The filter namelist can control all of this and you do not need to have a working `advance_model.csh` script, or even a working ocean model (as long as you have input data files).
2. advance the model first and then assimilate data for the first assimilation period and stop.
3. advance, assimilate and advance again. This tests the whole DART facility.

Exploring the output

There are Matlab® scripts for perusing netCDF files in the `DART/matlab` directory. There are Matlab® scripts for exploring the performance of the assimilation in observation-space (after running *PROGRAM obs_diag (for observations that use the threed_sphere location module)* to explore the `obs_seq.final` file) - use the scripts starting with `'plot_'`, e.g. `DART/diagnostics/matlab/plot_*.m`. As always, there are some model-specific item you should know about in `DART/models/MITgcm_ocean/matlab`, and `DART/models/MITgcm_ocean/shell_scripts`.

6.131.2 Files

- input namelist files: `data, data.cal, input.nml`
- input data file: `filter_ics, perfect_ics`
- output data files: `[S,T,U,V,Eta].YYYYMMDD.HHMMSS.[data,meta]`

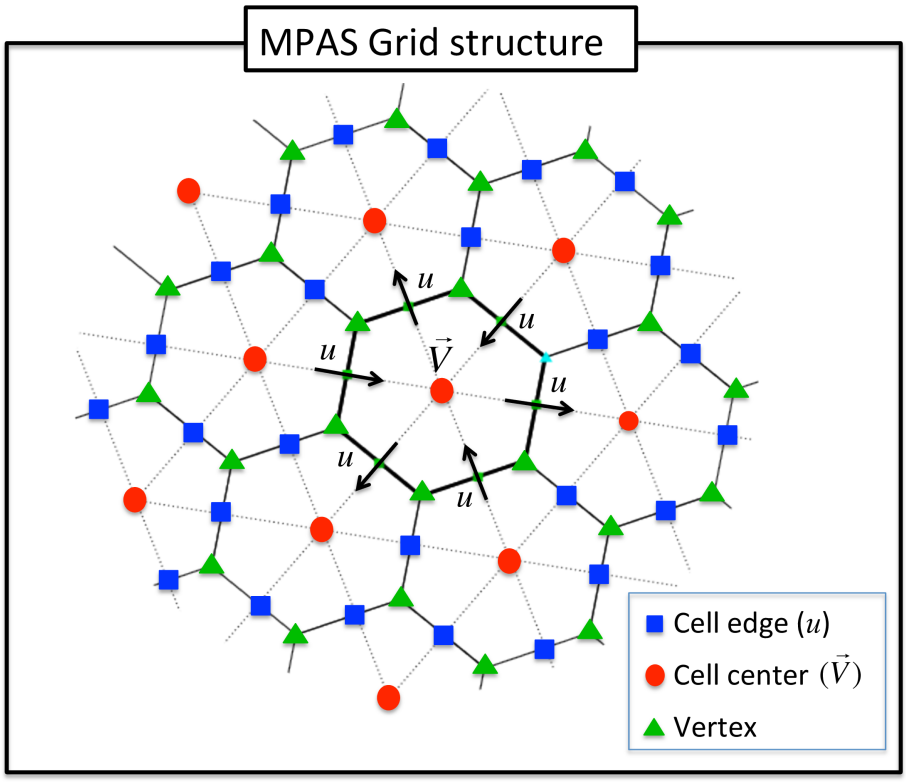
Please note that there are **many** more files needed to advance the ocean model, none of which are discussed here.

6.132 MPAS_ATM

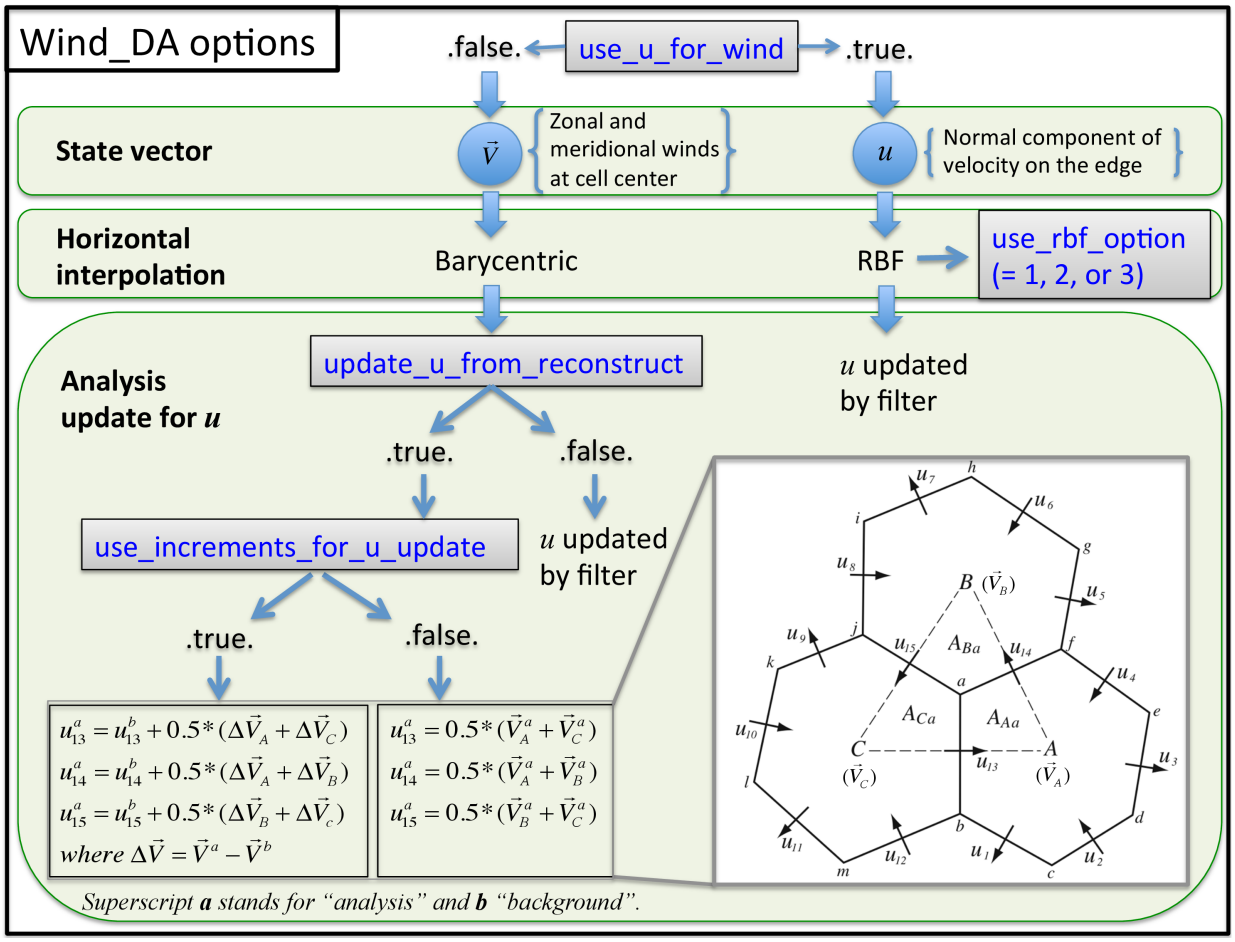
6.132.1 Overview

This document describes the DART interface module for the atmospheric component of the Model for Prediction Across Scales MPAS (or briefly, MPAS-ATM) global model, which uses an unstructured Voronoi grid mesh, formally Spherical Centriodal Voronoi Tesselations (SCVTs). This allows for both quasi-uniform discretization of the sphere and local refinement. The MPAS/DART interface was built on the SCVT-dual mesh and does not regrid to regular lat/lon grids. In the C-grid discretization, the normal component of velocity on cell edges is prognosed; zonal and meridional wind components are diagnosed on the cell centers. We provide several options to choose from in the assimilation of wind observations as shown below.

The grid terminology used in MPAS is as shown in the figure below:



The wind options during a DART assimilation are controlled by combinations of 4 different namelist values. The values determine which fields the forward operator uses to compute expected observation values; how the horizontal interpolation is computed in that forward operator; and how the assimilation increments are applied to update the wind quantities in the state vector. Preliminary results based on real data assimilation experiments indicate that performance is better when the zonal and meridional winds are used as input to the forward operator that uses Barycentric interpolation, and when the prognostic u wind is updated by the incremental method described in the figure below. However there remain scientific questions about how best to handle the wind fields under different situations. Thus we have kept all implemented options available for use in experimental comparisons. See the figure below for a flow-chart representation of how the 4 namelist items interact:



Cycling of MPAS/DART is run in a *restart* mode. As for all DART experiments, the overall design for an experiment is this: the DART program `filter` will read the initial condition file, the observation sequence file, and the DART namelist to decide whether or not to advance the MPAS-ATM model. All of the control of the execution of the MPAS model is done by DART directly. If the model needs to be advanced, `filter` makes a call to the shell to execute the script `advance_model.csh`, which is ENTIRELY responsible for getting all the input files, data files, namelists, etc. into a temporary directory, running the model, and copying the results back to the parent directory (which we call CENTRALDIR). The whole process hinges on setting the MPAS-ATM model namelist values such that it is doing a restart for every model advance. Unlike MPAS-ATM free forecast runs, the forecast step in MPAS/DART requires to set up one more namelist parameter called `config_do_DAcycling = .true.` in `&restart` section of `namelist.input` to recouple the state vectors (updated by filter) with the mass field for the restart mode. For more information, check the `advance_model.csh` script in `./shell_scripts/` directory.

Since DART is an ensemble algorithm, there are multiple analysis files for a single analysis time: one for each ensemble member. Because MPAS/DART is run in a restart mode, each member should keep its own MPAS restart file from the previous cycle (rather than having a single template file in CENTRALDIR). Creating the initial ensemble of states is an area of active research.

6.132.2 Namelist

This namelist is read from the file *input.nml*. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   init_template_filename       = 'mpas_init.nc',
   vert_localization_coord      = 3,
   assimilation_period_days     = 0,
   assimilation_period_seconds  = 21600,
   model_perturbation_amplitude = 0.0001,
   log_p_vert_interp            = .true.,
   calendar                     = 'Gregorian',
   use_u_for_wind               = .false.,
   use_rbf_option               = 2,
   update_u_from_reconstruct    = .true.,
   use_increments_for_u_update  = .true.,
   highest_obs_pressure_mb      = 100.0,
   sfc_elev_max_diff            = -1.0,
   outside_grid_level_tolerance = -1.0,
   extrapolate                  = .false.,
   debug                        = 0,
/
```

Item	Type	Description
init_template_filename	character(len=256) *[default: 'mpas_init.nc']*	The name of the MPAS analysis file to be read and/or written by the DART programs for the state data.
highest_obs_pressure_mb	real(r8) *[default: 100.0]*	Observations higher than this pressure are ignored. Set to -1.0 to ignore this test. For models with a prescribed top boundary layer, trying to assimilate very high observations results in problems because the model damps out any changes the assimilation tries to make. With adaptive algorithms this results in larger and larger coefficients as the assimilation tries to effect state vector change.
assimilation_period_days	integer *[default: 0]*	The number of days to advance the model for each assimilation. Even if the model is being advanced outside of the DART filter program, the assimilation period should be set correctly. Only observations with a time within +/- 1/2 this window size will be assimilated.
assimilation_period_seconds	integer *[default: 21600]*	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
vert_localization_coord	integer *[default: 3]*	Vertical coordinate for vertical localization. • 1 = model level • 2 = pressure (in pascals) • 3 = height (in meters) • 4 = scale height (unitless)
sfc_elev_max_diff	real(r8)*[default: -1.0]*	If > 0, the maximum difference, in meters, between an observation marked as a 'surface obs' as the vertical type (with the surface elevation, in meters, as the numerical vertical location), and the surface elevation as defined by the model. Observations further away from the surface than this threshold are rejected and not assimilated. If the value is negative, this test is skipped.
log_p_vert_interp	logical *[default: .true.]*	If `.true.`, vertical interpolation is done in log-pressure. Otherwise, linear.
use_u_for_wind	logical *[default: .false.]*	If `.false.`, zonal and meridional winds at cell centers are used for the wind observation operator [default]. In that case, triangular meshes are used for the barycentric (e.g., area-weighted) interpolation. If `.true.`, wind vectors at an arbitrary (e.g., observation) point are reconstructed from the normal component of velocity on cell edges *(u)* using radial basis functions (RBFs) provided by

The &mpas_vars_nml namelist within input.nml contains the list of MPAS variables that make up the DART state vector. The order the items are specified controls the order of the data in the state vector, so it should not be changed without regenerating all DART initial condition or restart files. These variables are directly updated by the filter assimilation.

Any variables whose values cannot exceed a given minimum or maximum can be listed in mpas_state_bounds. When the data is written back into the MPAS NetCDF files values outside the allowed range will be detected and changed. Data inside the DART state vector and data written to the DART diagnostic files will not go through this test and values may exceed the allowed limits. Note that changing values at the edges of the distribution means it is no longer completely gaussian. In practice this technique has worked effectively, but if the assimilation is continually trying to move the values outside the permitted range the results may be of poor quality. Examine the diagnostics for these fields carefully when using bounds to restrict their values.

```
&mpas_vars_nml
   mpas_state_variables = 'theta',                 'QTY_POTENTIAL_TEMPERATURE',
                          'uReconstructZonal',     'QTY_U_WIND_COMPONENT',
                          'uReconstructMeridional','QTY_V_WIND_COMPONENT',
                          'qv',                    'QTY_VAPOR_MIXING_RATIO',
                          'qc',                    'QTY_CLOUDWATER_MIXING_RATIO',
                          'surface_pressure',      'QTY_SURFACE_PRESSURE'
   mpas_state_bounds    = 'qv','0.0','NULL','CLAMP',
                          'qc','0.0','NULL','CLAMP',
/
```

Item	Type	Description
mpas_vars_nml	character(len=NF90_MAX_NAME) and MAX_MPAS_VARIABLES)	The table that both specifies which MPAS-ATM variables will be placed in the state vector, and also relates these variables to the corresponding DART kinds. The first column in each pair must be the exact NetCDF name of a field in the MPAS file. The second column in each pair must be a KIND known to the DART system. See the obs_kind_mod.f90 file within assimilation_code/modules/observations/ for known names. This file is autogenerated when DART builds filter for a particular model, so run quickbuild.sh in the work directory first before examining this file. Use the generic kind list in the obs_kind_mod tables, not the specific type list.
mpas_state_bounds	character(len=NF90_MAX_NAME) and MAX_MPAS_VARIABLES)	List only MPAS-ATM variables that must restrict their values to remain between given lower and upper bounds. Columns are: NetCDF variable name, min value, max value, and action to take for out-of-range values. Either min or max can have the string 'NULL' to indicate no limiting will be done. If the action is 'CLAMP' out of range values will be changed to the corresponding bound and execution continues; 'FAIL' stops the executable if out of range values are detected.

6.132.3 Grid Information

As the forward operators use the unstructured grid meshes in MPAS-ATM, the DART/MPAS interface needs to read static variables related to the grid structure from the MPAS ATM 'history' file (specified in model_analysis_filename). These variables are used to find the closest cell to an observation point in the cartesian coordinate (to avoid the polar issues).

integer :: nCells	the number of cell centers
integer :: nEdges	the number of cell edges
integer :: nVertices	the number of cell vertices
integer :: nVertLevels	the number of vertical levels for mass fields
integer :: nVertLevelsP1	the number of vertical levels for vertical velocity
integer :: nSoilLevels	the number of soil levels
real(r8) :: latCell(:)	the latitudes of the cell centers [-90,90]
real(r8) :: lonCell(:)	the longitudes of the cell centers [0, 360]
real(r8) :: latEdge(:)	the latitudes of the edges [-90,90], if edge winds are used.
real(r8) :: lonEdge(:)	the longitudes of the edges [0, 360], if edge winds are used.
real(r8) :: xVertex(:)	The cartesian location in x-axis of the vertex
real(r8) :: yVertex(:)	The cartesian location in y-axis of the vertex
real(r8) :: zVertex(:)	The cartesian location in z-axis of the vertex
real(r8) :: xEdge(:)	The cartesian location in x-axis of the edge, if edge winds are used.
real(r8) :: yEdge(:)	The cartesian location in y-axis of the edge, if edge winds are used.
real(r8) :: zEdge(:)	The cartesian location in z-axis of the edge, if edge winds are used.
real(r8) :: zgrid(:,:)	geometric height at cell centers (nCells, nVertLevelsP1)
integer :: CellsOnVertex(:,:)	list of cell centers defining a triangle
integer :: edgesOnCell(:,:)	list of edges on each cell
integer :: verticesOnCell(:,:)	list of vertices on each cell
integer :: edgeNormalVectors(:,:)	unit direction vectors on the edges (only used if *use_u_for_wind* = .true.)

6.132.4 model_mod variable storage

The `&mpas_vars_nml` within `input.nml` defines the list of MPAS variables used to build the DART state vector. Combined with an MPAS analysis file, the information is used to determine the size of the DART state vector and derive the metadata. To keep track of what variables are contained in the DART state vector, an array of a user-defined type called "progvar" is available with the following components:

```
type progvartype
   private
   character(len=NF90_MAX_NAME) :: varname
   character(len=NF90_MAX_NAME) :: long_name
   character(len=NF90_MAX_NAME) :: units
   character(len=NF90_MAX_NAME), dimension(NF90_MAX_VAR_DIMS) :: dimname
   integer, dimension(NF90_MAX_VAR_DIMS) :: dimlens
   integer :: xtype         ! netCDF variable type (NF90_double, etc.)
   integer :: numdims       ! number of dimensions - excluding TIME
   integer :: numvertical   ! number of vertical levels in variable
   integer :: numcells      ! number of cell locations (typically cell centers)
   integer :: numedges      ! number of edge locations (edges for normal velocity)
   logical :: ZonHalf       ! vertical coordinate for mass fields (nVertLevels)
   integer :: varsize       ! variable size (dimlens(1:numdims))
   integer :: index1        ! location in dart state vector of first occurrence
   integer :: indexN        ! location in dart state vector of last  occurrence
   integer :: dart_kind
   character(len=paramname_length) :: kind_string
   logical  :: clamping     ! does variable need to be range-restricted before
   real(r8) :: range(2)     ! lower and upper bounds for the data range.
   logical  :: out_of_range_fail  ! is out of range fatal if range-checking?
end type progvartype
```

(continues on next page)

(continued from previous page)

```
type(progvartype), dimension(max_state_variables) :: progvar
```

The variables are simply read from the MPAS analysis file and stored in the DART state vector such that all quantities for one variable are stored contiguously. Within each variable; they are stored vertically-contiguous for each horizontal location. From a storage standpoint, this would be equivalent to a Fortran variable dimensioned x(nVertical,nHorizontal,nVariables). The fastest-varying dimension is vertical, then horizontal, then variable … naturally, the DART state vector is 1D. Each variable is also stored this way in the MPAS analysis file.

6.132.5 Compilation

The DART interface for MPAS-ATM can be compiled with various fortran compilers such as (but not limited to) gfortran, pgf90, and intel. It has been tested on a Mac and NCAR IBM supercomputer (yellowstone).

Note: While MPAS requires the PIO (Parallel IO) and pNetCDF (Parallel NetCDF) libraries, DART uses only the plain NetCDF libraries. If an altered NetCDF library is required by the parallel versions, there may be incompatibilities between the run-time requirements of DART and MPAS. Static linking of one or the other executable, or swapping of modules between executions may be necessary.

6.132.6 Conversions

A Welcome Development

MPAS files no longer need to be converted to DART formatted files, they can be read in directly from a input file list!

Analysis File NetCDF header

The header of an MPAS analysis file is presented below - simply for context. Keep in mind that **many** variables have been removed for clarity. Also keep in mind that the multi-dimensional arrays listed below have the dimensions reversed from the Fortran convention. **Note:** the variables marked 'available in dart' are available as metadata variables in DART. Just to be perfectly clear, they are not 'state'.

```
$ ncdump -h mpas_init.nc
netcdf mpas_analysis {
dimensions:
        StrLen = 64 ;
        Time = UNLIMITED ; // (1 currently)
        nCells = 10242 ;                                    available in DART
        nEdges = 30720 ;                                    available in DART
        maxEdges = 10 ;
        maxEdges2 = 20 ;
        nVertices = 20480 ;                                 available in DART
        TWO = 2 ;
        THREE = 3 ;
        vertexDegree = 3 ;
        FIFTEEN = 15 ;
        TWENTYONE = 21 ;
        R3 = 3 ;
```

(continues on next page)

(continued from previous page)

```
        nVertLevels = 41 ;                                  available in DART
        nVertLevelsP1 = 42 ;                                available in DART
        nMonths = 12 ;
        nVertLevelsP2 = 43 ;
        nSoilLevels = 4 ;                                   available in DART
variables:
        char xtime(Time, StrLen) ;                          available in DART
        double latCell(nCells) ;                            available in DART
        double lonCell(nCells) ;                            available in DART
        double latEdge(nEdges) ;                            available in DART
        double lonEdge(nEdges) ;                            available in DART
        int indexToEdgeID(nEdges) ;
        double latVertex(nVertices) ;
        double lonVertex(nVertices) ;
     double xVertex(nVertices) ;                         available in DART
     double yVertex(nVertices) ;                         available in DART
     double zVertex(nVertices) ;                         available in DART
     double xEdge(nVertices) ;                           available in DART
     double yEdge(nVertices) ;                           available in DART
     double zEdge(nVertices) ;                           available in DART
        int indexToVertexID(nVertices) ;
        int cellsOnEdge(nEdges, TWO) ;
        int nEdgesOnCell(nCells) ;
        int nEdgesOnEdge(nEdges) ;
        int edgesOnCell(nCells, maxEdges) ;                 available in DART
        int edgesOnEdge(nEdges, maxEdges2) ;
        double weightsOnEdge(nEdges, maxEdges2) ;
        double dvEdge(nEdges) ;
        double dcEdge(nEdges) ;
        double angleEdge(nEdges) ;
        double edgeNormalVectors(nEdges, R3) ;              available in DART
        double cellTangentPlane(nEdges, TWO, R3) ;
        int cellsOnCell(nCells, maxEdges) ;
        int verticesOnCell(nCells, maxEdges) ;              available in DART
        int verticesOnEdge(nEdges, TWO) ;
        int edgesOnVertex(nVertices, vertexDegree) ;
        int cellsOnVertex(nVertices, vertexDegree) ;        available in DART
        double kiteAreasOnVertex(nVertices, vertexDegree) ;
        double rainc(Time, nCells) ;
        double cuprec(Time, nCells) ;
        double cutop(Time, nCells) ;
        double cubot(Time, nCells) ;
        double relhum(Time, nCells, nVertLevels) ;
        double qsat(Time, nCells, nVertLevels) ;
        double graupelnc(Time, nCells) ;
        double snownc(Time, nCells) ;
        double rainnc(Time, nCells) ;
        double graupelncv(Time, nCells) ;
        double snowncv(Time, nCells) ;
        double rainncv(Time, nCells) ;
        double sr(Time, nCells) ;
        double surface_temperature(Time, nCells) ;
```

(continues on next page)

(continued from previous page)

```
        double surface_pressure(Time, nCells) ;
        double coeffs_reconstruct(nCells, maxEdges, R3) ;
        double theta_base(Time, nCells, nVertLevels) ;
        double rho_base(Time, nCells, nVertLevels) ;
        double pressure_base(Time, nCells, nVertLevels) ;
        double exner_base(Time, nCells, nVertLevels) ;
        double exner(Time, nCells, nVertLevels) ;
        double h_divergence(Time, nCells, nVertLevels) ;
        double uReconstructMeridional(Time, nCells, nVertLevels) ;
        double uReconstructZonal(Time, nCells, nVertLevels) ;
        double uReconstructZ(Time, nCells, nVertLevels) ;
        double uReconstructY(Time, nCells, nVertLevels) ;
        double uReconstructX(Time, nCells, nVertLevels) ;
        double pv_cell(Time, nCells, nVertLevels) ;
        double pv_vertex(Time, nVertices, nVertLevels) ;
        double ke(Time, nCells, nVertLevels) ;
        double rho_edge(Time, nEdges, nVertLevels) ;
        double pv_edge(Time, nEdges, nVertLevels) ;
        double vorticity(Time, nVertices, nVertLevels) ;
        double divergence(Time, nCells, nVertLevels) ;
        double v(Time, nEdges, nVertLevels) ;
        double rh(Time, nCells, nVertLevels) ;
        double theta(Time, nCells, nVertLevels) ;
        double rho(Time, nCells, nVertLevels) ;
        double qv_init(nVertLevels) ;
        double t_init(nCells, nVertLevels) ;
        double u_init(nVertLevels) ;
        double pressure_p(Time, nCells, nVertLevels) ;
        double tend_theta(Time, nCells, nVertLevels) ;
        double tend_rho(Time, nCells, nVertLevels) ;
        double tend_w(Time, nCells, nVertLevelsP1) ;
        double tend_u(Time, nEdges, nVertLevels) ;
        double qv(Time, nCells, nVertLevels) ;
        double qc(Time, nCells, nVertLevels) ;
        double qr(Time, nCells, nVertLevels) ;
        double qi(Time, nCells, nVertLevels) ;
        double qs(Time, nCells, nVertLevels) ;
        double qg(Time, nCells, nVertLevels) ;
        double tend_qg(Time, nCells, nVertLevels) ;
        double tend_qs(Time, nCells, nVertLevels) ;
        double tend_qi(Time, nCells, nVertLevels) ;
        double tend_qr(Time, nCells, nVertLevels) ;
        double tend_qc(Time, nCells, nVertLevels) ;
        double tend_qv(Time, nCells, nVertLevels) ;
        double qnr(Time, nCells, nVertLevels) ;
        double qni(Time, nCells, nVertLevels) ;
        double tend_qnr(Time, nCells, nVertLevels) ;
        double tend_qni(Time, nCells, nVertLevels) ;
```

6.132.7 Files

filename	purpose
input.nml	to read the namelist - model_mod_nml and mpas_vars_nml
mpas_init.nc	provides model state, and 'valid_time' of the model state
static.nc	provides grid dimensions
true_state.nc	the time-history of the "true" model state from an OSSE
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.132.8 References

The Data Assimilation section in the MPAS documentation found at http://mpas-dev.github.io.

6.133 PROGRAM `mpas_dart_obs_preprocess`

6.133.1 Overview

Program to preprocess observations, with specific knowledge of the MPAS grid.

This program can superob (average) aircraft and satellite wind obs if they are too dense, based on the given MPAS ATM grid. It will average all observations of the same type in each grid cell. The averaging grid can be different than the grid used for the assimilation run.

This program can read up to 10 additional obs_seq files and merge their data in with the basic obs_sequence file which is the main input.

This program can reject surface observations if the elevation encoded in the observation is too different from the mpas surface elevation.

This program can exclude observations above a specified height or pressure.

This program can exclude observations outside a given time window defined by the specified analysis time and a window width in hours.

This program can overwrite the incoming Data QC value with another.

6.133.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&mpas_obs_preproc_nml

  file_name_input          = 'obs_seq.old'
  file_name_output         = 'obs_seq.new'

  sonde_extra              = 'obs_seq.rawin'
```

(continues on next page)

(continued from previous page)

```
  land_sfc_extra            = 'obs_seq.land_sfc'
  metar_extra               = 'obs_seq.metar'
  marine_sfc_extra          = 'obs_seq.marine'
  sat_wind_extra            = 'obs_seq.satwnd'
  profiler_extra            = 'obs_seq.profiler'
  gpsro_extra               = 'obs_seq.gpsro'
  acars_extra               = 'obs_seq.acars'
  gpspw_extra               = 'obs_seq.gpspw'
  trop_cyclone_extra        = 'obs_seq.tc'

  overwrite_obs_time        = .false.
  windowing_obs_time        = .false.
  windowing_int_hour        = 1.5

  obs_boundary              = 0.0
  increase_bdy_error        = .false.
  maxobsfac                 = 2.5
  obsdistbdy                = 15.0

  sfc_elevation_check       = .false.
  sfc_elevation_tol         = 300.0
  obs_pressure_top          = 0.0
  obs_height_top            = 2.0e10

  include_sig_data          = .true.
  tc_sonde_radii            = -1.0
  superob_qc_threshold      = 4

  superob_aircraft          = .false.
  aircraft_horiz_int        = 36.0
  aircraft_pres_int         = 2500.0

  superob_sat_winds         = .false.
  sat_wind_horiz_int        = 100.0
  sat_wind_pres_int         = 2500.0

  overwrite_ncep_satwnd_qc = .false.
  overwrite_ncep_sfc_qc     = .false.

  max_num_obs               = 1000000
/
```

Item	Type	Description
Generic parameters:		
file_name_input	character(len=129)	The input obs_seq file.
file_name_output	character(len=129)	The output obs_seq file.
sonde_extra, land_sfc_extra, metar_extra, marine_sfc_extra, sat_wind_extra, profiler_extra, gpsro_extra, acars_extra, gpspw_extra, trop_cyclone_extra	character(len=129)	The names of additional input obs_seq files, which if they exist, will be merged in with the obs from the file_name_input obs_seq file. If the files do not exist, they are silently ignored without error.
max_num_obs	integer	Must be larger than the total number of observations to be processed.
Parameters to reduce observation count:		
sfc_elevation_check	logical	If true, check the height of surface observations against the surface height in the model. Observations further away than the specified tolerance will be excluded.
sfc_elevation_tol	real(r8)	If sfc_elevation_check is true, the maximum difference between the elevation of a surface observation and the model surface height, in meters. If the difference is larger than this value, the observation is excluded.
obs_pressure_top	real(r8)	Observations with a vertical coordinate in pressure which are located above this pressure level (i.e. the obs vertical value is smaller than the given pressure) will be excluded.
obs_height_top	real(r8)	Observations with a vertical coordinate in height which are located above this height value (i.e. the obs vertical value is larger than the given height) will be excluded.
Radio/Rawinsonde-specific parameters:		
include_sig_data	logical	If true, include significant level data from radiosondes.
tc_sonde_radii	real(r8)	If greater than 0.0 remove any sonde observations closer than this distance in Kilometers to the center of a Tropical Cyclone.
Aircraft-specific parameters:		
superob_aircraft	logical	If true, average all aircraft observations within the same MPAS grid cell, at the given vertical levels. The output obs will be only a single observation per cell, per vertical level.
aircraft_pres_int	real(r8)	If superob_aircraft is true, the vertical distance in pressure which defines a series of superob vertical bins.
superob_qc_threshold	integer	If superob_aircraft is true, the Quality Control threshold at which observations are ignored when doing superob averaging. The value spec-

6.133.3 Modules used

```
types_mod
obs_sequence_mod
utilities_mod
obs_kind_mod
time_manager_mod
model_mod
netcdf
```

6.133.4 Files

- Input namelist ; `input.nml`
- Input MPAS state netCDF file: `mpas_init.nc`
- Input obs_seq files (as specified in namelist)
- Output obs_seq file (as specified in namelist)

File formats

This utility can read one or more obs_seq files and combine them while doing the rest of the processing. It uses the standard DART observation sequence file format. It uses the grid information from an MPAS file to define the bins for combining nearby aircraft and satellite wind observations.

6.133.5 References

- Developed by Soyoung Ha, based on the WRF observation preprocessor contributed by Ryan Torn.

6.134 MPAS OCN

Attention: `mpas_ocn` was being developed with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `mpas_ocn` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.134.1 Overview

The **MPAS OCN** interface for **Data Assimilation Research Testbed (DART)** is under development.
Since MPAS OCN uses netcdf files for their restart mechanism, a namelist-controlled set of variables is used to build the DART state vector. Each variable must also correspond to a DART "QUANTITY"; required for the DART interpolate routines. For example:

```
&mpas_vars_nml
   mpas_state_variables = 'uReconstructZonal',      'QTY_U_WIND_COMPONENT',
                          'uReconstructMeridional', 'QTY_V_WIND_COMPONENT',
                          'w',                      'QTY_VERTICAL_VELOCITY',
                          'theta',                  'QTY_POTENTIAL_TEMPERATURE',
                          'qv',                     'QTY_VAPOR_MIXING_RATIO',
                          'qc',                     'QTY_CLOUDWATER_MIXING_RATIO',
                          'qr',                     'QTY_RAINWATER_MIXING_RATIO',
                          'qi',                     'QTY_ICE_MIXING_RATIO',
                          'qs',                     'QTY_SNOW_MIXING_RATIO',
                          'qg',                     'QTY_GRAUPEL_MIXING_RATIO',
                          'surface_pressure',       'QTY_SURFACE_PRESSURE'
   /
```

These variables are then adjusted to be consistent with observations and stuffed back into the same netCDF analysis files. Since DART is an ensemble algorithm, there are multiple analysis files for a single analysis time: one for each ensemble member. Creating the initial ensemble of states is an area of active research.

DART reads grid information from the MPAS OCN 'history' file, I have tried to keep the variable names the same. Internal to the DART code, the following variables exist:

integer :: nCells	the number of Cell Centers
integer :: nEdges	the number of Cell Edges
integer :: nVertices	the number of Cell Vertices
integer :: nVertLevels	the number of vertical level midpoints
integer :: nVertLevelsP1	the number of vertical level edges
integer :: nSoilLevels	the number of soil level ?midpoints?
real(r8) :: latCell(:)	the latitudes of the Cell Centers (-90,90)
real(r8) :: lonCell(:)	the longitudes of the Cell Centers [0, 360)
real(r8) :: zgrid(:,:)	cell center geometric height at cell centers (ncells,nvert)
integer :: CellsOnVertex(:,:)	list of cell centers defining a triangle

6.134.2 model_mod variable storage

`input.nml&mpas_vars_nml` defines the list of MPAS variables used to build the DART state vector. Combined with an MPAS analysis file, the information is used to determine the size of the DART state vector and derive the metadata. To keep track of what variables are contained in the DART state vector, an array of a user-defined type called "progvar" is available with the following components:

```
type progvartype
   private
   character(len=NF90_MAX_NAME) :: varname
   character(len=NF90_MAX_NAME) :: long_name
   character(len=NF90_MAX_NAME) :: units
   character(len=NF90_MAX_NAME), dimension(NF90_MAX_VAR_DIMS) :: dimname
   integer, dimension(NF90_MAX_VAR_DIMS) :: dimlens
   integer :: xtype         ! netCDF variable type (NF90_double, etc.)
   integer :: numdims       ! number of dims - excluding TIME
   integer :: numvertical   ! number of vertical levels in variable
```

(continues on next page)

(continued from previous page)

```
   integer :: numcells      ! number of horizontal locations (typically cell centers)
   logical :: ZonHalf       ! vertical coordinate has dimension nVertLevels
   integer :: varsize       ! prod(dimlens(1:numdims))
   integer :: index1        ! location in dart state vector of first occurrence
   integer :: indexN        ! location in dart state vector of last  occurrence
   integer :: dart_kind
   character(len=paramname_length) :: kind_string
   logical  :: clamping     ! does variable need to be range-restricted before
   real(r8) :: range(2)     ! being stuffed back into MPAS analysis file.
end type progvartype

type(progvartype), dimension(max_state_variables) :: progvar
```

The variables are simply read from the MPAS analysis file and stored in the DART state vector such that all quantities for one variable are stored contiguously. Within each variable; they are stored vertically-contiguous for each horizontal location. From a storage standpoint, this would be equivalent to a Fortran variable dimensioned x(nVertical,nHorizontal,nVariables). The fastest-varying dimension is vertical, then horizontal, then variable … naturally, the DART state vector is 1D. Each variable is also stored this way in the MPAS analysis file.

The DART interface for MPAS (atm)

was compiled with the gfortran 4.2.3 compilers and run on a Mac.
The DART components were built with the following `mkmf.template` settings:

```
FC = gfortran
LD = gfortran
NETCDF = /Users/thoar/GNU
INCS = -I${NETCDF}/include
LIBS = -L${NETCDF}/lib -lnetcdf -lcurl -lhdf5_hl -lhdf5 -lz  -lm
FFLAGS = -O0 -fbounds-check -frecord-marker=4 -ffpe-trap=invalid $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

Converting between DART files and MPAS analysis files

is relatively straighforward. Given the namelist mechanism for determining the state variables and the MPAS history netCDF files exist, - everything that is needed is readily determined.
There are two programs - both require the list of MPAS variables to use in the DART state vector: the `mpas_vars_nml` namelist in the `input.nml` file. The MPAS file name being read and/or written is - in all instances - specified by the `model_nml:model_analysis_filename` variable in the `input.nml` namelist file.

PROGRAM model_to_dart for MPAS OCN	converts an MPAS analysis file (nominally named `mpas_analysis.nc`) into a DART-compatible file normally called `dart_ics` . We usually wind up linking the actual analysis file to a static name that is used by DART.
dart_to_model.f90	inserts the DART output into an existing MPAS analysis netCDF file by overwriting the variables in the analysis netCDF file. There are two different types of DART output files, so there is a namelist option to specify if the DART file has two time records or just one (if there are two, the first one is the 'advance_to' time, followed by the 'valid_time' of the ensuing state). `dart_to_model` updates the MPAS analysis file specified in `input.nml&model_nml:model_analysis_filename`. If the DART file contains an 'advance_to' time, separate control information is written to an auxiliary file that is used by the `advance_model.csh` script.

The header of an MPAS analysis file is presented below - simply for context. Keep in mind that **many** variables have been removed for clarity. Also keep in mind that the multi-dimensional arrays listed below have the dimensions reversed from the Fortran convention.

```
366 mirage2:thoar% ncdump -h mpas_analysis.nc
netcdf mpas_analysis {
dimensions:
        StrLen = 64 ;
        Time = UNLIMITED ; // (1 currently)
        nCells = 10242 ;                                    available in DART
        nEdges = 30720 ;                                    available in DART
        maxEdges = 10 ;
        maxEdges2 = 20 ;
        nVertices = 20480 ;                                 available in DART
        TWO = 2 ;
        THREE = 3 ;
        vertexDegree = 3 ;                                  available in DART
        FIFTEEN = 15 ;
        TWENTYONE = 21 ;
        R3 = 3 ;
        nVertLevels = 41 ;                                  available in DART
        nVertLevelsP1 = 42 ;                                available in DART
        nMonths = 12 ;
        nVertLevelsP2 = 43 ;
        nSoilLevels = 4 ;                                   available in DART
variables:
        char xtime(Time, StrLen) ;                          available in DART
        double latCell(nCells) ;                            available in DART
        double lonCell(nCells) ;                            available in DART
        double latEdge(nEdges) ;
        double lonEdge(nEdges) ;
        int indexToEdgeID(nEdges) ;
        double latVertex(nVertices) ;
        double lonVertex(nVertices) ;
        int indexToVertexID(nVertices) ;
        int cellsOnEdge(nEdges, TWO) ;
        int nEdgesOnCell(nCells) ;
        int nEdgesOnEdge(nEdges) ;
```

(continues on next page)

(continued from previous page)

```
        int edgesOnCell(nCells, maxEdges) ;
        int edgesOnEdge(nEdges, maxEdges2) ;
        double weightsOnEdge(nEdges, maxEdges2) ;
        double dvEdge(nEdges) ;
        double dcEdge(nEdges) ;
        double angleEdge(nEdges) ;
        double edgeNormalVectors(nEdges, R3) ;
        double cellTangentPlane(nEdges, TWO, R3) ;
        int cellsOnCell(nCells, maxEdges) ;
        int verticesOnCell(nCells, maxEdges) ;
        int verticesOnEdge(nEdges, TWO) ;
        int edgesOnVertex(nVertices, vertexDegree) ;
        int cellsOnVertex(nVertices, vertexDegree) ;      available in DART
        double kiteAreasOnVertex(nVertices, vertexDegree) ;
        double rainc(Time, nCells) ;
        double cuprec(Time, nCells) ;
        double cutop(Time, nCells) ;
        double cubot(Time, nCells) ;
        double relhum(Time, nCells, nVertLevels) ;
        double qsat(Time, nCells, nVertLevels) ;
        double graupelnc(Time, nCells) ;
        double snownc(Time, nCells) ;
        double rainnc(Time, nCells) ;
        double graupelncv(Time, nCells) ;
        double snowncv(Time, nCells) ;
        double rainncv(Time, nCells) ;
        double sr(Time, nCells) ;
        double surface_temperature(Time, nCells) ;
        double surface_pressure(Time, nCells) ;
        double coeffs_reconstruct(nCells, maxEdges, R3) ;
        double theta_base(Time, nCells, nVertLevels) ;
        double rho_base(Time, nCells, nVertLevels) ;
        double pressure_base(Time, nCells, nVertLevels) ;
        double exner_base(Time, nCells, nVertLevels) ;
        double exner(Time, nCells, nVertLevels) ;
        double h_divergence(Time, nCells, nVertLevels) ;
        double uReconstructMeridional(Time, nCells, nVertLevels) ;
        double uReconstructZonal(Time, nCells, nVertLevels) ;
        double uReconstructZ(Time, nCells, nVertLevels) ;
        double uReconstructY(Time, nCells, nVertLevels) ;
        double uReconstructX(Time, nCells, nVertLevels) ;
        double pv_cell(Time, nCells, nVertLevels) ;
        double pv_vertex(Time, nVertices, nVertLevels) ;
        double ke(Time, nCells, nVertLevels) ;
        double rho_edge(Time, nEdges, nVertLevels) ;
        double pv_edge(Time, nEdges, nVertLevels) ;
        double vorticity(Time, nVertices, nVertLevels) ;
        double divergence(Time, nCells, nVertLevels) ;
        double v(Time, nEdges, nVertLevels) ;
        double rh(Time, nCells, nVertLevels) ;
        double theta(Time, nCells, nVertLevels) ;
        double rho(Time, nCells, nVertLevels) ;
```

(continues on next page)

(continued from previous page)

```
        double qv_init(nVertLevels) ;
        double t_init(nCells, nVertLevels) ;
        double u_init(nVertLevels) ;
        double pressure_p(Time, nCells, nVertLevels) ;
        double tend_theta(Time, nCells, nVertLevels) ;
        double tend_rho(Time, nCells, nVertLevels) ;
        double tend_w(Time, nCells, nVertLevelsP1) ;
        double tend_u(Time, nEdges, nVertLevels) ;
        double qv(Time, nCells, nVertLevels) ;
        double qc(Time, nCells, nVertLevels) ;
        double qr(Time, nCells, nVertLevels) ;
        double qi(Time, nCells, nVertLevels) ;
        double qs(Time, nCells, nVertLevels) ;
        double qg(Time, nCells, nVertLevels) ;
        double tend_qg(Time, nCells, nVertLevels) ;
        double tend_qs(Time, nCells, nVertLevels) ;
        double tend_qi(Time, nCells, nVertLevels) ;
        double tend_qr(Time, nCells, nVertLevels) ;
        double tend_qc(Time, nCells, nVertLevels) ;
        double tend_qv(Time, nCells, nVertLevels) ;
        double qnr(Time, nCells, nVertLevels) ;
        double qni(Time, nCells, nVertLevels) ;
        double tend_qnr(Time, nCells, nVertLevels) ;
        double tend_qni(Time, nCells, nVertLevels) ;
```

6.134.3 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Consider yourself forewarned that character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
namelist /model_nml/  model_analysis_filename, &
          assimilation_period_days, assimilation_period_seconds, &
          model_perturbation_amplitude, output_state_vector, calendar, debug
```

This namelist is read in a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the model dynamical timestep. This also specifies the MPAS analysis file that will be read and/or written by the different program units.

Contents	Type	Description
model_analysis_filename	character(len=256) *[default: 'mpas_analysis.nc']*	Character string specifying the name of the MPAS analysis file to be read and/or written by the different program units.
output_state_vector	logical *[default: .true.]*	The switch to determine the form of the state vector in the output netCDF files. If `.true.` the state vector will be output exactly as DART uses it ... one long array. If `.false.`, the state vector is parsed into prognostic variables and output that way – much easier to use with 'ncview', for example.
assimilation_period_days	integer *[default: 1]*	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer *[default: 0]*	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
model_perturbation_amplitude	real(r8) *[default: 0.2]*	Reserved for future use.
calendar	character(len=32) *[default: 'Gregorian']*	Character string specifying the calendar being used by MPAS.
debug	integer *[default: 0]*	The switch to specify the run-time verbosity. `0` is as quiet as it gets. `> 1` provides more run-time messages. `> 5` provides ALL run-time messages.

Example namelist

```
&model_nml
   model_analysis_filename      = 'mpas_restart.nc';
   assimilation_period_days     = 0,
   assimilation_period_seconds  = 60,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .true.,
   calendar                     = 'Gregorian',
   debug                        = 0
   /
```

```
namelist /mpas_vars_nml/ mpas_state_variables
```

This namelist is read from `input.nml` and contains the list of MPAS variables that make up the DART state vector.

Contents	Type	Description
mpas_variable	character(len=NF90_MAX_NAME):: dimension(160) *[default: see example]*	The table that relates the GITM variables to use to build the DART state vector, and the corresponding DART kinds for those variables.

Example

The following mpas_vars_nml is just for demonstration purposes. You application will likely involve a different DART state vector.

```
&mpas_vars_nml
   mpas_state_variables = 'theta',                 'QTY_POTENTIAL_TEMPERATURE',
                          'uReconstructZonal',     'QTY_U_WIND_COMPONENT',
                          'uReconstructMeridional','QTY_V_WIND_COMPONENT',
                          'w',                     'QTY_VERTICAL_VELOCITY',
                          'qv',                    'QTY_VAPOR_MIXING_RATIO',
                          'qc',                    'QTY_CLOUDWATER_MIXING_RATIO',
                          'qr',                    'QTY_RAINWATER_MIXING_RATIO',
                          'qi',                    'QTY_ICE_MIXING_RATIO',
                          'qs',                    'QTY_SNOW_MIXING_RATIO',
                          'qg',                    'QTY_GRAUPEL_MIXING_RATIO'
                          'surface_pressure',      'QTY_SURFACE_PRESSURE'
   /
```

The variables are simply read from the MPAS analysis file and stored in the DART state vector such that all quantities for one variable are stored contiguously. Within each variable; they are stored vertically-contiguous for each horizontal location. From a storage standpoint, this would be equivalent to a Fortran variable dimensioned x(nVertical,nHorizontal,nVariables). The fastest-varying dimension is vertical, then horizontal, then variable … naturally, the DART state vector is 1D. Each variable is also stored this way in the MPAS analysis file.

6.134.4 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
obs_kind_mod
mpi_utilities_mod
random_seq_mod
```

Warning: DAReS staff began creating the MPAS_OCN interface to DART in preparation for the model's inclusion as the ocean component of the Community Earth System Model (CESM). The plans for including MPAS_OCN in CESM were abandoned and the Modular Ocean Model version 6 (MOM6) was included instead. Thus, the documentation on this page after this point describes an incomplete interface. Please contact DAReS staff by emailing dart@ucar.edu if you want to use DART with MPAS_OCN.

6.134.5 Public interfaces

Only a select number of interfaces used are discussed here. Each module has its own discussion of their routines.

Required interface routines

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

Unique interface routines

use model_mod, only :	get_gridsize
	restart_file_to_sv
	sv_to_restart_file
	get_gitm_restart_filename
	get_base_time
	get_state_time

use location_mod, only :	get_close_o bs

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

Interface routine descriptions

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector. Required.

model_size	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

adv_1step is not used for the gitm model. Advancing the model is done through the advance_model script. This is a NULL_INTERFACE, provided only for compatibility with the DART requirements.

x	State vector of length model_size.
time	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

get_state_meta_data returns metadata about a given element of the DART representation of the model state vector. Since the DART model state vector is a 1D array and the native model grid is multidimensional, get_state_meta_data returns information about the native model state vector representation. Things like the location, or the type of the variable (for instance: temperature, u wind component, ...). The integer values used to indicate different variable types in var_type are themselves defined as public interfaces to model_mod if required.

index_in	Index of state vector element about which information is requested.
location	Returns the 3D location of the indexed state variable. The location_ type comes from DART/assimilation_code/location/threed_sphere/location_mod.f90. Note that the lat/lon are specified in degrees by the user but are converted to radians internally.
var_type	Returns the type of the indexed state variable as an optional argument. The type is one of the list of supported observation types, found in the block of code starting ! Integer definitions for DART TYPES in DART/assimilation_code/modules/observations/obs_kind_mod.f90

The list of supported variables in DART/assimilation_code/modules/observations/obs_kind_mod.f90 is created by preprocess.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given a model state, model_interpolate returns the value of the desired observation type (which could be a state variable) that would be observed at the desired location. The interpolation method is either completely specified by the model, or uses some standard 2D or 3D scalar interpolation routines. Put another way, model_interpolate will apply the forward operator **H** to the model state to create an observation at the desired location.

If the interpolation is valid, istatus = 0. In the case where the observation operator is not defined at the given location (e.g. the observation is below the lowest model level, above the top level, or 'dry'), interp_val is returned as 0.0 and istatus = 1.

x	A model state vector.
location	Location to which to interpolate.
itype	Integer indexing which type of observation is desired.
obs_val	The interpolated value from the model.
istatus	Integer flag indicating the success of the interpolation. success == 0, failure == anything else

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

`get_model_time_step` returns the forecast length to be used as the "model base time step" in the filter. This is the minimum amount of time the model can be advanced by `filter`. *This is also the assimilation window*. All observations within (+/-) one half of the forecast length are used for the assimilation. In the `GITM` case, this is set from the namelist values for `input.nml&model_nml:assimilation_period_days, assimilation_period_seconds`.

`var`	Smallest time step of model.

call static_init_model()

`static_init_model` is called for runtime initialization of the model. The namelists are read to determine runtime configuration of the model, the grid coordinates, etc. There are no input arguments and no return values. The routine sets module-local private attributes that can then be queried by the public interface routines.
See the GITM documentation for all namelists in `gitm_in` . Be aware that DART reads the GITM `&grid_nml` namelist to get the filenames for the horizontal and vertical grid information as well as the topography information.
The namelists (all mandatory) are:
`input.nml&model_mod_nml`,
`gitm_in&time_manager_nml`,
`gitm_in&io_nml`,
`gitm_in&init_ts_nml`,
`gitm_in&restart_nml`,
`gitm_in&domain_nml`, and
`gitm_in&grid_nml`.

call end_model()

`end_model` is used to clean up storage for the model, etc. when the model is no longer needed. There are no arguments and no return values. The grid variables are deallocated.

call init_time(time)

```
type(time_type), intent(out) :: time
```

`init_time` returns the time at which the model will start if no input initial conditions are to be used. This is frequently used to spin-up models from rest, but is not meaningfully supported for the GITM model. The only time this routine would get called is if the `input.nml&perfect_model_obs_nml:start_from_restart` is .false., which is not supported in the GITM model.

time	the starting time for the model if no initial conditions are to be supplied. This is hardwired to 0.0

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

init_conditions returns default initial conditions for model; generally used for spinning up initial model states. For the GITM model it is just a stub because the initial state is always provided by the input files.

x	Initial conditions for state vector. This is hardwired to 0.0

ierr = nc_write_model_atts(ncFileID)

```
integer             :: nc_write_model_atts
integer, intent(in) :: ncFileID
```

nc_write_model_atts writes model-specific attributes to an opened netCDF file: In the GITM case, this includes information like the coordinate variables (the grid arrays: ULON, ULAT, TLON, TLAT, ZG, ZC, KMT, KMU), information from some of the namelists, and either the 1D state vector or the prognostic variables (SALT,TEMP,UVEL,VVEL,PSURF). All the required information (except for the netCDF file identifier) is obtained from the scope of the model_mod module. Both the input.nml and gitm_in files are preserved in the netCDF file as variables inputnml and gitm_in, respectively.

ncFileID	Integer file descriptor to previously-opened netCDF file.
ierr	Returns a 0 for successful completion.

nc_write_model_atts is responsible for the model-specific attributes in the following DART-output netCDF files: true_state.nc, preassim.nc, and analysis.nc.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer,                intent(in) :: ncFileID
real(r8), dimension(:), intent(in) :: statevec
integer,                intent(in) :: copyindex
integer,                intent(in) :: timeindex
integer                            :: ierr
```

nc_write_model_vars writes a copy of the state variables to a NetCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state. Whether the state vector is parsed into prognostic variables (SALT, TEMP, UVEL, VVEL, PSURF) or simply written

as a 1D array is controlled by `input.nml&model_mod_nml:output_state_vector`. If `output_state_vector = .true.` the state vector is written as a 1D array (the simplest case, but hard to explore with the diagnostics). If `output_state_vector = .false.` the state vector is parsed into prognostic variables before being written.

`ncFileID`	file descriptor to previously-opened netCDF file.
`statevec`	A model state vector.
`copyindex`	Integer index of copy to be written.
`timeindex`	The timestep counter for the given state.
`ierr`	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state, `pert_model_state` produces a perturbed model state. This is used to generate ensemble initial conditions perturbed around some control trajectory state when one is preparing to spin-up ensembles. Since the DART state vector for the GITM model contains both 'wet' and 'dry' cells, it is imperative to provide an interface to perturb **just** the wet cells (`interf_provided == .true.`).

The magnitude of the perturbation is wholly determined by `input.nml&model_mod_nml:model_perturbation_amplitude` and **utterly, completely fails**.

A more robust perturbation mechanism is needed. Until then, avoid using this routine by using your own ensemble of initial conditions. This is determined by setting `input.nml&filter_nml:start_from_restart = .false.`

`state`	State vector to be perturbed.
`pert_state`	The perturbed state vector.
`interf_provided`	Because of the 'wet/dry' issue discussed above, this is always `.true.`, indicating a model-specific perturbation is available.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, & num_close, close_ind [, dist])

```
type(get_close_type),              intent(in ) :: gc
type(location_type),               intent(in ) :: base_obs_loc
integer,                           intent(in ) :: base_obs_kind
type(location_type), dimension(:), intent(in ) :: obs
integer,             dimension(:), intent(in ) :: obs_kind
integer,                           intent(out) :: num_close
integer,             dimension(:), intent(out) :: close_ind
real(r8), optional,  dimension(:), intent(out) :: dist
```

Given a DART location (referred to as "base") and a set of locations, and a definition of 'close' - return a subset of locations that are 'close', as well as their distances to the DART location and their indices. This routine intentionally masks a routine of the same name in `location_mod` because we want to be able to discriminate against selecting 'dry land' locations.

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes, `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

For vertical distance computations, the general philosophy is to convert all vertical coordinates to a common coordinate. This coordinate type is defined in the namelist with the variable "vert_localization_coord".

`gc`	Structure to allow efficient identification of locations 'close' to a given location.
`base_obs_loc`	Single given location.
`base_obs_kind`	Kind of the single location.
`obs`	List of candidate locations.
`obs_kind`	Kind associated with candidate locations.
`num_close`	Number of locations close to the given location.
`close_ind`	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

ens_mean_for_model normally saves a copy of the ensemble mean to module-local storage. This is a NULL_INTERFACE for the GITM model. At present there is no application which requires module-local storage of the ensemble mean. No storage is allocated.

ens_mean	State vector containing the ensemble mean.

Unique interface routine descriptions

call get_gridsize(num_x, num_y, num_z)

```
integer, intent(out) :: num_x, num_y, num_z
```

get_gridsize returns the dimensions of the compute domain. The horizontal gridsize is determined from gitm_restart.nc.

num_x	The number of longitudinal gridpoints.
num_y	The number of latitudinal gridpoints.
num_z	The number of vertical gridpoints.

call restart_file_to_sv(filename, state_vector, model_time)

```
character(len=*),       intent(in)    :: filename
real(r8), dimension(:), intent(inout) :: state_vector
type(time_type),        intent(out)   :: model_time
```

restart_file_to_sv Reads a GITM netCDF format restart file and packs the desired variables into a DART state vector. The desired variables are specified in the gitm_vars_nml namelist.

filename	The name of the netCDF format GITM restart file.
state_vector	the 1D array containing the concatenated GITM variables.
model_time	the time of the model state. The last time in the netCDF restart file.

call sv_to_restart_file(state_vector, filename, statedate)

```
real(r8), dimension(:), intent(in) :: state_vector
character(len=*),       intent(in) :: filename
type(time_type),        intent(in) :: statedate
```

`sv_to_restart_file` updates the variables in the GITM restart file with values from the DART vector `state_vector`. The last time in the file must match the `statedate`.

`filename`	the netCDF-format GITM restart file to be updated.
`state_vector`	the 1D array containing the DART state vector.
`statedate`	the 'valid_time' of the DART state vector.

call get_gitm_restart_filename(filename)

```
character(len=*), intent(out) :: filename
```

`get_gitm_restart_filename` returns the name of the gitm restart file - the filename itself is in private module storage.

`filename`	The name of the GITM restart file.

time = get_base_time(filehandle)

```
integer,          intent(in) :: filehandle -OR-
character(len=*), intent(in) :: filehandle
type(time_type),  intent(out) :: time
```

`get_base_time` extracts the start time of the experiment as contained in the netCDF restart file. The file may be specified by either a character string or the integer netCDF fid.

time = get_state_time(filehandle)

```
integer,          intent(in) :: filehandle -OR-
character(len=*), intent(in) :: filehandle
type(time_type),  intent(out) :: time
```

`get_state_time` extracts the time of the model state as contained in the netCDF restart file. In the case of multiple times in the file, the last time is the time returned. The file may be specified by either a character string or the integer netCDF fid.

6.134.6 Files

filename	purpose
input.nml	to read the model_mod namelist
gitm_vars.nml	to read the `gitm_vars_nml` namelist
gitm_restart.nc	provides grid dimensions, model state, and 'valid_time' of the model state
true_state.nc	the time-history of the "true" model state from an OSSE
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.134.7 References

- none

6.134.8 Private components

N/A

6.135 PROGRAM `model_to_dart` for MPAS OCN

Attention: `mpas_ocn` was being developed with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `mpas_ocn` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.135.1 Overview

`model_to_dart` is the program that reads an MPAS OCN analysis file (nominally named `mpas_restart.nc`) and creates a DART state vector file (e.g. `perfect_ics, filter_ics, ...`). The MPAS analysis files have a **Time** UNLIMITED Dimension, which indicates there may (at some point) be more than one timestep in the file. The DART routines are currently designed to use the LAST timestep. If the Time dimension of length 3, we use the third timestep. A warning message is issued and indicates exactly the time being used.

`input.nml&mpas_vars_nml` defines the list of MPAS variables used to build the DART state vector. This namelist is more fully described in the *MPAS OCN* documentation. For example:

```
&mpas_vars_nml
   mpas_state_variables = 'temperature',  'QTY_TEMPERATURE',
                          'salinity',     'QTY_SALINITY',
                          'rho',          'QTY_DENSITY',
                          'u',            'QTY_EDGE_NORMAL_SPEED',
                          'h',            'QTY_SEA_SURFACE_HEIGHT'
```

(continues on next page)

(continued from previous page)

```
                          'tracer1',      'QTY_TRACER_CONCENTRATION'
   /
```

Conditions required for successful execution of `model_to_dart` are:

- a valid `input.nml` namelist file for DART which contains
- a MPAS OCN analysis file (nominally named `mpas_analysis.nc`).

Since this program is called repeatedly for every ensemble member, we have found it convenient to link the MPAS OCN analysis files to a static input filename (e.g. `mpas_analysis.nc`). The default DART filename is `dart_ics` - this may be moved or linked as necessary.

6.135.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_to_dart_nml
   model_to_dart_output_file = 'dart_ics'
   /
```

```
&model_nml
   model_analysis_filename  = 'mpas_analysis.nc'
   /

(partial namelist)
```

```
&mpas_vars_nml
   mpas_state_variables = '',
   mpas_state_bounds = '',
   /
```

The model_to_dart namelist includes:

Item	Type	Description
model_to_dart_output_file	character(len=128)	The name of the DART file containing the model state derived from the MPAS analysis file.

Two more namelists need to be mentioned. The model_nml namelist specifies the MPAS analysis file to be used as the source. The mpas_vars_nml namelist specifies the MPAS variables that will comprise the DART state vector.

For example:

```
&mpas_vars_nml
   mpas_state_variables = 'temperature',  'QTY_TEMPERATURE',
                          'salinity',     'QTY_SALINITY',
                          'rho',          'QTY_DENSITY',
                          'u',            'QTY_EDGE_NORMAL_SPEED',
                          'h',            'QTY_SEA_SURFACE_HEIGHT'
                          'tracer1',      'QTY_TRACER_CONCENTRATION'
   /
```

6.135.3 Modules used

```
assim_model_mod.f90
types_mod.f90
location_mod.f90
model_to_dart.f90
model_mod.f90
null_mpi_utilities_mod.f90
obs_kind_mod.f90
random_seq_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.135.4 Files read

- MPAS analysis file; `mpas_analysis.nc`
- DART namelist file; `input.nml`

6.135.5 Files written

- DART initial conditions/restart file; e.g. `dart_ics`

6.135.6 References

none

6.136 NCOMMAS

Attention: NCOMMAS works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using NCOMMAS with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.136.1 Overview

NCOMMAS 7_1 may now be used with the **Data Assimilation Research Testbed (DART)**.
Since NCOMMAS uses netCDF files or their restart mechanisms, it was possible to make a namelist-controlled set of variables to be included in the DART state vector. Each variable must also correspond to a DART "KIND"; required for the DART interpolate routines. For example,

```
&ncommas_vars_nml
   ncommas_state_variables = 'U',   'QTY_U_WIND_COMPONENT',
                             'V',   'QTY_V_WIND_COMPONENT',
                             'W',   'QTY_VERTICAL_VELOCITY',
                             'TH',  'QTY_POTENTIAL_TEMPERATURE',
                             'DBZ', 'QTY_RADAR_REFLECTIVITY',
                             'WZ',  'QTY_VERTICAL_VORTICITY',
                             'PI',  'QTY_EXNER_FUNCTION',
                             'QV',  'QTY_VAPOR_MIXING_RATIO',
                             'QC',  'QTY_CLOUDWATER_MIXING_RATIO',
                             'QR',  'QTY_RAINWATER_MIXING_RATIO',
                             'QI',  'QTY_ICE_MIXING_RATIO',
                             'QS',  'QTY_SNOW_MIXING_RATIO',
                             'QH',  'QTY_GRAUPEL_MIXING_RATIO'   /
```

These variables are then adjusted to be consistent with observations and stuffed back into the same netCDF restart files. Since DART is an ensemble algorithm, there are multiple restart files for a single restart time: one for each ensemble member. Creating the initial ensemble of states is an area of active research.
DART reads the grid information for NCOMMAS from the restart file specified in the DART `input.nml&model_nml:ncommas_restart_filename` and checks for the existence and shape of the desired state variables. This not only determines the size of the DART state vector, but DART also inherits much of the metadata for the variables from the NCOMMAS restart file. When DART is responsible for starting/stopping NCOMMAS, the information is conveyed through the command line arguments to NCOMMAS.

NCOMMAS 7_1

was compiled with the Intel 10.1 compilers and run on a linux cluster running SLES10. Initially, DART simply runs 'end-to-end' at every assimilation time, while the NCOMMAS ensemble mechanism is responsible for slicing and dicing the observation sequences and running `correct_ensemble` at the desired times. This is a complete role-reversal from the normal DART operation.

The DART components were built with the following settings:

```
MPIFC = mpif90
MPILD = mpif90
FC = ifort
LD = ifort
INCS = -I/coral/local/netcdf-3.6.3_intel-10.1-64/include
LIBS = -L/coral/local/netcdf-3.6.3_intel-10.1-64/lib -lnetcdf
FFLAGS = -pc64 -fpe0 -mp -O0 -vec-report0 $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

Converting between DART files and NCOMMAS restart files

is blissfully straighforward. Given the namelist mechanism for determining the state variables and the fact that the NCOMMAS netCDF file has all the grid and time information in it - everything that is needed can be readily determined.

There are two programs - both require the list of NCOMMAS variables to use in the DART state vector: the `ncommas_vars_nml` namelist in the `ncommas_vars.nml` file.

PROGRAM ncommas_to_dart	converts the ncommas restart file `ncommas_restart.nc` into a DART-compatible file normally called `dart_ics` . We usually wind up linking the restart file to a static name that is used by DART.
PROGRAM dart_to_ncommas	inserts the DART output into an existing ncommas restart netCDF file by overwriting the variables in the ncommas restart netCDF file. There are two different types of DART output files, so there is a namelist option to specify if the DART file has two time records or just one (if there are two, the first one is the 'advance_to' time, followed by the 'valid_time' of the ensuing state). `dart_to_ncommas` determines the ncommas restart file name from the `input.nml` `model_nml:ncommas_restart_filename`. If the DART file contains an 'advance_to' time, `dart_to_ncommas` creates a new `&time_manager_nml` for ncommas in a file called `ncommas_in.DART` which can be used to control the length of the ncommas integration.

Generating the initial ensemble

Creating the initial ensemble is an area of active research. The ncommas model cannot take one single model state and generate its own ensemble (typically done with pert_model_state).

The ensemble has to come from 'somewhere else'. At present, it may be sufficient to use a climatological ensemble; e.g., using the ncommas restarts for '1 January 00Z' from 50 consecutive years from a hindcast experiment.

There is **not yet** a `shell_scripts/MakeInitialEnsemble.csh` script to demonstrate how to convert a set of ncommas netCDF restart files into a set of DART files that have a consistent timestamp. If you simply convert each ncommas file to a DART file using `ncommas_to_dart`, each DART file will have a 'valid time' that reflects the ncommas time of that state - instead of an ensemble of states reflecting one single time. The restart_file_tool can be

used to overwrite the timestep in the header of each DART initial conditions file. The namelist for this program must look something like:

```
&restart_file_tool_nml
  input_file_name              = "dart_input",
  output_file_name             = "dart_output",
  ens_size                     = 1,
  single_restart_file_in       = .true.,
  single_restart_file_out      = .true.,
  write_binary_restart_files   = .true.,
  overwrite_data_time          = .true.,
  new_data_days                = 145731,
  new_data_secs                = 0,
  input_is_model_advance_file  = .false.,
  output_is_model_advance_file = .false.,
  overwrite_advance_time       = .false.,
  new_advance_days             = -1,
  new_advance_secs             = -1,
  gregorian_cal                = .true.  /
```

The time of days = *145731* seconds = *0* relates to 00Z 1 Jan 2000 in the DART world.

6.136.2 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Consider yourself forewarned that character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
namelist /model_nml/  ncommas_restart_filename, &
          assimilation_period_days, assimilation_period_seconds, &
          model_perturbation_amplitude, output_state_vector, calendar, debug
```

This namelist is read in a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the model dynamical timestep.

Contents	Type	Description
output_state_vector	logical *[default: .true.]*	The switch to determine the form of the state vector in the output netCDF files. If `.true.` the state vector will be output exactly as DART uses it ... one long array. If `.false.`, the state vector is parsed into prognostic variables and output that way – much easier to use with 'ncview', for example.
assimilation_period_days	integer *[default: 1]*	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer *[default: 0]*	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
model_perturbation_amplitude	real(r8) *[default: 0.2]*	Reserved for future use.
calendar	character(len=32) *[default: 'Gregorian']*	Character string specifying the calendar being used by NCOMMAS.
debug	integer *[default: 0]*	The switch to specify the run-time verbosity. `0` is as quiet as it gets. `> 1` provides more run-time messages. `> 5` provides ALL run-time messages. All values above 0 will also write a netCDF file of the grid information and perform a grid interpolation test.

Example model namelist

```
&model_nml
   ncommas_restart_filename     = 'ncommas_restart.nc';
   assimilation_period_days     = 1,
   assimilation_period_seconds  = 0,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .true.,
   calendar                     = 'Gregorian',
   debug                        = 0
   /
```

```
namelist /ncommas_vars_nml/ ncommas_state_variables
```

This namelist is read in a file called `ncommas_vars.nml` and contains the list of NCOMMAS variables that make up the DART state vector.

Contents	Type	Description
ncommas_state_variables	character(len=NF90_MAX_NAME):: dimension(160) *[default: see example]*	The table that relates the NCOMMAS variables to use to build the DART state vector, and the corresponding DART kinds for those variables.

Ncommas_vars namelist

```
&ncommas_vars_nml
   ncommas_state_variables = 'U',    'QTY_U_WIND_COMPONENT',
                             'V',    'QTY_V_WIND_COMPONENT',
                             'W',    'QTY_VERTICAL_VELOCITY',
                             'TH',   'QTY_POTENTIAL_TEMPERATURE',
                             'DBZ',  'QTY_RADAR_REFLECTIVITY',
                             'WZ',   'QTY_VERTICAL_VORTICITY',
                             'PI',   'QTY_EXNER_FUNCTION',
                             'QV',   'QTY_VAPOR_MIXING_RATIO',
                             'QC',   'QTY_CLOUDWATER_MIXING_RATIO',
                             'QR',   'QTY_RAINWATER_MIXING_RATIO',
                             'QI',   'QTY_ICE_MIXING_RATIO',
                             'QS',   'QTY_SNOW_MIXING_RATIO',
                             'QH',   'QTY_GRAUPEL_MIXING_RATIO'
  /
```

6.136.3 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
obs_kind_mod
mpi_utilities_mod
random_seq_mod
```

6.136.4 Public interfaces

Only a select number of interfaces used are discussed here. Each module has its own discussion of their routines.

Required interface routines

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

Unique interface routines

use model_mod, only :	get_gridsize
	restart_file_to_sv
	sv_to_restart_file
	get_ncommas_restart_filename
	get_base_time
	get_state_time

use location_mod, only :	get_close_o bs

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

Required interface routines

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector. Required.

model_size	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

adv_1step is not used for the ncommas model. Advancing the model is done through the advance_model script. This is a NULL_INTERFACE, provided only for compatibility with the DART requirements.

x	State vector of length model_size.
time	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

get_state_meta_data returns metadata about a given element of the DART representation of the model state vector. Since the DART model state vector is a 1D array and the native model grid is multidimensional, get_state_meta_data returns information about the native model state vector representation. Things like the location, or the type of the variable (for instance: temperature, u wind component, ...). The integer values used to indicate different variable types in var_type are themselves defined as public interfaces to model_mod if required.

index_in	Index of state vector element about which information is requested.
location	Returns the 3D location of the indexed state variable. The location_ type comes from DART/assimilation_code/location/threed_sphere/location_mod.f90. Note that the lat/lon are specified in degrees by the user but are converted to radians internally.
var_type	Returns the type of the indexed state variable as an optional argument. The type is one of the list of supported observation types, found in the block of code starting ! Integer definitions for DART TYPES in DART/assimilation_code/modules/observations/obs_kind_mod.f90

The list of supported variables in DART/assimilation_code/modules/observations/obs_kind_mod.f90 is created by preprocess.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given a model state, model_interpolate returns the value of the desired observation type (which could be a state variable) that would be observed at the desired location. The interpolation method is either completely specified by the model, or uses some standard 2D or 3D scalar interpolation routines. Put another way, model_interpolate will apply the forward operator **H** to the model state to create an observation at the desired location.
If the interpolation is valid, istatus = 0. In the case where the observation operator is not defined at the given location (e.g. the observation is below the lowest model level, above the top level, or 'dry'), interp_val is returned as 0.0 and istatus = 1.

x	A model state vector.
location	Location to which to interpolate.
itype	Integer indexing which type of observation is desired.
obs_val	The interpolated value from the model.
istatus	Integer flag indicating the success of the interpolation. success == 0, failure == anything else

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

get_model_time_step returns the forecast length to be used as the "model base time step" in the filter. This is the minimum amount of time the model can be advanced by filter. *This is also the assimilation window*. All observations within (+/-) one half of the forecast length are used for the assimilation. In the ncommas case, this is set from the namelist values for input.nml&model_nml:assimilation_period_days, assimilation_period_seconds.

var	Smallest time step of model.

call static_init_model()

static_init_model is called for runtime initialization of the model. The namelists are read to determine runtime configuration of the model, the grid coordinates, etc. There are no input arguments and no return values. The routine sets module-local private attributes that can then be queried by the public interface routines.

See the ncommas documentation for all namelists in ncommas_in . Be aware that DART reads the ncommas &grid_nml namelist to get the filenames for the horizontal and vertical grid information as well as the topography information.

The namelists (all mandatory) are:

input.nml&model_mod_nml,

ncommas_in&time_manager_nml,

ncommas_in&io_nml,

ncommas_in&init_ts_nml,

ncommas_in&restart_nml,

ncommas_in&domain_nml, and

ncommas_in&grid_nml.

call end_model()

end_model is used to clean up storage for the model, etc. when the model is no longer needed. There are no arguments and no return values. The grid variables are deallocated.

call init_time(time)

```
type(time_type), intent(out) :: time
```

init_time returns the time at which the model will start if no input initial conditions are to be used. This is frequently used to spin-up models from rest, but is not meaningfully supported for the ncommas model. The only time this routine would get called is if the input.nml&perfect_model_obs_nml:start_from_restart is .false., which is not supported in the ncommas model.

time	the starting time for the model if no initial conditions are to be supplied. This is hardwired to 0.0

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

init_conditions returns default initial conditions for model; generally used for spinning up initial model states. For the ncommas model it is just a stub because the initial state is always provided by the input files.

x	Initial conditions for state vector. This is hardwired to 0.0

ierr = nc_write_model_atts(ncFileID)

```
integer             :: nc_write_model_atts
integer, intent(in) :: ncFileID
```

nc_write_model_atts writes model-specific attributes to an opened netCDF file: In the ncommas case, this includes information like the coordinate variables (the grid arrays: ULON, ULAT, TLON, TLAT, ZG, ZC, KMT, KMU), information from some of the namelists, and either the 1D state vector or the prognostic variables (SALT,TEMP,UVEL,VVEL,PSURF). All the required information (except for the netCDF file identifier) is obtained from the scope of the model_mod module. Both the input.nml and ncommas_in files are preserved in the netCDF file as variables inputnml and ncommas_in, respectively.

ncFileID	Integer file descriptor to previously-opened netCDF file.
ierr	Returns a 0 for successful completion.

nc_write_model_atts is responsible for the model-specific attributes in the following DART-output netCDF files: true_state.nc, preassim.nc, and analysis.nc.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer,                intent(in) :: ncFileID
real(r8), dimension(:), intent(in) :: statevec
integer,                intent(in) :: copyindex
integer,                intent(in) :: timeindex
integer                            :: ierr
```

nc_write_model_vars writes a copy of the state variables to a NetCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state. Whether the state vector is parsed into prognostic variables (SALT, TEMP, UVEL, VVEL, PSURF) or simply written as a 1D array is controlled by input.nml&model_mod_nml:output_state_vector. If output_state_vector = .true. the state vector is written as a 1D array (the simplest case, but hard to explore with the diagnostics). If output_state_vector = .false. the state vector is parsed into prognostic variables before being written.

ncFileID	file descriptor to previously-opened netCDF file.
statevec	A model state vector.
copyindex	Integer index of copy to be written.
timeindex	The timestep counter for the given state.
ierr	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state, `pert_model_state` produces a perturbed model state. This is used to generate ensemble initial conditions perturbed around some control trajectory state when one is preparing to spin-up ensembles. Since the DART state vector for the ncommas model contains both 'wet' and 'dry' cells, it is imperative to provide an interface to perturb **just** the wet cells (`interf_provided == .true.`).
The magnitude of the perturbation is wholly determined by
`input.nml&model_mod_nml:model_perturbation_amplitude` and **utterly, completely fails**.
A more robust perturbation mechanism is needed. Until then, avoid using this routine by using your own ensemble of initial conditions. This is determined by setting `input.nml&filter_nml:start_from_restart = .false.`

`state`	State vector to be perturbed.
`pert_state`	The perturbed state vector.
`interf_provided`	Because of the 'wet/dry' issue discussed above, this is always `.true.`, indicating a model-specific perturbation is available.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, & num_close, close_ind [, dist])

```
type(get_close_type),              intent(in ) :: gc
type(location_type),               intent(in ) :: base_obs_loc
integer,                           intent(in ) :: base_obs_kind
type(location_type), dimension(:), intent(in ) :: obs
integer,             dimension(:), intent(in ) :: obs_kind
integer,                           intent(out) :: num_close
integer,             dimension(:), intent(out) :: close_ind
real(r8), optional,  dimension(:), intent(out) :: dist
```

Given a DART location (referred to as "base") and a set of locations, and a definition of 'close' - return a subset of locations that are 'close', as well as their distances to the DART location and their indices. This routine intentionally masks a routine of the same name in `location_mod` because we want to be able to discriminate against selecting 'dry land' locations.

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes, `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

For vertical distance computations, the general philosophy is to convert all vertical coordinates to a common coordinate. This coordinate type is defined in the namelist with the variable "vert_localization_coord".

`gc`	Structure to allow efficient identification of locations 'close' to a given location.
`base_obs_loc`	Single given location.
`base_obs_kind`	Kind of the single location.
`obs`	List of candidate locations.
`obs_kind`	Kind associated with candidate locations.
`num_close`	Number of locations close to the given location.
`close_ind`	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

`ens_mean_for_model` normally saves a copy of the ensemble mean to module-local storage. This is a NULL_INTERFACE for the ncommas model. At present there is no application which requires module-local storage of the ensemble mean. No storage is allocated.

`ens_mean`	State vector containing the ensemble mean.

Unique interface routines

call get_gridsize(num_x, num_y, num_z)

```
integer, intent(out) :: num_x, num_y, num_z
```

`get_gridsize` returns the dimensions of the compute domain. The horizontal gridsize is determined from `ncommas_restart.nc`.

`num_x`	The number of longitudinal gridpoints.
`num_y`	The number of latitudinal gridpoints.
`num_z`	The number of vertical gridpoints.

call restart_file_to_sv(filename, state_vector, model_time)

```
character(len=*),       intent(in)    :: filename
real(r8), dimension(:), intent(inout) :: state_vector
type(time_type),        intent(out)   :: model_time
```

`restart_file_to_sv` Reads a NCOMMAS netCDF format restart file and packs the desired variables into a DART state vector. The desired variables are specified in the `ncommas_vars_nml` namelist.

`filename`	The name of the netCDF format NCOMMAS restart file.
`state_vector`	the 1D array containing the concatenated NCOMMAS variables.
`model_time`	the time of the model state. The last time in the netCDF restart file.

call sv_to_restart_file(state_vector, filename, statedate)

```
real(r8), dimension(:), intent(in) :: state_vector
character(len=*),       intent(in) :: filename
type(time_type),        intent(in) :: statedate
```

`sv_to_restart_file` updates the variables in the NCOMMAS restart file with values from the DART vector `state_vector`. The last time in the file must match the `statedate`.

`filename`	the netCDF-format ncommas restart file to be updated.
`state_vector`	the 1D array containing the DART state vector.
`statedate`	the 'valid_time' of the DART state vector.

call get_ncommas_restart_filename(filename)

```
character(len=*), intent(out) :: filename
```

get_ncommas_restart_filename returns the name of the NCOMMAS restart file - the filename itself is in private module storage.

filename	The name of the NCOMMAS restart file.

time = get_base_time(filehandle)

```
integer,          intent(in) :: filehandle -OR-
character(len=*), intent(in) :: filehandle
type(time_type),  intent(out) :: time
```

get_base_time extracts the start time of the experiment as contained in the netCDF restart file. The file may be specified by either a character string or the integer netCDF fid.

time = get_state_time(filehandle)

```
integer,          intent(in) :: filehandle -OR-
character(len=*), intent(in) :: filehandle
type(time_type),  intent(out) :: time
```

get_state_time extracts the time of the model state as contained in the netCDF restart file. In the case of multiple times in the file, the last time is the time returned. The file may be specified by either a character string or the integer netCDF fid.

6.136.5 Files

filename	purpose
input.nml	to read the model_mod namelist
ncommas_vars.nml	to read the ncommas_vars_nml namelist
ncommas_restart.nc	provides grid dimensions, model state, and 'valid_time' of the model state
true_state.nc	the time-history of the "true" model state from an OSSE
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.136.6 References

- none

6.136.7 Private components

N/A

6.137 NOAH, NOAH-MP

6.137.1 Overview

The Manhattan-compliant version of the NOAH (technically NOAH-MP) supports NOAH-MP V3.6 and was largely updated in support of the data assimilation efforts with **wrf_hydro**. Experiments to perform data assimilation strictly with the NOAH-MP model have been run at the University of Texas at Austin by **Jingjing Liang**. We know other people are using DART and NOAH-MP. *however, we have not had the chance to update the documentation for the Manhattan release.* Consequently, we readily welcome any advice on how to improve the documentation and heartily encourage participation.

The NOAH **Land Surface Model** and **Data Assimilation Research Testbed (DART)** may now be used for assimilation experiments. The Classic or Lanai version should be considered an 'alpha' release – the code has only been tested for a single column configuration of NOAH.

Any of the variables in the NOAH restart file are available to be adjusted by the assimilation. The list of variables is set though a simple namelist interface. Since we are testing in a column configuration, there is no practical reason not to include all the variables necessary for a bit-for-bit restart: *SOIL_T*, *SOIL_M*, *SOIL_W*, *SKINTEMP*, *SNODEP*, *WEASD*, *CANWAT*, and *QFX*. These variables are then adjusted to be consistent with real observations and stuffed back into the same netCDF restart files. Since DART is an ensemble algorithm there are multiple restart files for a single restart time; one for each ensemble member. Creating the initial ensemble of land surface states is an area of active research. At present, it may be sufficient to use a climatological ensemble; e.g., using the restarts for '1 January 00Z' from 50 consecutive years.

There is reason to believe that the ensemble system will benefit from having unique atmospheric forcing for each ensemble member. A reasonable ensemble size is 50 or 80 or so.

DART reads the NOAH namelist `&NOAHLSM_OFFLINE` from a file called `namelist.hrldas` for several pieces of information. DART is responsible for starting/stopping NOAH; the restart information is conveyed through the NOAH namelist. **Unpleasant Reality #1 :** managing the tremendous number of hourly forcing files for every ensemble member is tedious. To facilitate matters, the DART/NOAH system uses a *single* netCDF file for each ensemble member that contains ALL of the forcing for that ensemble member.

dart_to_noah.f90	**updates** some or all of a NOAH restart file with the posterior DART state vector. There is the ability to selectively avoid updating the NOAH variables. This allows one to include NOAH variables in the DART state vector to aid in the application of observation operators, etc., without having to modify those variables in the NOAH restart file. [dart_to_noah.html]

Running a "Perfect Model" experiment ... OSSE

The example requires a basic knowledge of running NOAH. Four scripts are provided to demonstrate how to set up and run a perfect model experiment for a single site - with one caveat. You must provide your own initial ensemble for the experiment. The scripts are not intended to be black boxes. You are expected to read them and modify them to your own purpose.

The scripts assume the directory containing the DART executables is `${DARTDIR}/work`, and assume that the directory containing the NOAH executables is `${NOAHDIR}/Run`.

1. `shell_scripts/setup_pmo.csh`	This script stages the run of *program perfect_model_obs*. The directory where you run the script is called `CENTRALDIR` and will be the working directory for the experiment. The required input observation sequence file must be created in the normal DART way. This `obs_seq.in` file must exist before running this script. All the necessary data files and exectuables for a perfect model experiment get copied to CENTRALDIR so that you may run multiple experiments at the same time - in separate `CENTRALDIRs`.
2. `shell_scripts/run_pmo.csh`	very simply - it advances NOAH and applies the observation operator to put the "perfect" observations in an observation sequence file that can then be used for an assimilation.
3. `shell_scripts/setup_filter.csh`	builds upon the work of `setup_pmo.csh` and stages a PRE-EXISTING initial ensemble.
4. `shell_scripts/run_filter.csh`	Actually runs the filtering (assimilation) experiment.

Generating the initial ensemble

Creating the initial ensemble of soil moisture states is an area of active research. The ensemble must come from 'somewhere else'. At present, it may be sufficient to use a climatological ensemble; e.g., using the NOAH restarts for '1 January 00Z' from 50 consecutive years from a hindcast experiment. It may also be sufficient to take a single model state, replicate it N times and force each of the N instances with different atmospheric conditions for 'a long time'.

By The Way

Experience has shown that having a paired (unique) atmospheric forcing maintains the ensemble spread during an assimilation better than simply forcing all the ensemble members with one single atmospheric state.

DART has routines to perturb a single NOAH state and generate its own ensemble (typically done with `pert_model_state`), but this produces model states that are incompatible with NOAH. We are interested in adopting/adapting strategies to create sensible initial conditions for NOAH.

If you have an algorithm you believe will be useful, please contact us!

6.137.2 Observations

Some novel observations come from the Cosmic-ray Soil Moisture Observing System: COSMOS and are processed by DART routines in the `$DARTROOT/observations/COSMOS` directory.

DART has a very object-oriented approach to observation support. All observations that are intended to be supported must be preprocessed (see `$DARTROOT/preprocess/` into a single `obs_def_mod.f90` and `obs_kind_mod.f90` in the standard DART way.

Exploring the Output

There are Matlab® scripts for exploring the performance of the assimilation in observation-space (after running `obs_diag`). See `$DARTROOT/diagnostics/threed_sphere/obs_diag.html` to explore the *obs_seq.final* file) - use the scripts starting with `plot_`, i.e. `$DARTROOT/diagnostics/matlab/plot_*.m*`. As always, there are some model-specific items Matlab® will need to know about in `$DARTROOT/models/NOAH/matlab`.

The `Prior_Diag.nc` and `Posterior_Diag.nc` (and possibly `True_State.nc`) netCDF files have the model prognostic variables before and after the assimilation. The `./matlab` scripts for NOAH are under development.

It is also worthwhile to convert your `obs_seq.final` file to a netCDF format obs_sequence file with `obs_seq_to_netcdf`. See `$DARTROOT/obs_sequence/obs_seq_to_netcdf.html` and use any of the standard plots. Be aware that the COSMOS site-specific metadata will not get conveyed to the netCDF file.

6.137.3 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist. The standard values are shown below:

```
&model_nml
   lsm_model_choice             = 'noahMP_36'
   domain_shapefiles            = 'RESTART.2003051600_DOMAIN1_01'
   assimilation_period_days     =    0
   assimilation_period_seconds  = 3600
   model_perturbation_amplitude = 0.2
   perturb_distribution         = 'gaussian'
   debug                        = 0
   polar                        = .false.
   periodic_x                   = .false.
   periodic_y                   = .false.
   lsm_variables = 'SOIL_T',   'QTY_SOIL_TEMPERATURE',   '0.0',  'NA', 'UPDATE',
                   'SMC',      'QTY_SOIL_MOISTURE',      '0.0', '1.0', 'UPDATE',
                   'WA',       'QTY_AQUIFER_WATER',      '0.0',  'NA', 'UPDATE',
                   'SNEQV',    'QTY_SNOW_WATER',         '0.0',  'NA', 'UPDATE',
                   'FSNO',     'QTY_SNOWCOVER_FRAC',     '0.0', '1.0', 'UPDATE'
  /
```

This namelist is read from a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the NOAH model dynamical timestep.

Item	Type	Description
lsm_model_choice	character(len=256)	The version of the NOAH namelist to read
domain_shapefiles	an array of character(len=256)	The name of the NOAH RESTART files to use to specify the shape of the variables and geographic metadata. One per domain.
assimilation_period_days	integer	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
model_perturbation_amplitude	real(r8)	The amount of noise to add when trying to perturb a single state vector to create an ensemble. Only used when `input.nml` is set with `&filter_nml:start_from_restart = .false.`. See also *Generating the initial ensemble*. units: standard deviation of the specified distribution the mean at the value of the state vector element.
perturb_distribution	character(len=256)	The switch to determine the distribution of the perturbations used to create an initial ensemble from a single model state. Valid values are : `lognormal` or `gaussian`
periodic_x	logical	Switch to determine if the configuration has periodicity in the X direction.
periodic_y	logical	Switch to determine if the configuration has periodicity in the Y direction.
lsm_variables	character(len=32):: dimension(5,40)	The list of variable names in the NOAH restart file to use to create the DART state vector and their corresponding DART kind. [default: see example below]

The columns of `lsm_variables` needs some explanation. Starting with the column 5, `UPDATE` denotes whether or not to replace the variable with the Posterior (i.e. assimilated) value. Columns 3 and 4 denote lower and upper bounds that should be enforced when writing to the files used to restart the model. These limits are not enforced for the DART diagnostic files. Column 2 specifies the relationship between the netCDF variable name for the model and the corresponding DART QUANTITY.

The DART 'QTY's match what the `model_mod` knows how to interpolate, so you can't just add a new quantity and expect it to work. There is a complex interplay between `obs_def_mod` and `preprocess`, and `model_mod` that defines what QUANTITIES are supported. There is only a single QUANTITY that works with each variable and the example shows the current QUANTITYs. Support for these QUANTITYs was provided by running `preprocess` with the following namelist settings:

```
&preprocess_nml
    input_obs_kind_mod_file = '../../../assimilation_code/modules/observations/DEFAULT_
↪obs_kind_mod.F90'
   output_obs_kind_mod_file = '../../../assimilation_code/modules/observations/obs_kind_
↪mod.f90'
     input_obs_def_mod_file = '../../../observations/forward_operators/DEFAULT_obs_def_
↪mod.F90'
    output_obs_def_mod_file = '../../../observations/forward_operators/obs_def_mod.f90'
   input_files              = '../../../observations/forward_operators/obs_def_land_mod.
↪f90',
                              '../../../observations/forward_operators/obs_def_COSMOS_
↪mod.f90',
                              '../../../observations/forward_operators/obs_def_GRACE_mod.
↪f90'
```

(continues on next page)

(continued from previous page)

```
/
```

NOAHLSM_OFFLINE NAMELIST

```
namelist /NOAHLSM_OFFLINE/
   hrldas_constants_file, &
   indir, outdir,  &
   restart_filename_requested, &
   khour,  kday, &
   forcing_timestep, &
   noah_timestep,  &
   output_timestep, &
   restart_frequency_hours, &
   split_output_count, &
   nsoil, &
   zsoil
```

The remaining variables are not used by DART - but are used by NOAH. Since DART verifies namelist accuracy, any namelist entry in NOAHLSM_OFFLINE that is not in the following list will cause a FATAL DART ERROR.

```
zlvl, zlvl_wind, iz0tlnd, sfcdif_option, update_snow_from_forcing,
start_year, start_month, start_day, start_hour, start_min,
external_fpar_filename_template, external_lai_filename_template,
subwindow_xstart, subwindow_xend, subwindow_ystart, subwindow_yend
```

This namelist is read from a file called `namelist.hrldas`. This namelist is the same one that is used by NOAH. The values are explained in full in the NOAH documentation. Only the namelist variables of interest to DART are discussed. All other namelist variables are ignored by DART - but mean something to NOAH.

Item	Type	Description
hrldas_constants_file	character(len=256)	The name of the netCDF file containing the grid information. [default: `wrfinput`]
indir	character(len=256)	The DART/NOAH environment requires all the input files to be in the current working directory. [default: `'.'`]
outdir	character(len=256)	The DART/NOAH environment requires all output files are in the current working directory. [default: `'.'`]
restart_filename_requested	character(len=256)	The name of the file containing the grid information. The default value is implicitly used by the scripting examples. Change at your own risk. [default: `'restart.nc'`]
khour	integer	The duration (in hours) of the model integration. [default: `1`]
kday	integer	The duration (in days) of the model integration. [default: `0`]
forcing_timestep	integer	The timestep (in seconds) of the atmospheric forcing. [default: `3600`]
noah_timestep	integer	The internal (dynamical) timestep (in seconds). [default: `3600`]
output_timestep	integer	The output interval (in seconds). [default: `3600`]
restart_frequency_hours	integer	How often the NOAH restart files get written. [default: `1`]
split_output_count	integer	should be 1 or bad things happen. [default: `1`]
nsoil	integer	The number of soil interfaces. As I understand it, NOAH requires this to be 4. [default: `4`]
zsoil	integer(NSOLDX)	The depth (in meters) of the soil interfaces. [default: `-0.1, -0.4, -1.0, -2.04`]

Example

Note: the FORCING_FILE_DIRECTORY line is not required by NOAH but IS required by DART - specifically in the *advance_model.csh* script.

```
### THIS IS FOR DART ###
FORCING_FILE_DIRECTORY = "/path/to/your/forcing/files"

&NOAHLSM_OFFLINE
   HRLDAS_CONSTANTS_FILE = "wrfinput"
   INDIR  = "."
   OUTDIR = "."
   RESTART_FILENAME_REQUESTED = "restart.nc"
   KHOUR                  = 1
   FORCING_TIMESTEP       = 3600
   NOAH_TIMESTEP          = 3600
   OUTPUT_TIMESTEP        = 3600
   RESTART_FREQUENCY_HOURS = 1
   SPLIT_OUTPUT_COUNT     = 1
   NSOIL=4
   ZSOIL(1) = -0.10
   ZSOIL(2) = -0.40
   ZSOIL(3) = -1.00
   ZSOIL(4) = -2.00
/
```

6.137.4 Input Files

filename	purpose
input.nml	to read the model_mod namelist
namelist.hrldas	to read the NOAHLSM_OFLINE namelist
wrfinput	provides NOAH grid information
&model_nml:noah_netcdf_filename	the RESTART file containing the NOAH model state.

6.138 null_model

6.138.1 Overview

DART interface module for the 'null_model'. This model provides very simple models for evaluating filtering algorithms. It can provide simple linear growth around a fixed point, a random draw from a Gaussian, or combinations of the two. Namelist controls can set the width of the Gaussian and change both the model advance method and the expected observation interpolation method.

The 18 public interfaces are standardized for all DART compliant models. These interfaces allow DART to advance the model, get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for model state variables.

6.138.2 Namelist

The &model_nml namelist is read from the input.nml file. Namelists start with an ampersand & and terminate with a slash /. Character strings that contain a / must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size           = 2,
   delta_t              = 0.05,
   time_step_days       = 0,
   time_step_seconds    = 3600
   noise_amplitude      = 0.0_r8
   advance_method       = 'simple'
   interpolation_method = 'standard'
/
```

Description of each namelist entry

Item	Type	Description
model_size	integer	Model size.
delta_t	real(r8)	Internal model timestep parameter.
time_step_days	integer	Minimum model advance time in days.
time_step_seconds	integer	Minimum model advance time in seconds.
noise_amplitude	real(r8)	If greater than 0.0 sets the standard deviation of the added Gaussian noise during the model advance.
advance_method	character(64)	Controls the model advance method. The default is 'simple' timestepping. A 4-step Runga Kutta method can be selected with the string 'rk'.
interpolation_method	character(64)	Controls how the expected value of an observation is computed. The default is 'standard' which uses a linear interpolation between the two surrounding model points. Other options include 'square' which returns the square of the computed value, 'opposite_side' which adds on a value from the opposite side of the cyclical domain, and 'average' which averages 15 points to get the expected value. Model size should be > 15 to use the last option.

6.138.3 Files

filename	purpose
input.nml	to read the model_mod namelist
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.139 PBL_1D

6.139.1 Overview

The PBL_1D directory has been deprecated in favor of using the WRF/DART model interface. There is now support for WRF single column mode built into the standard model_mod in that directory.

If you are interested in more information on this configuration, please email us at dart@ucar.edu.

If you really want the files that used to be in this directory, check them out from the Kodiak release of DART.

6.140 pe2lyr

Attention: `pe2lyr` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `pe2lyr` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.140.1 Overview

DART standard interfaces for a two-layer isentropic primitive equation model.

The 16 public interfaces are standardized for all DART compliant models. These interfaces allow DART to advance the model, get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for model state variables.

This model is a 2-layer, isentropic, primitive equation model on a sphere. TODO: add more detail here, including equations, etc.

Contact: Jeffrey.S.Whitaker@noaa.gov

6.140.2 Other modules used

```
types_mod
time_manager_mod
utilities_mod
random_seq_mod
threed_sphere/location_mod
```

6.140.3 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the size of the model as an integer. For this model the default grid size is 96 (lon) by 48 (lat) by 2 levels, and 3 variables (U, V, Z) at each grid location, for a total size of 27,648. There are alternative include files which, if included at compile time instead of the default file, defines a grid at twice and 4 times this resolution. They have corresponding truncation values of T63 and T127 (the default grid uses T31).

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Advances the model for a single time step. The time associated with the initial model state is also input although it is not used for the computation.

`x`	State vector of length model_size.
`time`	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Returns metadata about a given element, indexed by `index_in`, in the model state vector. The `location` defines where the state variable is located.
For this model, the default grid is a global lat/lon grid, 96 (lon) by 48 (lat) by 2 levels. The variable types are U, V, and Z:

- 1 = TYPE_u
- 2 = TYPE_v
- 901 = TYPE_z

Grids at twice and 4 times the resolution can be compiled in instead by using one of the alternative header files (see `resolt31.h` (the default), `resolt63.h`, and `resolt127.h`).

`index_in`	Index of state vector element about which information is requested.
`location`	The location of state variable element.
var_type	The type of the state variable element.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
```

(continues on next page)

(continued from previous page)

```
real(r8),              intent(out) :: obs_val
integer,               intent(out) :: istatus
```

Given a state vector, a location, and a model state variable type, interpolates the state variable field to that location and returns the value in obs_val. The istatus variable is always returned as 0 (OK).

x	A model state vector.
location	Location to which to interpolate.
itype	Type of state field to be interpolated.
obs_val	The interpolated value from the model.
istatus	Integer value returning 0 for successful, other values can be defined for various failures.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the the time step of the model; the smallest increment in time that the model is capable of advancing the state in a given implementation. For this model the default value is 20 minutes (1200 seconds), but also comes with header files with times steps of 10 and 5 minutes (for higher grid resolution and truncation constants).

var	Smallest time step of model.

call static_init_model()

Used for runtime initialization of a model, for instance calculating storage requirements, initializing model parameters, etc. This is the first call made to a model by any DART compliant assimilation routines.
In this model, it allocates space for the grid, and initializes the grid locations, data values, and various parameters, including spherical harmonic weights.

call end_model()

A stub since the pe2lyr model does no cleanup.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns the time at which the model will start if no input initial conditions are to be used. This model sets the time to 0.

time	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns default initial conditions for model; generally used for spinning up initial model states. This model sets the default state vector based on the initialized fields in the model. (TODO: which are what?)

x	Initial conditions for state vector.

ierr = nc_write_model_atts(ncFileID)

```
integer             :: nc_write_model_atts
integer, intent(in) :: ncFileID
```

This routine writes the model-specific attributes to a netCDF file. This includes coordinate variables and any metadata, but NOT the model state vector. This model writes out the data as U, V, and Z arrays on a lat/lon/height grid, so the attributes are organized in the same way.

ncFileID	Integer file descriptor to previously-opened netCDF file.
ierr	Returns a 0 for successful completion.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                             :: nc_write_model_vars
integer,                intent(in) :: ncFileID
real(r8), dimension(:), intent(in) :: statevec
integer,                intent(in) :: copyindex
integer,                intent(in) :: timeindex
```

This routine writes the model-specific state vector (data) to a netCDF file. This model writes out the data as U, V, and Z arrays on a lat/lon/height grid.

ncFileID	file descriptor to previously-opened netCDF file.
statevec	A model state vector.
copyindex	Integer index of copy to be written.
timeindex	The timestep counter for the given state.
ierr	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state vector, perturbs this vector. Used to generate initial conditions for spinning up ensembles. This model has no code to generate these values, so it returns `interf_provided` as .false. and the default algorithms in filter are then used by the calling code.

state	State vector to be perturbed.
pert_state	Perturbed state vector
interf_provided	Returned false; interface is not implemented.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

In distance computations any two locations closer than the given `maxdist` will be considered close by the `get_close_obs()` routine. Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

gc	The get_close_type which stores precomputed information about the locations to speed up searching
maxdist	Anything closer than this will be considered close.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Given a location and kind, compute the distances to all other locations in the `obs` list. The return values are the number of items which are within maxdist of the base, the index numbers in the original obs list, and optionally the distances. The `gc` contains precomputed information to speed the computations.
Pass-through to the 3-D sphere locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

Stub only. Not needed by this model.

`ens_mean`	State vector containing the ensemble mean.

This model currently has no values settable by namelist.

6.140.4 Files

- The model source is in pe2lyr_mod.f90, and the spherical harmonic code is in spharmt_mod.f90. The various resolution settings are in resolt31.h, resolt63.h, and resolt127.h.

6.140.5 References

Zou, X., Barcilon, A., Navon, I.M., Whitaker, J., Cacuci, D.G.. 1993: An Adjoint Sensitivity Study of Blocking in a Two-Layer Isentropic Model. Monthly Weather Review: Vol. 121, No. 10, pp. 2833-2857.

6.140.6 Private components

N/A

6.141 POP

6.141.1 Overview

This document describes the DART interface to the Parallel Ocean Program (POP). It covers the *Development history* of the interface with two implementations of POP:

- the Los Alamos National Laboratory Parallel Ocean Program (LANL POP), and
- the Community Earth System Model Parallel Ocean Program 2 (CESM POP2; Smith et al. 2010[1]).

This document also provides *Detailed instructions for using DART and CESM POP2 on NCAR's supercomputer*, including information about the availability of restart files for *Creating an initial ensemble* of model states and *Observation sequence files* for assimilation.

6.141.2 Development History

When the DART interface to POP was originally developed circa 2009-2010, the interface worked with both the LANL POP and CESM POP2 implementations of POP.

LANL POP

In years subsequent to the initial development of the DART interface, the Computer, Computational, and Statistical Sciences Division at LANL transitioned from using POP as their primary ocean model to using the Model for Prediction Across Scales-Ocean (MPAS-Ocean). Thus it became difficult for staff in the Data Assimilation Research Section (DAReS) at NCAR to maintain access to the LANL POP source code. As a result, LANL POP has been tested using DART's Lanai framework but has not been tested using DART's Manhattan framework. If you intend to use LANL POP with DART Manhattan, contact DAReS staff for assistance by emailing dart@ucar.edu.

[1] Smith, R., and Coauthors, 2010: The Parallel Ocean Program (POP) Reference Manual Ocean Component of the Community Climate System Model (CCSM) and Community Earth System Model (CESM). National Center for Atmospheric Research, http://www.cesm.ucar.edu/ models/cesm1.0/pop2/doc/sci/POPRefManual.pdf.

CESM POP2

The NCAR implementation of POP, CESM POP2, has been used extensively with DART throughout multiple generations of NCAR's supercomputer (Bluefire, Yellowstone & Cheyenne) and multiple iterations of NCAR's earth system model (CCSM4, CESM1 and CESM2). CESM POP2 is supported under DART's Manhattan framework.

For DART's CESM POP2 interface, the CESM Interactive Ensemble facility is used to manage the ensemble and the Flux Coupler is responsible for stopping POP2 at the times required to perform an assimilation. CESM runs continuously and all of the DART routines run at each assimilation time.

6.141.3 Detailed instructions for using DART and CESM POP2 on NCAR's supercomputer

If you're using NCAR's supercomputer, you can run the setup scripts after making minor edits to set details that are specific to your project. The setup scripts create a CESM case in which POP is configured using a 1° horizontal grid, and uses the eddy parametrization of Gent and McWilliams (1990).[2] The CICE model is active and atmospheric forcing is provided by the CAM6 DART Reanalysis.

The filesystem attached to NCAR's supercomputer is known as the Globally Accessible Data Environment (GLADE). All filepaths on GLADE have the structure:

```
/glade/*
```

If you aren't using NCAR's supercomputer, take note of when the `/glade/` filepath is present in the setup scripts, since this will indicate sections that you must alter in order to get the scripts to work on your supercomputer. Additionally, you'll need to generate your own initial condition and observation sequence files or you'll need to copy these files from GLADE. If you want to copy these files from GLADE and don't have access, contact DAReS staff by emailing dart@ucar.edu for assistance.

6.141.4 Summary

To use DART and CESM POP2 on NCAR's supercomputer, you will need to complete the following steps.

1. Configure the scripts for your specific experiment by editing `DART_params.csh`.
2. Stage your initial ensemble using `copy_POP_JRA_restarts.py`.
3. Run the appropriate DART setup script to create and build the CESM case.

If the DART setup script runs to completion, it will print instructions to the screen. Follow these instructions to submit your case.

6.141.5 Shell scripts

Since CESM requires many third-party modules in order to compile, it is often difficult to compile older versions of CESM because the older modules become unavailable. You should attempt to use the most recent setup scripts. The Discuss CESM bulletin board specifies which releases of CESM are supported.

The setup scripts are stored in:

```
DART/models/POP/shell_scripts
```

in subdirectories that correspond releases of CESM. For example:

[2] Gent, P. R., and J. C. McWilliams, 1990: Isopycnal Mixing in Ocean Circulation Models. *Journal of Physical Oceanography*, **20**, 150–155, doi:10.1175/1520-0485(1990)020<0150:IMIOCM>2.0.CO;2.

```
DART/models/POP/shell_scripts/cesm2_1
```

contains scripts that should be used with CESM releases 2.1.0-2.1.3.

copy_POP_JRA_restarts.py

This script stages an intial ensemble of POP2 restart files by copying files from a prior experiment run by *Who Kim*. Thanks Who!

These restart files can be used as an initial ensemble of model states. The files are kept in a directory on GLADE that is owned by the Climate and Global Dynamics (CGD) Ocean Section:

```
/glade/campaign/cgd/oce/people/whokim/csm/g210.G_JRA.v14.gx1v7.01
```

Unless you're already a member of the CGD Ocean Section, you must be granted access to this directory by CISL. Use the Service Desk to request permission. If you're unable to get permission, contact DAReS staff for assistance by emailing dart@ucar.edu.

Filepaths beginning with `/glade/campaign/*` can't be accessed from NCAR's supercomputer nodes. You must log on to NCAR's data visualization computer to copy files from `/glade/campaign/*`.

This python script was created by *Dan Amrhein*. Thanks Dan!

Script name	Description
copy_POP_JRA_restarts.py	This script copies restart files from the g210.G_JRA.v14.gx1v7.01 experiment that are saved in campaign storage. You must be granted access to the CGD Ocean Section campaign storage directory and be logged on to NCAR's data visualization computer in order to run this script. The assignment of the `stagedir` variable in this script should match the assignment of the `stagedir` variable in `DART_params.csh`.

In order to use this script, log in to NCAR's data visualization computer and use python to run the script. For example:

```
$ cd DART/models/POP/shell_scripts/cesm2_1
$ python copy_POP_JRA_restarts.py
```

DART_params.csh

This is the essential script you must edit to get your cases to build properly. While you need to configure this script, you don't need to run this script. It is run by the setup scripts.

Script name	Description
DART_params.csh	This script contains most, if not all, of the variables that you need to set in order to build and run cases. You must read this file carefully and configure the variables to match your needs. The assignment of the `stagedir` variable in this script should match the assignment of the `stagedir` variable in `copy_POP_JRA_restarts.py`.

Setup scripts

These are the primary scripts used to setup CESM cases in which data assimilation is enabled in POP2. The only variable that you might need to set in these scripts is the `extra_string` variable. It is appended to the end of the CESM case name. You can use it to differentiate experiments with the same configuration.

Script name	Description
`setup_CESM_perfect_model.csh`	This script creates a CESM case with a single model instance in order to run DART's `perfect_model_obs` program to collect observations from the model run.
`setup_CESM_hybrid_ensemble.csh`	This script creates a CESM case with multiple model instances in order to run DART's `filter` program to complete assimilation.

After configuring your experiment in `DART_params.csh`, you can setup a case by running these scripts. For example, to setup an assimilation experiment:

```
$ cd DART/models/POP/shell_scripts/cesm2_1
$ ./setup_CESM_hybrid_ensemble.csh
```

If the setup scripts run to completion, they will print instructions that you can follow to use CESM's case submit tool to begin a model integration.

CESM_DART_config.csh

This script is copied by the setup scripts into the CESM case directory. It configures CESM to run DART.

Script name	Description
`CESM_DART_config.csh`	This script is copied into the CESM case directory where it configures CESM to run DART.

Runtime scripts

These scripts are copied into the CESM case directory. They are called by CESM and contain the logic to run DART's `perfect_model_obs` or `filter` programs. You shouldn't need to run these scripts directly, unless they exit before completion and halt a CESM integration. In this case you may need to run the script directly to complete an assimilation in order to continue the integration.

Script name	Description
`perfect_model.csh`	This script runs `perfect_model_obs` to collect synthetic data in a single-instance CESM case.
`assimilate.csh`	This script runs `filter` to perform assimilation in a multi-instance CESM case.

6.141.6 Other files needed for assimilation

Creating an initial ensemble

Karspeck et al. (2013)[3] find that an ensemble of 1 January model states selected from a multi-decade free-running integration of POP2 can be used as an initial ensemble.

If you have access to CGD's Ocean Section directory on `/glade/campaign` you can use the *copy_POP_JRA_restarts.py* script to stage a collection of POP restart files from Who Kim's multi-century `g210.G_JRA.v14.gx1v7.01` experiment to serve as an initial ensemble. This experiment uses the JRA-55 dataset for atmospheric forcing (Tsujino et al. 2018[4]).

Observation sequence files

When `setup_CESM_hybrid_ensemble.csh` is used to create an assimilation experiment, `DART_params.csh` configures the experiment to assimilate observation sequence files from the World Ocean Database 2013 (WOD13; Boyer et al. 2013[5]).

The WOD13 dataset comprises data from 2005-01-01 to 2016-12-31 and contains the following observation types:

FLOAT_SALINITY	FLOAT_TEMPERATURE
DRIFTER_SALINITY	DRIFTER_TEMPERATURE
GLIDER_SALINITY	GLIDER_TEMPERATURE
MOORING_SALINITY	MOORING_TEMPERATURE
BOTTLE_SALINITY	BOTTLE_TEMPERATURE
CTD_SALINITY	CTD_TEMPERATURE
XCTD_SALINITY	XCTD_TEMPERATURE
APB_SALINITY	APB_TEMPERATURE
XBT_TEMPERATURE	

The W0D13 observations have already been converted into DART's observation sequence file format by *Fred Castruccio*. Thanks Fred! The files are stored in the following directory on GLADE:

```
/glade/p/cisl/dares/Observations/WOD13
```

The subdirectories are formatted in `YYYYMM` order.

Observation sequence files converted from the World Ocean Database 2009 (WOD09; Johnson et al. 2009[6]), which comprises data from 1960-01-01 to 2008-12-31, are also stored in the following directory on GLADE:

```
/glade/p/cisl/dares/Observations/WOD09
```

These observation sequence files can be assimilated by changing the `BASEOBSDIR` variable in `DART_params.csh`.

DART extracts the following variables from the POP2 restart files and adjusts them to be consistent with the observations: `SALT_CUR`, `TEMP_CUR`, `UVEL_CUR`, `VVEL_CUR`, and `PSURF_CUR`.

[3] Karspeck, A., Yeager, S., Danabasoglu, G., Hoar, T. J., Collins, N. S., Raeder, K. D., Anderson, J. L, Tribbia, J. 2013: An ensemble adjustment Kalman filter for the CCSM4 ocean component. *Journal of Climate*, **26**, 7392-7413, doi:10.1175/JCLI-D-12-00402.1.

[4] Tsujino, H., Urakawa, S., Nakano, H., Small, R. J., Kim, W. M., Yeager, S. G., … Yamazaki, D., 2018: JRA-55 based surface dataset for driving ocean-sea-ice models (JRA55-do). *Ocean Modelling*, **130**, 79-139, doi:10.1016/j.ocemod.2018.07.002.

[5] Boyer, T.P., J. I. Antonov, O. K. Baranova, C. Coleman, H. E. Garcia, A. Grodsky, D. R. Johnson, R. A. Locarnini, A. V. Mishonov, T.D. O'Brien, C.R. Paver, J.R. Reagan, D. Seidov, I. V. Smolyar, and M. M. Zweng, 2013: World Ocean Database 2013, NOAA Atlas NESDIS 72, S. Levitus, Ed., A. Mishonov, Technical Ed.; Silver Spring, MD, 209 pp., doi:10.7289/V5NZ85MT.

[6] Johnson, D.R., T.P. Boyer, H.E. Garcia, R.A. Locarnini, O.K. Baranova, and M.M. Zweng, 2009. World Ocean Database 2009 Documentation. Edited by Sydney Levitus. NODC Internal Report 20, NOAA Printing Office, Silver Spring, MD, 175 pp., http://www.nodc.noaa.gov/OC5/WOD09/pr_wod09.html.

Data atmosphere streams files

The setup scripts configure the CESM case with atmospheric forcing from the CAM6 DART Reanalysis. The coupler history files from this reanalysis are referenced in `user_datm.streams*template` files. These `user_datm.streams*template` files are contained in the same directory as the setup scripts and are configured and copied into the CESM case directory by the setup scripts.

6.141.7 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand, `&`, and terminate with a slash, `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

The variables and their default values are listed here:

```
&model_nml
   assimilation_period_days     = -1
   assimilation_period_seconds  = -1
   model_perturbation_amplitude = 0.2
   binary_grid_file_format      = 'big_endian'
   debug                        = 0,
   model_state_variables        = 'SALT_CUR ', 'QTY_SALINITY             ', 'UPDATE',
                                  'TEMP_CUR ', 'QTY_POTENTIAL_TEMPERATURE', 'UPDATE',
                                  'UVEL_CUR ', 'QTY_U_CURRENT_COMPONENT  ', 'UPDATE',
                                  'VVEL_CUR ', 'QTY_V_CURRENT_COMPONENT  ', 'UPDATE',
                                  'PSURF_CUR', 'QTY_SEA_SURFACE_PRESSURE ', 'UPDATE'
/
```

This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the ocean model dynamical timestep.

Item	Type	Description
assimilation_period_days	integer	The number of days to advance the model for each assimilation. If both assimilation_period_days and assimilation_period_seconds are 0; the value of the POP namelist variables restart_freq and restart_freq_opt are used to determine the assimilation period. *WARNING:* in the CESM framework, the restart_freq is set to a value that is not useful so DART defaults to 1 day - even if you are using POP in the LANL framework.
assimilation_period_seconds	integer	In addition to assimilation_period_days, the number of seconds to advance the model for each assimilation. Make sure you read the description of assimilation_period_days.
model_perturbation_amplitude	real(r8)	Reserved for future use.
binary_grid_file_format	character(len=32)	The POP grid files are in a binary format. Valid values are native, big_endian, or little_endian. Modern versions of Fortran allow you to specify the endianness of the file you wish to read when they are opened as opposed to needing to set a compiler switch or environment variable.
debug	integer	The switch to specify the run-time verbosity. • 0 is as quiet as it gets. • > 1 provides more run-time messages. • > 5 provides ALL run-time messages. All values above 0 will also write a netCDF file of the grid information and perform a grid interpolation test.
model_state_variables	character(:,3)	Strings that associate POP variables with a DART quantity and whether or not to write the updated values to the restart files. These variables will be read from the POP restart file and modified by the assimilation. Some (perhaps all) will be used by the forward observation operators. If the 3rd column is 'UPDATE', the output files will have the modified (assimilated,posterior) values. If the 3rd column is 'NO_COPY_BACK', that variable will not be written to the restart files. **The DART diagnostic files will always have the (modified) posterior values.** Diagnostic variables that are useful for the calculation of the forward observation operator but have no im-

6.141.8 References

6.142 MODULE dart_pop_mod (POP)

6.142.1 Overview

dart_pop_mod provides a consistent collection of routines that are useful for multiple programs e.g. dart_to_pop, pop_to_dart, etc.

6.142.2 Namelist

There are no namelists unique to this module. It is necessary for this module to read some of the POP namelists, and so they are declared in this module. In one instance, DART will read the time_manager_nml namelist and **write** an updated version to control the length of the integration of POP. All other information is simply read from the namelists and is used in the same context as POP itself. The POP documentation should be consulted. **Only the variables of interest to DART are described in this document.**
All namelists are read from a file named pop_in.

```
namelist /time_manager_nml/  allow_leapyear, stop_count, stop_option
```

dart_to_pop controls the model advance of LANL/POP by creating a &time_manager_nml in pop_in.DART **IFF** the DART state being converted has the 'advance_to_time' record. The pop_in.DART must be concatenated with the other namelists needed by POP into a file called pop_in . We have chosen to store the other namelists (which contain static information) in a file called pop_in.part2. Initially, the time_manager_nml is stored in a companion file called pop_in.part1 and the two files are concatenated into the expected pop_in - then, during the course of an assimilation experiment, DART keeps writing out a new time_manager_nml with new integration information - which gets appended with the static information in pop_in.part2

Contents	Type	Description
allow_leapyear	logical	DART ignores the setting of this parameter. All observations must use a Gregorian calendar. There are pathological cases, but if you are doing data assimilation, just use the Gregorian calendar. *[default: .true.]*
stop_count	integer	the number of model advance steps to take. *[default: 1]*
stop_option	character(len=64)	The units for the number of model advance steps (stop_count) to take. *[default: 'ndays']*

```
namelist /io_nml/  luse_pointer_files, pointer_filename
```

Contents	Type	Description
luse_pointer_files	logical	switch to indicate the use of pointer files or not. If `.true.`, a pointer file is used to contain the name of the restart file to be used. DART requires this to be `.true`. *[default: .true.]*
pointer_filename	character(len=100)	The name of the pointer file. All of the DART scripts presume and require the use of the default. Each ensmeble member gets its own pointer file. *[default: rpointer.ocn.[1-N].restart]*

```
namelist /restart_nml/  restart_freq_opt, restart_freq
```

Contents	Type	Description
luse_pointer_files	logical	switch to indicate the use of pointer files or not. If `.true.`, a pointer file is used to contain the name of the restart file to be used. DART requires this to be `.true`. *[default: .true.]*
pointer_filename	character(len=100)	The name of the pointer file. All of the DART scripts presume and require the use of the default. Each ensmeble member gets its own pointer file. *[default: rpointer.ocn.[1-N].restart]*

```
namelist /init_ts_nml/  init_ts_option, init_ts_file, init_ts_file_fmt
```

The `dart_pop_mod:initialize_module()` routine reads `pop_in` . There are several code stubs for future use that may allow for a more fully-supported POP namelist implementation. This namelist is one of them. Until further notice, the `init_ts_nml` is completely ignored by DART.

Contents	Type	Description
init_ts_option	character(len=64)	NOT USED by DART. All T,S information comes from a netCDF restart file named `pop.r.nc` *[default: 'restart']*
init_ts_file	character(len=100)	NOT USED by DART. All T,S information comes from `pop.r.nc` *[default: 'pop.r']*
init_ts_file_fmt	character(len=64)	NOT USED by DART. The file format is `'nc'` *[default: 'nc']*

```
namelist /domain_nml/  ew_boundary_type
```

DART needs to know if the East-West domain is cyclic for spatial interpolations. Presently, DART has only been tested for the dipole grid, which is cyclic E-W and closed N-S.

Contents	Type	Description
ew_boundary_type	character(len=64)	switch to indicate whether the East-West domain is cyclic or not. DART/POP has not been tested in a regional configuration, so DART requires this to be `'cyclic'`. *[default: 'cyclic']*

```
namelist /grid_nml/  horiz_grid_opt,  vert_grid_opt,  topography_opt, &
                                horiz_grid_file, vert_grid_file, topography_file
```

The POP grid information comes in several files: horizontal grid lat/lons in one, the vertical grid spacing in another, and the topography (lowest valid vertical level) in a third.
Here is what we can get from the (binary) horizontal grid file:

```
real(r8), dimension(:,:) :: ULAT,  &! latitude  (radians) of U points
real(r8), dimension(:,:) :: ULON,  &! longitude (radians) of U points
real(r8), dimension(:,:) :: HTN ,  &! length (cm) of north edge of T box
real(r8), dimension(:,:) :: HTE ,  &! length (cm) of east  edge of T box
real(r8), dimension(:,:) :: HUS ,  &! length (cm) of south edge of U box
real(r8), dimension(:,:) :: HUW ,  &! length (cm) of west  edge of U box
real(r8), dimension(:,:) :: ANGLE  &! angle
```

The vertical grid file is ascii, with 3 columns/line:

```
cell thickness(in cm)   cell center(in m)   cell bottom(in m)
```

Here is what we can get from the topography file:

```
integer, dimension(:,:), :: KMT    &! k index of deepest grid cell on T grid
```

These must be derived or come from someplace else . . .

```
KMU               k index of deepest grid cell on U grid
HT                real(r8) value of deepest valid T depth (in cm)
HU                real(r8) value of deepest valid U depth (in cm)
```

Contents	Type	Description
horiz_grid_opt, vert_grid_opt, topography_opt	character(len=64)	switch to indicate whether or not the grids will come from an external file or not. DART requires ALL of these to be `'file'`. *[default: 'file']*
horiz_grid_file	character(len=100)	The name of the binary file containing the values for the horizontal grid. The **dimensions** of the grid are read from `pop.r.nc`. It would have been nice to include the actual grid information in the netCDF files. *[default: 'horiz_grid.gx3v5.r8ieee.le']*
vert_grid_file	character(len=100)	The name of the ASCII file containing the values for the vertical grid. The file must contain three columns of data pertaining to the cell thickness (in cm), the cell center (in meters), and the cell bottom (in meters). Again, it would have been nice to include the vertical grid information in the netCDF files. *[default: 'vert_grid.gx3v5']*
topography_grid_file	character(len=100)	The name of the binary file containing the values for the topography information. The **dimensions** of the grid are read from `pop.r.nc`. *[default: 'topography.gx3v5.r8ieee.le']*

6.142.3 Other modules used

```
types_mod
time_manager_mod
utilities_mod
typesizes
netcdf
```

6.142.4 Public interfaces

Only a select number of interfaces used are discussed here. Each module has its own discussion of their routines.

Interface routines

use dart_pop_mod, only :	get_pop_calendar
	set_model_time_step
	get_horiz_grid_dims
	get_vert_grid_dim
	read_horiz_grid
	read_topography
	read_vert_grid
	write_pop_namelist
	get_pop_restart_filename

Required interface routines

call get_pop_calendar(calstring)

```
character(len=*), intent(out) :: calstring
```

Returns a string containing the type of calendar in use.

`calstring`	DART/POP uses a 'gregorian' calendar.

poptimestep = set_model_time_step()

```
type(time_type), intent(out) :: poptimestep
```

`set_model_time_step` returns the model time step that was set in the restart_nml`restart_freq`. This is the minimum amount of time DART thinks the POP model can advance. Indirectly, this specifies the minimum assimilation interval.

`poptimestep`	the minimum assimilation interval

call get_horiz_grid_dims(Nx, Ny)

```
integer, intent(out) :: Nx, Ny
```

`get_horiz_grid_dims` reads `pop.r.nc` to determine the number of longitudes and latitudes.

`Nx`	the length of the 'i' dimension in the POP restart file. The number of longitudes in use.
`Ny`	the length of the 'j' dimension in the POP restart file. The number of latitudes in use.

call get_vert_grid_dim(Nz)

```
integer, intent(out) :: Nz
```

`get_vert_grid_dim` reads `pop.r.nc` to determine the number of vertical levels in use.

`Nz`	the length of the 'k' dimension in the POP restart file. The number of vertical levels in use.

call read_horiz_grid(nx, ny, ULAT, ULON, TLAT, TLON)

```
integer,                    intent(in)  :: nx, ny
real(r8), dimension(nx,ny), intent(out) :: ULAT, ULON, TLAT, TLON
```

`read_horiz_grid` reads the direct access binary files containing the POP grid information. **The first record is REQUIRED to be 'ULAT', the second record is REQUIRED to be 'ULON'.**

`nx`	The number of longitudes in the grid.
`ny`	The number of latitudes in the grid.
`ULAT`	The matrix of latitudes for the UVEL and VVEL variables. Units are degrees [-90,90].
`ULON`	The matrix of longitudes for the UVEL and VVEL variables. Units are degrees. [0,360]
`TLAT`	The matrix of latitudes for the SALT and TEMP variables. Units are degrees [-90,90].
`TLON`	The matrix of longitudes for the SALT and TEMP variables. Units are degrees. [0,360]

call read_topography(nx, ny, KMT, KMU)

```
integer,                   intent(in)  :: nx, ny
integer, dimension(nx,ny), intent(out) :: KMT, KMU
```

`read_topography` reads the direct access binary files containing the POP topography information. **The first record is REQUIRED to be 'KMT'.** 'KMU' is calculated from 'KMT'.

nx	The number of longitudes in the grid.
ny	The number of latitudes in the grid.
KMT	The matrix containing the lowest valid depth index at grid centroids.
KMU	The matrix containing the lowest valid depth index at grid corners.

call read_vert_grid(nz, ZC, ZG)

```
integer,                intent(in)  :: nz
real(r8), dimension(nz), intent(out) :: ZC, ZG
```

read_vert_grid reads the ASCII file containing the information about the vertical levels. The file must contain three columns of data pertaining to; 1) the cell thickness (in cm),
2) the cell center (in meters),
and 3) the cell bottom (in meters).

nz	The number of vertical levels.
ZC	The depth (in meters) at the grid centers.
ZG	The depth (in meters) at the grid edges.

call write_pop_namelist(model_time, adv_to_time)

```
type(time_type), intent(in)  :: model_time
type(time_type), intent(in)  :: adv_to_time
```

write_pop_namelist writes the POP namelist time_manager_nml with the information necessary to advance POP to the next assimilation time. The namelist is written to a file called pop_in.DART. Presently, DART is configured to minimally advance POP for 86400 seconds - i.e. 1 day. The forecast length (the difference between 'model_time' and 'adv_to_time') must be an integer number of days with the current setup. An error will result if it is not.

model_time	The 'valid' time of the current model state.
adv_to_time	The time of the next assimilation.

call get_pop_restart_filename(filename)

```
character(len=*), intent(out) :: filename
```

`get_pop_restart_filename` returns the filename containing the POP restart information. At this point the filename is **hardwired** to `pop.r.nc`, but may become more flexible in future versions. The filename may be derived from the `restart_nml` but is currently ignored.

`filename`	The name of the POP restart file.

6.142.5 Files

filename	purpose
pop_in	to read the POP namelists
pop.r.nc	provides grid dimensions and 'valid_time' of the model state
`&grid_nml` "horiz_grid_file"	contains the values of the horizontal grid
`&grid_nml` "vert_grid_file"	contains the number and values of the vertical levels
`&grid_nml` "topography_grid_file"	contains the indices of the wet/dry cells
pop_in.DART	to control the integration of the POP model advance

6.142.6 References

- none

6.142.7 Private components

N/A

6.143 ROMS

There are several DART users who have working DART interface code to the Regional Ocean Modeling System (ROMS), as the model is a community ocean model funded by the Office of Naval Research. Please visit MyRoms for more information on the model.

The lead developers are at Rutgers and UCLA, but the list of associate developers is extensive. Please read ROMS developers for more information.

If you are interested in running DART with this model please contact the DART group at dart@ucar.edu for more information. We are currently working with collaborators to optimize the model_mod interface and associated scripting to run data assimilation experiments with this model. We may be able to put you in contact with the right people to get a copy of the code.

6.143.1 Overview

This document describes the relationship between ROMS and DART and provides an overview of how to perform ensemble data assimilation with ROMS to provide ocean states that are consistent with the information provided by various ocean observations.

Running ROMS is complicated. It is **strongly** recommended that you become very familiar with running ROMS before you attempt a ROMS-DART assimilation experiment. Running DART is complicated. It is **strongly** recommended that you become very familiar with running DART before you attempt a ROMS-DART assimilation experiment. Running ROMS-DART takes expertise in both areas.

We recommend working through the *DART tutorial* to learn the concepts of ensemble data assimilation and the capabilities of DART.

The ROMS code is not distributed with DART, it can be obtained from the ROMS website. There you will also find instructions on how to compile and run ROMS. DART can use the 'verification observations' from ROMS (basically the estimate of the observation at the location and time computed as the model advances) so it would be worthwhile to become familiar with that capability of ROMS.

DART calls these 'precomputed forward operators'. DART can also use observations from the World Ocean Database - WOD. The conversion from the WOD formats to the DART observation sequence format is accomplished by the converters in the `DART/observations/obs_converters/WOD` directory.

The DART forward operators require interpolation from the ROMS terrain-following and horizontally curvilinear orthogonal coordinates to the observation location. Please contact us for more information about this interpolation.

6.143.2 Generating an initial ensemble

The ROMS interface provides the ability to create an ensemble of initial ROMS history files from an initial file by using the *PROGRAM perturb_single_instance*. You can specify an ensemble of any size in the `perturb_single_instance` namelist in `input.nml` and this program will randomly perturb the temperature and salinity fields of an initial ROMS history file to generate the ensemble.

6.143.3 A note about filenames

During the course of an experiment, many files are created. To make them unique, the *ocean_time* is converted from "seconds since 1900-01-01 00:00:00" to the equivalent number of DAYS. An *integer* number of days. The intent is to tag the filename to reflect the valid time of the model state. This could be used as the DSTART for the next cycle, so it makes sense to me. The confusion comes when applied to the observation files.

The input observation files for the ROMS 4DVAR system typically have a DSTART that designates the start of the forecast cycle and the file must contain observation from DSTART to the end of the forecast. Makes sense.

The model runs to the end of the forecast, harvesting the verification observations along the way. So then DART converts all those verification observations and tags that file … with the same time tag as all the other output files … which reflects the *ocean_time* (converted to days). The input observation file to ROMS will have a different DSTART time in the filename than the corresponding verification files. Ugh. You are free to come up with a better plan.

These are just examples…after all; hopefully good examples.

6.143.4 Procedure

The procedure to perform an assimilation experiment is outlined in the following steps:

1. Compile ROMS (as per the ROMS instructions).
2. Compile all the DART executables (in the normal fashion).
3. Stage a directory with all the files required to advance an ensemble of ROMS models and DART.
4. Modify the run-time controls in `ocean.in`, `s4dvar.in` and `input.nml`. Since ROMS has a *Bin/subsitute* command, it is used to replace temporary placeholders with actual values at various parts during the process.
5. Advance all the instances of ROMS; each one will produce a restart file and a verification observation file.
6. Convert all the verification observation files into a single DART observation sequence file with the `convert_roms_obs.f90` program in `DART/observations/obs_converters/ROMS/`.
7. Run filter to assimilate the data (DART will read and update the ROMS files directly - no conversion is necessary.)
8. Update the control files for ROMS in preparation for the next model advance.

6.143.5 Shell scripts

The `shell_scripts` directory has several scripts that are intended to provide examples. These scripts **WILL** need to be modified to work on your system and are heavily internally commented. It will be necessary to read through and understand the scripts. As mentioned before, the ROMS *Bin/subsitute* command is used to replace temporary placeholders with actual values at various parts during the process.

Script	Description
ensemble.sh	Was written by Hernan Arango to run an ensemble of ROMS models. It is an appropriate example of what is required from the ROMS perspective. It does no data assimilation.
stage_experiment.csh	prepares a directory for an assimilation experiment. The idea is basically that everything you need should be assembled by this script and that this should only be run ONCE per experiment. After everything is staged in the experiment directory, another script can be run to advance the model and perform the assimilation. *stage_experiment.csh* will also modify some of the template scripts and copy working versions into the experiment directory. This script may be run interactively, i.e. from the UNIX command line.
submit_multiple_cycles.csh	is an executable script that submits a series of dependent jobs to an LSF queuing system. Each job runs *cycle.csh* in the experiment directory and only runs if the previous dependent job completes successfully.
cycle.csh.template	is a non-executable template that is modified by *stage_experiment.csh* and results in an exectuable *cycle.csh* in the experiment directory. *cycle.csh* is designed to be run as a batch job and advances the ROMS model states one-by-one for the desired forecast length. The assimilation is performed and the control information for the next ROMS forecast is updated. Each model execution and *filter* use the same set of MPI tasks.
submit_multiple_jobs_slurm.csh	is an executable script that submits a series of dependent jobs to an LSF queuing system. It is possible to submit **many** jobs to the queue, but the jobs run one-at-a-time. Every assimilation cycle is divided into two scripts to be able to efficiently set the resources for each phase. *advance_ensemble.csh* is a job array that advances each ROMS instance in separate jobs. When the entire job array finishes - and only if they all finish correctly - will the next job start to run. *run_filter.csh* performs the assimilation and prepares the experiment directory for another assimilation cycle. *submit_multiple_jobs_slurm.csh* may be run from the command line in the experiment directory. Multiple assimilation cycles can be specified, so it is possible to put **many** jobs in the queue.
advance_ensemble.csh.template	is a non-executable template that is modified by *stage_experiment.csh* and results in an exectuable *advance_ensemble.csh* in the experiment directory. *advance_ensemble.csh* is designed to submit an job array to the queueing system (PBS,SLURM, or LSF) to advance the ensemble members in separate jobs.
run_filter.csh.template	is a non-executable template that is modified by *stage_experiment.csh* and results in an exectuable *run_filter.csh* in the experiment directory. *run_filter.csh* is very similar to *cycle.csh* but does not advance the ROMS model instances.

The variables from ROMS that are copied into the DART state vector are controlled by the *input.nml model_nml* namelist. See below for the documentation on the &model_nml entries. The state vector should include all variables needed to apply the forward observation operators as well as the prognostic variables important to restart ROMS.

The example *input.nml model_nml* demonstrates how to construct the DART state vector. The following table explains in detail each entry for the *variables* namelist item:

Variable name	This is the ROMS variable name as it appears in the ROMS netCDF file.
DART QUANTITY	This is the character string of the corresponding DART QUANTITY. The complete list of possible DART QUANTITY values is available in the `obs_def_mod` that is built by `preprocess`.
minimum	If the variable is to be updated in the ROMS restart file, this specifies the minimum value. If set to 'NA', there is no minimum value.
maximum	If the variable is to be updated in the ROMS restart file, this specifies the maximum value. If set to 'NA', there is no maximum value.
update	The updated variable may or may not be written to the ROMS restart file. *'UPDATE'* means the variable in the restart file is updated. This is case-insensitive. *'NO_COPY_BACK'* (or anything else) means the variable in the restart file remains unchanged.

6.143.6 Namelist

This namelist is read from the file *input.nml*. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. The default namelist is presented below, a more realistic namelist is presented at the end of this section.

```
&model_nml
  roms_filename               = 'roms_input.nc'
  assimilation_period_days    = 1
  assimilation_period_seconds = 0
  vert_localization_coord     = 3
  debug                       = 0
  variables                   = ''
/
```

Item	Type	Description
roms_filename	character(len=256)	This is the name of the file used to provide information about the ROMS variable dimensions, etc.
assi milation_period_days, assimi lation_period_seconds	integer	Combined, these specify the width of the assimilation window. The current model time is used as the center time of the assimilation window. All observations in the assimilation window are assimilated. BEWARE: if you put observations that occur before the beginning of the assimilation_period, DART will error out because it cannot move the model 'back in time' to process these observations.
variables	character(:, 5)	A 2D array of strings, 5 per ROMS variable to be added to the dart state vector. 1. ROMS field name - must match netCDF variable name exactly 2. DART QUANTITY - must match a valid DART QTY_xxx exactly 3. minimum physical value - if none, use 'NA' 4. maximum physical value - if none, use 'NA' 5. case-insensitive string describing whether to copy the updated variable into the ROMS restart file ('UPDATE') or not (any other value). There is generally no point copying diagnostic variables into the restart file. Some diagnostic variables may be useful for computing forward operators, however.
ve rt_localization_coord	integer	Vertical coordinate for vertical localization. • 1 = model level • 2 = pressure (in pascals) • 3 = height (in meters) • 4 = scale height (unitless) Currently, only 3 (height) is supported for ROMS.

A more realistic ROMS namelist is presented here, along with one of the more unusual settings that is generally necessary when running ROMS. The *use_precomputed_FOs_these_obs_types* variable needs to list the observation types that are present in the ROMS verification observation file.

```
&model_nml
  roms_filename                = 'roms_input.nc'
  assimilation_period_days     = 1
  assimilation_period_seconds  = 0
  vert_localization_coord      = 3
  debug                        = 1
  variables = 'temp',   'QTY_TEMPERATURE',          'NA', 'NA', 'update',
              'salt',   'QTY_SALINITY',            '0.0', 'NA', 'update',
              'u',      'QTY_U_CURRENT_COMPONENT',  'NA', 'NA', 'update',
              'v',      'QTY_V_CURRENT_COMPONENT',  'NA', 'NA', 'update',
              'zeta',   'QTY_SEA_SURFACE_HEIGHT'    'NA', 'NA', 'update'
/
&obs_kind_nml
  evaluate_these_obs_types = ''
  assimilate_these_obs_types =          'SATELLITE_SSH',
                                        'SATELLITE_SSS',
                                        'XBT_TEMPERATURE',
                                        'CTD_TEMPERATURE',
                                        'CTD_SALINITY',
                                        'ARGO_TEMPERATURE',
                                        'ARGO_SALINITY',
                                        'GLIDER_TEMPERATURE',
                                        'GLIDER_SALINITY',
                                        'SATELLITE_BLENDED_SST',
                                        'SATELLITE_MICROWAVE_SST',
                                        'SATELLITE_INFRARED_SST'
  use_precomputed_FOs_these_obs_types = 'SATELLITE_SSH',
                                        'SATELLITE_SSS',
                                        'XBT_TEMPERATURE',
                                        'CTD_TEMPERATURE',
                                        'CTD_SALINITY',
                                        'ARGO_TEMPERATURE',
                                        'ARGO_SALINITY',
                                        'GLIDER_TEMPERATURE',
                                        'GLIDER_SALINITY',
                                        'SATELLITE_BLENDED_SST',
                                        'SATELLITE_MICROWAVE_SST',
                                        'SATELLITE_INFRARED_SST'
/
```

6.144 ROSE

Attention: `rose` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `rose` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.144.1 Overview

The rose model is an atmospheric model for the Mesosphere Lower-Thermosphere (MLT). The DART interface was developed by Tomoko Matsuo (now at CU-Boulder).

The source code for rose is not distributed with DART, thus the DART/models/rose/work/workshop_setup.csh script is SUPPOSED to fail without the rose code.

The rose model is a research model that is still being developed. The DART components here are simply to help the rose developers with the DART framework.

As of Mon Mar 22 17:23:20 MDT 2010 the rose project has been substantially streamlined. There is no need for the trans_time and build_nml routines. dart_to_model has assumed those responsibilities.

6.145 Simple advection

6.145.1 Overview

This simple advection model simulates a wind field using Burger's Equation with an upstream semi-lagrangian differencing on a periodic one-dimensional domain. This diffusive numerical scheme is stable and forcing is provided by adding in random gaussian noise to each wind grid variable independently at each timestep. The domain mean value of the wind is relaxed to a constant fixed value set by the namelist parameter `mean_wind`. The random forcing magnitude is set by namelist parameter `wind_random_amp` and the damping of the mean wind is controlled by parameter `wind_damping_rate`. An Eulerian option with centered in space differencing is also provided and can be used by setting namelist parameter `lagrangian_for_wind` to `.false.` The Eulerian differencing is both numerically unstable and subject to shock formation. However, it can sometimes be made stable in assimilation mode (see recent work by Majda and collaborators).

The model state includes a single passive tracer that is advected by the wind field using semi-lagrangian upstream differencing. The state also includes a tracer source value at each gridpoint. At each time step, the source is added into the concentration at each gridpoint. There is also a constant global destruction of tracer that is controlled by the namelist parameter destruction_rate. The appropriate percentage of tracer is destroyed at each gridpoint at each timestep.

The model also includes an associated model for the tracer source rate. At each gridpoint, there is a value of the time mean source rate and a value of the phase offset for a diurnal component of the source rate. The diurnal source rate has an amplitude that is proportional to the source rate (this proportion is controlled by namelist parameter `source_diurnal_rel_amp`). At each grid point, the source is the sum of the source rate plus the appropriate diurnally varying component. The phase_offset at the gridpoint controls the diurnal phase. The namelist parameter `source_phase_noise` controls the amplitude of random gaussian noise that is added into the source phase at each time step. If `source_phase_noise` is zero then the phase offset is fixed. Finally, the time mean source rate is constant in time in the present model version. The time mean source rate controls the amplitude of the diurnal cycle of the tracer source.

For the simple advection model, DART advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation for model state variables.

The simple advection model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced in Section 25 of the *DART tutorial*. and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

6.145.2 Namelist

The &model_nml namelist is read from the input.nml file. Namelists start with an ampersand & and terminate with a slash /. Character strings that contain a / must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   num_grid_points        = 10,
   grid_spacing_meters    = 100000.0,
   time_step_days         = 0,
   time_step_seconds      = 3600,
   mean_wind              = 20.0,
   wind_random_amp        = 0.00027778,
   wind_damping_rate      = 0.0000027878,
   lagrangian_for_wind    = .true.,
   destruction_rate       = 0.000055556,
   source_random_amp_frac = 0.00001,
   source_damping_rate    = 0.0000027878,
   source_diurnal_rel_amp = 0.05,
   source_phase_noise     = 0.0,
   output_state_vector  = .false.
/
```

Description of each namelist entry

Item	Type	Description
num_grid_points	integer	Number of grid points in model. State vector size is 5 times this number.
grid_spacing_meters	integer	Grid spacing in meters.
time_step_days	real(r8)	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	real(r8)	Number of seconds for dimensional timestep, mapped to delta_t.
mean_wind	real(r8)	Base wind velocity (expected value over time) in meters/second.
wind_random_amp	real(r8)	Random walk amplitude for wind in meters/second^2.
wind_damping_rate	real(r8)	Rate of damping towards mean wind value in fraction/second.
lagrangian_for_wind	logical	Can use Lagrangian (stable) or Eulerian (unstable) scheme for wind.
destruction_rate	real(r8)	Tracer destruction rate in fraction/second.
source_random_amp_frac	real(r8)	Random walk amplitude for source as a fraction of mean source (per second)^2.
source_damping_rate	real(r8)	Damping towards mean source rate in fraction/second.
source_diurnal_rel_amp	real(r8)	Relative amplitude of diurnal cycle of source (dimensionless).
source_phase_noise	real(r8)	Amplitude of gaussian noise to be added to source phase offset (per second).
output_state_vector	logical	Controls the output to netCDF files. If .true., output the raw dart state vector. If .false. output the prognostic version (gridded data) for easier plotting (recommended).

6.146 SQG

Attention: `sqg` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `sqg` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.146.1 Overview

This is a uniform PV two-surface QG+1 spectral model contributed by Rahul Majahan.

The underlying model is described in: Hakim, Gregory J., 2000: Role of Nonmodal Growth and Nonlinearity in Cyclogenesis Initial-Value Problems. J. Atmos. Sci., 57, 2951-2967. doi: 10.1175/1520-0469(2000)057<2951:RONGAN>2.0.CO;2

6.146.2 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
```

6.146.3 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

Optional namelist interface `&model_nml` may be read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = *get_model_size( )*

```
integer :: get_model_size
```

Returns the length of the model state vector.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Advances the model for a single time step. The time associated with the initial model state is also input although it is not used for the computation.

x	State vector of length model_size.
time	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Returns metadata about a given element, indexed by index_in, in the model state vector. The location defines where the state variable is located.

index_in	Index of state vector element about which information is requested.
location	The location of state variable element.
var_type	Returns the type (always 1) of the indexed state variable as an optional argument.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given model state, returns the value interpolated to a given location.

x	A model state vector.
location	Location to which to interpolate.
itype	Not used.
obs_val	The interpolated value from the model.
istatus	Quality control information, always returned 0.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the time step (forecast length) of the model;

var	Smallest time step of model.

call static_init_model()

Used for runtime initialization of model; reads namelist, initializes model parameters, etc. This is the first call made to the model by any DART-compliant assimilation routine.

call end_model()

A stub.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns the time at which the model will start if no input initial conditions are to be used. This is used to spin-up the model from rest.

time	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns default initial conditions for the model; generally used for spinning up initial model states.

x	Initial conditions for state vector.

ierr = nc_write_model_atts(ncFileID)

```
integer             :: nc_write_model_atts
integer, intent(in) :: ncFileID
```

Function to write model specific attributes to a netCDF file. At present, DART is using the NetCDF format to output diagnostic information. This is not a requirement, and models could choose to provide output in other formats. This function writes the metadata associated with the model to a NetCDF file opened to a file identified by ncFileID.

ncFileID	Integer file descriptor to previously-opened netCDF file.
ierr	Returns a 0 for successful completion.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                             :: nc_write_model_vars
integer,                intent(in) :: ncFileID
real(r8), dimension(:), intent(in) :: statevec
integer,                intent(in) :: copyindex
integer,                intent(in) :: timeindex
```

Writes a copy of the state variables to a netCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state.

ncFileID	file descriptor to previously-opened netCDF file.
statevec	A model state vector.
copyindex	Integer index of copy to be written.
timeindex	The timestep counter for the given state.
ierr	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state, produces a perturbed model state.

state	State vector to be perturbed.
pert_state	Perturbed state vector: NOT returned.
interf_provided	Returned false; interface is not implemented.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3D Sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3D Sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Pass-through to the 3D Sphere locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

A NULL INTERFACE in this model.

ens_mean	State vector containing the ensemble mean.

6.146.4 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input.

```
&model_nml
  output_state_vector = .false.
  channel_center = 45.0
  channel_width = 40.0
  assimilation_period_days = 0
  assimilation_period_seconds = 21600
  debug = .false.
/
```

This namelist is read in a file called `input.nml`

Contents	Type	Description
output_state_vector	logical	If .true. write state vector as a 1D array to the diagnostic output file. If .false. break state vector up into fields before writing to the outputfile.
channel_center	real(r8)	Channel center
channel_width	real(r8)	Channel width
assimilation_period_days	integer	Number of days for timestep
assimilation_period_seconds	integer	Number of seconds for timestep
debug	logical	Set to .true. for more output

6.146.5 Files

filename	purpose
input.nml	to read the model_mod namelist
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.146.6 References

The underlying model is described in:
Hakim, Gregory J., 2000: Role of Nonmodal Growth and Nonlinearity in Cyclogenesis Initial-Value Problems. J. Atmos. Sci., 57, 2951-2967. doi: 10.1175/1520-0469(2000)057<2951:RONGAN>2.0.CO;2

6.146.7 Private components

N/A

6.147 TIEGCM

6.147.1 Overview

The Thermosphere Ionosphere Electrodynamic General Circulation Model (TIEGCM) is developed by the NCAR High Altitude Observatory (HAO).

DART-TIEGCM has been used to assimilate neutral mass density retrieved from satellite-borne accelerometers and electron density obtained from ground-based and space-based GNSS signals. Unlike other ionospheric data assimilation applications, this approach allows simultaneous assimilation of thermospheric and ionospheric parameters by taking advantage of the coupling of plasma and neutral constituents described in TIEGCM. DART/TIEGCM's demonstrated capability to infer under-observed thermospheric parameters from abundant electron density observations has important implications for the future of upper atmosphere research.

DART is designed so that the TIEGCM source code can be used with no modifications. TIEGCM and DART run as separate executables. The TIEGCM 2.0 source code and User's Guide is available from HAO:

- TIEGCM 2.0 source code
- TIEGCM 2.0 User's Guide

6.147.2 DART-TIEGCM namelist options

The `model_nml` namelist contains the TIEGCM specific options for DART. `model_nml` is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   tiegcm_restart_file_name    = 'tiegcm_restart_p.nc'
   tiegcm_secondary_file_name  = 'tiegcm_s.nc'
   model_res                   = 5.0
   assimilation_period_seconds = 3600
   estimate_f10_7              = .false.
   f10_7_file_name             = 'f10_7.nc'
   debug                       = 0
   variables = 'NE',    'QTY_ELECTRON_DENSITY',          '1000.0',  'NA',      'restart',
↪    'UPDATE'
               'OP',    'QTY_DENSITY_ION_OP',            'NA',      'NA',      'restart',
↪    'UPDATE',
               'TI',    'QTY_TEMPERATURE_ION',           'NA',      'NA',      'restart',
↪    'UPDATE',
```

(continues on next page)

(continued from previous page)

```
                'TE',    'QTY_TEMPERATURE_ELECTRON',      'NA',      'NA',      'restart',
↪    'UPDATE',
                'OP_NM', 'QTY_DENSITY_ION_OP',            'NA',      'NA',      'restart',
↪    'UPDATE',
                'O1',    'QTY_ATOMIC_OXYGEN_MIXING_RATIO','0.00001', '0.99999', 'secondary
↪',  'NO_COPY_BACK',
                'O2',    'QTY_MOLEC_OXYGEN_MIXING_RATIO', '0.00001', '0.99999', 'secondary
↪',  'NO_COPY_BACK',
                'TN',    'QTY_TEMPERATURE',               '0.0',     '6000.0',  'secondary
↪',  'NO_COPY_BACK',
                'ZG',    'QTY_GEOMETRIC_HEIGHT',          'NA',      'NA',      'secondary
↪',  'NO_COPY_BACK',
  /
```

Namelist entry	Type	Description
tiegcm_restart_file_name	character(len=256)	The TIEGCM restart template
tiegcm_secondary_file_name	character(len=256)	The TIEGCM secondary template
model_resol	real(r8)	TIEGCM model resolution 5.0 or 2.5 degrees
assimilation_period_seconds	integer	This specifies the width of the assimilation window. The current model time is used as the center time of the assimilation window. All observations in the assimilation window are assimilated. BEWARE: if you put observations that occur before the beginning of the assimilation_period, DART will error out because it cannot move the model 'back in time' to process these observations. `assimilation_period_seconds` must be an integer number of TIEGCM dynamical timesteps (as specified by tiegcm.nml:STEP) AND be able to be expressed by tiegcm.nml:STOP. Since STOP has three components: day-of-year, hour, and minute, the `assimilation_period_seconds` must be an integer number of minutes.
estimate_f10_7	logical	Switch to specify that the f10.7 index should be estimated by augmenting the DART state vector with a scalar. The location of the f10.7 index is taken to be longitude of local noon and latitude zero.
f10_7_file_name	character(len=256)	If `estimate_f107=.true.` f10.7 will be part of the dart state. The variable f10_7 is read from `f10_7_file_name` An example f10_7.cdl file is given in the work directory
debug	integer	Set to 0 (zero) for minimal output. Successively larger values generate successively more output.
variables	character (MAX_NUM_VARIABLES * 6)	Six strings to describe the TIEGCM variables to be used in DART. A description of the six strings is given below.

```
variables = 'NAME', 'QTY', 'MIN', 'MAX', 'FILE', 'UPDATE'
```

`NAME` The variable name in the TIEGCM netCDF file.

`QTY` The DART quantity for the variable.

`MIN` The minimum bound (if any) for the variable. Enter 'NA' for no minimum.

`MAX` The a maximum bound (if any) for the variable.

`FILE` The tiegcm netcdf file containing the variable. 'restart' or 'secondary'

`UPDATE` filter will update the variable in the TIEGCM netcdf file. Use `NO_COPY_BACK` to prevent filter from updating the variable.

Below is an example showing the namelist options necessary to add f10.7 to the DART state

```
&filter_nml
   input_state_file_list        = 'restart_p_files.txt', 'secondary_files.txt', 'f10.7.
↪txt'
   output_state_file_list       = 'out_restart_p_files.txt', 'out_secondary_files.txt',
↪'out_f10.7.txt'

&model_nml
  estimate_f10_7 = .true.
  f10_7_file_name = 'f10_7.nc'
  variables =  'NE',    'QTY_ELECTRON_DENSITY',          '1000.0',  'NA',      'restart',
↪    'UPDATE'
               ...
               'ZG',    'QTY_GEOMETRIC_HEIGHT',          'NA',      'NA',      'secondary
↪',  'NO_COPY_BACK',
               'f10_7'  'QTY_1D_PARAMETER'               'NA',      'NA',      'calculate
↪', 'UPDATE'
```

6.147.3 References

- Matsuo, T., and E. A. Araujo-Pradere (2011), Role of thermosphere-ionosphere coupling in a global ionosphere specification, *Radio Science*, **46**, RS0D23, doi:10.1029/2010RS004576
- Lee, I. T., T, Matsuo, A. D. Richmond, J. Y. Liu, W. Wang, C. H. Lin, J. L. Anderson, and M. Q. Chen (2012), Assimilation of FORMOSAT-3/COSMIC electron density profiles into thermosphere/Ionosphere coupling model by using ensemble Kalman filter, *Journal of Geophysical Research*, **117**, A10318, doi:10.1029/2012JA017700
- Matsuo, T., I. T. Lee, and J. L. Anderson (2013), Thermospheric mass density specification using an ensemble Kalman filter, *Journal of Geophysical Research*, **118**, 1339-1350, doi:10.1002/jgra.50162
- Lee, I. T., H. F. Tsai, J. Y. Liu, Matsuo, T., and L. C. Chang (2013), Modeling impact of FORMOSAT-7/COSMIC-2 mission on ionospheric space weather monitoring, *Journal of Geophysical Research*, **118**, 6518-6523, doi:10.1002/jgra.50538
- Matsuo, T. (2014), Upper atmosphere data assimilation with an ensemble Kalman filter, in Modeling the Ionosphere-Thermosphere System, *Geophys. Monogr. Ser.*, vol. 201, edited by J. Huba, R. Schunk, and G. Khazanov, pp. 273-282, John Wiley & Sons, Ltd, Chichester, UK, doi:10.1002/9781118704417
- Hsu, C.-H., T. Matsuo, W. Wang, and J. Y. Liu (2014), Effects of inferring unobserved thermospheric and ionospheric state variables by using an ensemble Kalman filter on global ionospheric specification and forecasting, *Journal of Geophysical Research*, **119**, 9256-9267, doi:10.1002/2014JA020390
- Chartier, A., T. Matsuo, J. L. Anderson, G. Lu, T. Hoar, N. Collins, A. Coster, C. Mitchell, L. Paxton, G. Bust (2015), Ionospheric Data Assimilation and Forecasting During Storms, *Journal of Geophysical Research*, doi:10.1002/2014JA020799

6.148 WRF-Hydro

6.148.1 Overview

The Weather Research and Forecasting Hydrologic Model (WRF-Hydro) is a community modeling system and framework for hydrologic modeling and model coupling. WRF-Hydro is configured to use the Noah-MP Land Surface Model to simulate land surface processes. Combined with DART, the facility is called *HydroDART*.

The development of HydroDART was a collaboration between **James McCreight** of the Research Applications Laboratory of NCAR and **Moha Gharamti** of the Data Assimilation Research Section of NCAR.

Streamflow assimilation is an active area of research and provides many interesting research challenges.

6.148.2 Description of this directory within the DART repository

Contents of the `$DARTROOT/models/wrf_hydro/`:

```
├── ensemble_config_files/
      # Files which configure ensembles in wrfhydropy.
├── experiment_config_files/
      # File which configure hydro_dart_py experiments.
├── hydro_dart_py/
      # Python package/library for configuring and executing experiments.
├── python/
      # Python scripts for various purposes.
├── R/
      # R scripts for various purposes.
├── shell_scripts/
      # Shell scripts for various purposes.
├── templates/
      # Obsolete?
├── work/
      # Dart executables build directory and other testing.
├── model_mod.html
      # The model_mod documentation.
├── model_mod.nml
      # The model_mod namelist (subsumed by work/input.nml)
├── model_mod.f90
      # The model_mod code.
├── noah_hydro_mod.f90
      # Some model_mod interfaces more specific to Noah?
├── create_identity_streamflow_obs.f90
      # For creating identity streamflow obs for the NHDPlus-based
      # channel-network configuration of WRF-Hydro.
├── README.rst
      # This file.
```

6.148.3 To set up an experiment

To set up an experiment, consult the `./python/experiment` directory.

6.148.4 Description of external directories on GLADE

The gridded version of the model has bits/bobs in these directories:

- `/gpfs/fs1/work/jamesmcc/domains/public/croton_NY/Gridded/DOMAIN`
- `/gpfs/fs1/work/jamesmcc/domains/public/croton_NY/Gridded/RESTART`

Only the gridcells with flow are retained in the `qlink[1,2]`, `hlink` variables, so they must be unpacked in EXACTLY the same way as wrfHydo packs them from the grid to their 'sparse' representation.

6.148.5 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
     assimilation_period_days       = 0
     assimilation_period_seconds    = 3600
     lsm_model_choice               = 'noahMP'
     model_perturbation_amplitude   = 0.5
     perturb_distribution           = 'lognormal'
     max_link_distance              = 2000.0
     streamflow_4_local_multipliers = 0.0001
     debug                          = 0
     domain_order                   = 'hydro'
     domain_shapefiles              = 'restart.hydro.nc'
     lsm_variables    = 'SH2O',              'QTY_SOIL_LIQUID_WATER', '0.0',   'NA',
↪'NOUPDATE',
                        'SUBSURFACE_FLUX',   'QTY_SUBSURFACE',        '0.0',   'NA',
↪'NOUPDATE',
                        'OVERLAND_FLUX',     'QTY_OVERLAND_FLOW',     '0.0',   'NA',
↪'NOUPDATE'
     hydro_variables  = 'qlink1',            'QTY_STREAM_FLOW',       '0.0',   'NA',
↪'UPDATE',
                        'z_gwsubbas',        'QTY_AQUIFER_WATER',     'NA',    'NA',
↪'UPDATE'
     parameters       = 'qBucketMult',       'QTY_BUCKET_MULTIPLIER', '0.001', '50',
↪'UPDATE',
                        'qSfcLatRunoffMult', 'QTY_RUNOFF_MULTIPLIER', '0.001', '50',
↪'UPDATE'
/
```

This namelist is read from a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the NOAH model dynamical timestep.

Item	Type	Description
assimilation_period_days	integer	The number of days to advance the model for each assimilation. [default: 1]
assimilation_period_seconds	integer	In addition to assimilation_period_days, the number of seconds to advance the model for each assimilation. [default: 0]
lsm_model_choice	character(len=128)	case-insensitive specification of the Land Surface model. Valid values are noahmp and noahmp_36
model_perturbation_amplitude	real(r8)	The amount of noise to add when trying to perturb a single state vector to create an ensemble. Only used when input.nml is set with &filter_nml:start_from_restart = .false.. See also Generating the initial ensemble. units: standard deviation of the specified distribution the mean at the value of the state vector element.
perturb_distribution	character(len=256)	The switch to determine the distribution of the perturbations used to create an initial ensemble from a single model state. Valid values are : lognormal or gaussian
max_link_distance	real(r8)	The along-the-stream localization distance. In meters.
streamflow_4_local_multipliers	real(r8)	
debug	integer	The switch to specify the run-time verbosity. • 0 is as quiet as it gets • > 1 provides more run-time messages • > 5 provides ALL run-time messages All values above 0 will also write a netCDF file of the grid information and perform a grid interpolation test. [default: 0]
domain_order	character(len=256):: dimension(3)	There are three possible domains to include in the HydroDART state: hydro, parameters, lsm This variable specifies the ordering of the domains.
domain_shapefiles	character(len=256):: dimension(3)	There are input files used to determine the shape of the input variables and any geographic metadata. They must be specified in the same order as listed in domain_order
lsm_variables	character(len=32):: dimension(5,40)	The list of variable names in the NOAH restart file to use to create the DART state vector and their corresponding DART QUANTITY. [see example below]
hydro_variables	character(len=32):: dimension(5,40)	The list of variable names in the channel model file to use to create the DART state vector and their

The columns of `lsm_variables`, `hydro_variables`, and `parameters` needs some explanation. Starting with the column 5, `UPDATE` denotes whether or not to replace the variable with the Posterior (i.e. assimilated) value. Columns 3 and 4 denote lower and upper bounds that should be enforced when writing to the files used to restart the model. These limits are not enforced for the DART diagnostic files. Column 2 specifies the relationship between the netCDF variable name for the model and the corresponding DART QUANTITY.

Support for these QUANTITYs is provided by running `preprocess` with the following namelist settings:

```
&preprocess_nml
           overwrite_output = .true.
    input_obs_kind_mod_file = '../../../assimilation_code/modules/observations/DEFAULT_
↪obs_kind_mod.F90'
   output_obs_kind_mod_file = '../../../assimilation_code/modules/observations/obs_kind_
↪mod.f90'
     input_obs_def_mod_file = '../../../observations/forward_operators/DEFAULT_obs_def_
↪mod.F90'
    output_obs_def_mod_file = '../../../observations/forward_operators/obs_def_mod.f90'
   input_files              = '../../../observations/forward_operators/obs_def_
↪streamflow_mod.f90',
                              '../../../observations/forward_operators/obs_def_land_mod.
↪f90',
                              '../../../observations/forward_operators/obs_def_COSMOS_
↪mod.f90'
  /
```

6.149 WRF

6.149.1 Overview

DART interface module for the Weather Research and Forecasting (WRF) model. This page documents the details of the module compiled into DART that interfaces with the WRF data in the state vector.

6.149.2 WRF+DART Tutorial

There is additional overview and tutorial documentation for running a WRF/DART assimilation in *WRF/DART Tutorial Materials for the Manhattan Release.*

Please work through the tutorial in order to learn how to run WRF and DART.

Items of Note

- The `model_mod` reads WRF netCDF files directly to acquire the model state data. The `wrf_to_dart` and `dart_to_wrf` programs are no longer necessary.
- A netCDF file named `wrfinput_d01` is required and must be at the same resolution and have the same surface elevation data as the files converted to create the DART initial conditions. No data will be read from this file, but the grid information must match exactly.

The model interface code supports WRF configurations with multiple domains. Data for all domains is read into the DART state vector. During the computation of the forward operators (getting the estimated observation values from each ensemble member), the search starts in the domain with the highest number, which is generally the finest nest or one of multiple finer nests. The search stops as soon as a domain contains the observation location, working its way

from largest number to smallest number domain, ending with domain 1. For example, in a 4 domain case the data in the state vector that came from `wrfinput_d04` is searched first, then `wrfinput_d03`, `wrfinput_d02`, and finally `wrfinput_d01`.

The forward operator is computed from the first domain grid that contains the lat/lon of the observation. During the assimilation phase, when the state values are adjusted based on the correlations and assimilation increments, all points in all domains that are within the localization radius are adjusted, regardless of domain. The impact of an observation on the state depends only on the distance between the observation and the state vector point, and the regression coefficient based on the correlation between the distributions of the ensemble of state vector points and the ensemble of observation forward operator values.

The fields from WRF that are copied into the DART state vector are controlled by namelist. See below for the documentation on the &model_nml entries. The state vector should include all fields needed to restart a WRF run. There may be additional fields needed depending on the microphysics scheme selected. See the ascii file `wrf_state_variables_table` in the `models/wrf` directory for a list of fields that are often included in the DART state.

6.149.3 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   default_state_variables     = .true.
   wrf_state_variables         = 'NULL'
   wrf_state_bounds            = 'NULL'
   num_domains                 = 1
   calendar_type               = 3
   assimilation_period_seconds = 21600
   allow_obs_below_vol         = .false.
   vert_localization_coord     = 3
   center_search_half_length   = 500000.
   center_spline_grid_scale    = 10
   circulation_pres_level      = 80000.0
   circulation_radius          = 108000.0
   sfc_elev_max_diff           = -1.0
   polar                       = .false.
   periodic_x                  = .false.
   periodic_y                  = .false.
   scm                         = .false.
   allow_perturbed_ics         = .false.   # testing purposes only
/

   # Notes for model_nml:
   # (1) vert_localization_coord must be one of:
   #     1 = model level
   #     2 = pressure
   #     3 = height
   #     4 = scale height
   # (2) see bottom of this file for explanations of polar, periodic_x,
   #     periodic_y, and scm
   # (3) calendar = 3 is GREGORIAN, which is what WRF uses.
```

(continues on next page)

(continued from previous page)

```
# (4) if 'default_state_variables' is .true. the model_mod.f90 code will
#     fill the state variable table with the following wrf vars:
#        U, V, W, PH, T, MU
#     you must set it to false before you change the value
#     of 'wrf_state_variables' and have it take effect.
# (5) the format for 'wrf_state_variables' is an array of 5 strings:
#     wrf netcdf variable name, dart QTY_xxx string, type string (must be
#     unique, will soon be obsolete, we hope), 'UPDATE', and '999' if the
#     array is part of all domains.  otherwise, it is a string with the domain
#     numbers (e.g. '12' for domains 1 and 2, '13' for domains 1 and 3).
#   example:
# wrf_state_variables='U','QTY_U_WIND_COMPONENT','TYPE_U','UPDATE','999',
#                     'V','QTY_V_WIND_COMPONENT','TYPE_V','UPDATE','999',
#                     'W','QTY_VERTICAL_VELOCITY','TYPE_W','UPDATE','999',
#                     'T','QTY_POTENTIAL_TEMPERATURE','TYPE_T','UPDATE','999',
#                     'PH','QTY_GEOPOTENTIAL_HEIGHT','TYPE_GZ','UPDATE','999',
#                     'MU','QTY_PRESSURE','TYPE_MU','UPDATE','999',
#                     'QVAPOR','QTY_VAPOR_MIXING_RATIO','TYPE_QV','UPDATE','999',
#                     'QCLOUD','QTY_CLOUD_LIQUID_WATER','TYPE_QC','UPDATE','999',
#                     'QRAIN','QTY_RAINWATER_MIXING_RATIO','TYPE_QR','UPDATE','999',
#                     'U10','QTY_U_WIND_COMPONENT','TYPE_U10','UPDATE','999',
#                     'V10','QTY_V_WIND_COMPONENT','TYPE_V10','UPDATE','999',
#                     'T2','QTY_TEMPERATURE','TYPE_T2','UPDATE','999',
#                     'TH2','QTY_POTENTIAL_TEMPERATURE','TYPE_TH2','UPDATE','999',
#                     'Q2','QTY_SPECIFIC_HUMIDITY','TYPE_Q2','UPDATE','999',
#                     'PSFC','QTY_PRESSURE','TYPE_PS','UPDATE','999',
# (6) the format for 'wrf_state_bounds' is an array of 4 strings:
#     wrf netcdf variable name, minimum value, maximum value, and either
#     FAIL or CLAMP.  FAIL will halt the program if an out of range value
#     is detected.  CLAMP will set out of range values to the min or max.
#     The special string 'NULL' will map to plus or minus infinity and will
#     not change the values.  arrays not listed in this table will not
#     be changed as they are read or written.
#
#
# polar and periodic_x are used in global wrf.  if polar is true, the
# grid interpolation routines will wrap over the north and south poles.
# if periodic_x is true, when the east and west edges of the grid are
# reached the interpolation will wrap.  note this is a separate issue
# from regional models which cross the GMT line; those grids are marked
# as having a negative offset and do not need to wrap; this flag controls
# what happens when the edges of the grid are reached.

# the scm flag is used for the 'single column model' version of WRF.
# it needs the periodic_x and periodic_y flags set to true, in which
# case the X and Y directions are periodic; no collapsing of the grid
# into a single location like the 3d-spherical polar flag implies.
```

Description of each namelist entry

Item	Type	Description
default_state_variables	logical	If *.true.*, the dart state vector contains the fields U, V, W, PH, T, MU, in that order, and only those. Any values listed in the *wrf_state_variables* namelist item will be ignored.
wrf_state_variables	character(:, 5)	A 2D array of strings, 5 per wrf array to be added to the dart state vector. If *default_state_variables* is *.true.*, this is ignored. When *.false.*, this list of array names controls which arrays and the order that they are added to the state vector. The 5 strings are: 1. WRF field name - must match netcdf name exactly 2. DART KIND name - must match a valid DART QTY_xxx exactly 3. TYPE_NN - will hopefully be obsolete, but for now NN should match the field name. 4. the string UPDATE. at some future point, non-updatable fields may become part of the state vector. 5. A numeric string listing the domain numbers this array is part of. The specical string 999 means all domains. For example, '12' means domains 1 and 2, '13' means 1 and 3.
wrf_state_bounds	character(:, 4)	A 2D array of strings, 4 per wrf array. During the copy of data to and from the wrf netcdf file, variables listed here will have minimum and maximum values enforced. The 4 strings are: 1. WRF field name - must match netcdf name exactly 2. Minimum – specified as a string but must be a numeric value (e.g. '0.1') Can be 'NULL' to allow any minimum value. 3. Maximum – specified as a string but must be a numeric value (e.g. '0.1') Can be 'NULL' to allow any maximum value. 4. Action – valid strings are 'CLAMP', 'FAIL', 'FAIL' means if a value is found outside the range, the code fails with an error. 'CLAMP' simply sets the out of range

The following items used to be in the WRF namelist but have been removed. The first 4 are no longer needed, and the last one was moved to the `&dart_to_wrf_nml` namelist in 2010. In the Lanai release having these values in the namelist does not cause a fatal error, but more recent versions of the code will fail if any of these values are specified. Remove them from your namelist to avoid errors.

Item	Type	Description
`surf_obs`	logical	OBSOLETE – now an error to specify this.
`soil_data`	logical	OBSOLETE – now an error to specify this.
`h_diab`	logical	OBSOLETE – now an error to specify this.
`num_moist_vars`	integer	OBSOLETE – now an error to specify this.
`adv_mod_command`	character(len=32)	OBSOLETE – now an error to specify this.

6.149.4 Files

- model_nml in input.nml
- wrfinput_d01, wrfinput_d02, ... (one file for each domain)
- netCDF output state diagnostics files

6.149.5 References

http://www2.mmm.ucar.edu/wrf/users/docs/user_guide_V3/contents.html

6.150 PROGRAM `replace_wrf_fields`

6.150.1 Overview

Program to copy various fields from one WRF netCDF file to another.

There are many existing utilities to process netCDF files, i.e. the NCO operators and NCL scripts, which have more functionality than this program. The only purpose for having this one is that it is a standalone program with no prerequisites or dependencies other than the netCDF libraries. If you already have other tools available they can do the same functions that this program does.

This program copies the given data fields from the input file to the output file, failing if their sizes, shapes, or data types do not match exactly. The expected use is to copy fields which are updated by the WRF program but are not part of the DART state vector, for example, sea surface temperature or soil fields. After DART has updated the WRF restart `wrfinput_d01` file, this program can be used to update other fields in the file before running the model.

6.150.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&replace_wrf_fields_nml
   fieldnames = 'SST',
   fieldlist_file = '',
   fail_on_missing_field = .true.
```

(continues on next page)

(continued from previous page)

```
   debug = .false.,
   /
```

Item	Type	Description
field-names	character(len=129)(:)	An array of ASCII field names to be copied from the input netCDF file to the output netCDF file. The names must match exactly, and the size and shape of the data must be the same in the input and output files for the data to be copied. If the field names are set here, the fieldlist_file item must be ' '.
field-list_file	character(len=129)	An alternative to an explicit list of field names to copy. This is a single string, the name of a file which contains a single field name, one per line. If this option is set, the fieldnames namelist item must be ' '.
fail_on_missing_field	logical	If any fields in the input list are not found in either the input or output netcdf files, fail if this is set to true. If false, a warning message will be printed but execution will continue.
debug	logical	If true, print out debugging messages about which fields are found in the input and output files.

6.150.3 Modules used

```
types_mod
utilities_mod
parse_args_mod
```

6.150.4 Files

- input namelist ; `input.nml`
- Input - output WRF state netCDF files; `wrfinput_d01, wrfinput_d02, ...`
- fieldlist_file (if specified in namelist)

File formats

This utility works on any pair of netCDF files, doing a simple read and copy from one to the other.

Error codes and conditions

Routine	Message	Comment
replace_wrf_fields	Usage: echo infile.nc outfile.nc | ./replace_wrf_fields	The program did not read 2 filenames from the console.
replace_wrf_fields	cannot specify both fieldnames and fieldlist_file	In the namelist you must either specify an explicit list of fieldnames to copy between the files, or give a single filename which contains the list of field names. You cannot specify both.
replace_wrf_fields	field not found in input/output file	If 'fail_on_missing_field' is true in the namelist and a field is not found in either the input or output file.
replace_wrf_fields	field does not match	If the input and output files have different sizes, number of dimensions, or data types, the program cannot copy the data.

6.150.5 References

- none

6.151 PROGRAM `wrf_dart_obs_preprocess`

6.151.1 Overview

Program to preprocess observations, with specific knowledge of the WRF domain.

This program will exclude all observations outside of the given WRF domain. There are options to exclude or increase the error values of obs close to the domain boundaries. The program can superob (average) aircraft and satellite wind obs if they are too dense.

This program can read up to 9 additional obs_seq files and merge their data in with the basic obs_sequence file which is the main input.

This program can reject surface observations if the elevation encoded in the observation is too different from the wrf surface elevation.

This program can exclude observations above a specified height or pressure.

This program can overwrite the incoming Data QC value with another.

6.151.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&wrf_obs_preproc_nml

  file_name_input          = 'obs_seq.old'
  file_name_output         = 'obs_seq.new'

  sonde_extra              = 'obs_seq.rawin'
```

(continues on next page)

(continued from previous page)

```
  land_sfc_extra            = 'obs_seq.land_sfc'
  metar_extra               = 'obs_seq.metar'
  marine_sfc_extra          = 'obs_seq.marine'
  sat_wind_extra            = 'obs_seq.satwnd'
  profiler_extra            = 'obs_seq.profiler'
  gpsro_extra               = 'obs_seq.gpsro'
  acars_extra               = 'obs_seq.acars'
  trop_cyclone_extra        = 'obs_seq.tc'

  overwrite_obs_time        = .false.

  obs_boundary              = 0.0
  increase_bdy_error        = .false.
  maxobsfac                 = 2.5
  obsdistbdy                = 15.0

  sfc_elevation_check       = .false.
  sfc_elevation_tol         = 300.0
  obs_pressure_top          = 0.0
  obs_height_top            = 2.0e10

  include_sig_data          = .true.
  tc_sonde_radii            = -1.0

  superob_aircraft          = .false.
  aircraft_horiz_int        = 36.0
  aircraft_pres_int         = 2500.0

  superob_sat_winds         = .false.
  sat_wind_horiz_int        = 100.0
  sat_wind_pres_int         = 2500.0

  overwrite_ncep_satwnd_qc = .false.
  overwrite_ncep_sfc_qc     = .false.
/
```

Item	Type	Description
Generic parameters:		
file_name_input	character(len=129)	The input obs_seq file.
file_name_output	character(len=129)	The output obs_seq file.
sonde_extra, land_sfc_extra, metar_extra, marine_sfc_extra, sat_wind_extra, profiler_extra, gpsro_extra, acars_extra, trop_cyclone_extra	character(len=129)	The names of additional input obs_seq files, which if they exist, will be merged in with the obs from the file_name_input obs_seq file. If the files do not exist, they are silently ignored without error.
overwrite_obs_time	logical	If true, replace the incoming observation time with the analysis time. Not recommended.
Boundary-specific parameters:		
obs_boundary	real(r8)	Number of grid points around domain boundary which will be considered the new extent of the domain. Observations outside this smaller area will be excluded.
increase_bdy_error	logical	If true, observations near the domain boundary will have their observation error increased by maxobsfac.
maxobsfac	real(r8)	If increase_bdy_error is true, multiply the error by a ramped factor. This item sets the maximum error.
obsdistbdy	real(r8)	If increase_bdy_error is true, this defines the region around the boundary (in number of grid points) where the observation error values will be altered. This is ramped, so when you reach the innermost points the change in observation error is 0.0.
Parameters to reduce observation count :		
sfc_elevation_check	logical	If true, check the height of surface observations against the surface height in the model.
sfc_elevation_tol	real(r8)	If sfc_elevation_check is true, the maximum difference between the elevation of a surface observation and the model surface height, in meters. If the difference is larger than this value, the observation is excluded.
obs_pressure_top	real(r8)	Observations with a vertical coordinate in pressure which are located above this pressure level (i.e. the obs vertical value is smaller than the given pressure) will be excluded.
obs_height_top	real(r8)	Observations with a vertical coordinate in height which are located above this height value (i.e. the obs vertical value is larger than the given height) will be excluded.
Radio/Rawinsonde-specific parameters :		
include_sig_data	logical	If true, include significant level data from radiosondes.

6.151.3 Modules used

```
types_mod
obs_sequence_mod
utilities_mod
obs_kind_mod
time_manager_mod
model_mod
netcdf
```

6.151.4 Files

- Input namelist ; `input.nml`
- Input WRF state netCDF files; `wrfinput_d01, wrfinput_d02, ...`
- Input obs_seq files (as specified in namelist)
- Output obs_seq file (as specified in namelist)

File formats

This utility can read one or more obs_seq files and combine them while doing the rest of the processing. It uses the standard DART observation sequence file format.

6.151.5 References

- Generously contributed by Ryan Torn.

6.152 MODULE model_mod

6.152.1 Overview

Every model that is DART compliant must provide an interface as documented here. The file `models/template/model_mod.f90` provides the fortran interfaces for a minimal implementation meeting these requirements. When adding a new model to DART you can either start by modifying a `model_mod.f90` file from a similar model already in DART or start with the template file. Either way, the supplied interface must match these descriptions exactly; no details of the underlying model can impact the interface.

Several of the routines listed below are allowed to be a NULL INTERFACE. This means the subroutine or function name must exist in this file, but it is ok if it contains no executable code.

A few of the routines listed below are allowed to be a PASS-THROUGH INTERFACE. This means the subroutine or function name can be listed on the 'use' line from the `location_mod`, and no subroutine or function with that name is supplied in this file. Alternatively, this file can provide an implementation which calls the underlying routines from the `location_mod` and then alters or augments the results based on model-specific requirements.

The system comes with several types of location modules for computing distances appropriately. Two of the ones most commonly used are for data in a 1D system and for data in a 3D spherical coordinate system. Make the selection by listing the appropriate choice from `location/*/location_mod.f90` in the corresponding `path_names_*` file at compilation time.

6.152.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
 /
```

Models are free to include a model namelist which can be read when `static_init_model` is called. A good example can be found in the lorenz_96 `model_mod.f90`.

6.152.3 Other modules used

```
types_mod
time_manager_mod
location_mod (multiple choices here)
utilities_mod
POSSIBLY MANY OTHERS DEPENDING ON MODEL DETAILS
```

6.152.4 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	shortest_time_between_assimilations
	static_init_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_copies
	get_close_obs
	get_close_state
	convert_vertical_obs
	convert_vertical_state
	read_model_time
	write_model_time
	end_model

A namelist interface `&model_nml` may be defined by the module, in which case it will be read from file `input.nml`. The details of the namelist are always model-specific (there are no generic namelist values).

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer(i8) :: get_model_size
```

Returns the length of the model state vector. Required.

model_size	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Does a single timestep advance of the model. The input value of the vector x is the starting condition and x must be updated to reflect the changed state after a timestep. The time argument is intent in and is used for models that need to know the date/time to compute a timestep, for instance for radiation computations. This interface is only called if the namelist parameter async is set to 0 in `perfect_model_obs` or `filter` or if the program `integrate_model` is to be used to advance the model state as a separate executable. If one of these options is not going to be used (the model will *only* be advanced as a separate model-specific executable), this can be a NULL INTERFACE. (The subroutine name must still exist, but it can contain no code and it will not be called.)

x	State vector of length model_size.
time	Current time of the model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Given an integer index into the state vector, returns the associated location. An optional argument returns the generic quantity of this item, e.g. QTY_TEMPERATURE, QTY_DENSITY, QTY_SALINITY, QTY_U_WIND_COMPONENT. This interface is required to be functional for all applications.

index_in	Index of state vector element about which information is requested.
location	The location of state variable element.
var_type	The generic quantity of the state variable element.

call model_interpolate(state_handle, ens_size, location, obs_quantity, expected_obs, istatus)

```
type(ensemble_type),    intent(in)  :: state_handle
integer,                intent(in)  :: ens_size
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: obs_quantity
real(r8),               intent(out) :: expected_obs(ens_size)
integer,                intent(out) :: istatus(ens_size)
```

Given a handle containing information for a state vector, an ensemble size, a location, and a model state variable quantity interpolates the state variable field to that location and returns an ensemble-sized array of values in `expected_obs(:)`. The `istatus(:)` array should be 0 for successful ensemble members and a positive value for failures. The `obs_quantity` variable is one of the quantity (QTY) parameters defined in the *MODULE obs_kind_mod* file and defines the quantity to interpolate. In low-order models that have no notion of kinds of variables this argument may be ignored. For applications in which only perfect model experiments with identity observations (i.e. only the value of a particular state variable is observed), this can be a NULL INTERFACE. Otherwise it is required (which is the most common case).

`state_handle`	The handle to the state structure containing information about the state vector about which information is requested.
`ens_size`	The ensemble size.
`location`	Location to which to interpolate.
`obs_quantity`	Quantity of state field to be interpolated.
`expected_obs`	The interpolated values from the model.
`istatus`	Integer values return 0 for success. Other positive values can be defined for various failures.

var = shortest_time_between_assimilations()

```
type(time_type) :: shortest_time_between_assimilations
```

Returns the smallest increment in time that the model is capable of advancing the state in a given implementation. The actual value may be set by the model_mod namelist (depends on the model). This interface is required for all applications.

`var`	Smallest advance time of the model.

call static_init_model()

Called to do one time initialization of the model. As examples, might define information about the model size or model timestep, read in grid information, read a namelist, set options, etc. In models that require pre-computed static data, for instance spherical harmonic weights, these would also be computed here. Can be a NULL INTERFACE for the simplest models.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Companion interface to init_conditions. Returns a time that is somehow appropriate for starting up a long integration of the model. At present, this is only used if the `perfect_model_obs` namelist parameter `read_input_state_from_file = .false.` If this option should not be used in `perfect_model_obs`, calling this routine should issue a fatal error.

time	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns a model state vector, x, that is some sort of appropriate initial condition for starting up a long integration of the model. At present, this is only used if the `perfect_model_obs` namelist parameter `read_input_state_from_file = .false.` If this option should not be used in `perfect_model_obs`, calling this routine should issue a fatal error.

x	Initial conditions for state vector.

call nc_write_model_atts(ncFileID, domain_id)

```
integer, intent(in) :: ncFileID
integer, intent(in) :: domain_id
```

This routine writes the model-specific attributes to netCDF files that DART creates. This includes coordinate variables and any metadata, but NOT the actual model state vector. `models/template/model_mod.f90` contains code that can be used for any model as-is.

The typical sequence for adding new dimensions, variables, attributes:

```
NF90_OPEN             ! open existing netCDF dataset
   NF90_redef         ! put into define mode
   NF90_def_dim       ! define additional dimensions (if any)
   NF90_def_var       ! define variables: from name, kind, and dims
   NF90_put_att       ! assign attribute values
NF90_ENDDEF           ! end definitions: leave define mode
   NF90_put_var       ! provide values for variable
NF90_CLOSE            ! close: save updated netCDF dataset
```

ncFileID	Integer file descriptor to previously-opened netCDF file.
domain_id	integer describing the domain (which can be a nesting level, a component model …) Models with nested grids are decomposed into 'domains' in DART. The concept is extended to refer to 'coupled' models where one model component may be the atmosphere, another component may be the ocean, or land, or ionosphere … these would be referenced as different domains.

call nc_write_model_vars(ncFileID, domain_id, state_ens_handle [, memberindex] [, timeindex])

```
integer,             intent(in) :: ncFileID
integer,             intent(in) :: domain_id
type(ensemble_type), intent(in) :: state_ens_handle
integer, optional,   intent(in) :: memberindex
integer, optional,   intent(in) :: timeindex
```

This routine may be used to write the model-specific state vector (data) to a netCDF file. Only used if `model_mod_writes_state_variables = .true.`

Typical sequence for adding new dimensions,variables,attributes:

```
NF90_OPEN             ! open existing netCDF dataset
   NF90_redef         ! put into define mode
   NF90_def_dim       ! define additional dimensions (if any)
   NF90_def_var       ! define variables: from name, kind, and dims
   NF90_put_att       ! assign attribute values
NF90_ENDDEF           ! end definitions: leave define mode
   NF90_put_var       ! provide values for variable
NF90_CLOSE            ! close: save updated netCDF dataset
```

ncFileID	file descriptor to previously-opened netCDF file.
domain_id	integer describing the domain (which can be a nesting level, a component model ...)
state_ens_handle	The handle to the state structure containing information about the state vector about which information is requested.
memberindex	Integer index of ensemble member to be written.
timeindex	The timestep counter for the given state.

call pert_model_copies(state_ens_handle, ens_size, pert_amp, interf_provided)

```
type(ensemble_type), intent(inout) :: state_ens_handle
integer,             intent(in)    :: ens_size
real(r8),            intent(in)    :: pert_amp
logical,             intent(out)   :: interf_provided
```

Given an ensemble handle, the ensemble size, and a perturbation amplitude; perturb the ensemble. Used to generate initial conditions for spinning up ensembles. If the `model_mod` does not want to do this, instead allowing the default algorithms in `filter` to take effect, `interf_provided =&nbps;.false.` and the routine can be trivial. Otherwise, `interf_provided` must be returned as `.true.`

state_ens_handle	The handle containing an ensemble of state vectors to be perturbed.
ens_size	The number of ensemble members to perturb.
pert_amp	the amplitude of the perturbations. The interpretation is based on the model-specific implementation.
interf_provided	Returns false if model_mod cannot do this, else true.

call get_close_obs(gc, base_loc, base_type, locs, loc_qtys, loc_types, num_close, close_ind [, dist] [, state_handle)

```
type(get_close_type),          intent(in)  :: gc
type(location_type),           intent(in)  :: base_loc
integer,                       intent(in)  :: base_type
type(location_type),           intent(in)  :: locs(:)
integer,                       intent(in)  :: loc_qtys(:)
integer,                       intent(in)  :: loc_types(:)
integer,                       intent(out) :: num_close
integer,                       intent(out) :: close_ind(:)
real(r8),            optional, intent(out) :: dist(:)
type(ensemble_type), optional, intent(in)  :: state_handle
```

Given a location and quantity, compute the distances to all other locations in the `obs` list. The return values are the number of items which are within maxdist of the base, the index numbers in the original obs list, and optionally the distances. The `gc` contains precomputed information to speed the computations.

In general this is a PASS-THROUGH ROUTINE. It is listed on the use line for the locations_mod, and in the public list for this module, but has no subroutine declaration and no other code in this module:

```
use location_mod, only: get_close_obs

public :: get_close_obs
```

However, if the model needs to alter the values or wants to supply an alternative implementation it can intercept the call like so:

```
use location_mod, only: &
        lm_get_close_obs => get_close_obs

public :: get_close_obs
```

In this case a local `get_close_obs()` routine must be supplied. To call the original code in the location module use:

```
call lm_get_close_obs(gc, base_loc, ...)
```

This subroutine will be called after `get_close_maxdist_init` and `get_close_obs_init`.

In most cases the PASS-THROUGH ROUTINE will be used, but some models need to alter the actual distances depending on the observation or state vector kind, or based on the observation or state vector location. It is reasonable in this case to leave `get_close_maxdist_init()` and `get_close_obs_init()` as pass-through routines and intercept only `get_close_obs()`. The local `get_close_obs()` can first call the location mod routine and let it return a list of values, and then inspect the list and alter or remove any entries as needed. See the CAM and WRF model_mod files for examples of this use.

gc	The get_close_type which stores precomputed information about the locations to speed up searching
base_loc	Reference location. The distances will be computed between this location and every other location in the obs list
base_type	The DART quantity at the base_loc
locs(:)	Compute the distance between the base_loc and each of the locations in this list
loc_qtys(:)	The corresponding quantity of each item in the locs list
loc_types(:)	The corresponding type of each item in the locs list. This is not available in the default implementation but may be used in custom implementations.
num_close	The number of items from the locs list which are within maxdist of the base location
close_ind(:)	The list of index numbers from the locs list which are within maxdist of the base location
dist(:)	If present, return the distance between each entry in the close_ind list and the base location. If not present, all items in the obs list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.
state_handle	The handle to the state structure containing information about the state vector about which information is requested.

call get_close_state(gc, base_loc, base_type, state_loc, state_qtys, state_indx, num_close, close_ind [, dist, state_handle])

```
type(get_close_type),          intent(in)    :: gc
type(location_type),           intent(inout) :: base_loc
integer,                       intent(in)    :: base_type
type(location_type),           intent(inout) :: state_loc(:)
integer,                       intent(in)    :: state_qtys(:)
integer(i8),                   intent(in)    :: state_indx(:)
integer,                       intent(out)   :: num_close
integer,                       intent(out)   :: close_ind(:)
real(r8),            optional, intent(out)   :: dist(:)
type(ensemble_type), optional, intent(in)    :: state_handle
```

Given a location and quantity, compute the distances to all other locations in the state_loc list. The return values are the number of items which are within maxdist of the base, the index numbers in the original state_loc list, and optionally the distances. The gc contains precomputed information to speed the computations.
In general this is a PASS-THROUGH ROUTINE. It is listed on the use line for the locations_mod, and in the public list for this module, but has no subroutine declaration and no other code in this module:

```
use location_mod, only: get_close_state

public :: get_close_state
```

However, if the model needs to alter the values or wants to supply an alternative implementation it can intercept the call like so:

```
use location_mod, only: &
        lm_get_close_state => get_close_state
```

(continues on next page)

(continued from previous page)

```
public :: get_close_state
```

In this case a local `get_close_state()` routine must be supplied. To call the original code in the location module use:

```
call loc_get_close_state(gc, base_loc, ...)
```

This subroutine will be called after `get_close_maxdist_init` and `get_close_state_init`.
In most cases the PASS-THROUGH ROUTINE will be used, but some models need to alter the actual distances depending on the observation or state vector kind, or based on the observation or state vector location. It is reasonable in this case to leave `get_close_maxdist_init()` and `get_close_state_init()` as pass-through routines and intercept only `get_close_state()`. The local `get_close_state()` can first call the location mod routine and let it return a list of values, and then inspect the list and alter or remove any entries as needed. See the CAM and WRF model_mod files for examples of this use.

gc	The get_close_type which stores precomputed information about the locations to speed up searching
base_loc	Reference location. The distances will be computed between this location and every other location in the list
base_type	The DART quantity at the base_loc
state_loc(:)	Compute the distance between the base_loc and each of the locations in this list
state_qtys(:)	The corresponding quantity of each item in the state_loc list
state_indx(:)	The corresponding DART index of each item in the state_loc list. This is not available in the default implementation but may be used in custom implementations.
num_close	The number of items from the state_loc list which are within maxdist of the base location
close_ind(:)	The list of index numbers from the state_loc list which are within maxdist of the base location
dist(:)	If present, return the distance between each entry in the close_ind list and the base location. If not present, all items in the state_loc list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.
state_handle	The handle to the state structure containing information about the state vector about which information is requested.

call convert_vertical_obs(state_handle, num, locs, loc_qtys, loc_types, which_vert, status)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: num
type(location_type), intent(in)  :: locs(:)
integer,             intent(in)  :: loc_qtys(:)
integer,             intent(in)  :: loc_types(:)
integer,             intent(in)  :: which_vert
integer,             intent(out) :: status(:)
```

Converts the observations to the desired vertical localization coordinate system. Some models (toy models with no 'real' observations) will not need this. Most (real) models have observations in one or more coordinate systems (pressure, height) and the model is generally represented in only one coordinate system. To be able to interpolate the model state

to the observation location, or to compute the true distance between the state and the observation, it is necessary to convert everything to a single coodinate system.

state_handle	The handle to the state.
num	the number of observation locations
locs	the array of observation locations
loc_qtys	the array of observation quantities.
loc_types	the array of observation types.
which_vert	the desired vertical coordinate system. There is a table in the location_mod.f90 that relates integers to vertical coordinate systems.
status	Success or failure of the vertical conversion. If istatus = 0, the conversion was a success. Any other value is a failure.

call convert_vertical_state(state_handle, num, locs, loc_qtys, loc_types, which_vert, status)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: num
type(location_type), intent(in)  :: locs(:)
integer,             intent(in)  :: loc_qtys(:)
integer(i8),         intent(in)  :: loc_indx(:)
integer,             intent(in)  :: which_vert
integer,             intent(out) :: status(:)
```

Converts the state to the desired vertical localization coordinate system. Some models (toy models with no 'real' observations) will not need this. To compute the true distance between the state and the observation, it is necessary to convert everything to a single coodinate system.

state_handle	The handle to the state.
num	the number of state locations
locs	the array of state locations
loc_qtys	the array of state quantities.
loc_indx	the array of state indices.
which_vert	the desired vertical coordinate system. There is a table in the location_mod.f90 that relates integers to vertical coordinate systems.
status	Success or failure of the vertical conversion. If istatus = 0, the conversion was a success. Any other value is a failure.

model_time = read_model_time(filename)

```
character(len=*), intent(in) :: filename
type(time_type)              :: model_time
```

Reads the valid time of the model state in a netCDF file. There is a default routine in assimilation_code/modules/io/dart_time_io_mod.f90 that can be used as a pass-through. That routine will read the **last** timestep of a 'time' variable - which is the same strategy used for reading netCDF files that have multiple timesteps in them. If your model

has some other representation of time (i.e. it does not use a netCDF variable named 'time') - you will have to write this routine.

ncid	handle to an open netCDF file
dart_time	The current time of the model state.

call write_model_time(ncid, dart_time)

```
integer,          intent(in) :: ncid
type(time_type),  intent(in) :: dart_time
```

Writes the assimilation time to a netCDF file. There is a default routine in `assimilation_code/modules/io/dart_time_io_mod.f90` that can be used as a pass-through. If your model has some other representation of time (i.e. it does not use a netCDF variable named 'time') - you will have to write this routine.

ncid	handle to an open netCDF file
dart_time	The current time of the model state.

call end_model()

Does any shutdown and clean-up needed for model. Can be a NULL INTERFACE if the model has no need to clean up storage, etc.

6.152.5 Files

- Models are free to read and write files as they see fit.

6.152.6 References

1. none

6.152.7 Private components

N/A

6.153 MODULE model_mod

6.153.1 Overview

Every model that is DART compliant must provide an set of interfaces that will be called by DART code. For models which have no special code for some of these routines, they can pass through the call to this default module, which satisfies the call but does no work. To use these routines in a `model_mod.f90`, add at the top:

```
use default_model_mod, only : xxx, yyy
```

and then leave them in the public list.

6.153.2 Namelist

The default routines have no namelist.

6.153.3 Other modules used

```
types_mod
time_manager_mod
location_mod
utilities_mod
netcdf_utilities_mod
ensemble_manager_mod
dart_time_io_mod
```

6.153.4 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	shortest_time_between_assimilations
	static_init_model
	init_time
	fail_init_time
	init_conditions
	fail_init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_copies
	get_close_obs
	get_close_state
	convert_vertical_obs
	convert_vertical_state
	read_model_time
	write_model_time
	end_model

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = *get_model_size( )*

```
integer(i8) :: get_model_size
```

Returns the length of the model state vector as 1. Probably not what you want. The model_mod should set this to the right size and not use this routine.

model_size	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Throws a fatal error. If the model_mod can advance the model it should provide a real routine. This default routine is intended for use by models which cannot advance themselves from inside filter.

x	State vector of length model_size.
time	Current time of model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Sets the location to missing and the variable type to 0. The model_mod should provide a routine that sets a real location and a state vector type for the requested item in the state vector.

index_in	Index of state vector element about which information is requested.
location	The location of state variable element.
var_type	The generic quantity of the state variable element.

call model_interpolate(state_handle, ens_size, location, obs_quantity, expected_obs, istatus)

```
type(ensemble_type),    intent(in)  :: state_handle
integer,                intent(in)  :: ens_size
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: obs_quantity
real(r8),               intent(out) :: expected_obs(ens_size)
integer,                intent(out) :: istatus(ens_size)
```

Sets the expected obs to missing and returns an error code for all obs. This routine should be supplied by the model_mod.

state_handle	The handle to the state structure containing information about the state vector about which information is requested.
ens_size	The ensemble size.
location	Location to which to interpolate.
obs_quantity	Quantity of state field to be interpolated.
expected_obs	The interpolated values from the model.
istatus	Integer values return 0 for success. Other positive values can be defined for various failures.

var = shortest_time_between_assimilations()

```
type(time_type) :: shortest_time_between_assimilations
```

Returns 1 day.

var	Smallest advance time of the model.

call static_init_model()

Does nothing.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns a time of 0.

time	Initial model time.

call fail_init_time(time)

```
type(time_type), intent(out) :: time
```

Throws a fatal error. This is appropriate for models that cannot start from arbitrary initial conditions.

time	NOT SET. Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns x(:) = 0.0

x	Initial conditions for state vector.

call fail_init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Throws a fatal error. This is appropriate for models that cannot start from arbitrary initial conditions.

x	NOT SET: Initial conditions for state vector.

call nc_write_model_atts(ncFileID, domain_id)

```
integer, intent(in) :: ncFileID
integer, intent(in) :: domain_id
```

Does nothing.

ncFileID	Integer file descriptor to previously-opened netCDF file.
domain_id	integer describing the domain (which can be a nesting level, a component model ...) Models with nested grids are decomposed into 'domains' in DART. The concept is extended to refer to 'coupled' models where one model component may be the atmosphere, another component may be the ocean, or land, or ionosphere ... these would be referenced as different domains.

call nc_write_model_vars(ncFileID, domain_id, state_ens_handle [, memberindex] [, timeindex])

```
integer,             intent(in) :: ncFileID
integer,             intent(in) :: domain_id
type(ensemble_type), intent(in) :: state_ens_handle
integer, optional,   intent(in) :: memberindex
integer, optional,   intent(in) :: timeindex
```

Does nothing

ncFileID	file descriptor to previously-opened netCDF file.
domain_id	integer describing the domain (which can be a nesting level, a component model ...)
state_ens_handle	The handle to the state structure containing information about the state vector about which information is requested.
memberindex	Integer index of ensemble member to be written.
timeindex	The timestep counter for the given state.

call pert_model_copies(state_ens_handle, ens_size, pert_amp, interf_provided)

```
type(ensemble_type), intent(inout) :: state_ens_handle
integer,             intent(in)    :: ens_size
real(r8),            intent(in)    :: pert_amp
logical,             intent(out)   :: interf_provided
```

Returns 'interface provided' flag as false, so the default perturb routine in DART will add small amounts of gaussian noise to all parts of the state vector.

state_ens_handle	The handle containing an ensemble of state vectors to be perturbed.
ens_size	The number of ensemble members to perturb.
pert_amp	the amplitude of the perturbations. The interpretation is based on the model-specific implementation.
interf_provided	Returns false if model_mod cannot do this, else true.

call get_close_obs(gc, base_loc, base_type, locs, loc_qtys, loc_types, num_close, close_ind [, dist] [, state_handle)

```
type(get_close_type),          intent(in)  :: gc
type(location_type),           intent(in)  :: base_loc
integer,                       intent(in)  :: base_type
type(location_type),           intent(in)  :: locs(:)
integer,                       intent(in)  :: loc_qtys(:)
integer,                       intent(in)  :: loc_types(:)
integer,                       intent(out) :: num_close
integer,                       intent(out) :: close_ind(:)
real(r8),            optional, intent(out) :: dist(:)
type(ensemble_type), optional, intent(in)  :: state_handle
```

Passes the call through to the location module code.

gc	The get_close_type which stores precomputed information about the locations to speed up searching
base_loc	Reference location. The distances will be computed between this location and every other location in the obs list
base_type	The DART quantity at the base_loc
locs(:)	Compute the distance between the base_loc and each of the locations in this list
loc_qtys(:)	The corresponding quantity of each item in the locs list
loc_types(:)	The corresponding type of each item in the locs list. This is not available in the default implementation but may be used in custom implementations.
num_close	The number of items from the locs list which are within maxdist of the base location
close_ind(:)	The list of index numbers from the locs list which are within maxdist of the base location
dist(:)	If present, return the distance between each entry in the close_ind list and the base location. If not present, all items in the obs list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.
state_handle	The handle to the state structure containing information about the state vector about which information is requested.

call get_close_state(gc, base_loc, base_type, state_loc, state_qtys, state_indx, num_close, close_ind, dist, state_handle)

```
type(get_close_type), intent(in)    :: gc
type(location_type),  intent(inout) :: base_loc
integer,              intent(in)    :: base_type
type(location_type),  intent(inout) :: state_loc(:)
integer,              intent(in)    :: state_qtys(:)
integer(i8),          intent(in)    :: state_indx(:)
integer,              intent(out)   :: num_close
integer,              intent(out)   :: close_ind(:)
real(r8),             intent(out)   :: dist(:)
type(ensemble_type),  intent(in)    :: state_handle
```

Passes the call through to the location module code.

gc	The get_close_type which stores precomputed information about the locations to speed up searching
base_loc	Reference location. The distances will be computed between this location and every other location in the obs list
base_type	The DART quantity at the base_loc
state_loc(:)	Compute the distance between the base_loc and each of the locations in this list
state_qtys(:)	The corresponding quantity of each item in the state_loc list
state_indx(:)	The corresponding DART index of each item in the state_loc list. This is not available in the default implementation but may be used in custom implementations.
num_close	The number of items from the state_loc list which are within maxdist of the base location
close_ind(:)	The list of index numbers from the state_loc list which are within maxdist of the base location
dist(:)	If present, return the distance between each entry in the close_ind list and the base location. If not present, all items in the state_loc list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.
state_handle	The handle to the state structure containing information about the state vector about which information is requested.

call convert_vertical_obs(state_handle, num, locs, loc_qtys, loc_types, which_vert, status)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: num
type(location_type), intent(in)  :: locs(:)
integer,             intent(in)  :: loc_qtys(:)
integer,             intent(in)  :: loc_types(:)
integer,             intent(in)  :: which_vert
integer,             intent(out) :: status(:)
```

Passes the call through to the location module code.

state_handle	The handle to the state.
num	the number of observation locations
locs	the array of observation locations
loc_qtys	the array of observation quantities.
loc_types	the array of observation types.
which_vert	the desired vertical coordinate system. There is a table in the location_mod.f90 that relates integers to vertical coordinate systems.
status	Success or failure of the vertical conversion. If istatus = 0, the conversion was a success. Any other value is a failure.

call convert_vertical_state(state_handle, num, locs, loc_qtys, loc_indx, which_vert, status)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: num
type(location_type), intent(in)  :: locs(:)
integer,             intent(in)  :: loc_qtys(:)
integer(i8),         intent(in)  :: loc_indx(:)
integer,             intent(in)  :: which_vert
integer,             intent(out) :: status(:)
```

Passes the call through to the location module code.

state_handle	The handle to the state.
num	the number of state locations
locs	the array of state locations
loc_qtys	the array of state quantities.
loc_indx	the array of state vector indices.
which_vert	the desired vertical coordinate system. There is a table in the location_mod.f90 that relates integers to vertical coordinate systems.
status	Success or failure of the vertical conversion. If istatus = 0, the conversion was a success. Any other value is a failure.

model_time = read_model_time(filename)

```
character(len=*), intent(in) :: filename
type(time_type)              :: model_time
```

Passes the call through to the dart_time_io module code.

filename	netCDF file name
model_time	The current time of the model state.

call write_model_time(ncid, dart_time)

```
integer,          intent(in) :: ncid
type(time_type),  intent(in) :: dart_time
```

Passes the call through to the dart_time_io module code.

ncid	handle to an open netCDF file
dart_time	The current time of the model state.

call end_model()

Does nothing.

6.153.5 Files

none

6.153.6 References

1. none

6.153.7 Private components

N/A

6.154 Contributors' guide

6.154.1 Contributing to DART

This section describes how you can contribute your work to DART. Because DART is an open-source project, your contributions are welcome. Many user-provided contributions have widely benefited the earth science community.

To ensure you aren't duplicating efforts, contact DAReS staff by emailing dart@ucar.edu before you expend considerable development time.

All of the source code is hosted in the DART GitHub repository.

Before you start developing, you should be familiar with the GitHub workflow. The GitHub worflow involves:

1. Creating a *fork* of the DART project. A fork is a publically visible copy of the repository that is stored in your GitHub account.
2. Creating a *branch* for your feature with an appropriate name for your project, and when you are finished with your changes you can commit them to your fork. After testing locally on your machine, you can push them to your fork.

   Important: At this point, everyone can see the changes you made on your fork.

   When you are ready to begin the conversation about merging your work into the original project (called the DART repository master), you can create a pull request, which will show your changes. After reviewing and testing your changes, the pull request will be addressed appropriately by the DART development team.

6.154.2 Keeping your work private until you publish

You may want to keep your work private until it is ready for publication or public viewing.

Follow these steps to hide sensitive code until you are ready to contribute it to DART your work has been published.

1. First, create a public fork of the DART repository by following the steps listed above.
2. Next, create a private repository on GitHub.com. The name of your private repository is arbitrary, since only you and your private collaborators can see it.
3. Add your public fork as a remote repository of your private repository. Your remote repository can be named "public_fork" or "upstream."
4. Add additional team members, if necessary.
5. Instead of pulling and pushing from your public fork, develop on your private repository.

Note: Only three collaborators are allowed on a free non-institutional private repository. DAReS staff can collaborate with you on your private repository, but keep this three collaborator limit in mind if you using a free GitHub account.

6.155 Requesting features and reporting bugs

DAReS staff uses GitHub's project management tools to track development.

To request a feature or to request a bug fix, use the GitHub issue tracker on the DART repository.

Make sure you explore the *closed* Issues! There's some good stuff in there.

6.156 Mailing list

DAReS staff send periodic updates to DART users. These updates summarize changes to the DART repository, including recent bug fixes.

To subscribe to the DART users mailing list, see Dart-users.

The mailing list is not generally used for discussion so emails are infrequent.

Discussions are encouraged on the GitHub DART Issues forum.

6.157 DART Manhattan Differences from Lanai Release Notes

6.157.1 Overview

This document includes an overview of the changes in the DART system since the Lanai release. For further details on any of these items look at the HTML documentation for that specific part of the system.

The two most significant changes in the Manhattan version of DART are it can support running models with a state vector larger than the memory of a single task, removing a limit from the Lanai version of DART. It also reads and writes NetCDF files directly instead of requiring a conversion from one file to another. There are many other smaller changes, detailed below.

Manhattan supported models:

- 9var
- bgrid_solo
- cam-fv
- cice
- clm
- cm1
- forced_lorenz_96
- ikeda
- lorenz_63
- lorenz_84
- lorenz_96
- lorenz_96_2scale
- lorenz_04
- mpas_atm (NetCDF overwrite not supported for update_u_from_reconstruct = .true.)

- null_model
- POP
- ROMS
- simple_advection
- wrf

If your model of interest is not on the list consider checking out the 'Classic' release of DART, which is Lanai plus bug fixes and minor enhancements. All models previously supported by Lanai are still in DART 'Classic'.

These are the major differences between the Lanai/Classic and Manhattan releases of DART:

- Read and write NetCDF restarts
- Calculation of forward operators
- Vertical conversion of observation locations
- Diagnostic file changes
- *State Stucture*
- model_mod interface changes
- Observation Quantity replaces Kind
- Perturbation of the state

6.157.2 NetCDF restart files

The programs filter and perfect_model_obs now read/write directly from NetCDF files rather than having to run converters (`model_to_dart` and `dart_to_model`). To facilitate this there is a new required call `add_domain` which must be called during `static_init_model`. It can be called multiple times in static_model_mod, e.g. once for each NetCDF file that contains state variables. There are three ways to add a domain:

- **From File** : This is for models which have NetCDF restart files
  - `dom_id = add_domain(template_file, num_vars, var_names, ... )`
- **From Spec** : Creates a skeleton structure for a domain (currently only used in bgrid_solo)
  - `dom_id = add_domain(num_vars, var_names, ... )`
  - `call add_dimension_to_variable(dom_id, var_id, dim_nam, dim_size)`
  - `call finished_adding_domain`
- **From Blank** : This is for small models such as lorenz_96 and no NetCDF restarts
  - `dom_id = add_domain(model_size)`

For models without NetCDF restarts, use `add_domain(model_size)`. This is the minimum amount of information needed by DART to create a netdcf file. For models with NetCDF restarts use `add_domain(info_file, num_vars, var_names)` which lets DART read the NetCDF dimensions for a list of variables from a file (`info_file`). There are several routines that can be used together to create a domain from a description: `add_domain, add_dimension_to_variable, finished_adding_domain`. This can be used in models such as bgrid_solo where the model is spun up in perfect_model_obs, but the model itself has variable structure (3D variables with names). See Additions/Changes to existing namelists for how to use NetCDF IO.

Note when using NetCDF restarts, inflation files are NetCDF also. The inflation mean and inflation standard deviation are in separate files when you use NetCDF restarts. See *Netcdf Inflation Files* for details.

6.157.3 Calculation of forward operators

The forward operator code in model_mod now operates on an array of state values. See *Forward Operators* for more detail about distributed vs. non-distributed forward operators. In distributed mode the forward operators for all ensemble members are calculated in the same `model_interpolate` call. In non-distributed mode, the forward operators for all ensemble members a task owns (1-ens_size) are calculated at once.

6.157.4 Vertical conversion of observation and state locations

The vertical conversion of observation locations is done before the assimilation by default. This can be changed by namelist options.

In Lanai this calculation is done in the assimilation as part of `get_close_obs` if a model_mod does vertical conversion. Note that not all models do vertical conversion or even have a concept of vertical location, but every model_mod must have the following routines:

```
call set_vertical_localization_coord(vert_localization_coord)

call convert_vertical_obs(ens_handle, num, locs, loc_qtys, loc_types, &
                          which_vert, status)

call convert_vertical_state(ens_handle, num, locs, loc_qtys, loc_indx, &
                            which_vert, istatus)
```

If there are NOT multiple choices for a vertical coordinate (e.g. cartesian, one dimensional), all these routines can be no-ops.

If there are multiple types of vertical coordinates, the convert routines must be able to convert between them. The 'set_vertical_localization_coord()' routine should be called from 'static_init_model()' to set what localization coordinate type is being requested.

The three routines related to vertical coordinates/localization choices are:

- `set_vert_localization_coord` - sets the vertical localization coordiate (not required if there is no vertical conversion)
- `convert_vertical_obs` - converts observation location to required vertical type (does nothing if there is no vertical conversion)
- `convert_vertical_state` - converts state vector location to required vertical type (does nothing if there is no vertical conversion)

6.157.5 DART diagnostic file changes

For large models DART format diagnostic files (`Prior_Diag.nc` and `Posterior_Diag.nc`) have been replaced with separate files for each copy that would have gone into Prior_Diag.nc and Posterior_Diag.nc.

For Prior_Diag.nc:

- **Mean and standard deviation**: preassim_mean.nc preassim_sd.nc
- **Inflation mean and standard deviation** (if state space inflation is used): preassim_priorinf_mean.nc preassim_priorinf_sd.nc
- **The number of ensemble members specifed** in filter_nml (num_output_state_members): preassim_member_####.nc

For Posterior_Diag.nc:

- **Mean and standard deviation**: postassim_mean.nc postassim_sd.nc
- **Inflation mean and standard deviation** (if state space inflation is used): postassim_priorinf_mean.nc postassim_priorinf_sd.nc
- **The number of ensemble members specifed** in filter_nml (num_output_state_members): postassim_member_####.nc

The `num_output_state_members` are not written separately from the restarts. Note that restarts will have been clamped if any clamping is applied (given as an arguement to add_domain). This is *different* to Posterior_Diag.nc which contains unclamped values. Note also that there are 2 more "stages" which might be output, in addition to the preassim and postassim discussed here.

For models with multiple domains the filenames above are appended with the domain number, e.g. preassim_mean.nc becomes preassim_mean_d01.nc, preassim_mean_d02.nc, etc.

Changes to nc_write_model_atts

`nc_write_model_atts` now has 2 arguments:

- ncid - open netcdf file identifier
- domain_id - domain number being written

The calling code will write the model state, so this routine should only add attributes and optionally, non-state information like grid arrays.

This routine will only be called if DART is creating an output NetCDF file from scratch. This may include any of the preassim, postassim, or output files.

Changes to nc_write_model_vars

`nc_write_model_vars` is currently unused (and in fact uncalled). It remains for possible future expansion.

6.157.6 Model_mod.f90 interface changes

The model_mod.f90 file contains all code that is specific to any particular model. The code in this file is highly constrained since these routines are *called by* other code in the DART system. All routine interfaces – the names, number of arguments, and the names of those arguments – must match the prescribed interfaces exactly. Since not all required interfaces are needed for every model there are default routines provided that can be referenced from a 'use' statement and then the routine name can be put in the module 'public' list without any code for that routine having to be written in the model_mod.f90 file.

The following 18 routines are required:

- static_init_model
- get_model_size
- get_state_meta_data
- shortest_time_between_assimilations
- model_interpolate
- end_model
- nc_write_model_atts
- nc_write_model_vars

- init_time
- init_conditions
- adv_1step
- pert_model_copies
- get_close_obs
- get_close_state
- convert_vertical_obs
- convert_vertical_state
- read_model_time
- write_model_time

Here is an example of code from the top of a model_mod file, including the modules where the default routines live and the required public list.

```
use     location_mod, only : location_type, get_close_type, &
                             get_close_obs, get_close_state, &
                             convert_vertical_obs, convert_vertical_state, &
                             set_location, set_location_missing, &
                             set_vertical_localization_coord
use    utilities_mod, only : register_module, error_handler, &
                             E_ERR, E_MSG
                             ! nmlfileunit, do_output, do_nml_file, do_nml_term,  &
                             ! find_namelist_in_file, check_namelist_read
use netcdf_utilities_mod, only : nc_add_global_attribute, nc_synchronize_file, &
                                 nc_add_global_creation_time, &
                                 nc_begin_define_mode, nc_end_define_mode
use state_structure_mod, only : add_domain
use ensemble_manager_mod, only : ensemble_type
use dart_time_io_mod, only  : read_model_time, write_model_time
use default_model_mod, only : pert_model_copies, nc_write_model_vars

implicit none
private

! required by DART code - will be called from filter and other
! DART executables.  interfaces to these routines are fixed and
! cannot be changed in any way.
public :: static_init_model,      &
          get_model_size,         &
          get_state_meta_data,    &
          shortest_time_between_assimilations, &
          model_interpolate,      &
          end_model,              &
          nc_write_model_atts,    &
          adv_1step,              &
          init_time,              &
          init_conditions

! public but in another module
```

(continues on next page)

(continued from previous page)

```
public :: nc_write_model_vars,    &
          pert_model_copies,      &
          get_close_obs,          &
          get_close_state,        &
          convert_vertical_obs,   &
          convert_vertical_state, &
          read_model_time,        &
          write_model_time
```

6.157.7 Observation quantity replaces kinds

Historically there has been confusion about the terms for specific observation types (which often include the name of the instrument collecting the data) and the generic quantity that is being measured (e.g. temperature). The previous terms for these were 'types' and 'kinds', respectively.

Starting with the Manhattan release we have tried to clarify the terminology and make the interfaces consistent. The following table lists the original names from the Lanai/Classic release and the replacement routines in Manhattan.

All code that is part of the DART code repository has been updated to use the replacment routines, but if you have your own utilities written using this code, you will need to update your code. Contact us (dart@ucar.edu) for help if you have any questions.

```
public subroutines, existing name on left, replacement on right:

assimilate_this_obs_kind()     =>     assimilate_this_type_of_obs(type_index)
evaluate_this_obs_kind()       =>       evaluate_this_type_of_obs(type_index)
use_ext_prior_this_obs_kind()  =>  use_ext_prior_this_type_of_obs(type_index)

get_num_obs_kinds()      =>  get_num_types_of_obs()
get_num_raw_obs_kinds()  =>  get_num_quantities()

get_obs_kind_index()     => get_index_for_type_of_obs(type_name)
get_obs_kind_name()      => get_name_for_type_of_obs(type_index)

get_raw_obs_kind_index()  =>  get_index_for_quantity(quant_name)
get_raw_obs_kind_name()   =>  get_name_for_quantity(quant_index)

get_obs_kind_var_type()  =>  get_quantity_for_type_of_obs(type_index)

get_obs_kind()      =>  get_obs_def_type_of_obs(obs_def)
set_obs_def_kind()  =>  set_obs_def_type_of_obs(obs_def)

get_kind_from_menu()      =>  get_type_of_obs_from_menu()

read_obs_kind()     =>   read_type_of_obs_table(file_unit, file_format)
write_obs_kind()    =>  write_type_of_obs_table(file_unit, file_format)

maps obs_seq nums to specific type nums, only used in read_obs_seq:
map_def_index()  => map_type_of_obs_table()

removed.  apparently unused, and simply calls get_obs_kind_name():
get_obs_name()
```

(continues on next page)

(continued from previous page)

```
apparently unused anywhere, removed:
add_wind_names()
do_obs_form_pair()

public integer parameter constants and subroutine formal argument names,
old on left, new on right:

KIND_ => QTY_
kind => quantity

TYPE_ => TYPE_
type => type_of_obs

integer parameters:
max_obs_generic  =>  max_defined_quantities  (not currently public, leave private)
max_obs_kinds    =>  max_defined_types_of_obs
```

6.157.8 Additions/changes to existing namelists

Quality_control_nml

These namelist options used to be in filter_nml, now they are in quality_control_nml.

```
&quality_control_nml
   input_qc_threshold          = 3,
   outlier_threshold           = 4,
   enable_special_outlier_code = .false.
/
```

New namelist variables

filter_nml

```
&filter_nml
   single_file_in               = .false.,
   single_file_out              = .false.,

   input_state_file_list        = 'null',
   output_state_file_list       = 'null',
   input_state_files            = 'null',
   output_state_files           = 'null',

   stages_to_write              = 'output'
   write_all_stages_at_end      = .false.
   output_restarts              = .true.
   output_mean                  = .true.
   output_sd                    = .true.

   perturb_from_single_instance = .false.,
```

(continues on next page)

(continued from previous page)

```
   perturbation_amplitude       = 0.2_r8,

   distributed_state            = .true.
/
```

Item	Type	Description
single_file_in	logical	True means that all of the restart and inflation information is read from a single NetCDF file. False means that you must specify an input_state_file_list and DART will be expecting input_{priorinf,postinf}_{mean,sd}.nc files for inflation.
single_file_out	logical	True means that all of the restart and inflation information is written to a single NetCDF file. False means that you must specify a output_state_files and DART will be output files specified in the list. Inflation files will be written in the form input_{priorinf,postinf}_{mean,sd}.nc.
input_state_files	character array	This is used for single file input for low order models. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .true.
output_state_files	character array	This is used for single file input for low order models. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .true.
input_state_file_list	character array	A list of files containing input model restarts. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .false.
output_state_file_list	character array	A list of files containing output model restarts. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .false.
stages_to_write	character array	Controls which stages to write. Case-insensitive input. Currently there are six options: • `input` – writes input mean and sd only • `forecast` – before assimilation, before prior inflation is applied • `preassim` – before assimilation, before prior inflation is applied • `postassim` – after assimilation, before posterior inflation is applied • `analysis` – after assimilation, after posterior inflation is applied

NetCDF reads and writes:

For **input** file names:

- give `input_state_file_list` a file for each domain, each of which contains a list of restart files. An example of an 'input_list.txt' might look something like :

```
advance_temp1/wrfinput_d01
advance_temp2/wrfinput_d01
advance_temp3/wrfinput_d01
advance_temp4/wrfinput_d01
advance_temp5/wrfinput_d01
....
```

- if no `input_state_file_list` is provided then default filenames will be used e.g. input_member_####.nc, input_priorinf_mean.nc, input_priorinf_sd.nc

For **output** file names:

- give `output_state_file_list` a file for each domain, each of which contains a list of restart files. An example of an 'input_list.txt' might for WRF might look something like :

```
wrf_out_d01.0001.nc
wrf_out_d01.0002.nc
wrf_out_d01.0003.nc
wrf_out_d01.0004.nc
wrf_out_d01.0005.nc
....
```

  if you would like to simply like to overwrite your previous data input_list.txt = output_list.txt

- if no `output_state_files` is provided then default filenames will be used e.g. output_member_####.nc, output_priorinf_mean.nc, output_priorinf_sd.nc

For small models you may want to use `single_file_in`, `single_file_out` which contains all copies needed to run filter.

State_vector_io_nml

```
&state_vector_io_nml
   buffer_state_io          = .false.,
   single_precision_output  = .false.,
/
```

When `buffer_state_io` is `.false.` the entire state is read into memory at once if .true. variables are read one at a time. If your model can not fit into memory at once this must be set to `.true.` .

`single_precision_output` allows you to run filter in double precision but write NetCDF files in single presision

Assim_tools_nml

```
&assim_tools_nml
   distribute_mean  = .true.
/
```

In previous DART releases, each processor gets a copy of the mean (in ens_mean_for_model). In RMA DART, the mean is distributed across all processors. However, a user can choose to have a copy of the mean on each processor by setting `distribute_mean = .false.` . Note that the mean state is accessed through `get_state` whether distribute_mean is `.true.` or `.false.`

Removed from existing namelists

```
&filter_nml
   input_qc_threshold           = 3,
   outlier_threshold            = 4,
   enable_special_outlier_code = .false.
   start_from_restart           = .false.
   output_inflation             = .true.
   output_restart               = .true.
   /
```

NOTE : `output_restart` has been renamed to `output_restarts`. **``output_inflation`` is no longer supported** and only writes inflation files if `inf_flavor > 1`

```
&ensemble_manager_nml
   single_restart_file_out = .true.
   perturbation_amplitude  = 0.2,
   /
```

```
&assim_manager_nml
   write_binary_restart_files = .true.,
   netCDF_large_file_support  = .false.
   /
```

6.157.9 Perturbations

The option to perturb one ensemble member to produce an ensemble is in filter_nml:`perturb_from_single_instance`. The model_mod interface is now `pert_model_copies` not `pert_model_state`. Each task perturbs every ensemble member for its own subsection of state. This is more complicated than the Lanai routine `pert_model_state`, where a whole state vector is available. If a model_mod does not provide a perturb interface, filter will do the perturbing with an amplitude set in filter_nml:perturbation_amplitude. Note the perturb namelist options have been removed from ensemble_manager_nml

6.158 Forward Operators

In data assimilation the process of computing an observation's expected value from a model state is called the `forward operator`.

In DART the forward operator code is kept separate from any model-specific details. This is to enable the same forward operator to run with any model of the same type (atmosphere, ocean, etc). This is accomplished by restricting the forward operator code to only interpolating state fields through a call to the `interpolate()` function. It can make no other assumptions about the model-specific details, for example the grid size or spacing.

A simple forward operator computes the expected value by calling the `model_mod.f90` `interpolate()` function on a field contained in the model state and returning the value. The base DART code does this without requiring the user to add additional code.

Generally a model's interpolate function should be written so it only does spatial interpolation on fields in the model state. Derived values and more complex computations should be placed in forward operator code.

A more complex forward operator either modifies the state value returned by `interpolate` or it calls interpolate for multiple state fields and calculates an expected value from those. The convention is to make a separate Fortran module for each forward operator and place the code in a `observations/forward_operator/obs_def_xxx_mod.f90` file, where xxx is specific to the type of forward operator.

Forward operator code will be called by other DART code and so the subroutine calling sequences are fixed. See documentation for *MODULE obs_def_mod* for more details.

The routine that must return an ensemble-sized array of expected values is usually called `get_expected_xxx`. It is passed a handle to access the state values and the observation location. It must return an array of expected values, one for each ensemble member, for that observation type at that location.

6.159 Approaches for Common Situations

Users who are familiar with data assimilation systems which are tightly integrated with a particular model often assume that it is either impossible or too inefficient to separate the model details from the forward operator code. In our experience with running DART across a wide range of models, this is not the case.

Here are approaches successfully used for some common situations.

6.159.1 Summing or Integrating a Column Value

Some instruments return an observation of a vertically integrated value, e.g. Total Precipitable Water. If this field is not already in the model state the forward operator needs to sample the model in a vertical column and sum the moisture values to return the total water in the column.

There are 2 common approaches to implementing code of this type.

1. Use a synthetic set of pressure or height vertical values at the horizontal location. The forward operator code uses the horizontal components of the requested location and constructs a set of synthetic locations with fixed steps in pressure or height in the vertical, calls interpolate for each vertical location, and sums the results.
2. Use a synthetic set of model levels in the vertical at the horizontal location. The forward operator code uses the horizontal components of the requested location and construct a set of synthetic locations with model level numbers in the vertical, calls interpolate for each vertical location, and sums the results. This approach has the advantage of minimizing the interpolation needed for the model-specific code to compute the vertical value.

In both cases the code must expect errors when the vertical coordinate moves outside of the model grid. Often the code will verify the horizontal location is within the grid, then loop over vertical levels to count the expected number of

successful locations. It can then allocate arrays of that size and call interpolate for the actual state vector fields required for this observation type.

6.159.2 Optional Precomputed Fields

If a complex computation is already being performed by the model for an observed type, those values can be added to the state vector and the forward operator code can interpolate values from that field directly. However if this forward operator is used when that field is not present it should have code to do a computation using other fields present in the model state.

See the radar forward operator for an example in the 'expected fall velocity' section. If this has already been computed by the model and is available in the model state, the code interpolates in this field and returns. If not, it interpolates values for other consitituent fields and does a generic computation.

6.160 Parallelism Implementation Details

The forward operator code is called for each observation to be assimilated. DART does this in parallel with each MPI task computing the values for all ensemble members for a subset of the observations. The state data for the ensembles is distributed across all tasks, so MPI one-sided communication is used for tasks to retrieve needed data values from other tasks.

See sections below for more details on other parallelism options and how this differs from previous versions of DART.

The forward operator is performed in `get_obs_ens_distrb_state`. A limited call tree for `get_obs_ens_distrb_state` is shown below.

The QC_LOOP is in `get_obs_ens_distrb_state` because the qc across the ensemble is known. The window opening and closing in `get_obs_ens_distrb_state` is as follows:

1. State window created (tasks can access other tasks's memory)
2. Forward operator called
3. State window destroyed (tasks can no longer access other tasks's memory)
4. QC calculated for the ensemble

However, there may be occasions where having only the first ens_size tasks perform the forward operator is desired. For example, if the forward operator is being read from a file, or the forward operator uses a large portion of the state. Or when debugging it may be easier to have 1 task per ensemble member.

To transpose and do the forward operators without using MPI one-sided communication, you can use the filter_nml namelist option `distribute_state = .false.` The process is the same as above except the window creation and destruction are transposing the state.

1. State window created (state ensemble is transposed var complete)
2. Forward operator called
3. State window destroyed (state ensemble is tranaposed to copy complete)
4. QC calculated for the ensemble

Note, that if you have fewer tasks than ensemble members some tasks will still be doing vectorized forward operators (because they own more than one ensemble member).

6.160.1 State access

Model_mod routines no longer get an array containing the state. The state is accessed through the function `get_state`.

```
x = get_state(i, state_handle)
```

where x is the state at index i. `state_handle` is passed from above. During model_interpolate `get_state` returns an array. Durring `get_state` returns a single value (the mean state).

6.161 Other Parallelism Options

In releases of DART before Manhattan (Lanai and earlier), transposes of the ensemble of states were used. If all data values for a single ensemble were needed an all-to-some transpose was used to collect the state vector data onto the first N tasks, where N is the ensemble size. If all ensemble values for a single item were needed a some-to-all transpose was used to spread a subset of the state vector data to all tasks, where tasks contained all ensemble member values for the items they had. This is `transpose mode` where the full state vector must fit into a single task's memory.

The current Manhattan release runs in `distributed mode` where state data is read into memory and distributed across all tasks. Each task stores all ensemble member values for a subset of the items in the state vector. The entire filter run keeps this data distribution and the data is only collected back into individual ensembles at output time. Collection and I/O can be done on a field-by-field basis so the entire state vector is never present at one time on a task.

In the Manhattan release it is still possible to run in transpose mode which avoids using MPI one-sided communication. It is generally only feasible for models with smaller state vector sizes since the first N tasks will need to store an entire state vector in memory. However, in most cases run time is still faster for distributed mode even for smaller models because computing forward operators for an entire ensemble at once often amortizes the overhead parts of the computation when computing N values.

Transpose mode collects the entire state vector for an ensemble onto the first N tasks. The forward operator code has access to the entire state vector and can reference any item with a simple array access. Each of the first N tasks computes the expected value for all observations for that single ensemble member. Tasks N+1 and higher are idle during this part of the execution.

In distributed mode each task computes the expected values for a subset of the observations. The observations are equally distributed round-robin. For K observations and T tasks each task gets K/T obs. MPI one-sided communication is used to retrieve needed state values from other tasks.

6.162 Netcdf Inflation Files

The filter_nml now read restart and inflation files directly from NetCDF files

Netcdf inflation files are no longer special files. DART format inflation files were always 2 copies in one file (mean and standard devation). Taking away this special status of inflation files has the advantage that all copies (restarts, ensemble mean, ensemble standard deviation, inflation mean, inflation sd, etc.) can all be treated the same for IO purposes. Since there are two inflation files when reading/writing netcdf the filenames are different to DART format restart files.

The names of the netcdf inflation files are now fixed.

Input inflation file names

The filter_nml option:

```
inf_in_file_name = prior_inflation_ics, post_inflation_ics
```

has been **deprecated** and for 1 domain filter is expecting to read:

input_{priorinf,postinf}_mean.nc
input_{priorinf,postinf}_sd.nc

For multiple domains filter is expecting to read:

input_{priorinf,postinf}_mean_d01.nc
input_{priorinf,postinf}_sd_d01.nc
input_{priorinf,postinf}_mean_d02.nc
input_{priorinf,postinf}_sd_d02.nc

where d0* is the domain number.

Output inflation file names

The filter_nml option:

`inf_out_file_name = prior_inflation_restart, post_inflation_restart`

has been **deprecated** and for 1 domain filter is expecting to read:

output_{priorinf,postinf}_mean.nc
output_{priorinf,postinf}_sd.nc

For multiple domains filter is expecting to write:

prior_inflation_restart_mean_d01
prior_inflation_restart_sd_d01
prior_inflation_restart_mean_d02
prior_inflation_restart_sd_d02

where d0* is the domain number.

6.163 State Stucture

state_structure_mod is a module that holds all the domain, variable, dimension info about the model_mods in the state. Note it stores only **metadata** about the state, not the actual state variables themselves.

It is the foundation for two parts of the code:

- Read/write state variables from/to netcdf files
- Calculate DART index from x,y,z variable indices and the inverse: x,y,z, variable from DART index.

Inside `static_init_model` a call is made to `add_domain`. This call is *required* as it communicates to the state structure that a new domain has been added to the state. The state structure keeps track of the number of domains in the state. These may be multiple domains in one model_mod, e.g. nested domains in WRF, or multiple model_mods, e.g. POP coupled with CAM. The minimum amount of information `add_domain` needs is model size which means

vector of length model size has been added to the state. This equivalent to Lanai where the only information filter has is that the model is a vector of length model_size. For models with netcdf restart files you supply `add_domain` with:

- a netcdf file
- the number of variables to read from the file
- the name of the variables
- Optionally:
  - the DART KINDS of the variables
  - clamping upper and lower bounds
  - update/not update this variable

For models that are spun up in perfect_model_obs you can manually describe the variables so you can create netcdf files containing the varibles in the model state, e.g. Temperature, Surface Pressure, etc. There are 3 steps to this process:

1. Supply `add_domain` with almost the same arguments as you would for a netcdf file, but skip the first arguement (netcdf filename).
2. For each variable, loop around the required number of dimensions and call `add_dimension_to_variable`
3. Call `finished_adding_domain` to let the state structure know that you have finished adding dimensions to variables.

6.163.1 DART index

To get the dart index for an i,j,k,variable in a domain use:

```
get_dart_vector_index(i, j, k, dom_id, var_id)
```

To get the i,j,k, variable, domain from the dart index use:

```
get_model_variable_indices(dart_index, i, j, k, var_id, dom_id)
```

Note That (i,j,k) needs to be converted to (lon, lat, lev) or to whatever grid the variable is on. `get_dim_name` can be used to get the dimension name from i,j,k if needed.

6.163.2 Unlimited dimensions: io vs model_mod routines

Some model restart files have an unlimited dimension. For IO purposes, e.g. creating netcdf files, the unlimited dimension is used. For state structure accessor functions called be the model_mod the unlimited dimension is ignored. So if you have a variable TEMPERATURE in your netcdf file, with dimensions (lon, lat, level, time) the IO routines will see a 4D variable, but `get_num_dims` used in model_mod will return 3D.

6.164 Filter async modes

6.164.1 Options for parallelism both in DART and in the model advances

Simplest case, async=0:

This is a single MPI executable, with each call to the model being simply a subroutine call from each MPI task.
To the DART mpi intro document
Parallel advance, async=2:

The filter executable is one MPI program, and the model is a single, sequential executable. Each MPI task uses the unix "system()" call to invoke a shell script (advance_model.csh) which runs the models as independent programs.
To the DART mpi intro document
Other views of how the async=2 option is structured; these may be more or less helpful.
Parallel advance, async=2:

Parallel advance, async=2, second version:

Parallel model advance, async=2, showing how data is communicated between filter and the model thru intermediate files. IC are 'initial condition' files, UD are 'updated' files.

Parallel model advance, async=4:

The filter executable is one MPI program, and the model is also an MPI program. The filter executable communicates with the runme_filter shell script, which sequentially invokes mpirun to advance each of the model runs, one per ensemble member, still using advance_model.csh.
To the DART mpi intro document
Parallel model advance, async=4, showing how data is communicated between filter and the model thru intermediate files. IC are 'initial condition' files, UD are 'updated' files.

6.165 Distributed State

The key part of DART is having a state that is physically distributed across processors. The location in memory of any part of the state vector (which processor and where in memory on that processor) is completely under the control of filter, not model_mod. This improvement was released in, and was the major reason for, Manhattan.

Implications of this:

- The model_mod never gets a whole state vector to use. So no whole vector for a forward operator, and no whole vector for the mean.
- The model_mod can not make any assumptions about the order of elements in the state. Currently, filter is ordering variables in the order they are listed in add_domain and with the dimension order of the netcdf file.

So, how does the model_mod access the state without having the vector and not knowing the state order? - state accessor routines.

6.165.1 State accessor routines

Getting the dart index

```
function get_dart_vector_index(i, j, k, dom_id, var_id)
```

get_dart_vector_index returns the dart index for a given i,j,k of a variable. Note if the variable is 1D j and k are ignored. If a variable is 2D k is ignored. Note only variables upto 3D are supported, but this could be extended to support up to 7 dimensional variables (or whatever fortran and netcdf will support).

Getting the state at a given dart index

```
function x = get_state(index, state_handle)
```

get_state returns the state x at the given *index*. state_handle is a derived type which contains the state information. *state_handle* is passed to the model_mod from above. *get_state* returns an array of values (the whole ensemble at index) during model_mod and a single value (the mean) during *get_close_obs* or *vert_convert*.

If you have an array of indices, for example a forward operator which is located in different levels on different ensemble members you can use *get_state_array*. An example of this is in CAM when an observation is in pressure, the level an observation is in depends on the state and so can vary across the ensemble.

```
subroutine get_state_array(x(:), index(:), state_handle)
```

The code inside *get_state_array* will do the minimum amount of communication to get you the indices you need. For example if

```
index = [3 4 3 3 4 3]
```

get_state_array will only do 2 mpi communications and return

```
x = [state(3), state(4), state(3), state(3), state(4), state(3)]
```

A limited module diagram is shown below. A -> B means A uses B:

filter_mod and *assim_tools_mod* take care of making data available for use with *get_state*. Note *get_state* will only return data during *model_interpolate*, *get_close_obs*, or *vert_convert*. If you use *get_state* outside these routines you will get and error.

Compilation Notes

The Remote Memory Access programming model DART employs uses mpi_windows. There are 2 ways to compile window mods for mpi and non-mpi filter. This is taken care of automatically when you run quickbuild.sh. However, if you use mpi, there is a choice of mpi_window mods:

- cray_win_mod.f90
- no_cray_win_mod.f90

We have these two modules that you can swap in your path_names files because the MPI 2 standard states:
Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI_ALLOC_MEM.
MPI_ALLOC_MEM uses cray pointers, thus we have supplied a window module that uses cray pointers. However, no_cray_win_mod.f90 is the default since some versions of gfortran (4.9.0) do not support cray pointers. These different modules will go away when we swap to MPI 3.

To change to cray_win_mod.f90 edit DART/build-system/buildfunctions.sh

```
windowsrc=cray_win
```

6.166 MODULE location_mod (channel)

6.166.1 Overview

THIS HAS NOT BEEN UPDATED YET - ONLY COPIED FROM 3D SPHERE VERSION

THIS HAS NOT BEEN UPDATED YET - ONLY COPIED FROM 3D SPHERE VERSION

THIS HAS NOT BEEN UPDATED YET - ONLY COPIED FROM 3D SPHERE VERSION

The DART framework needs to be able to compute distances between locations, to pass location information to and from the model interface code (model_mod.f90), and to be able to read and write location information to files. DART isolates all this location information into separate modules so that the main algorithms can operate with the same code independent of whether the model uses latitude/longitude/height, 1D unit sphere coordinates, cylindrical coordinates, etc. DART provides about half a dozen possible coordinate systems, and others can be added. The most common one for geophysical models is this one: threed_sphere.

This location module provides a representation of a physical location on a 3-D spherical shell, using latitude and longitude plus a vertical component with choices of vertical coordinate type such as pressure or height in meters. A type that abstracts the location is provided along with operators to set, get, read, write, and compute distances between locations. This is a member of a class of similar location modules that provide the same abstraction for different represenations of physical space.

Location-independent code

All types of location modules define the same module name `location_mod`. Therefore, the DART framework and any user code should include a Fortran 90 `use` statement of `location_mod`. The selection of which location module will be compiled into the program is controlled by which source file name is specified in the `path_names_xxx` file, which is used by the `mkmf_xxx` scripts.

All types of location modules define the same Fortran 90 derived type `location_type`. Programs that need to pass location information to subroutines but do not need to interpret the contents can declare, receive, and pass this derived type around in their code independent of which location module is specified at compile time. Model and location-independent utilities should be written in this way. However, as soon as the contents of the location type needs to be accessed by user code then it becomes dependent on the exact type of location module that it is compiled with.

Usage of distance routines

Regardless of the fact that the distance subroutine names include the string 'obs', there is nothing specific to observations in these routines. They work to compute distances between any set of locations. The most frequent use of these routines in the filter code is to compute the distance between a single observation and items in the state vector, and also between a single observation and other nearby observations. However, any source for locations is supported.

In simpler location modules (like the `oned` version) there is no need for anything other than a brute force search between the base location and all available state vector locations. However in the case of large geophysical models which typically use the `threed_sphere` locations code, the brute-force search time is prohibitive. The location code pre-processes all locations into a set of *bins* and then only needs to search the lists of locations in nearby bins when looking for locations that are within a specified distance.

The expected calling sequence of the `get_close` routines is as follows:

```
call get_close_maxdist_init()  ! is called before get_close_obs_init()
call get_close_obs_init()

call get_close_obs()           ! called many, many times

call get_close_obs_destroy()
```

In the `threed_sphere` implementation the first routine initializes some data structures, the second one bins up the list of locations, and then the third one is called multiple times to find all locations within a given radius of some reference location, and to optionally compute the exact separation distance from the reference location. The last routine deallocates the space. See the documentation below for the specific details for each routine.

All 4 of these routines must be present in every location module but in most other versions all but `get_close_obs()` are stubs. In this `threed_sphere` version of the locations module all are fully implemented.

Interaction with model_mod.f90 code

The filter and other DART programs could call the `get_close` routines directly, but typically do not. They declare them (in a `use` statement) to be in the `model_mod` module, and all model interface modules are required to supply them. However in many cases the model_mod only needs to contain another `use` statement declaring them to come from the `location_mod` module. Thus they 'pass through' the model_mod but the user does not need to provide a subroutine or any code for them.

However, if the model interface code wants to intercept and alter the default behavior of the get_close routines, it is able to. Typically the model_mod still calls the location_mod routines and then adjusts the results before passing them back to the calling code. To do that, the model_mod must be able to call the routines in the location_mod which have the same names as the subroutines it is providing. To allow the compiler to distinguish which routine is to be called where, we use the Fortran 90 feature which allows a module routine to be renamed in the use statement. For example, a common case is for the model_mod to want to supply additions to the get_close_obs() routine only. At the top of the model_mod code it would declare:

```
use location_mod, only :: location_get_close_obs => get_close_obs,    &
                          get_close_maxdist_init, get_close_obs_init, &
                          get_close_obs_destroy
```

That makes calls to the maxdist_init, init, and destroy routines simply pass through to the code in the location_mod, but the model_mod must supply a get_close_obs() subroutine. When it wants to call the code in the location_mod it calls `location_get_close_obs()`.

One use pattern is for the model_mod to call the location get_close_obs() routine without the `dist` argument. This returns a list of any potentially close locations without computing the exact distance from the base location. At this point the list of locations is a copy and the model_mod routine is free to alter the list in any way it chooses: it can change the locations to make certain types of locations appear closer or further away from the base location; it can convert the vertical coordinates into a common coordinate type so that calls to the `get_dist()` routine can do full 3d distance computations and not just 2d (the vertical coordinates must match between the base location and the locations in the list in order to compute a 3d distance). Then typically the model_mod code loops over the list calling the `get_dist()` routine to get the actual distances to be returned to the calling code. To localize in the vertical in a particular unit type, this is the place where the conversion to that vertical unit should be done.

Horizontal distance only

If *horiz_distance_only* is .true. in the namelist, then the vertical coordinate is ignored and only the great-circle distance between the two locations is computed, as if they were both on the surface of the sphere.

If *horiz_distance_only* is .false. in the namelist then the appropriate normalization constant determines the relative impact of vertical and horizontal separation. Since only a single localization distance is specified, and the vertical scales might have very different distance characteristics, the vert_normalization_xxx values can be used to scale the vertical appropriately to control the desired influence of observations in the vertical.

Precomputation for run-time search efficiency

For search efficiency all locations are pre-binned. The surface of the sphere is divided up into *nlon* by *nlat* boxes and the index numbers of all items (both state vector entries and observations) are stored in the appropriate box. To locate all points close to a given location, only the locations listed in the boxes within the search radius must be checked. This speeds up the computations, for example, when localization controls which state vector items are impacted by any given observation. The search radius is the localization distance and only those state vector items in boxes closer than the radius to the observation location are processed.

The default values have given good performance on many of our existing model runs, but for tuning purposes the box counts have been added to the namelist to allow adjustment. By default the code prints some summary information about how full the average box is, how many are empty, and how many items were in the box with the largest count. The namelist value *output_box_info* can be set to .true. to get even more information about the box statistics. The best performance will be obtained somewhere between two extremes; the worst extreme is all the points are located in just a few boxes. This degenerates into a (slow) linear search through the index list. The other extreme is a large number of empty or sparsely filled boxes. The overhead of creating, managing, and searching a long list of boxes will impact performance. The best performance lies somewhere in the middle, where each box contains a reasonable number of values, more or less evenly distributed across boxes. The absolute numbers for best performance will certainly vary from case to case.

For latitude, the *nlat* boxes are distributed evenly across the actual extents of the data. (Locations are in radians, so the maximum limits are the poles at $-\pi/2$ and $+\pi/2$. For longitude, the code automatically determines if the data is spread around more than half the sphere, and if so, the boxes are distributed evenly across the entire sphere (longitude range 0 to 2π). If the data spans less than half the sphere in longitude, the actual extent of the data is determined (including correctly handling the cyclic boundary at 0) and the boxes are distributed only within the data extent. This simplifies the actual distance calculations since the distance from the minimum longitude box to the maximum latitude box cannot be shorter going the other way around the sphere. In practice, for a global model the boxes are evenly distributed across the entire surface of the sphere. For local or regional models, the boxes are distributed only across the the extent of the local grid.

For efficiency in the case where the boxes span less than half the globe, the 3D location module needs to be able to determine the greatest longitude difference between a base point at latitude ϕ_s and all points that are separated from that point by a central angle of θ. We might also want to know the latitude, ϕ_f, at which the largest separation occurs. Note also that an intermediate form below allows the computation of the maximum longitude difference at a particular latitude.

The central angle between a point at latitude ϕ_s and a second point at latitude ϕ_f that are separated in longitude by $\Delta\lambda$ is:

$$\theta = cos^{-1}(sin\phi_s sin\phi_f + cos\phi_s cos\phi_f cos\Delta\lambda)$$

Taking the cos of both sides gives:

$$cos\theta = (sin\phi_s sin\phi_f + cos\phi_s cos\phi_f cos\Delta\lambda)$$

Solving for $cos\Delta\lambda$ gives:

$$cos\Delta\lambda = \frac{a - bsin\phi_f}{ccos\phi_f}$$

$$cos\Delta\lambda = \frac{a}{csec\phi_f} - \frac{b}{ctan\phi_f}$$

where $a = cos\theta$, $b = sin\phi_s$, and $c = cos\phi_s$. We want to maximize $\Delta\lambda$ which implies minimizing $cos\Delta\lambda$ subject to constraints.

Taking the derivative with respect to ϕ_f gives:

$$\frac{dcos\Delta\lambda}{d\phi_f} = \frac{a}{csec\phi_f tan\phi_f} - \frac{b}{csec^2\phi_f} = 0$$

Factoring out $sec\phi_f$ which can never be 0 and using the definitions of sec and tan gives:

$$\frac{asin\phi_f}{ccos\phi_f} - \frac{b}{ccos\phi_f} = 0$$

Solving in the constrained range from 0 to $\pi/2$ gives:

$$sin\phi_f = \frac{b}{a} = \frac{sin\phi_s}{cos\theta}$$

So knowing base point (ϕ_s, λ_s), latitude ϕ_f, and distance θ we can use the great circle equation to find the longitude difference at the greatest separation point:

$$\Delta\lambda = cos^{-1}\left(\frac{a - bsin\phi_f}{ccos\phi_f}\right)$$

Note that if the angle between the base point and a pole is less than or equal to the central angle, all longitude differences will occur as the pole is approached.

6.166.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand `&` and terminate with a slash `/` . Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&location_nml
   use_octree      = .false.
   nboxes          = 1000
   maxdepth        = 4
   filled          = 10
   output_box_info = .false.
   print_box_level = 0
   compare_to_correct = .false.
/
```

Items in this namelist either control the way in which distances are computed and/or influence the code performance.

Item	Type	Description
use_octree	logical	There are two variations of search code. For now, this must be set to .false.
nboxes	integer	An optimization parameter which controls how many boxes the space is divided up into for precomputing nearby points. Larger numbers use more memory but may make searching faster if the model contains a large grid.
filled	integer	An optimization parameter for the octree code. Set the lower item count limit where a box no longer splits.
output_box_info	logical	If true, print more details about the distribution of locations across the array of boxes.
print_box_level	integer	If output_box_info is true, controls the amount of output.
compare_to_correct	logical	If true do an exhaustive (and slow) search to ensure the results are the same as using optimized search code. Should only be used for debugging.

6.166.3 Other modules used

```
types_mod
utilities_mod
random_seq_mod
```

6.166.4 Public interfaces

`use location_mod, only :`	location_type
	get_close_type
	get_location
	set_location
	write_location
	read_location
	interactive_location
	set_location_missing
	query_location
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	get_close_obs_destroy
	get_dist
	LocationDims
	LocationName
	LocationLName
	horiz_dist_only
	vert_is_undef

continues on next page

Table 4 – continued from previous page

	vert_is_surface
	vert_is_pressure
	vert_is_scale_height
	vert_is_level
	vert_is_height
	VERTISUNDEF
	VERTISSURFACE
	VERTISLEVEL
	VERTISPRESSURE
	VERTISHEIGHT
	VERTISSCALEHEIGHT
	operator(==)
	operator(/=)

Namelist interface `&location_nml` must be read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

type location_type

```
   private
   real(r8) :: lon, lat, vloc
   integer  :: which_vert
end type location_type
```

Provides an abstract representation of physical location on a three-d spherical shell.

Component	Description
lon	longitude in radians
lat	latitude in radians
vloc	vertical location, units as selected by which_vert
which_vert	type of vertical location: -2=no specific vert location; -1=surface; 1=level; 2=pressure; 3=height, 4=scale height

The vertical types have parameters defined for them so they can be referenced by name instead of number.

type get_close_type

```
   private
   integer  :: num
   real(r8) :: maxdist
   integer, pointer :: lon_offset(:, :)
   integer, pointer :: obs_box(:)
```

(continues on next page)

(continued from previous page)

```
   integer, pointer :: count(:, :)
   integer, pointer :: start(:, :)
end type get_close_type
```

Provides a structure for doing efficient computation of close locations.

Component	Description
num	Number of locations in list
maxdist	Threshhold distance. Anything closer is close.
lon_offset	Dimensioned nlon by nlat. For a given offset in longitude boxes and difference in latitudes, gives max distance from base box to a point in offset box.
obs_box	Dimensioned num. Gives index of what box each location is in.
count	Dimensioned nlon by nlat. Number of obs in each box.
start	Dimensioned nlon by nlat. Index in straight storage list where obs in each box start.

var = get_location(loc)

```
real(r8), dimension(3)          :: get_location
type(location_type), intent(in) :: loc
```

Extracts the longitude and latitude (converted to degrees) and the vertical location from a location type and returns in a 3 element real array.

get_location	The longitude and latitude (in degrees) and vertical location
loc	A location type

var = set_location(lon, lat, vert_loc, which_vert)

```
type(location_type)   :: set_location
real(r8), intent(in)    :: lon
real(r8), intent(in)    :: lat
real(r8), intent(in)    :: vert_loc
integer,  intent(in)    :: which_vert
```

Returns a location type with the input longitude and latitude (input in degrees) and the vertical location of type specified by which_vert.

set_location	A location type
lon	Longitude in degrees
lat	Latitude in degrees
vert_loc	Vertical location consistent with which_vert
which_vert	The vertical location type

call write_location(locfile, loc [, fform, charstring])

```
integer,                intent(in)       ::  locfile
type(location_type),    intent(in)       ::  loc
character(len=*), optional, intent(in)   ::  fform
character(len=*), optional, intent(out)  ::  charstring
```

Given an integer IO channel of an open file and a location, writes the location to this file. The *fform* argument controls whether write is "FORMATTED" or "UNFORMATTED" with default being formatted. If the final *charstring* argument is specified, the formatted location information is written to the character string only, and the `locfile` argument is ignored.

`locfile`	the unit number of an open file.
`loc`	location type to be written.
fform	Format specifier ("FORMATTED" or "UNFORMATTED"). Default is "FORMATTED" if not specified.
charstring	Character buffer where formatted location string is written if present, and no output is written to the file unit.

var = read_location(locfile [, fform])

```
type(location_type)                    :: read_location
integer, intent(in)                    :: locfile
character(len=*), optional, intent(in) :: fform
```

Reads a location_type from a file open on channel locfile using format *fform* (default is formatted).

`read_location`	Returned location type read from file
`locfile`	Integer channel opened to a file to be read
fform	Optional format specifier ("FORMATTED" or "UNFORMATTED"). Default "FORMATTED".

call interactive_location(location [, set_to_default])

```
type(location_type), intent(out) :: location
logical, optional, intent(in)    :: set_to_default
```

Use standard input to define a location type. With set_to_default true get one with all elements set to 0.

`location`	Location created from standard input
set_to_default	If true, sets all elements of location type to 0

var = query_location(loc [, attr])

```
real(r8)                                :: query_location
type(location_type), intent(in)         :: loc
character(len=*), optional, intent(in) :: attr
```

Returns the value of which_vert, latitude, longitude, or vertical location from a location type as selected by the string argument attr. If attr is not present or if it is 'WHICH_VERT', the value of which_vert is converted to real and returned. Otherwise, attr='LON' returns longitude, attr='LAT' returns latitude and attr='VLOC' returns the vertical location.

query_location	Returns longitude, latitude, vertical location, or which_vert (converted to real)
loc	A location type
attr	Selects 'WHICH_VERT', 'LON', 'LAT' or 'VLOC'

var = set_location_missing()

```
type(location_type) :: set_location_missing
```

Returns a location with all elements set to missing values defined in types module.

set_location_missing	A location with all elements set to missing values

call get_close_maxdist_init(gc,maxdist, [maxdist_list])

```
type(get_close_type), intent(inout) :: gc
real(r8), intent(in)                :: maxdist
real(r8), intent(in), optional      :: maxdist_list(:)
```

Sets the threshhold distance. maxdist is in units of radians. Anything closer than this is deemed to be close. This routine must be called first, before the other get_close routines. It allocates space so it is necessary to call get_close_obs_destroy when completely done with getting distances between locations.

If the last optional argument is not specified, maxdist applies to all locations. If the last argument is specified, it must be a list of exactly the length of the number of specific types in the obs_kind_mod.f90 file. This length can be queried with the get_num_types_of_obs() function to get count of obs types. It allows a different maximum distance to be set per base type when get_close() is called.

gc	Data for efficiently finding close locations.
maxdist	Anything closer than this number of radians is a close location.
maxdist_list	If specified, must be a list of real values. The length of the list must be exactly the same length as the number of observation types defined in the obs_def_kind.f90 file. (See get_num_types_of_obs() to get count of obs types.) The values in this list are used for the obs types as the close distance instead of the maxdist argument.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type),             intent(inout) :: gc
integer,                          intent(in)    :: num
type(location_type), dimension(:) intent(in)    :: obs
```

Initialize storage for efficient identification of locations close to a given location. Allocates storage for keeping track of which 'box' each location in the list is in. Must be called after `get_close_maxdist_init`, and the list of locations here must be the same as the list of locations passed into `get_close_obs()`. If the list changes, `get_close_obs_destroy()` must be called, and both the initialization routines must be called again. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

`gc`	Structure that contains data to efficiently find locations close to a given location.
`num`	The number of locations in the list.
`obs`	The locations of each element in the list, not used in 1D implementation.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind, dist)

```
type(get_close_type),              intent(in)  :: gc
type(location_type),               intent(in)  :: base_obs_loc
integer,                           intent(in)  :: base_obs_kind
type(location_type), dimension(:), intent(in)  :: obs
integer,             dimension(:), intent(in)  :: obs_kind
integer,                           intent(out) :: num_close
integer,             dimension(:), intent(out) :: close_ind
real(r8), optional,  dimension(:), intent(out) :: dist
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

If called without the optional *dist* argument, all locations that are potentially close are returned, which is likely a superset of the locations that are within the threshold distance specified in the `get_close_maxdist_init()` call. This can be useful to collect a list of potential locations, and then to convert all the vertical coordinates into one consistent unit (pressure, height in meters, etc), and then the list can be looped over, calling get_dist() directly to get the exact distance, either including vertical or not depending on the setting of `horiz_dist_only`.

gc	Structure to allow efficient identification of locations close to a given location.
base_obs_loc	Single given location.
base_obs_kind	Kind of the single location.
obs	List of locations from which close ones are to be found.
obs_kind	Kind associated with locations in obs list.
num_close	Number of locations close to the given location.
close_ind	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call get_close_obs_destroy(gc)

```
type(get_close_type), intent(inout) :: gc
```

Releases memory associated with the gc derived type. Must be called whenever the list of locations changes, and then get_close_maxdist_init and get_close_obs_init must be called again with the new locations list.

gc	Data for efficiently finding close locations.

var = get_dist(loc1, loc2, [, kind1, kind2, no_vert])

```
real(r8)                        :: get_dist
type(location_type), intent(in) :: loc1
type(location_type), intent(in) :: loc2
integer, optional,   intent(in) :: kind1
integer, optional,   intent(in) :: kind2
logical, optional,   intent(in) :: no_vert
```

Returns the distance between two locations in radians. If horiz_dist_only is set to .TRUE. in the locations namelist, it computes great circle distance on sphere. If horiz_dist_only is false, then it computes an ellipsoidal distance with the horizontal component as above and the vertical distance determined by the types of the locations and the normalization constants set by the namelist for the different vertical coordinate types. The vertical normalization gives the vertical distance that is equally weighted as a horizontal distance of 1 radian. If *no_vert* is present, it overrides the value in the namelist and controls whether vertical distance is included or not.

The kind arguments are not used by the default location code, but are available to any user-supplied distance routines which want to do specialized calculations based on the kinds associated with each of the two locations.

loc1	First of two locations to compute distance between.
loc2	Second of two locations to compute distance between.
kind1	DART kind associated with location 1.
kind2	DART kind associated with location 2.
no_vert	If true, no vertical component to distance. If false, vertical component is included.
var	distance between loc1 and loc2.

var = vert_is_undef(loc)

```
logical                          :: vert_is_undef
type(location_type), intent(in) :: loc
```

Returns true if which_vert is set to undefined, else false. The meaning of 'undefined' is specific; it means there is no particular vertical location associated with this type of measurement; for example a column-integrated value.

vert_is_undef	Returns true if vertical coordinate is set to undefined.
loc	A location type

var = vert_is_surface(loc)

```
logical                          :: vert_is_surface
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for surface, else false.

vert_is_surface	Returns true if vertical coordinate type is surface
loc	A location type

var = vert_is_pressure(loc)

```
logical                          :: vert_is_pressure
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for pressure, else false.

vert_is_pressure	Returns true if vertical coordinate type is pressure
loc	A location type

var = vert_is_scale_height(loc)

```
logical                          :: vert_is_scale_height
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for scale_height, else false.

vert_is_scale_height	Returns true if vertical coordinate type is scale_height
loc	A location type

var = vert_is_level(loc)

```
logical                          :: vert_is_level
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for level, else false.

vert_is_level	Returns true if vertical coordinate type is level
loc	A location type

var = vert_is_height(loc)

```
logical                          :: vert_is_height
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for height, else false.

vert_is_height	Returns true if vertical coordinate type is height
loc	A location type

var = has_vertical_localization()

```
logical :: has_vertical_localization
```

Returns .TRUE. if the namelist variable `horiz_dist_only` is .FALSE. meaning that vertical separation between locations is going to be computed by `get_dist()` and by `get_close_obs()`.

This routine should perhaps be renamed to something like 'using_vertical_for_distance' or something similar. The current use for it is in the localization code inside filter, but that doesn't make this a representative function name. And at least in current usage, returning the opposite setting of the namelist item makes the code read more direct (fewer double negatives).

loc1 == loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types have identical values, else false.

loc1 /= loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types do NOT have identical values, else false.

```
integer, parameter :: VERTISUNDEF       = -2
integer, parameter :: VERTISSURFACE     = -1
integer, parameter :: VERTISLEVEL       =  1
integer, parameter :: VERTISPRESSURE    =  2
integer, parameter :: VERTISHEIGHT      =  3
integer, parameter :: VERTISSCALEHEIGHT =  4
```

Constant parameters used to differentiate vertical types.

```
integer, parameter :: LocationDims = 3
```

This is a **constant**. Contains the number of real values in a location type. Useful for output routines that must deal transparently with many different location modules.

```
character(len=129), parameter :: LocationName = "loc3Dsphere"
```

This is a **constant**. A parameter to identify this location module in output metadata.

```
character(len=129), parameter :: LocationLName =

       "threed sphere locations: lon, lat, vertical"
```

This is a **constant**. A parameter set to "threed sphere locations: lon, lat, vertical" used to identify this location module in output long name metadata.

6.166.5 Files

filename	purpose
input.nml	to read the location_mod namelist

6.166.6 References

1. none

6.166.7 Private components

N/A

6.167 MODULE location_mod

6.167.1 Overview

DART provides a selection of options for the coordinate system in which all observations and all model state vector locations are described. All executables are built with a single choice from the available location modules. The names of these modules are all `location_mod`.

6.167.2 Introduction

The core algorithms of DART work with many different models which have a variety of coordinate systems. This directory provides code for creating, setting/getting, copying location information (coordinates) independently of the actual specific coordinate information. It also contains distance routines needed by the DART algorithms.

Each of the different location_mod.f90 files provides the same set of interfaces and defines a 'module location_mod', so by selecting the proper version in your path_names_xxx file you can compile your model code with the main DART routines.

- *MODULE location_mod (threed_sphere)*: The most frequently used version for real-world 3d models. It uses latitude and longitude for horizontal coordinates, plus a vertical coordinate which can be meters, pressure, model level, surface, or no specific vertical location.
- *MODULE (1D) location_mod*: The most frequently used for small models (e.g. the Lorenz family). It has a cyclic domain from 0 to 1.
- others:
  - *MODULE location_mod (threed_cartesian)*: A full 3D X,Y,Z coordinate system.
  - column: no x,y but 1d height, pressure, or model level for vertical.
  - annulus: a hollow 3d cylinder with azimuth, radius, and depth.

- twod: a periodic 2d domain with x,y coordinates between 0 and 1.
- twod_sphere: a 2d shell with latitude, longitude pairs.
- threed: a periodic 3d domain with x,y,z coordinates between 0 and 1.
- *MODULE location_mod (channel)*: a 3d domain periodic in x, limited in y, and unlimited z.

Other schemes can be added, as needed by the models. Possible ideas are a non-periodic version of the 1d, 2d cartesian versions. Email dart at ucar.edu if you have a different coordinate scheme which we might want to support.

6.167.3 Namelist

Each location module option has a different namelist. See the specific documentation for the location option of choice.

6.167.4 Files

- none

6.167.5 References

- none

6.167.6 Private components

N/A

6.168 MODULE (1D) location_mod

6.168.1 Overview

The DART framework needs to be able to compute distances between locations, to pass location information to and from the model interface code (in model_mod.f90), and to be able to read and write location information to files. DART isolates all this location information into separate modules so that the main algorithms can operate with the same code independent of whether the model uses latitude/longitude/height, one-d unit sphere coordinates, cylindrical coordinates, etc. DART provides about half a dozen possible coordinate systems, and others can be added.

This locations module provides a representation of a physical location on a periodic 1D domain with location values between 0 and 1. A type that abstracts the location is provided along with operators to set, get, read, write, and compute distances between locations. This is a member of a class of similar location modules that provide the same abstraction for different represenations of physical space.

All possible location modules define the same module name `location_mod`. Therefore, the DART framework and any user code should include a Fortran 90 'use' statement of 'location_mod'. The selection of exactly which location module is compiled is specified by the source file name in the `path_names_xxx` file, which is read by the `mkmf_xxx` scripts.

The model-specific `model_mod.f90` files need to define four `get_close` routines, but in most cases they can simply put a `use` statement at the top which uses the routines in the locations module, and they do not have to provide any additional code.

However, if the model interface code wants to intercept and alter the default behavior of the get_close routines, they are able to. The correct usage of the `get_close` routines is as follows:

```
call get_close_maxdist_init()  ! must be called before get_close_obs_init()
call get_close_obs_init()
...
call get_close_obs()           ! many, many times
...
call get_close_obs_destroy()
```

Regardless of the fact that the names include the string 'obs', they are intended for use with any group of locations in the system, frequently state vector items or observations, but any location is acceptable.

6.168.2 Namelist

This version of the locations module does not have any namelist input.

6.168.3 Other modules used

```
types_mod
utilities_mod
random_seq_mod
```

6.168.4 Public interfaces

use location_mod, only :	location_type
	get_close_type
	get_location
	set_location
	write_location
	read_location
	interactive_location
	set_location_missing
	query_location
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	get_close_obs_destroy
	get_dist
	LocationDims
	LocationName
	LocationLName
	horiz_dist_only
	vert_is_undef
	vert_is_surface
	vert_is_pressure
	vert_is_level
	vert_is_height
	VERTISUNDEF
	VERTISSURFACE
	VERTISLEVEL
	VERTISPRESSURE
	VERTISHEIGHT
	operator(==)

There is currently no namelist interface for the 1D location module.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

type location_type

```
   private
   real(r8) :: x
end type location_type
```

Provides an abstract representation of physical location on a one-dimensional periodic domain.

Component	Description
x	Location has range 0 to 1

type get_close_type

```
   private
   integer  :: num
   real(r8) :: maxdist
end type get_close_type
```

Provides a structure for doing efficient computation of close locations. Doesn't do anything in the 1D implementation except provide appropriate stubs.

Component	Description
num	Number of locations in list
maxdist	Threshhold distance. Anything closer is close.

var = get_location(loc)

```
real(r8)                        :: get_location
type(location_type), intent(in) :: loc
```

Extracts the real location value, range 0 to 1, from a location type.

`get_location`	The real value for a location
`loc`	A location derived type

var = set_location(x)

```
type(location_type)   :: set_location
real(r8), intent(in)  :: x
```

Returns a location type with the location x.

set_location	A location derived type
x	Location value in the range 0. to 1.

call write_location(locfile, loc [, fform, charstring])

```
integer,                intent(in)       ::  locfile
type(location_type),    intent(in)       ::  loc
character(len=*), optional, intent(in)   ::  fform
character(len=*), optional, intent(out)  ::  charstring
```

Given an integer IO channel of an open file and a location, writes the location to this file. The *fform* argument controls whether write is "FORMATTED" or "UNFORMATTED" with default being formatted. If the final *charstring* argument is specified, the formatted location information is written to the character string only, and the `locfile` argument is ignored.

locfile	the unit number of an open file.
loc	location type to be written.
fform	Format specifier ("FORMATTED" or "UNFORMATTED"). Default is "FORMATTED" if not specified.
charstring	Character buffer where formatted location string is written if present, and no output is written to the file unit.

var = read_location(locfile [, fform])

```
type(location_type)                    :: read_location
integer, intent(in)                    :: locfile
character(len=*), optional, intent(in) :: fform
```

Reads a location_type from a file open on channel locfile using format *fform* (default is formatted).

read_location	Returned location type read from file
locfile	Integer channel opened to a file to be read
fform	Optional format specifier ("FORMATTED" or "UNFORMATTED"). Default "FORMATTED".

call interactive_location(location [, set_to_default])

```
type(location_type), intent(out) :: location
logical, optional, intent(in)    :: set_to_default
```

Use standard input to define a location type. With set_to_default true get one with all elements set to 0.

`location`	Location created from standard input
set_to_default	If true, sets all elements of location type to 0

var = query_location(loc [, attr])

```
real(r8)                                :: query_location
type(location_type), intent(in)         :: loc
character(len=*), optional, intent(in) :: attr
```

Returns the location value if attr = 'X' or if attr is not passed.

`query_location`	Returns value of x.
`loc`	A location type
attr	Selects 'X'

var = set_location_missing()

```
type(location_type) :: set_location_missing
```

Returns a location with location set to missing value from types_mod.

`set_location_missing`	A location set to missing value

call get_close_maxdist_init(gc,maxdist , [maxdist_list])

```
type(get_close_type), intent(inout) :: gc
real(r8), intent(in)                :: maxdist
real(r8), intent(in), optional      :: maxdist_list(:)
```

Sets the threshhold distance. Anything closer than this is deemed to be close. This routine must be called first, before the other `get_close` routines. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

gc	Data for efficiently finding close locations.
maxdist	Anything closer than this distance is a close location.
maxdist_list	Ignored for this location type.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type),              intent(inout) :: gc
integer,                           intent(in)    :: num
type(location_type), dimension(:) intent(in)    :: obs
```

Initialize storage for efficient identification of locations close to a given location. The oned implementation is minimal and just records the number of locations here. Must be called after `get_close_maxdist_init`, and the list of locations here must be the same as the list of locations passed into `get_close_obs()`. If the list changes, `get_close_obs_destroy()` must be called, and both the initialization routines must be called again. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

gc	Structure that contains data to efficiently find locations close to a given location.
num	The number of locations in the list.
obs	The locations of each element in the list, not used in 1D implementation.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind, dist)

```
type(get_close_type),              intent(in)  :: gc
type(location_type),               intent(in)  :: base_obs_loc
integer,                           intent(in)  :: base_obs_kind
type(location_type), dimension(:), intent(in)  :: obs
integer, dimension(:),             intent(in)  :: obs_kind
integer,                           intent(out) :: num_close
integer, dimension(:),             intent(out) :: close_ind
real(r8), dimension(:),            intent(out) :: dist
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

gc	Structure to allow efficient identification of locations close to a given location.
base_obs_loc	Single given location.
base_obs_kind	Kind of the single location.
obs	List of locations from which close ones are to be found.
obs_kind	Kind associated with locations in obs list.
num_close	Number of locations close to the given location.
close_ind	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call get_close_obs_destroy(gc)

```
type(get_close_type), intent(inout) :: gc
```

Releases memory associated with the gc derived type. Must be called whenever the list of locations changes, and then get_close_maxdist_init and get_close_obs_init must be called again with the new locations list.

gc	Data for efficiently finding close locations.

var = get_dist(loc1, loc2, [, kind1, kind2])

```
real(r8)                        :: get_dist
type(location_type), intent(in) :: loc1
type(location_type), intent(in) :: loc2
integer, optional,   intent(in) :: kind1
integer, optional,   intent(in) :: kind2
```

Return the distance between 2 locations. Since this is a periodic domain, the shortest distance may wrap around.

The kind arguments are not used by the default location code, but are available to any user-supplied distance routines which want to do specialized calculations based on the kinds associated with each of the two locations.

loc1	First of two locations to compute distance between.
loc2	Second of two locations to compute distance between.
kind1	DART kind associated with location 1.
kind2	DART kind associated with location 2.
var	distance between loc1 and loc2.

var = vert_is_undef(loc)

```
logical                          :: vert_is_undef
type(location_type), intent(in) :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

vert_is_undef	Always returns .FALSE.
loc	A location type

var = vert_is_surface(loc)

```
logical                          :: vert_is_surface
type(location_type), intent(in) :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

vert_is_surface	Always returns .FALSE.
loc	A location type

var = vert_is_pressure(loc)

```
logical                          :: vert_is_pressure
type(location_type), intent(in) :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

vert_is_pressure	Always returns .FALSE.
loc	A location type

var = vert_is_level(loc)

```
logical                          :: vert_is_level
type(location_type), intent(in) :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

vert_is_level	Always returns .FALSE.
loc	A location type

var = vert_is_height(loc)

```
logical                          :: vert_is_height
type(location_type), intent(in) :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

vert_is_height	Always returns .FALSE.
loc	A location type

var = has_vertical_localization()

```
logical :: has_vertical_localization
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

See note in threed_sphere locations module about the function name.

loc1 == loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types have identical values, else false.

loc1 /= loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types do NOT have identical values, else false.

```
integer, parameter :: VERTISUNDEF    = -2
integer, parameter :: VERTISSURFACE  = -1
integer, parameter :: VERTISLEVEL    =  1
integer, parameter :: VERTISPRESSURE =  2
integer, parameter :: VERTISHEIGHT   =  3
```

This locations module has no vertical coordinate, but for compatibility with other location modules, these are defined.

```
integer, parameter :: LocationDims = 1
```

This is a **constant**. Contains the number of real values in a location type. Useful for output routines that must deal transparently with many different location modules.

```
character(len=129), parameter :: LocationName = "loc1d"
```

This is a **constant**. A parameter to identify this location module in output metadata.

```
character(len=129), parameter :: LocationLName = "location on unit circle"
```

This is a **constant**. A parameter to identify this location module in output long name metadata.

6.168.5 Files

None.

6.168.6 References

1. none

6.168.7 Private components

N/A

6.169 MODULE location_mod (threed_cartesian)

6.169.1 Overview

The DART framework needs to be able to compute distances between locations, to pass location information to and from the model interface code (model_mod.f90), and to be able to read and write location information to files. DART isolates all this location information into separate modules so that the main algorithms can operate with the same code independent of whether the model uses latitude/longitude/height, 1D unit cartesian coordinates, cylindrical coordinates, etc. DART provides about half a dozen possible coordinate systems, and others can be added. The most common one for geophysical models is the *MODULE location_mod (threed_sphere)* version. This document describes an alternative 3D cartesian coordinate system.

Note that only one location module can be compiled into any single DART executable, and most earth observational data is generated in [latitude, longitude, vertical pressure or height] coordinates - the threed_sphere option. The cartesian and 3D sphere locations cannot be mixed or used together.

This location module provides a representation of a physical location in an [X, Y, Z] cartesian coordinate space. A type that abstracts the location is provided along with operators to set, get, read, write, and compute distances between locations. This is a member of a class of similar location modules that provide the same abstraction for different represenations of physical space.

Location-independent code

All types of location modules define the same module name `location_mod`. Therefore, the DART framework and any user code should include a Fortran 90 `use` statement of `location_mod`. The selection of which location module will be compiled into the program is controlled by which source file name is specified in the `path_names_xxx` file, which is used by the `mkmf_xxx` scripts.

All types of location modules define the same Fortran 90 derived type `location_type`. Programs that need to pass location information to subroutines but do not need to interpret the contents can declare, receive, and pass this derived type around in their code independent of which location module is specified at compile time. Model and location-independent utilities should be written in this way. However, as soon as the contents of the location type needs to be accessed by user code then it becomes dependent on the exact type of location module that it is compiled with.

Usage of distance routines

Regardless of the fact that the distance subroutine names include the string 'obs', there is nothing specific to observations in these routines. They work to compute distances between any set of locations. The most frequent use of these routines in the filter code is to compute the distance between a single observation and items in the state vector, and also between a single observation and other nearby observations. However, any source for locations is supported.

In simpler location modules (like the `oned` version) there is no need for anything other than a brute force search between the base location and all available state vector locations. However in the case of large geophysical models which typically use the `threed_cartesian` locations code, the brute-force search time is prohibitive. The location code pre-processes all locations into a set of *bins* and then only needs to search the lists of locations in nearby bins when looking for locations that are within a specified distance.

The expected calling sequence of the `get_close` routines is as follows:

```
call get_close_maxdist_init()  ! is called before get_close_obs_init()
call get_close_obs_init()

call get_close_obs()           ! called many, many times

call get_close_obs_destroy()
```

In the threed_cartesian implementation the first routine initializes some data structures, the second one bins up the list of locations, and then the third one is called multiple times to find all locations within a given radius of some reference location, and to optionally compute the exact separation distance from the reference location. The last routine deallocates the space. See the documentation below for the specific details for each routine.

All 4 of these routines must be present in every location module but in most other versions all but get_close_obs() are stubs. In this threed_cartesian version of the locations module all are fully implemented.

Interaction with model_mod.f90 code

The filter and other DART programs could call the get_close routines directly, but typically do not. They declare them (in a use statement) to be in the model_mod module, and all model interface modules are required to supply them. However in many cases the model_mod only needs to contain another use statement declaring them to come from the location_mod module. Thus they 'pass through' the model_mod but the user does not need to provide a subroutine or any code for them.

However, if the model interface code wants to intercept and alter the default behavior of the get_close routines, it is able to. Typically the model_mod still calls the location_mod routines and then adjusts the results before passing them back to the calling code. To do that, the model_mod must be able to call the routines in the location_mod which have the same names as the subroutines it is providing. To allow the compiler to distinguish which routine is to be called where, we use the Fortran 90 feature which allows a module routine to be renamed in the use statement. For example, a common case is for the model_mod to want to supply additions to the get_close_obs() routine only. At the top of the model_mod code it would declare:

```
use location_mod, only :: location_get_close_obs => get_close_obs,    &
                          get_close_maxdist_init, get_close_obs_init, &
                          get_close_obs_destroy
```

That makes calls to the maxdist_init, init, and destroy routines simply pass through to the code in the location_mod, but the model_mod must supply a get_close_obs() subroutine. When it wants to call the code in the location_mod it calls location_get_close_obs().

One use pattern is for the model_mod to call the location get_close_obs() routine without the dist argument. This returns a list of any potentially close locations without computing the exact distance from the base location. At this point the list of locations is a copy and the model_mod routine is free to alter the list in any way it chooses: for example, it can change the locations to make certain types of locations appear closer or further away from the base location. Then typically the model_mod code loops over the list calling the get_dist() routine to get the actual distances to be returned to the calling code.

Horizontal distance only

This option is not supported for the threed_cartesian option.

Precomputation for run-time search efficiency

For search efficiency all locations are pre-binned. For the non-octree option, the total list of locations is divided up into *nx* by *ny* by *nz* boxes and the index numbers of all items (both state vector entries and observations) are stored in the appropriate box. To locate all points close to a given location, only the locations listed in the boxes within the search radius must be checked. This speeds up the computations, for example, when localization controls which state vector items are impacted by any given observation. The search radius is the localization distance and only those state vector items in boxes closer than the radius to the observation location are processed.

The default values have given good performance on many of our existing model runs, but for tuning purposes the box counts have been added to the namelist to allow adjustment. By default the code prints some summary information about how full the average box is, how many are empty, and how many items were in the box with the largest count. The namelist value *output_box_info* can be set to .true. to get even more information about the box statistics. The best performance will be obtained somewhere between two extremes; the worst extreme is all the points are located in just a few boxes. This degenerates into a (slow) linear search through the index list. The other extreme is a large number of empty or sparsely filled boxes. The overhead of creating, managing, and searching a long list of boxes will impact performance. The best performance lies somewhere in the middle, where each box contains a reasonable number of values, more or less evenly distributed across boxes. The absolute numbers for best performance will certainly vary from case to case.

6.169.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&location_nml
   nx                  = 10
   ny                  = 10
   nz                  = 10
   x_is_periodic       = .false.
   min_x_for_periodic  = -888888.0
   max_x_for_periodic  = -888888.0
   y_is_periodic       = .false.
   min_y_for_periodic  = -888888.0
   max_y_for_periodic  = -888888.0
   z_is_periodic       = .false.
   min_z_for_periodic  = -888888.0
   max_z_for_periodic  = -888888.0
   compare_to_correct  = .false.
   output_box_info     = .false.
   print_box_level     = 0
   debug               = 0
  /
```

Items in this namelist either control the way in which distances are computed and/or influence the code performance.

Item	Type	Description
nx, ny, nz	integer	The number of boxes in each dimension to use to speed the searches. This is **not** the number of gridcells.
x_is_periodic, y_is_periodic, z_is_periodic	logical	If .true., the domain wraps in the coordinate.
min_x_for_periodic, max_x_for_periodic	real(r8)	The minimum and maximum values that are considered to be identical locations if `x_is_periodic = .true.`
min_y_for_periodic, max_y_for_periodic	real(r8)	The minimum and maximum values that are considered to be identical locations if `y_is_periodic = .true.`
min_z_for_periodic, max_z_for_periodic	real(r8)	The minimum and maximum values that are considered to be identical locations if `z_is_periodic = .true.`
compare_to_correct	logical	If true, do an exhaustive search for the closest point. Only useful for debugging because the performance cost is prohibitive.
output_box_info	logical	Print out debugging info.
print_box_level	logical	If output_box_info is true, how detailed should the output be.
debug	integer	The higher the number, the more verbose the run-time output. 0 (zero) is the minimum run-time output.

6.169.3 Other modules used

```
types_mod
utilities_mod
random_seq_mod
```

6.169.4 Public interfaces

use location_mod, only :	location_type
	get_close_type
	get_location
	set_location
	write_location
	read_location
	interactive_location
	set_location_missing
	query_location
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	get_close_obs_destroy
	get_dist
	LocationDims
	LocationName
	LocationLName
	horiz_dist_only
	vert_is_undef
	vert_is_surface
	vert_is_pressure
	vert_is_scale_height
	vert_is_level
	vert_is_height
	operator(==)
	operator(/=)

Namelist interface `&location_nml` must be read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

type location_type

```
   private
   real(r8) :: x, y, z
end type location_type
```

Provides an abstract representation of physical location in a 3D cartesian space.

Component	Description
x, y, z	location in each dimension

type get_close_type

```
   private
   integer, pointer  :: loc_box(:)          ! (nloc); List of loc indices in boxes
   integer, pointer  :: count(:, :, :)      ! (nx, ny, nz); # of locs in each box
   integer, pointer  :: start(:, :, :)      ! (nx, ny, nz); Start of list of locs in␣
↪this box
   real(r8)          :: bot_x, top_x        ! extents in x, y, z
   real(r8)          :: bot_y, top_y
   real(r8)          :: bot_z, top_z
   real(r8)          :: x_width, y_width, z_width    ! widths of boxes in x,y,z
   real(r8)          :: nboxes_x, nboxes_y, nboxes_z ! based on maxdist how far to search
end type get_close_type
```

Provides a structure for doing efficient computation of close locations.

var = get_location(loc)

```
real(r8), dimension(3)          :: get_location
type(location_type), intent(in) :: loc
```

Extracts the x, y, z locations from a location type and returns in a 3 element real array.

`get_location`	The x,y,z values
`loc`	A location type

var = set_location(x, y, z) var = set_location(lon, lat, height, radius)

```
type(location_type) :: set_location
real(r8), intent(in)    :: x
real(r8), intent(in)    :: y
real(r8), intent(in)    :: z
```

or

```
type(location_type) :: set_location
real(r8), intent(in)    :: lon
real(r8), intent(in)    :: lat
real(r8), intent(in)    :: height
real(r8), intent(in)    :: radius
```

Returns a location type with the input [x,y,z] or allows the input to be specified as locations on the surface of a sphere with a specified radius and height above the surface.

`set_location`	A location type
`x, y, z`	Coordinates along each axis
`lon, lat`	Longitude, Latitude in degrees
`height`	Vertical location in same units as radius (e.g. meters)
`radius`	The radius of the sphere in same units as height (e.g. meters)

call write_location(locfile, loc [, fform, charstring])

```
integer,                intent(in)       ::  locfile
type(location_type),    intent(in)       ::  loc
character(len=*), optional, intent(in)   ::  fform
character(len=*), optional, intent(out)  ::  charstring
```

Given an integer IO channel of an open file and a location, writes the location to this file. The *fform* argument controls whether write is "FORMATTED" or "UNFORMATTED" with default being formatted. If the final *charstring* argument is specified, the formatted location information is written to the character string only, and the `locfile` argument is ignored.

`locfile`	the unit number of an open file.
`loc`	location type to be written.
fform	Format specifier ("FORMATTED" or "UNFORMATTED"). Default is "FORMATTED" if not specified.
charstring	Character buffer where formatted location string is written if present, and no output is written to the file unit.

var = read_location(locfile [, fform])

```
type(location_type)                    :: read_location
integer, intent(in)                    :: locfile
character(len=*), optional, intent(in) :: fform
```

Reads a location_type from a file open on channel locfile using format *fform* (default is formatted).

`read_location`	Returned location type read from file
`locfile`	Integer channel opened to a file to be read
fform	Optional format specifier ("FORMATTED" or "UNFORMATTED"). Default "FORMATTED".

call interactive_location(location [, set_to_default])

```
type(location_type), intent(out) :: location
logical, optional, intent(in)    :: set_to_default
```

Use standard input to define a location type. With set_to_default true get one with all elements set to 0.

`location`	Location created from standard input
set_to_default	If true, sets all elements of location type to 0

var = query_location(loc [, attr])

```
real(r8)                               :: query_location
type(location_type), intent(in)        :: loc
character(len=*), optional, intent(in) :: attr
```

Returns the value of x, y, z depending on the attribute specification. If attr is not present, returns 'x'.

`query_location`	Returns x, y, or z.
`loc`	A location type
attr	Selects 'X', 'Y', 'Z'. If not specified, 'X' is returned.

var = set_location_missing()

```
type(location_type) :: set_location_missing
```

Returns a location with all elements set to missing values defined in types module.

set_location_missing	A location with all elements set to missing values

call get_close_maxdist_init(gc,maxdist, [maxdist_list])

```
type(get_close_type), intent(inout) :: gc
real(r8), intent(in)                :: maxdist
real(r8), intent(in), optional      :: maxdist_list(:)
```

Sets the threshhold distance. maxdist is in units of radians. Anything closer than this is deemed to be close. This routine must be called first, before the other get_close routines. It allocates space so it is necessary to call get_close_obs_destroy when completely done with getting distances between locations.

If the last optional argument is not specified, maxdist applies to all locations. If the last argument is specified, it must be a list of exactly the length of the number of specific types in the obs_kind_mod.f90 file. This length can be queried with the get_num_types_of_obs() function to get count of obs types. It allows a different maximum distance to be set per base type when get_close() is called.

gc	Data for efficiently finding close locations.
maxdist	Anything closer than this number of radians is a close location.
maxdist_list	If specified, must be a list of real values. The length of the list must be exactly the same length as the number of observation types defined in the obs_def_kind.f90 file. (See get_num_types_of_obs() to get count of obs types.) The values in this list are used for the obs types as the close distance instead of the maxdist argument.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type),             intent(inout) :: gc
integer,                          intent(in)    :: num
type(location_type), dimension(:) intent(in)    :: obs
```

Initialize storage for efficient identification of locations close to a given location. Allocates storage for keeping track of which 'box' each location in the list is in. Must be called after get_close_maxdist_init, and the list of locations here must be the same as the list of locations passed into get_close_obs(). If the list changes, get_close_obs_destroy() must be called, and both the initialization routines must be called again. It allocates space so it is necessary to call get_close_obs_destroy when completely done with getting distances between locations.

gc	Structure that contains data to efficiently find locations close to a given location.
num	The number of locations in the list.
obs	The locations of each element in the list, not used in 1D implementation.

call get_close_obs(gc, base_obs_loc, base_obs_type, obs, obs_kind, num_close, close_ind, dist)

```
type(get_close_type),              intent(in)  :: gc
type(location_type),               intent(in)  :: base_obs_loc
integer,                           intent(in)  :: base_obs_type
type(location_type), dimension(:), intent(in)  :: obs
integer,             dimension(:), intent(in)  :: obs_kind
integer,                           intent(out) :: num_close
integer,             dimension(:), intent(out) :: close_ind
real(r8), optional,  dimension(:), intent(out) :: dist
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

Note that the base location is passed with the specific type associated with that location. The list of potential close locations is matched with a list of generic kinds. This is because in the current usage in the DART system the base location is always associated with an actual observation, which has both a specific type and generic kind. The list of potentially close locations is used both for other observation locations but also for state variable locations which only have a generic kind.

If called without the optional *dist* argument, all locations that are potentially close are returned, which is likely a superset of the locations that are within the threshold distance specified in the `get_close_maxdist_init()` call.

`gc`	Structure to allow efficient identification of locations close to a given location.
`base_obs_loc`	Single given location.
`base_obs_type`	Specific type of the single location.
`obs`	List of locations from which close ones are to be found.
`obs_kind`	Generic kind associated with locations in obs list.
`num_close`	Number of locations close to the given location.
`close_ind`	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call get_close_obs_destroy(gc)

```
type(get_close_type), intent(inout) :: gc
```

Releases memory associated with the `gc` derived type. Must be called whenever the list of locations changes, and then `get_close_maxdist_init` and `get_close_obs_init` must be called again with the new locations list.

`gc`	Data for efficiently finding close locations.

var = get_dist(loc1, loc2, [, type1, kind2, no_vert])

```
real(r8)                         :: get_dist
type(location_type), intent(in) :: loc1
type(location_type), intent(in) :: loc2
integer, optional,   intent(in) :: type1
integer, optional,   intent(in) :: kind2
```

Returns the distance between two locations.

The type and kind arguments are not used by the default location code, but are available to any user-supplied distance routines which want to do specialized calculations based on the types/kinds associated with each of the two locations.

`loc1`	First of two locations to compute distance between.
`loc2`	Second of two locations to compute distance between.
type1	DART specific type associated with location 1.
kind2	DART generic kind associated with location 2.
`var`	distance between loc1 and loc2.

var = vert_is_undef(loc)

```
logical                          :: vert_is_undef
type(location_type), intent(in) :: loc
```

Always returns .false.

`vert_is_undef`	Always returns .false.
`loc`	A location type

var = vert_is_surface(loc)

```
logical                          :: vert_is_surface
type(location_type), intent(in) :: loc
```

Always returns .false.

`vert_is_surface`	Always returns .false.
`loc`	A location type

var = vert_is_pressure(loc)

```
logical                          :: vert_is_pressure
type(location_type), intent(in) :: loc
```

Always returns .false.

vert_is_pressure	Always returns .false.
loc	A location type

var = vert_is_scale_height(loc)

```
logical                          :: vert_is_scale_height
type(location_type), intent(in) :: loc
```

Always returns .false.

vert_is_scale_height	Always returns .false.
loc	A location type

var = vert_is_level(loc)

```
logical                          :: vert_is_level
type(location_type), intent(in) :: loc
```

Always returns .false.

vert_is_level	Always returns .false.
loc	A location type

var = vert_is_height(loc)

```
logical                          :: vert_is_height
type(location_type), intent(in) :: loc
```

Always returns .false.

vert_is_height	Always returns .false.
loc	A location type

var = has_vertical_localization()

```
logical :: has_vertical_localization
```

Always returns .false.

This routine should perhaps be renamed to something like 'using_vertical_for_distance' or something similar. The current use for it is in the localization code inside filter, but that doesn't make this a representative function name. And at least in current usage, returning the opposite setting of the namelist item makes the code read more direct (fewer double negatives).

loc1 == loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types have identical values, else false.

loc1 /= loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types do NOT have identical values, else false.

```
integer, parameter :: LocationDims = 3
```

This is a **constant**. Contains the number of real values in a location type. Useful for output routines that must deal transparently with many different location modules.

```
character(len=129), parameter :: LocationName = "loc3Dcartesian"
```

This is a **constant**. A parameter to identify this location module in output metadata.

```
character(len=129), parameter :: LocationLName =

       "threed cartesian locations: x, y, z"
```

This is a **constant**. A parameter set to "threed cartesian locations: x, y, z" used to identify this location module in output long name metadata.

6.169.5 Files

filename	purpose
input.nml	to read the location_mod namelist

6.169.6 References

1. none

6.169.7 Private components

N/A

6.170 MODULE location_mod (threed_sphere)

6.170.1 Overview

The DART framework needs to be able to compute distances between locations, to pass location information to and from the model interface code (`model_mod.f90`), and to be able to read and write location information to files. DART isolates all this location information into separate modules so that the main algorithms can operate with the same code independent of whether the model uses latitude/longitude/height, 1D unit sphere coordinates, cylindrical coordinates, etc. DART provides about half a dozen possible coordinate systems, and others can be added. The most common one for geophysical models is this one: threed_sphere.

This location module provides a representation of a physical location on a 3-D spherical shell, using latitude and longitude plus a vertical component with choices of vertical coordinate type such as pressure or height in meters. A type that abstracts the location is provided along with operators to set, get, read, write, and compute great circle distances between locations. This is a member of a class of similar location modules that provide the same abstraction for different represenations of physical space.

6.170.2 Usage

The location routines are general purpose code that can be used for a variety of utilities. The following discussion is specifically restricted to how the location namelist settings affect the execution of the `filter` assimilation program.

Issues related to changing the results of an assimilation based on the location module settings:

- Whether and how to treat the vertical separation when computing distances between two locations.
- Whether to use different distances in the vertical for different observation types.

Issues related to changing the results of an assimilation based on code in the model-specific `model_mod.f90` module:

- Whether the model-specific code needs to convert vertical coordinates.
- Whether the model-specific code alters the distances in some other way.

Issues related to the speed/efficiency of an assimilation based on the location module settings:

- Whether to use a faster but less precise distance computation.
- Whether to change the number of internal bins used to more quickly find nearby locations.

Vertical issues

The localization distance during an assimilation – the maximum separation between an observation and a state vector item potentially affected by the assimilation – is set in the &assim_tools_nml namelist (the `cutoff` item).

Setting `horiz_dist_only = .TRUE.` in the namelist means the great circle distances will be computed using only the latitude and longitudes of the two locations, ignoring the vertical components of the locations. The cutoff is specified in radians to be independent of the radius of the sphere. For the Earth the radius is nominally 6,371 Km. To compute the horizontal only localization distance, multiply 6,371 Km by the cutoff to get the distance in Km. The cutoff is by definition 1/2 the distance to where the increments go to 0, so multiply that result by 2 to get the maximum distance at which an observation can alter the state.

Setting `horiz_dist_only = .FALSE.` in the namelist means the code will compute a 3D distance, including the vertical separation. In this case, the `vert_normalization_xxx` namelist values will be used to convert from pressure, height, model level, or scale heights into radians so the distances are computed in a consistent unit system. In practice, multiply the cutoff by the normalization factor (and then again by 2) to get the maximum vertical separation in each of the given units.

When including vertical separation the potential area of influence of an assimilated observation is an ellipsoid with the observation at the center. The horizontal radius is defined by the cutoff and the vertical radius is defined by the normalization factors.

See examples below for specific examples that highlight some vertical localization issues.

Different vertical factors per observation type

Generally a single cutoff value and a single set of normalization factors are sufficient for most assimilations. The localization distances define the maximum range of impact of an observation, but there still must be a positive or negative correlation between the state ensemble and the forward operator/expected obs ensemble for the values to change.

However, the &assim_tools_nml namelist includes the option to set a different cutoff on a per-observation-type basis. There are corresponding items in the location module namelist to similiarly control the vertical distance contribution on a per-observation, per-vertical-type basis.

Model-dependent vertical conversion issues

If the model supports either a different vertical coordinate than the vertical coordinate of the observations, or if the user wants to localize in a different vertical coordinate than the observations or state vector items, the model-specific `model_mod.f90` code will have to provide a conversion between different vertical coordinates. This cannot be done by the location module since most vertical conversions require additional model-specific information such as temperature, moisture, depth, surface elevation, model levels, etc.

Once the locations have the same vertical units the location module code can compute the distances between them. It is an error to ask for the distance between two locations with different vertical coordinates unless you have set the namelist to horizontal distances only.

There is a vertical type of VERTISUNDEF (Vertical is Undefined). This is used to describe observations where there is no specific single vertical location, for example the position of a hurricane or a column integrated quantity. In this case the location code computes only horizontal distances between any pair of observations in which one or both have an undefined vertical location.

Model-dependent distance adjustments

The calls to routines that collect the distances between locations for the assimilation code pass through the model-specific `model_mod.f90` code. This allows the code to alter the actual distances to either increase or decrease the effect of an observation on the state or on other observations.

For example, if the top of a model is externally constrained then modifications by the assimilation code may lead to bad results. The model-specific code can compute the actual distances between two locations and then increase it artificially as you reach the top of the model, so observations have smaller and smaller impacts. For ocean models, the distances to points on land can be set to a very large value and those points will be unaffected by the assimilation.

Approximate distances

For regional models this should usually be `.FALSE.` in the namelist.

For global models this is usually set to `.TRUE.` which allows the code to run slightly faster by precomputing tables of sines, cosines, and arc cosines used in the distance computations. Values are linearly interpolated between entries in the table which leads to minor roundoff errors. These are negligible in a global model but may be significant in models that over a small region of the globe.

Internal bin counts

The default settings for `nlon` and `nlat` are usually sufficient. However if this is a high resolution model with a large state vector the assimilation may run faster by doubling these values or multiplying them by 4. (The `nlon` item must be odd; compute the value and subtract 1.) These values set the number of internal bins used inside the code to pre-sort locations and make it faster to retrieve all locations close to another location. A larger bin count uses more memory but shortens the linear part of the location search.

Examples and questions involving vertical issues

Example of specifying a cutoff based on a distance in kilometers

The Earth radius is nominally 6,371 Km. If you want the maximum horizontal distance that an observation can possibly influence something in the model state to be X km, then set the cutoff to be (X / 6,371) / 2. Remember the actual impact will depend on a combination of this distance and the regression coefficient computed from the distribution of forward operator values and the ensemble of values in the model state.

Cutoff and half-widths

Q: Why is the cutoff specified as half the distance to where the impact goes to 0, and why is it called 'cutoff'?

A: Because the original paper by Gaspari & Cohn used that definition in this paper which our localization function is based on.

Gaspari, G. and Cohn, S. E. (1999), Construction of correlation functions in two and three dimensions. Q.J.R. Meteorol. Soc., 125: 723-757. doi:10.1002/qj.49712555417

Computing vertical normalization values

Because distances are computed in radians, the vertical distances have to be translated to radians. To get a maximum vertical separation of X meters (if localizing in height), specify the vert_normalization_height of X / cutoff. If localizing in pressure, specify vert_normalization_pressure as X pascals / cutoff, etc.

Single vertical coordinate type

Vertical distances can only be computed between two locations that have the same vertical type. In practice this means if vertical localization is enabled all observations which have a vertical location need to be converted to a single vertical coordinate type, which matches the desired localization unit. The model state must also be able to be converted to the same vertical coordinate type.

For example, if some observations come with a vertical coordinate type of pressure and some with height, and you want to localize in height, the pressure coordinates need to be converted to an equivalant height. This usually requires information only available to the model interface code in the model_mod.f90 file, so a convert_vertical_obs() routine is called to do the conversion.

The locations of the model state are returned by the get_state_meta_data() routine in the model_mod.f90 file. If the vertical coordinate used in the state is not the same as the desired vertical localization type, they must also be converted using a convert_vertical_state() routine.

6.170.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand `&` and terminate with a slash `/` . Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&location_nml
    horiz_dist_only                = .true.
    vert_normalization_pressure    = 100000.0
    vert_normalization_height      = 10000.0
    vert_normalization_level       = 20.0
    vert_normalization_scale_height = 5.0
    approximate_distance           = .false.
    nlon                           = 71
    nlat                           = 36
    output_box_info                = .false.
    print_box_level                = 0
    special_vert_normalization_obs_types     = 'null'
    special_vert_normalization_pressures     = -888888.0
    special_vert_normalization_heights       = -888888.0
    special_vert_normalization_levels        = -888888.0
    special_vert_normalization_scale_heights = -888888.0
  /
```

Items in this namelist either control the way in which distances are computed and/or influence the code performance.

Item	Type	Description
horiz_dist_only	logical	If .TRUE. compute great-circle distance using the horizontal distance component only. If .FALSE. compute distances by including the vertical and horizontal separation. All distances are computed in radians; the corresponding vertical normalization factors are used to compute the vertical distance. The vertical coordinate system must be the same for both locations in order to compute a distance. However, if either location is VERTISUNDEF, or both are VERTISSURFACE, only a horizontal distance is computed. For any other combination of vertical coordinate systems this routine will fail because it cannot convert between vertical coordinate systems without model-specific information. The model_mod interface code may supply a get_close_obs() routine to intercept and convert the vertical coordinates before calling this get_close_obs() routine.
vert_normalization_pressure	real(r8)	The number of pascals equivalent to a horizontal distance of one radian.
vert_normalization_height	real(r8)	The number of meters equivalent to a horizontal distance of one radian.
vert_normalization_level	real(r8)	The number of model levels equivalent to a horizontal distance of one radian.
vert_normalization_scale_ht	real(r8)	The number of scale heights equivalent to a horizontal distance of one radian.
approximate_distance	logical	If true, uses a table lookup for fast approximate computation of distances on sphere. Distance computation can be a first order cost for some spherical problems so this can increase speed significantly at a loss of some precision. WARNING: This should be set to .FALSE. if you need to compute small distances accurately or you have a regional model.
nlon	integer	Used internally by the search code to speed the search for nearby locations. Number of boxes (bins) created in the longitude direction. Must be an odd number. (See discussion above for more information about this item.)
nlat	integer	Used internally by the search code to speed the search for nearby locations. Number of boxes (bins) created in the latitude direction. (See discussion above for more information about this item.)
output_box_info	logical	If true, print details about the distribution of locations across the array of boxes. `print_box_level` controls how much detail is printed.
print_box_level	integer	If `output_box_info = .true.`, `print_box_level` controls how much detail is printed. 0 = no detail. 1,2,3 are progressively more and more detail.
special_vert_normalization_obs_types	character(len=32), dimension(500)	If specified, must be a string array of observation specific types (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_TEMPERATURE, etc). For each type listed here, a vertical normalization value must be given which overrides the default vertical normalization values. Even if only one is going to be used, all 4 normalization values must be specified for each special type.
special_vert_normalization_pressures	real(r8), dimension(500)	The number of pascals equivalent to a horizontal distance of one radian, one value for each special observation type listed in the ' special_vert_normalization_obs_types' list.
special_vert_normalization_heights	real(r8), dimension(500)	The number of geopotential meters equivalent to a horizontal distance of one radian, one value for each special observation type listed in the ' special_vert_normalization_obs_types' list.
special_vert_normalization_scale_ht	real(r8), dimension(500)	The number of scale heights equivalent to a horizontal distance of one radian, one value for each special observation type listed in the ' special_vert_normalization_obs_types' list.
special_vert_normalization_levels	real(r8), dimension(500)	The number of model levels equivalent to a horizontal distance of one radian, one value for each special observation type listed in the ' special_vert_normalization_obs_types' list.

6.170.4 Discussion

Location-independent code

All types of location modules define the same module name `location_mod`. Therefore, the DART framework and any user code should include a Fortran 90 `use` statement of `location_mod`. The selection of which location module will be compiled into the program is controlled by which source file name is specified in the `path_names_xxx` file, which is used by the `mkmf_xxx` scripts.

All types of location modules define the same Fortran 90 derived type `location_type`. Programs that need to pass location information to subroutines but do not need to interpret the contents can declare, receive, and pass this derived type around in their code independent of which location module is specified at compile time. Model and location-independent utilities should be written in this way. However, as soon as the contents of the location type needs to be accessed by user code then it becomes dependent on the exact type of location module that it is compiled with.

Usage of distance routines

Regardless of the fact that the distance subroutine names include the string 'obs', there is nothing specific to observations in these routines. They work to compute distances between any set of locations. The most frequent use of these routines in the filter code is to compute the distance between a single observation and items in the state vector, and also between a single observation and other nearby observations. However, any source for locations is supported.

In simpler location modules (like the `oned` version) there is no need for anything other than a brute force search between the base location and all available state vector locations. However in the case of large geophysical models which typically use the `threed_sphere` locations code, the brute-force search time is prohibitive. The location code pre-processes all locations into a set of *bins* and then only needs to search the lists of locations in nearby bins when looking for locations that are within a specified distance.

The expected calling sequence of the `get_close` routines is as follows:

```
call get_close_init()
...
call get_close_obs()           ! called many, many times
...
call get_close_destroy()
```

`get_close_init()` initializes the data structures, `get_close_obs()` is called multiple times to find all locations within a given radius of some reference location, and to optionally compute the exact separation distance from the reference location. `get_close_destroy()` deallocates the space. See the documentation below for the specific details for each routine.

All 3 of these routines must be present in every location module but in most other versions all but `get_close_obs()` are stubs. In this `threed_sphere` version of the locations module all are fully implemented.

Interaction with model_mod.f90 code

The filter and other DART programs could call the `get_close` routines directly, but typically do not. They declare them (in a `use` statement) to be in the `model_mod` module, and all model interface modules are required to supply them. However in many cases the model_mod only needs to contain another `use` statement declaring them to come from the `location_mod` module. Thus they 'pass through' the model_mod but the user does not need to provide a subroutine or any code for them.

However, if the model interface code wants to intercept and alter the default behavior of the get_close routines, it is able to. Typically the model_mod still calls the location_mod routines and then adjusts the results before passing them back to the calling code. To do that, the model_mod must be able to call the routines in the location_mod which have

the same names as the subroutines it is providing. To allow the compiler to distinguish which routine is to be called where, we use the Fortran 90 feature which allows a module routine to be renamed in the use statement. For example, a common case is for the model_mod to want to supply additions to the get_close_obs() routine only. At the top of the model_mod code it would declare:

```
use location_mod, only :: get_close_init, get_close_destroy, &
                          location_get_close_obs => get_close_obs
```

That makes calls to the maxdist_init, init, and destroy routines simply pass through to the code in the location_mod, but the model_mod must supply a get_close_obs() subroutine. When it wants to call the code in the location_mod it calls `location_get_close_obs()`.

One use pattern is for the model_mod to call the location get_close_obs() routine without the `dist` argument. This returns a list of any potentially close locations without computing the exact distance from the base location. At this point the list of locations is a copy and the model_mod routine is free to alter the list in any way it chooses: it can change the locations to make certain types of locations appear closer or further away from the base location; it can convert the vertical coordinates into a common coordinate type so that calls to the `get_dist()` routine can do full 3d distance computations and not just 2d (the vertical coordinates must match between the base location and the locations in the list in order to compute a 3d distance). Then typically the model_mod code loops over the list calling the `get_dist()` routine to get the actual distances to be returned to the calling code. To localize in the vertical in a particular unit type, this is the place where the conversion to that vertical unit should be done.

Horizontal distance only

If *horiz_distance_only* is .true. in the namelist then the vertical coordinate is ignored and only the great-circle distance between the two locations is computed, as if they were both on the surface of the sphere.

If *horiz_distance_only* is .false. in the namelist then the appropriate normalization constant determines the relative impact of vertical and horizontal separation. Since only a single localization distance is specified, and the vertical scales might have very different distance characteristics, the vert_normalization_xxx values can be used to scale the vertical appropriately to control the desired influence of observations in the vertical.

Precomputation for run-time search efficiency

For search efficiency all locations are pre-binned. The surface of the sphere is divided up into *nlon* by *nlat* boxes and the index numbers of all items (both state vector entries and observations) are stored in the appropriate box. To locate all points close to a given location, only the locations listed in the boxes within the search radius must be checked. This speeds up the computations, for example, when localization controls which state vector items are impacted by any given observation. The search radius is the localization distance and only those state vector items in boxes closer than the radius to the observation location are processed.

The default values have given good performance on many of our existing model runs, but for tuning purposes the box counts have been added to the namelist to allow adjustment. By default the code prints some summary information about how full the average box is, how many are empty, and how many items were in the box with the largest count. The namelist value *output_box_info* can be set to .true. to get even more information about the box statistics. The best performance will be obtained somewhere between two extremes; the worst extreme is all the points are located in just a few boxes. This degenerates into a (slow) linear search through the index list. The other extreme is a large number of empty or sparsely filled boxes. The overhead of creating, managing, and searching a long list of boxes will impact performance. The best performance lies somewhere in the middle, where each box contains a reasonable number of values, more or less evenly distributed across boxes. The absolute numbers for best performance will certainly vary from case to case.

For latitude, the *nlat* boxes are distributed evenly across the actual extents of the data. (Locations are in radians, so the maximum limits are the poles at $-\pi/2$ and $+\pi/2$. For longitude, the code automatically determines if the data is spread around more than half the sphere, and if so, the boxes are distributed evenly across the entire sphere (longitude range

0 to 2π). If the data spans less than half the sphere in longitude, the actual extent of the data is determined (including correctly handling the cyclic boundary at 0) and the boxes are distributed only within the data extent. This simplifies the actual distance calculations since the distance from the minimum longitude box to the maximum latitude box cannot be shorter going the other way around the sphere. In practice, for a global model the boxes are evenly distributed across the entire surface of the sphere. For local or regional models, the boxes are distributed only across the the extent of the local grid.

For efficiency in the case where the boxes span less than half the globe, the 3D location module needs to be able to determine the greatest longitude difference between a base point at latitude ϕ_s and all points that are separated from that point by a central angle of θ. We might also want to know the latitude, ϕ_f, at which the largest separation occurs. Note also that an intermediate form below allows the computation of the maximum longitude difference at a particular latitude.

The central angle between a point at latitude ϕ_s and a second point at latitude ϕ_f that are separated in longitude by $\Delta\lambda$ is:

$$\theta = cos^{-1}(sin\phi_s sin\phi_f + cos\phi_s cos\phi_f cos\Delta\lambda)$$

Taking the cos of both sides gives:

$$cos\theta = (sin\phi_s sin\phi_f + cos\phi_s cos\phi_f cos\Delta\lambda)$$

Solving for $cos\Delta\lambda$ gives:

$$cos\Delta\lambda = \frac{a - bsin\phi_f}{ccos\phi_f}$$

$$cos\Delta\lambda = \frac{a}{csec\phi_f} - \frac{b}{ctan\phi_f}$$

where $a = cos\theta$, $b = sin\phi_s$, and $c = cos\phi_s$. We want to maximize $\Delta\lambda$ which implies minimizing $cos\Delta\lambda$ subject to constraints.

Taking the derivative with respect to ϕ_f gives:

$$\frac{dcos\Delta\lambda}{d\phi_f} = \frac{a}{csec\phi_f tan\phi_f} - \frac{b}{csec^2\phi_f} = 0$$

Factoring out $sec\phi_f$ which can never be 0 and using the definitions of sec and tan gives:

$$\frac{asin\phi_f}{ccos\phi_f} - \frac{b}{ccos\phi_f} = 0$$

Solving in the constrained range from 0 to $\pi/2$ gives:

$$sin\phi_f = \frac{b}{a} = \frac{sin\phi_s}{cos\theta}$$

So knowing base point (ϕ_s, λ_s), latitude ϕ_f, and distance θ we can use the great circle equation to find the longitude difference at the greatest separation point:

$$\Delta\lambda = cos^{-1}\left(\frac{a - bsin\phi_f}{ccos\phi_f}\right)$$

Note that if the angle between the base point and a pole is less than or equal to the central angle, all longitude differences will occur as the pole is approached.

6.170.5 Other modules used

```
types_mod
utilities_mod
random_seq_mod
obs_kind_mod
ensemble_manager_mod
```

6.170.6 Public interfaces

use location_mod, only :	location_type
	get_close_type
	get_location
	set_location
	write_location
	read_location
	interactive_location
	set_location_missing
	query_location
	get_close_init
	get_close_obs
	get_close_destroy
	get_dist
	get_maxdist
	LocationDims
	LocationName
	LocationLName
	horiz_dist_only
	vert_is_undef
	vert_is_surface
	vert_is_pressure
	vert_is_scale_height
	vert_is_level
	vert_is_height
	VERTISUNDEF
	VERTISSURFACE
	VERTISLEVEL
	VERTISPRESSURE
	VERTISHEIGHT
	VERTISSCALEHEIGHT
	operator(==)
	operator(/=)

Namelist interface `&location_nml` must be read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

type location_type

```
   private
   real(r8) :: lon, lat, vloc
   integer  :: which_vert
end type location_type
```

Provides an abstract representation of physical location on a three-d spherical shell.

Component	Description
lon	longitude in radians
lat	latitude in radians
vloc	vertical location, units as selected by which_vert
which_vert	type of vertical location: -2=no specific vert location; -1=surface; 1=level; 2=pressure; 3=height, 4=scale height

The vertical types have parameters defined for them so they can be referenced by name instead of number.

type get_close_type

```
   private
   integer  :: num
   real(r8) :: maxdist
   integer, pointer :: lon_offset(:, :)
   integer, pointer :: obs_box(:)
   integer, pointer :: count(:, :)
   integer, pointer :: start(:, :)
end type get_close_type
```

Provides a structure for doing efficient computation of close locations.

Component	Description
num	Number of locations in list
maxdist	Threshhold distance. Anything closer is close.
lon_offset	Dimensioned nlon by nlat. For a given offset in longitude boxes and difference in latitudes, gives max distance from base box to a point in offset box.
obs_box	Dimensioned num. Gives index of what box each location is in.
count	Dimensioned nlon by nlat. Number of obs in each box.
start	Dimensioned nlon by nlat. Index in straight storage list where obs in each box start.

var = get_location(loc)

```
real(r8), dimension(3)          :: get_location
type(location_type), intent(in) :: loc
```

Extracts the longitude and latitude (converted to degrees) and the vertical location from a location type and returns in a 3 element real array.

get_location	The longitude and latitude (in degrees) and vertical location
loc	A location type

var = set_location(lon, lat, vert_loc, which_vert)

```
type(location_type) :: set_location
real(r8), intent(in)    :: lon
real(r8), intent(in)    :: lat
real(r8), intent(in)    :: vert_loc
integer,  intent(in)    :: which_vert
```

Returns a location type with the input longitude and latitude (input in degrees) and the vertical location of type specified by which_vert.

set_location	A location type
lon	Longitude in degrees
lat	Latitude in degrees
vert_loc	Vertical location consistent with which_vert
which_vert	The vertical location type

call write_location(locfile, loc [, fform, charstring])

```
integer,               intent(in)       ::  locfile
type(location_type),   intent(in)       ::  loc
character(len=*), optional, intent(in)  ::  fform
character(len=*), optional, intent(out) ::  charstring
```

Given an integer IO channel of an open file and a location, writes the location to this file. The *fform* argument controls whether write is "FORMATTED" or "UNFORMATTED" with default being formatted. If the final *charstring* argument is specified, the formatted location information is written to the character string only, and the `locfile` argument is ignored.

locfile	the unit number of an open file.
loc	location type to be written.
fform	Format specifier ("FORMATTED" or "UNFORMATTED"). Default is "FORMATTED" if not specified.
charstring	Character buffer where formatted location string is written if present, and no output is written to the file unit.

var = read_location(locfile [, fform])

```
type(location_type)                    :: read_location
integer, intent(in)                    :: locfile
character(len=*), optional, intent(in) :: fform
```

Reads a location_type from a file open on channel locfile using format *fform* (default is formatted).

`read_location`	Returned location type read from file
`locfile`	Integer channel opened to a file to be read
fform	Optional format specifier ("FORMATTED" or "UNFORMATTED"). Default "FORMATTED".

call interactive_location(location [, set_to_default])

```
type(location_type), intent(out) :: location
logical, optional, intent(in)    :: set_to_default
```

Use standard input to define a location type. With set_to_default true get one with all elements set to 0.

`location`	Location created from standard input
set_to_default	If true, sets all elements of location type to 0

var = query_location(loc [, attr])

```
real(r8)                               :: query_location
type(location_type), intent(in)        :: loc
character(len=*), optional, intent(in) :: attr
```

Returns the value of which_vert, latitude, longitude, or vertical location from a location type as selected by the string argument attr. If attr is not present or if it is 'WHICH_VERT', the value of which_vert is converted to real and returned. Otherwise, attr='LON' returns longitude, attr='LAT' returns latitude and attr='VLOC' returns the vertical location.

`query_location`	Returns longitude, latitude, vertical location, or which_vert (converted to real)
`loc`	A location type
attr	Selects 'WHICH_VERT', 'LON', 'LAT' or 'VLOC'

var = set_location_missing()

```
type(location_type) :: set_location_missing
```

Returns a location with all elements set to missing values defined in types module.

set_location_missing	A location with all elements set to missing values

call get_close_init(gc, num, maxdist, locs [,maxdist_list])

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
real(r8),             intent(in)    :: maxdist
type(location_type),  intent(in)    :: locs(:)
real(r8), optional,   intent(in)    :: maxdist_list(:)
```

Initializes the get_close accelerator. `maxdist` is in units of radians. Anything closer than this is deemed to be close. This routine must be called first, before the other `get_close` routines. It allocates space so it is necessary to call `get_close_destroy` when completely done with getting distances between locations.

If the last optional argument is not specified, `maxdist` applies to all locations. If the last argument is specified, it must be a list of exactly the length of the number of specific types in the `obs_kind_mod.f90` file. This length can be queried with the get_num_types_of_obs() function to get count of obs types. It allows a different maximum distance to be set per base type when `get_close()` is called.

gc	Data for efficiently finding close locations.
num	The number of locations, i.e. the length of the `locs` array.
maxdist	Anything closer than this number of radians is a close location.
locs	The list of locations in question.
maxdist	If specified, must be a list of real values. The length of the list must be exactly the same length as the number of observation types defined in the obs_def_kind.f90 file. (See get_num_types_of_obs() to get count of obs types.) The values in this list are used for the obs types as the close distance instead of the maxdist argument.

call get_close_obs(gc, base_obs_loc, base_obs_type, obs, obs_kind, num_close, close_ind [, dist, ens_handle])

```
type(get_close_type),              intent(in)  :: gc
type(location_type),               intent(in)  :: base_obs_loc
integer,                           intent(in)  :: base_obs_type
type(location_type), dimension(:), intent(in)  :: obs
integer,             dimension(:), intent(in)  :: obs_kind
integer,                           intent(out) :: num_close
integer,             dimension(:), intent(out) :: close_ind
real(r8), optional,  dimension(:), intent(out) :: dist
type(ensemble_type), optional,     intent(in)  :: ens_handle
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must

be identical to the list of `obs` passed into the most recent call to `get_close_init()`. If the list of locations of interest changes `get_close_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

Note that the base location is passed with the specific type associated with that location. The list of potential close locations is matched with a list of generic kinds. This is because in the current usage in the DART system the base location is always associated with an actual observation, which has both a specific type and generic kind. The list of potentially close locations is used both for other observation locations but also for state variable locations which only have a generic kind.

If called without the optional *dist* argument, all locations that are potentially close are returned, which is likely a superset of the locations that are within the threshold distance specified in the `get_close_init()` call. This can be useful to collect a list of potential locations, and then to convert all the vertical coordinates into one consistent unit (pressure, height in meters, etc), and then the list can be looped over, calling get_dist() directly to get the exact distance, either including vertical or not depending on the setting of `horiz_dist_only`.

`gc`	Structure to allow efficient identification of locations close to a given location.
`base_obs_loc`	Single given location.
`base_obs_type`	Specific type of the single location.
`obs`	List of locations from which close ones are to be found.
`obs_kind`	Generic kind associated with locations in obs list.
`num_close`	Number of locations close to the given location.
`close_ind`	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.
ens_handle	Handle to an ensemble of interest.

call get_close_destroy(gc)

```
type(get_close_type), intent(inout) :: gc
```

Releases memory associated with the `gc` derived type. Must be called whenever the list of locations changes, and then `get_close_init` must be called again with the new locations list.

`gc`	Data for efficiently finding close locations.

var = get_dist(loc1, loc2, [, type1, kind2, no_vert])

```
real(r8)                        :: get_dist
type(location_type), intent(in) :: loc1
type(location_type), intent(in) :: loc2
integer, optional,   intent(in) :: type1
integer, optional,   intent(in) :: kind2
logical, optional,   intent(in) :: no_vert
```

Returns the distance between two locations in radians. If `horiz_dist_only` is set to .TRUE. in the locations namelist, it computes great circle distance on sphere. If `horiz_dist_only` is false, then it computes an ellipsoidal distance

with the horizontal component as above and the vertical distance determined by the types of the locations and the normalization constants set by the namelist for the different vertical coordinate types. The vertical normalization gives the vertical distance that is equally weighted as a horizontal distance of 1 radian. If *no_vert* is present, it overrides the value in the namelist and controls whether vertical distance is included or not.

The type and kind arguments are not used by the default location code, but are available to any user-supplied distance routines which want to do specialized calculations based on the types/kinds associated with each of the two locations.

loc1	First of two locations to compute distance between.
loc2	Second of two locations to compute distance between.
type1	DART specific type associated with location 1.
kind2	DART generic kind associated with location 2.
no_vert	If true, no vertical component to distance. If false, vertical component is included.
var	distance between loc1 and loc2.

var = get_maxdist(gc [, obs_type])

```
real(r8)                              :: var
type(get_close_type), intent(inout) :: gc
integer, optional,    intent(in)    :: obs_type
```

Since it is possible to have different cutoffs for different observation types, an optional argument *obs_type* may be used to specify which maximum distance is of interest. The cutoff is specified as the half-width of the tapering function, get_maxdist returns the full width of the tapering function.

gc	Data for efficiently finding close locations.
obs_type	The integer code specifying the type of observation.
var	The distance at which the tapering function is zero. Put another way, anything closer than this number of radians is a close location.

var = vert_is_undef(loc)

```
logical                           :: vert_is_undef
type(location_type), intent(in) :: loc
```

Returns true if which_vert is set to undefined, else false. The meaning of 'undefined' is specific; it means there is no particular vertical location associated with this type of measurement; for example a column-integrated value.

vert_is_undef	Returns true if vertical coordinate is set to undefined.
loc	A location type

var = vert_is_surface(loc)

```
logical                          :: vert_is_surface
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for surface, else false.

vert_is_surface	Returns true if vertical coordinate type is surface
loc	A location type

var = vert_is_pressure(loc)

```
logical                          :: vert_is_pressure
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for pressure, else false.

vert_is_pressure	Returns true if vertical coordinate type is pressure
loc	A location type

var = vert_is_scale_height(loc)

```
logical                          :: vert_is_scale_height
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for scale_height, else false.

vert_is_scale_height	Returns true if vertical coordinate type is scale_height
loc	A location type

var = vert_is_level(loc)

```
logical                          :: vert_is_level
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for level, else false.

vert_is_level	Returns true if vertical coordinate type is level
loc	A location type

var = vert_is_height(loc)

```
logical                          :: vert_is_height
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for height, else false.

vert_is_height	Returns true if vertical coordinate type is height
loc	A location type

var = has_vertical_localization()

```
logical :: has_vertical_localization
```

Returns .TRUE. if the namelist variable horiz_dist_only is .FALSE. meaning that vertical separation between locations is going to be computed by get_dist() and by get_close_obs().

This routine should perhaps be renamed to something like 'using_vertical_for_distance' or something similar. The current use for it is in the localization code inside filter, but that doesn't make this a representative function name. And at least in current usage, returning the opposite setting of the namelist item makes the code read more direct (fewer double negatives).

loc1 == loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types have identical values, else false.

loc1 /= loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types do NOT have identical values, else false.

```
integer, parameter :: VERTISUNDEF       = -2
integer, parameter :: VERTISSURFACE     = -1
integer, parameter :: VERTISLEVEL       =  1
integer, parameter :: VERTISPRESSURE    =  2
integer, parameter :: VERTISHEIGHT      =  3
integer, parameter :: VERTISSCALEHEIGHT =  4
```

Constant parameters used to differentiate vertical types.

```
integer, parameter :: LocationDims = 3
```

This is a **constant**. Contains the number of real values in a location type. Useful for output routines that must deal transparently with many different location modules.

```
character(len=129), parameter :: LocationName = "loc3Dsphere"
```

This is a **constant**. A parameter to identify this location module in output metadata.

```
character(len=129), parameter :: LocationLName =

       "threed sphere locations: lon, lat, vertical"
```

This is a **constant**. A parameter set to "threed sphere locations: lon, lat, vertical" used to identify this location module in output long name metadata.

6.170.7 Files

filename	purpose
input.nml	to read the location_mod namelist

6.170.8 References

1. none

6.170.9 Error codes and conditions

Routine	Message	Comment
initialize_module	nlon must be odd	Tuning parameter for number of longitude boxes must be odd for algorithm to function.
get_dist	Dont know how to compute vertical distance for unlike vertical coordinates	Need same which_vert for distances.
set_location	longitude (#) is not within range [0,360]	Is it really a longitude?
set_location	latitude (#) is not within range [-90,90]	Is it really a latitude?
set_location	which_vert (#) must be one of -2, -1, 1, 2, 3, or 4	Vertical coordinate type restricted to: -2 = no specific vertical location -1 = surface value 1 = (model) level 2 = pressure 3 = height 4 = scale height
read_location	Expected location header "loc3d" in input file, got ___	Probable mixing of other location modules in observation sequence processing.
nc_write_location	Various NetCDF-f90 interface error messages	From one of the NetCDF calls in nc_write_location

6.170.10 Future plans

Need to provide more efficient algorithms for getting close locations and document the nlon and nlat choices and their impact on cost.

The collection of 'val = vert_is_xxx()' routines should probably be replaced by a single call 'val = vert_is(loc, VERTISxxx)'.

See the note in the 'has_vertical_localization()' about a better name for this routine.

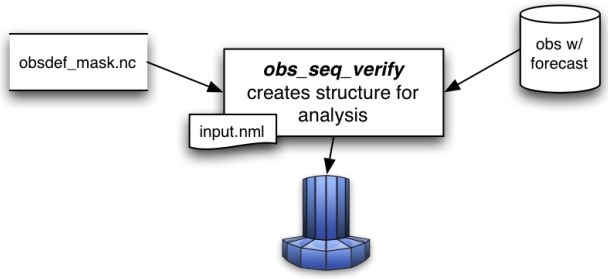
The use of 'obs' in all these routine names should probably be changed to 'loc' since there is no particular dependence that they be observations. They may need to have an associated DART kind, but these routines are used for DART state vector entries so it's misleading to always call them 'obs'.

6.170.11 Private components

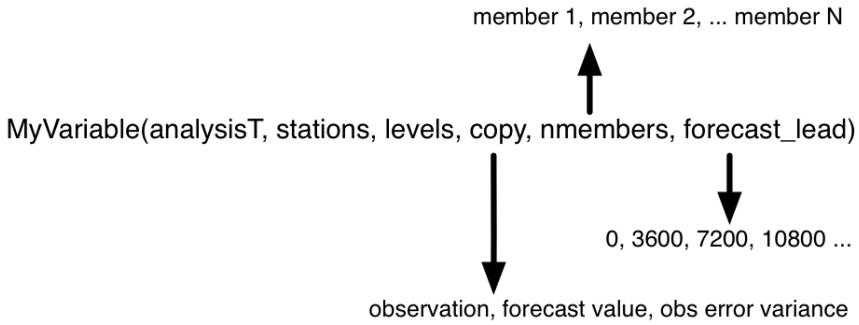
N/A

6.171 program obs_seq_verify

6.171.1 Overview



`obs_seq_verify` reorders the observations from a forecast run of DART into a structure that is amenable for the evaluation of the forecast. The big picture is that the verification locations and times identified in the `obsdef_mask.nc` and the observations from the forecast run (whose files **must** have an extension as in the following: `obs_seq.forecast.YYYYMMDDHH`) are put into a netCDF variable that looks like this:



`obs_seq_verify` can read in a series of observation sequence files - each of the files **must** contain the **entire forecast from a single analysis time**. The extension of each filename is **required** to reflect the analysis time. Use *program obs_sequence_tool* to concatenate multiple files into a single observation sequence file if necessary. *Only the individual ensemble members forecast values are used - the ensemble mean and spread (as individual copies) are completely ignored.* The individual "*prior ensemble member NNNN*" copies are used. As a special case, the "*prior ensemble mean*" copy is used *if and only if* there are no individual ensemble members present (i.e. `input.nml &filter_nml:num_output_obs_members` == *0*).

Dimension	Explanation
analysisT	This is the netCDF UNLIMITED dimension, so it is easy to 'grow' this dimension. This corresponds to the number of forecasts one would like to compare.
stations	The unique horizontal locations in the verification network.
levels	The vertical level at each location. Observations with a pressure vertical coordinate are selected based on their proximity to the mandatory levels as defined in *program obs_seq_coverage*. Surface observations or observations with undefined vertical coordinates are simply put into level 1.
copy	This dimension designates the quantity of interest; the observation, the forecast value, or the observation error variance. These quantities are the ones required to calculate the evaluation statistics.
nmembers	Each ensemble member contributes a forecast value.
forecast_lead	This dimension relates to the amount of time between the start of the forecast and the verification.

The USAGE section has more on the actual use of `obs_seq_verify`.

6.171.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_seq_verify_nml
   obs_sequences     = ''
   obs_sequence_list = ''
   station_template  = 'obsdef_mask.nc'
   netcdf_out        = 'forecast.nc'
   obtype_string     = 'RADIOSONDE_TEMPERATURE'
   print_every       = 10000
   verbose           = .true.
   debug             = .false.
   /
```

You can specify **either** `obs_sequences` **or** `obs_sequence_list` – not both. One of them has to be an empty string … i.e. ' '.

Item	Type	Description
obs_sequences	character(len=256), dimension(500)	Names of the observation sequence files - each of which **MUST** have an extension that defines the start of the forecast (the analysis time). The observation sequence filenames must be something like `obs_seq.forecast.YYYYMMDDHH` . If `obs_sequences` is specified, `obs_sequence_list` must be empty.
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. The observation sequence filenames **MUST** have an extension that defines the start of the forecast (the analysis time). The observation sequence filenames must be something like `obs_seq.forecast.YYYYMMDDHH`. `obs_sequence_list` can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program. If `obs_sequence_list` is specified, `obs_sequences` must be empty.
station_template	character(len=256)	The name of the netCDF file created by *program obs_seq_coverage* that contains the verification network description.
netcdf_out	character(len=256)	The base portion of the filename of the file that will contain the forecast quantities. Since each observation type of interest is processed with a separate run of `obs_seq_verify`, the observation type string is used to create a unique output filename.
calendar	character(len=129)	The type of the calendar used to interpret the dates.
obtype_string	character(len=32)	The observation type string that will be verified. The character string must match one of the standard DART observation types. This will be the name of the variable in the netCDF file, and will also be used to make a unique netCDF file name.
print_every	integer	Print run-time information for every "`print_every`" *n*-th observation.
verbose	logical	Print extra run-time information.
debug	logical	Print a frightening amount of run-time information.

6.171.3 Other modules used

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
models/your_model/model_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
observations/forward_operators/obs_def_mod.f90
```

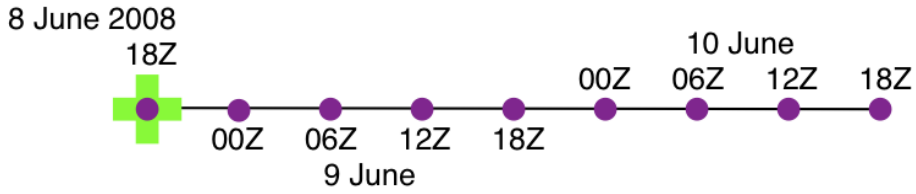
6.171.4 Files

- `input.nml` is used for *obs_seq_verify_nml*
- A netCDF file containing the metadata for the verification network. This file is created by *program obs_seq_coverage* to define the desired times and locations for the verification. (`obsdef_mask.nc` is the default name)
- One or more observation sequence files from `filter` run in *forecast* mode - meaning all the observations were flagged as *evaluate_only*. It is required/presumed that all the ensemble members are output to the observation sequence file (see num_output_obs_members). Each observation sequence file contains all the forecasts from a single analysis time and the filename extension must reflect the analysis time used to start the forecast. (`obs_seq.forecast.YYYYMMDDHH` is the default name)
- Every execution of `obs_seq_verify` results in one netCDF file that contains the observation being verified. If `obtype_string = 'METAR_U_10_METER_WIND'`, and `netcdf_out = 'forecast.nc'`; the resulting filename will be `METAR_U_10_METER_WIND_forecast.nc`.

6.171.5 Usage

`obs_seq_verify` is built in .../DART/models/*your_model*/work, in the same way as the other DART components. Once the forecast has completed, each observation type may be extracted from the observation sequence file and stuffed into the appropriate verification structure. Each observation type must be processed serially at this time, and each results in a separate output netCDF file. Essentially, `obs_seq_verify` sorts an unstructured, unordered set of observations into a predetermined configuration.

Example: a single 48-hour forecast that is evaluated every 6 hours



In this example, the `obsdef_mask.nc` file was created by running *program obs_seq_coverage* with the namelist specified in the single 48hour forecast evaluated every 6 hours example. The `obsdef_mask.txt` file was used to mask the input observation sequence files by *program obs_selection* and the result was run through *PROGRAM filter* with the observations marked as *evaluate_only* - resulting in a file called `obs_seq.forecast.2008060818`. This filename could also be put in a file called `verify_list.txt`.

Just to reiterate the example, both namelists for `obs_seq_coverage` and `obs_seq_verify` are provided below.

```
&obs_seq_coverage_nml
   obs_sequences      = ''
   obs_sequence_list  = 'coverage_list.txt'
   obs_of_interest    = 'METAR_U_10_METER_WIND'
                        'METAR_V_10_METER_WIND'
   textfile_out       = 'obsdef_mask.txt'
   netcdf_out         = 'obsdef_mask.nc'
   calendar           = 'Gregorian'
   first_analysis     =  2008, 6, 8, 18, 0, 0
```

(continues on next page)

(continued from previous page)

```
   last_analysis      =  2008, 6, 8, 18, 0, 0
   forecast_length_days          = 2
   forecast_length_seconds       = 0
   verification_interval_seconds = 21600
   temporal_coverage_percent     = 100.0
   lonlim1            =    0.0
   lonlim2            =  360.0
   latlim1            =  -90.0
   latlim2            =   90.0
   verbose            = .true.
   /

&obs_seq_verify_nml
   obs_sequences      = 'obs_seq.forecast.2008060818'
   obs_sequence_list  = ''
   station_template   = 'obsdef_mask.nc'
   netcdf_out         = 'forecast.nc'
   obtype_string      = 'METAR_U_10_METER_WIND'
   print_every        = 10000
   verbose            = .true.
   debug              = .false.
   /
```

The pertinent information from the `obsdef_mask.nc` file is summarized (from *ncdump -v experiment_times,analysis,forecast_lead obsdef_mask.nc*) as follows:

```
verification_times = 148812.75, 148813, 148813.25, 148813.5, 148813.75,
                             148814, 148814.25, 148814.5, 148814.75 ;

analysis           = 148812.75 ;

forecast_lead      = 0, 21600, 43200, 64800, 86400, 108000, 129600, 151200, 172800 ;
```

There is one analysis time, 9 forecast leads and 9 verification times. The analysis time is the same as the first verification time. The run-time output of `obs_seq_verify` and a dump of the resulting netCDF file follows:

```
[thoar@mirage2 work]$ ./obs_seq_verify |& tee my.verify.log
 Starting program obs_seq_verify
 Initializing the utilities module.
 Trying to log to unit           10
 Trying to open file dart_log.out

 ---------------------------------------
 Starting ... at YYYY MM DD HH MM SS =
                 2011  3  1 10  2 54
 Program obs_seq_verify
 ---------------------------------------

 set_nml_output Echo NML values to log file only
 Trying to open namelist log dart_log.nml
 ------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
-------------- ASSIMILATE_THESE_OBS_TYPES --------------
RADIOSONDE_TEMPERATURE
RADIOSONDE_U_WIND_COMPONENT
RADIOSONDE_V_WIND_COMPONENT
SAT_U_WIND_COMPONENT
SAT_V_WIND_COMPONENT
-------------- EVALUATE_THESE_OBS_TYPES --------------
RADIOSONDE_SPECIFIC_HUMIDITY
------------------------------------------------------

find_ensemble_size:  opening obs_seq.forecast.2008060818
location_mod: Ignoring vertical when computing distances; horizontal only
find_ensemble_size: There are   50 ensemble members.

fill_stations:  There are          221 stations of interest,
fill_stations: ...  and              9 times    of interest.
InitNetCDF:  METAR_U_10_METER_WIND_forecast.nc is fortran unit            5

obs_seq_verify:  opening obs_seq.forecast.2008060818
analysis            1 date is 2008 Jun 08 18:00:00

index    6 is prior ensemble member      1
index    8 is prior ensemble member      2
index   10 is prior ensemble member      3
...
index  100 is prior ensemble member     48
index  102 is prior ensemble member     49
index  104 is prior ensemble member     50

QC index           1  NCEP QC index
QC index           2  DART quality control

Processing obs        10000  of        84691
Processing obs        20000  of        84691
Processing obs        30000  of        84691
Processing obs        40000  of        84691
Processing obs        50000  of        84691
Processing obs        60000  of        84691
Processing obs        70000  of        84691
Processing obs        80000  of        84691

METAR_U_10_METER_WIND dimlen            1  is            9
METAR_U_10_METER_WIND dimlen            2  is           50
METAR_U_10_METER_WIND dimlen            3  is            3
METAR_U_10_METER_WIND dimlen            4  is            1
METAR_U_10_METER_WIND dimlen            5  is          221
METAR_U_10_METER_WIND dimlen            6  is            1
obs_seq_verify:  Finished successfully.

-------------------------------------
Finished ... at YYYY MM DD HH MM SS =
```

(continues on next page)

(continued from previous page)

```
                2011  3  1 10  3  7
 -------------------------------------

[thoar@mirage2 work]$ ncdump -h METAR_U_10_METER_WIND_forecast.nc
netcdf METAR_U_10_METER_WIND_forecast {
dimensions:
        analysisT = UNLIMITED ; // (1 currently)
        copy = 3 ;
        station = 221 ;
        level = 14 ;
        ensemble = 50 ;
        forecast_lead = 9 ;
        linelen = 129 ;
        nlines = 446 ;
        stringlength = 64 ;
        location = 3 ;
variables:
        char namelist(nlines, linelen) ;
                namelist:long_name = "input.nml contents" ;
        char CopyMetaData(copy, stringlength) ;
                CopyMetaData:long_name = "copy quantity names" ;
        double analysisT(analysisT) ;
                analysisT:long_name = "time of analysis" ;
                analysisT:units = "days since 1601-1-1" ;
                analysisT:calendar = "Gregorian" ;
                analysisT:missing_value = 0. ;
                analysisT:_FillValue = 0. ;
        int copy(copy) ;
                copy:long_name = "observation copy" ;
                copy:note1 = "1 == observation" ;
                copy:note2 = "2 == prior" ;
                copy:note3 = "3 == observation error variance" ;
                copy:explanation = "see CopyMetaData variable" ;
        int station(station) ;
                station:long_name = "station index" ;
        double level(level) ;
                level:long_name = "vertical level of observation" ;
        int ensemble(ensemble) ;
                ensemble:long_name = "ensemble member" ;
        int forecast_lead(forecast_lead) ;
                forecast_lead:long_name = "forecast lead time" ;
                forecast_lead:units = "seconds" ;
        double location(station, location) ;
                location:description = "location coordinates" ;
                location:location_type = "loc3Dsphere" ;
                location:long_name = "threed sphere locations: lon, lat, vertical" ;
                location:storage_order = "Lon Lat Vertical" ;
                location:units = "degrees degrees which_vert" ;
        int which_vert(station) ;
                which_vert:long_name = "vertical coordinate system code" ;
                which_vert:VERTISUNDEF = -2 ;
                which_vert:VERTISSURFACE = -1 ;
```

(continues on next page)

(continued from previous page)

```
                which_vert:VERTISLEVEL = 1 ;
                which_vert:VERTISPRESSURE = 2 ;
                which_vert:VERTISHEIGHT = 3 ;
                which_vert:VERTISSCALEHEIGHT = 4 ;
        double METAR_U_10_METER_WIND(analysisT, station, level, copy, ensemble, forecast_
↪lead) ;
                METAR_U_10_METER_WIND:long_name = "forecast variable quantities" ;
                METAR_U_10_METER_WIND:missing_value = -888888. ;
                METAR_U_10_METER_WIND:_FillValue = -888888. ;
        int original_qc(analysisT, station, forecast_lead) ;
                original_qc:long_name = "original QC value" ;
                original_qc:missing_value = -888888 ;
                original_qc:_FillValue = -888888 ;
        int dart_qc(analysisT, station, forecast_lead) ;
                dart_qc:long_name = "DART QC value" ;
                dart_qc:explanation1 = "1 == prior evaluated only" ;
                dart_qc:explanation2 = "4 == forward operator failed" ;
                dart_qc:missing_value = -888888 ;
                dart_qc:_FillValue = -888888 ;
// global attributes:
                :creation_date = "YYYY MM DD HH MM SS = 2011 03 01 10 03 00" ;
                :source = "$URL$" ;
                :revision = "$Revision$" ;
                :revdate = "$Date$" ;
                :obs_seq_file_001 = "obs_seq.forecast.2008060818" ;
}
[thoar@mirage2 work]$
```

Discussion

- the values of *ASSIMILATE_THESE_OBS_TYPES* and *EVALUATE_THESE_OBS_TYPES* are completely irrelevant - again - since `obs_seq_verify` is not actually doing an assimilation.
- The analysis time from the filename is used to determine which analysis from `obsdef_mask.nc` is being considered, and which set of verification times to look for. This is important.
- The individual `prior ensemble member` copies must be present! Since there are no observations being assimilated, there is no reason to choose the posteriors over the priors.
- There are 221 locations reporting METAR_U_10_METER_WIND observations at all 9 requested verification times.
- The `METAR_U_10_METER_WIND_forecast.nc` file has all the metadata to be able to interpret the *METAR_U_10_METER_WIND* variable.
- The *analysisT* dimension is the netCDF record/unlimited dimension. Should you want to increase the strength of the statistical results, you should be able to trivially `ncrcat` more (compatible) netCDF files together.

6.171.6 References

- none - but this seems like a good place to start: The Centre for Australian Weather and Climate Research - Forecast Verification Issues, Methods and FAQ

6.172 PROGRAM `wakeup_filter`

Attention: `wakeup_filter` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `wakeup_filter` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.172.1 Overview

Small auxiliary program for use in the "async=4" case where the main filter program is an MPI program and the model being run with DART is also an MPI program. The main MPI job script runs each of the model advances for the ensemble members, and then runs this program to restart the filter program.

6.172.2 Modules used

```
mpi_utilities_mod
```

6.172.3 Namelist

There are no namelist options for this program. It must be run as an MPI program with the same number of tasks as filter was originally started with.

6.172.4 Files

Named pipes (fifo) files are used to synchronize with the main MPI job run script, to ensure that the filter program and the script do not do a "busy-wait" in which they consume CPU cycles while they are waiting for each other. The fifo names are:

- filter_to_model.lock
- model_to_filter.lock
- filter_lockNNNNN (where NNNNN is the task number with leading 0s)

6.172.5 References

- Anderson, J., T. Hoar, K. Raeder, H. Liu, N. Collins, R. Torn, and A. Arellano, 2009: The Data Assimilation Research Testbed: A Community Facility. Bull. Amer. Meteor. Soc., 90, 1283-1296. DOI: 10.1175/2009BAMS2618.1

6.173 PROGRAM compare_states

6.173.1 Overview

Utility program to compare fields in two NetCDF files and print out the min and max values from each file and the min and max of the differences between the two fields. The default is to compare all numeric variables in the files, but specific variables can be specified in the namelist or in a separate file. The two input NetCDF filenames are read from the console or can be echo'd into the standard input of the program.

If you want to restrict the comparison to only specific variables in the files, specify the list of field names to compare either in the namelist, or put a list of fields, one per line, in a text file and specify the name of the text file. Only data arrays can be compared, not character arrays, strings, or attribute values.

Namelist interface `&compare_states_nml` must be read from file `input.nml`.

6.173.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&compare_states_nml
   do_all_numeric_fields   = .true.
   fieldnames              = ''
   fieldlist_file          = ''
   fail_on_missing_field   = .true.
   only_report_differences = .true.
   debug                   = .false.
  /
```

Item	Type	Description
do_all_numeric_fields	logical	If true, i.e., all integer, float, and double variables in the NetCDF files will have their values compared. If .false. the list of specific variables to be compared must be given either directly in the namelist in the `fieldnames` item, or else the field names must be listed in an ASCII file, one name per line, and the name of that file is specified in `fieldlist_file`.
fieldnames	character list	One or more names of arrays in the NetCDF files to be compared. Only read if `do_all_numeric_fields` is .false.
fieldlist_file	character	Name of a text file containing the fieldnames, one per line. It is an error to specify both the fieldnames namelist item and this one. Only read if `do_all_numeric_fields` is .false.
fail_on_missing_field	logical	If true. and any one of the field names is not found in both files it is a fatal error. If .false. a message is printed about the missing field but execution continues.
only_report_differences	logical	If true only print the name of the variable being tested; skip printing the variable value min and max if the two files are identical. If .false. print more details about both variables which differ and varibles with the same values.
debug	logical	If true print out debugging info.

6.173.3 Modules used

```
types_mod
utilities_mod
parse_args_mod
```

6.173.4 Files

- two NetCDF input files
- compare_states.nml
- field names text file (optionally)

6.173.5 References

- none

6.174 PROGRAM gen_sampling_err_table

6.174.1 Overview

Utility program which computes a table of values needed to apply Sampling Error Correction (SEC) during assimilation. These values are used to correct covariances based on small sample size statistics. See reference below.

To enable the sampling error correction algorithm in `filter`, set the namelist item &assim_tools_nml : sampling_error_correction to *.true.*, and copy the netCDF file gen_sampling_err_table/work/sampling_error_correction_table.nc into the run directory.

The name of the SEC table is always `sampling_error_correction_table.nc`. This is a NetCDF format file. If this file already exists in the current directory any tables for new ensemble sizes will be appended to the existing file. If the file does not exist a new file will be created by this tool. The resulting file should be copied into the current working directory when `filter` is run.

A file with almost 200 common ensemble sizes is distributed with the system. Any new ensemble sizes can be generated on demand. Be aware that the computation can be time consuming. The job may need to be submitted to a batch system if many new ensemble sizes are being generated, or start the job on a laptop and leave it to run overnight.

The file contains a "sparse array" of ensemble sizes. Only sizes which have an existing table are stored in the file so large ensemble sizes do not require a large jump in the size of the output file.

The structure of sampling_error_correction_table.nc is shown in this example.

```
0[1095] desktop:system_simulation/work % ncdump -v ens_sizes *nc
netcdf sampling_error_correction_table {
dimensions:
   bins = 200 ;
   ens_sizes = UNLIMITED ; // (40 currently)
variables:
   int count(ens_sizes, bins) ;
           count:description = "number of samples in each bin" ;
   double true_corr_mean(ens_sizes, bins) ;
   double alpha(ens_sizes, bins) ;
           alpha:description = "sampling error correction factors" ;
   int ens_sizes(ens_sizes) ;
           ens_sizes:description = "ensemble size used for calculation" ;

   // global attributes:
           :num_samples = 100000000 ;
           :title = "Sampling Error Corrections for fixed ensemble sizes." ;
           :reference = "Anderson, J., 2012: Localization and Sampling Error Correction
            in Ensemble Kalman Filter Data Assimilation. Mon. Wea. Rev., 140, 2359-2371,
            doi: 10.1175/MWR-D-11-00013.1." ;
           :version = "$Id: gen_sampling_err_table.f90 13014 2019-03-14 21:59:07Z␣
↪nancy@ucar.edu $" ;

 data:

 ens_sizes = 5,  6,  7,  8,  9, 10, 12, 14, 15, 16, 18, 20, 22, 24, 28, 30, 32, 36, 40,␣
↪44,
            48, 49, 50, 52, 56, 60, 64, 70, 72, 80, 84, 88, 90, 96, 100, 120, 140, 160,␣
↪180, 200
}
```

This program uses the random number generator to compute the correction factors. The generator is seeded with the ensemble size so repeated runs of the program will generate the same values for the tables.

6.174.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&gen_sampling_error_table_nml
   ens_sizes = -1
   debug = .false.
   /
```

Description of each namelist entry

ens_sizes
type: integer(200)

List of ensemble sizes to compute Sampling Error Correction tables for. These do not need to be in any particular order. Duplicates will be removed and any sizes which already have tables computed in the output file will be skipped. The file which comes with the system already has tables computed for these ensemble sizes:

```
ens_sizes = 3,  4,  5,  6,  7,  8,  9, 10,
   11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
   21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
   31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
   41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
   51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
   61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
   71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
   81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
   91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
   101, 102, 103, 104, 105, 106, 107, 108, 109, 110,
   111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
   121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
   131, 132, 133, 134, 135, 136, 137, 138, 139, 140,
   141, 142, 143, 144, 145, 146, 147, 148, 149, 150,
   151, 152, 153, 154, 155, 156, 157, 158, 159, 160,
   161, 162, 163, 164, 165, 166, 167, 168, 169, 170,
   171, 172, 173, 174, 175, 176, 177, 178, 179, 180,
   181, 182, 183, 184, 185, 186, 187, 188, 189, 190,
   191, 192, 193, 194, 195, 196, 197, 198, 199, 200
```

debug
type: logical

If true print out debugging info.

6.174.3 Examples

To add tables for ensemble sizes 220 and 256 run the program with this namelist:

```
&gen_sampling_error_table_nml
   ens_sizes = 220, 256,
   debug = .false.
   /
```

6.174.4 Modules used

```
types_mod
utilities_mod
random_seq_mod
netcdf
```

6.174.5 Files

- output file is always `sampling_error_correction_table.nc` If one exists new ensemble sizes will be appended. If it doesn't exist a new file will be created. This is a NetCDF format file.

6.174.6 References

- Ref: Anderson, J., 2012: Localization and Sampling Error Correction in Ensemble Kalman Filter Data Assimilation. Mon. Wea. Rev., 140, 2359-2371, doi: 10.1175/MWR-D-11-00013.1.

6.175 PROGRAM `perturb_single_instance`

6.175.1 Overview

Utility program to generate an ensemble of perturbed ensemble member restart files. This program can be run in parallel and used as a stand alone program.

6.175.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&perturb_single_instance
   ens_size               = ''
   input_files            = ''
   output_files           = ''
   output_file_list       = ''
   perturbation_amplitude = 0.0
   single_restart_file_in = .false.
  /
```

Item	Type	Description
ens_size	integer	Total number of ensemble members.
input_files	character(len=256),dimension(num_domains)	The restart file you would like to perturb from.
output_file_list	character(len=256)	A file containing a list of the desired output names.
output_files	character(len=256)	An array of filenames
perturbation_amplitude	real(r8)	The desired perturbation amplitude. If the model provides an interface then it will use that subroutine, otherwise it will simply add gaussian noise to the entire state, and this is the standard deviation.
single_restart_file_in	logical	A boolean, specifying if you have a single file restart, such as the case for lower order models.

Below is an example of a typical namelist for the perturb_single_instance.

```
&perturb_single_instance_nml
   ens_size          = 3
   input_files       = 'caminput.nc'
   output_files      = 'cam_pert1.nc','cam_pert2.nc','cam_pert3.nc'
/
```

6.175.3 Files

- inputfile.nc (description file that will be perturbed)
- output_file_list.txt (a file containing a list of restart files) and,
- perturb_single_instance.nml

6.175.4 References

- none

6.176 system simulation programs

6.176.1 Overview

A collection of standalone programs for simulating various properties of ensembles.

- `full_error.f90`
- `obs_sampling_err.f90`
- `sampling_error.f90`
- `system_simulation.f90`
- `test_sampling_err_table.f90`

- correl_error.f90
- sys_sim101.f90
- sys_sim101a.f90
- sys_sim102.f90
- sys_sim102b.f90
- sys_sim103.f90
- sys_sim104.f90
- sys_sim104b.f90
- sys_sim105.f90
- sys_sim2.f90
- sys_sim201.f90
- sys_sim202.f90
- sys_sim203.f90
- sys_sim3.f90
- sys_sim301.f90
- sys_sim302.f90
- sys_sim4.f90
- sys_sim401.f90
- sys_sim402.f90
- sys_sim5.f90
- sys_sim501.f90
- sys_sim502.f90

Note: Talk to Jeff Anderson about the programs in this directory.

6.177 PROGRAM compute_error

6.177.1 Overview

Utility program to compute the time-mean ensemble error and spread in the same manner that the DART MATLAB diagnostic routine 'plot_total_err' does. It runs from the command line, opens no windows, and outputs several types of numerical results on standard output. Grep for 'Total' to get the 2 lines with total error and total spread. Intended for scripts where only the numeric results are wanted instead of a time-series plot. This routine does not do any weighted computations.

The default is to compare a True_State.nc file output from perfect_model_obs to a Prior_Diag.nc file output from filter. Other filenames can be specified in the namelist. These files must have at least one overlapping value in the 'time' array. The statistics will be done on the overlapping time region only.

The output includes the min and max error and spread values, and the time index and time value where that occurs. There is also an option to recompute the time mean ensemble error and spread after skipping the first N times. This

can be useful to skip an initial error spike while the model is spinning up which can result in a larger than expected total error.

Namelist interface `&compute_error_nml` is read from file `input.nml`.

6.177.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&compute_error_nml
   truth_file_name   = 'true_state.nc'
   diag_file_name    = 'preassim.nc'
   skip_first_ntimes = 0
  /
```

Item	Type	Description
truth_file_name	character(len=256)	State-space diagnostic file from the 'perfect_model_obs' program.
diag_file_name	character(len=256)	State space diagnostic file output from the 'filter' program.
skip_first_ntimes	integer	If set to a value greater than 0, the error values will be recomputed a second time, skipping the first N times. This can be useful when running an experiment that has an initial error spike as the model spins up and then decays down to a more steady state.

6.177.3 Modules used

```
types_mod
utilities_mod
```

6.177.4 Files

- DART diagnosic files (True_State.nc, Prior_Diag.nc)
- compute_error.nml

6.177.5 References

- none

6.178 PROGRAM preprocess

6.178.1 Overview

Preprocess is a DART-supplied preprocessor program. Preprocess is used to insert observation specific code into DART at compile time.

In DART, forward operators are not specific to any one model. To achieve this separation between models and forward operators DART makes a distinction between an observation *type* and a physical *quantity*. For example, a radiosonde used to measure windspeed would be a *type* of observation. Zonal wind and meridional wind are *quantities* used to calculate windspeed. Specifying many observation types allows DART to be able to evaluate some observations and assimilate others even if the instruments measure the same quantity.

Preprocess takes user supplied observation and quantity files and combines them with template files to produce code for DART. Use the namelist option 'obs_type_files' to specify the input observation files and the namelist option 'quantity_files' to specify the input quantity files.

- If no quantity files are given, a default list of quantities is used.
- If no obs_type_files are given, only identity observations can be used in the filter (i.e. the state variable values are directly observed; forward operator is an identity)

The template files `DEFAULT_obs_def_mod.F90` and `DEFAULT_obs_kind_mod.F90` contain specially formatted comment lines. These comment lines are used as markers to insert observation specific information. Prepreocess relies these comment lines being used *verbatim*.

There is no need to to alter `DEFAULT_obs_def_mod.F90` or `DEFAULT_obs_kind_mod.F90`. Detailed instructions for adding new observation types can be found in *MODULE obs_def_mod*. New quantities should be added to a quantity file, for example a new atmosphere quantity should be added to `atmosphere_quantities_mod.f90`.

Every line in a quantity file between the start and end markers must be a comment or a quantity definition (QTY_string). Multiple name-value pairs can be specified for a quantity but are not required. For example, temperature may be defined: `! QTY_TEMPERATURE units="K" minval=0.0`. Comments are allowed between quantity definitions or on the same line as the definition. The code snippet below shows acceptable formats for quantity definitions

```
! BEGIN DART PREPROCESS QUANTITY DEFINITIONS
!
! Formats accepted:
```

(continues on next page)

(continued from previous page)

```
!
! QTY_string
! QTY_string name=value
! QTY_string name=value name2=value2
!
! QTY_string ! comments
!
! ! comment
!
! END DART PREPROCESS QUANTITY DEFINITIONS
```

The output files produced by preprocess are named `assimilation_code/modules/observations/obs_kind_mod.f90` and `observations/forward_operators/obs_def_mod.f90`, but can be renamed by namelist control if needed. Be aware that if you change the name of these output files, you will need to change the path_names files for DART executables.

6.178.2 Namelist

When you run preprocess, the namelist is read from the file `input.nml` in the directory where preprocess is run.

Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. These are the defaults:

```
&preprocess_nml
  overwrite_output        = .true.,
  input_obs_def_mod_file  = '../../../observations/forward_operators/DEFAULT_obs_def_mod.
↪F90',
  output_obs_def_mod_file = '../../../observations/forward_operators/obs_def_mod.f90',
  input_obs_qty_mod_file  = '../../../assimilation_code/modules/observations/DEFAULT_obs_
↪kind_mod.F90',
  output_obs_qty_mod_file = '../../../assimilation_code/modules/observations/obs_kind_
↪mod.f90',
  quantity_files          = '../../../assimilation_code/modules/observations/atmosphere_
↪quantities_mod.f90',
  obs_type_files          = '../../../observations/forward_operators/obs_def_reanalysis_
↪bufr_mod.f90',
                            '../../../observations/forward_operators/obs_def_rel_
↪humidity_mod.f90',
                            '../../../observations/forward_operators/obs_def_altimeter_
↪mod.f90'
 /
```

Item	Type	Description
input_obs_def_mod_file	character(len=256)	Path name of the template observation definition module to be preprocessed. The default is `../../observations/forward_operators/DEFAULT_obs_def_mod.F90`. This file must have the appropriate commented lines indicating where the different parts of the input special obs definition modules are to be inserted.
output_obs_def_mod_file	character(len=256)	Path name of output observation definition module to be created by preprocess. The default is `../../observations/forward_operators/obs_def_mod.f90`.
input_obs_qty_mod_file	character(len=256)	Path name of input obs quantity file to be preprocessed. The default path name is `../../../assimilation_code/modules/observations/DEFAULT_obs_kind_mod.F90`. This file must have the appropriate commented lines indicating where the different quantity modules are to be inserted.
output_obs_qty_mod_file	character(len=256)	Path name of output obs quantity module to be created by preprocess. The default is `../../../assimilation_code/modules/observations/obs_kind_mod.f90`.
obs_type_files	character(len=256), dimension(:)	A list of files containing observation definitions for the type of observations you want to use with DART. The maximum number of files is limited to MAX_OBS_TYPE_FILES = 1000. The DART obs_def files are in `observations/forward_operators/obs_def_*.mod.f90`.
overwrite_output	logical	By default, preprocess will overwrite the existing obs_kind_mod.f90 and obs_def_mod.f90 files. Set `overwrite_output = .false.` if you want to preprocess to not overwrite existing files.

6.178.3 Modules used

```
parse_arges_mod
types_mod
utilities_mod
```

Namelist interface `&preprocess_nml` must be read from file `input.nml`.

6.178.4 Files

- input_obs_def_mod_file, specified by namelist; usually `DEFAULT_obs_def_mod.F90`.
- output_obs_def_mod_file, specified by namelist; usually `obs_def_mod.f90`.
- input_obs_qty_mod_file, specified by namelist; usually `DEFAULT_obs_kind_mod.F90`.
- output_obs_qty_mod_file, specified by namelist; usually `obs_kind_mod.f90`.
- obs_type_files, specified by namelist; usually files like `obs_def_reanalysis_bufr_mod.f90`.
- quantity_files, specified by namelist; usually files like `atmosphere_quantities_mod.f90`.
- namelistfile; `input.nml`

6.178.5 References

- none

6.179 PROGRAM `obs_impact_tool`

6.179.1 Overview

The standard DART algorithms compute increments for an observation and then compute corresponding increments for each model state variable due to that observation. To do this, DART computes a sample regression coefficient using the prior ensemble distributions of a state variable and the observation. The increments for each member of the observation are multiplied by this regression coefficient and then added to the corresponding prior ensemble member for the state variable. However, in many cases, it is appropriate to reduce the impact of an observation on a state variable; this is called localization. The standard DART algorithms allow users to specify a localization that is a function of the horizontal (and optionally vertical) distance between the observation and the state variable. The localization is a value between 0 and 1 and multiplies the regression coefficient when updating state ensemble members.

Sometimes, it may be desirable to do an additional localization that is a function of the type of observation and the state vector quantity. This program allows users to construct a table that is read by filter at run-time to localize the impact of sets of observation types on sets of state vector quantities. Users can create named sets of observation types and sets of state vector quantities and specify a localization for the impact of the specified observation types on the state vector quantities.

An example would be to create a subset of observations of tracer concentration for a variety of tracers, and a subset of dynamic state variable quantities like temperatures and wind components. It has been common to set this localization value to 0 so that tracer observations have no impact on dynamic state quantities, however, the tool allows values between 0 and 1 to be specified.

This tool allows related collections of observation types and state vector quantities to be named and then express the relationship of the named groups to each other in a concise way. It can also define relationships by exceptions.

All the listed observation types and state vector quantities must be known by the system. If they are not, look at the &preprocess_nml :: input_items namelist which specifies which *obs_def_xxx_mod.f90* files are included, which is where observation types are defined. Quantities for different regimes (atmosphere, ocean, land, etc.) are defined in `assimilation_code/modules/observations/xxx_quantities_mod.f90` and explained in *MODULE obs_kind_mod*

Format of the input file can be any combination of these types of sections:

```
# hash mark starts a comment.

# the GROUP keyword starts a group and must be followed
# by a name.  All types or quantities listed before the END
# line becomes members of this group.

# GROUPs cannot contain nested groups.

GROUP groupname1
 QTY_xxx  QTY_xxx  QTY_xxx
 QTY_xxx                          # comments can be here
END GROUP

GROUP groupname2
 QTY_xxx
```

(continues on next page)

(continued from previous page)

```
 QTY_xxx
 QTY_xxx
 QTY_xxx
END GROUP

# GROUPs can also be defined by specifying ALL, ALLQTYS,
# or ALLTYPES and then EXCEPT and listing the types or
# quantities which should be removed from this group.
# ALL EXCEPT must be the first line in a group, and all
# subsequent items are removed from the list.
# The items listed after EXCEPT can include the names
# of other groups.

GROUP groupnameM
ALL EXCEPT QTY_xxx QTY_xxx
QTY_xxx
END GROUP

GROUP groupnameN
ALL EXCEPT groupnameY
END GROUP


# once any groups have been defined, a single instance
# of the IMPACT table is specified by listing a TYPE,
# QTY, or group in column 1, then a QTY or GROUP
# in column 2 (the second name cannot be a specific type).
# column 3 must be 0.0 or 1.0.  subsequent entries
# that overlap previous entries have precedence
# (last entry wins).

IMPACT
 QTY_xxx     QTY_xxx      0.0
 QTY_xxx     groupname1   0.0
 groupname1  QTY_xxx      0.0
 groupname1  groupname1   0.0
END IMPACT
```

Namelist interface `&obs_impact_tool_nml` must be read from file `input.nml`.

6.179.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_impact_tool_nml
  input_filename          = 'cross_correlations.txt'
  output_filename         = 'control_impact_runtime.txt'
  debug                   = .false.
  /
```

Item	Type	Description
input_filename	character(len=512)	Name of an ascii text file which describes how the interaction of observations to state vector values and observations to other observations should be controlled. See the Overview section for details about the format of the input file entries.
output_filename	character(len=512)	Name of an ascii text file which created by this tool. It can be read at filter run time to control the impact of observations on state vector items and other observation values. The format of this file is set by this tool and should not be modified by hand. Rerun this tool to recreate the file.
debug	logical	If true print out debugging info.

6.179.3 Examples

To prevent chemistry species from impacting the meterological variables in the model state, and vice versa:

```
GROUP chem
 QTY_CO QTY_NO QTY_C2H4
END GROUP

GROUP met
 ALLQTYS EXCEPT chem
END GROUP

IMPACT
 chem   met    0.0
 met    chem   0.0
END IMPACT
```

6.179.4 Modules used

```
types_mod
utilities_mod
parse_args_mod
```

6.179.5 Files

- two text files, one input and one output.
- obs_impact_tool.nml

6.179.6 References

- none

6.180 program create_fixed_network_seq

6.180.1 Overview

Reads in an observation sequence file and creates a second observation sequence file. Any time information in the input file is ignored entirely. All of the observations in the input file define a set of observations. The output sequence replicates this set multiple times, either with a fixed period in time or at arbitrarily selected times. The program is driven by input from standard input, either the terminal or a text file.

First, one must select either a regularly repeating time sequence of observations (option 1) or an arbitrarily repeating sequence (option 2). For the fixed period, the total number of observation times, the first observation time and the period of the observations is input and an output observation sequence is generated. For the arbitrary period, the user is queried for the number of observing times and then a set of monotonically increasing times. Finally, the user selects a file name (traditionally obs_seq.in) to which the output file is written. The format of the output file is controlled by the namelist options in obs_sequence_mod.

Any data values or quality control flags associated with the input set are replicated to the output, but this program is typically used with perfect model experiments to create observations without data, which are then filled in by running *program perfect_model_obs*.

6.180.2 Modules used

```
types_mod
utilities_mod
obs_def_mod
obs_sequence_mod
time_manager_mod
model_mod
```

6.180.3 Files

- Input observation sequence (set_def.out is standard).
- Output observation sequence (obs_seq.in is standard).

6.180.4 References

- none

6.181 program obs_loop

6.181.1 Overview

This program is a template that is intended to be modified by the user to do any desired operations on an observation sequence file.

6.181.2 Usage

This program is intended to be used as a template to read in observations from one obs_seq file and write them, optionally modified in some way, to another obs_seq file. It can be compiled and run as-is, but it simply makes an exact copy of the input file.

There are comments in the code (search for `MODIFY HERE`) where you can test values, types, times, error values, and either modify them or skip copying that observation to the output.

There are build files in `observations/utilities/oned` and `observations/utilities/threed_sphere` to build the `obs_loop` program.

6.181.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_loop_nml
   filename_in  = ''
   filename_out = ''
   print_only   = .false.
   calendar     = 'Gregorian'
   /
```

Items in this namelist set the input and output files.

Item	Type	Description
filename_in	character(len=256)	Observation sequence file to read
filename_out	character(len=256)	Observation sequence file to create and write. If this file exists it will be overwritten.
print_only	logical	If .TRUE. then do the work but only print out information about what would be written as output without actually creating the output file.
calendar	character(len=32)	The string name of a valid DART calendar type. (See the *MODULE time_manager_mod* documentation for a list of valid types.) The setting here does not change what is written to the output file; it only changes how the date information is printed to the screen in the informational messages.

6.181.4 Discussion

See the documentation in the obs_kind and obs_def modules for things you can query about an observation, and how to set (overwrite) existing values.

6.181.5 Building

There are build files in `observations/utilities/oned` and `observations/utilities/threed_sphere` to build the `obs_loop` program.

The `preprocess` program must be built and run first to define what set of observation types will be supported. See the *PROGRAM preprocess* for more details on how to define the list and run it. The `&preprocess_nml` namelist in the `input.nml` file must contain files with definitions for the combined set of all observation types which will be encountered over all input obs_seq files.

If you have observation types which are not part of the default list in the &preprocess_nml namelist, add them to the input.nml file and then run quickbuild.sh.

6.181.6 Files

filename	purpose
input.nml	to read the &obs_loop_nml namelist

6.181.7 References

1. none

6.181.8 Error codes and conditions

Routine	Message	Comment
obs_loop		
obs_loop		

6.181.9 Future plans

none

6.182 program perfect_model_obs

6.182.1 Overview

Main program for creating synthetic observation sequences given a model for use in filter assimilations. Reads in an observation sequence file which has only observation definitions and generates synthetic observation values for an output observation sequence file. The execution of perfect_model_obs is controlled by the input observation sequence file and the model time-stepping capabilities in a manner analogous to that used by the filter program.

6.182.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&perfect_model_obs_nml
   single_file_in             = .false.,
   read_input_state_from_file = .false.,
   input_state_files          = "",
   init_time_days             = 0,
   init_time_seconds          = 0,

   single_file_out            = .false.,
   output_state_files         = "",
   write_output_state_to_file = .false.,
   output_interval            = 1,

   async                      = 0,
   adv_ens_command            = "./advance_model.csh",
   tasks_per_model_advance    = 1,

   obs_seq_in_file_name       = "obs_seq.in",
   obs_seq_out_file_name      = "obs_seq.out",
   first_obs_days             = -1,
```

(continues on next page)

(continued from previous page)

```
   first_obs_seconds          = -1,
   last_obs_days              = -1,
   last_obs_seconds           = -1,
   obs_window_days            = -1,
   obs_window_seconds         = -1,

   trace_execution            = .false.,
   output_timestamps          = .false.,
   print_every_nth_obs        = 0,
   output_forward_op_errors   = .false.,
   silence                    = .false.,
/
```

Item	Type	Description
read_input_state_from_file	logical	If false, model_mod must provide the input state.
single_file_in	logical	Get all states from a single file.
input_state_files	character(len=256) dimension(MAX_NUM_DOMS)	A list of files, one per domain. Each file must be a text file containing the name of the NetCDF file to open.
write_output_state_to_file	logical	If false, state is not written out.
single_file_out	logical	Write all states to a single file.
output_state_files	character(len=256) dimension(MAX_NUM_DOMS)	A list of files, one per domain. Each file must be a text file containing the names of the NetCDF file to open.
init_time_days	integer	If negative, don't use. If non-negative, override the initial data time read from restart file.
init_time_seconds	integer	If negative don't use. If non-negative, override the initial data time read from restart file.
output_interval	integer	Output state and observation diagnostics every nth assimilation time, n is output_interval.
async	integer	Controls method for advancing model: • 0 = subroutine call • 2 = shell command, single task model • 4 = shell command, parallel model
adv_ens_command	character(len=129)	Command sent to shell if async == 2 or 4.
tasks_per_model_advance	integer	Number of tasks to use while advancing the model.
obs_seq_in_file_name	character(len=256)	File name from which to read an observation sequence.
obs_seq_out_file_name	character(len=256)	File name to which to write output observation sequence.
first_obs_days	integer	If negative, don't use. If non-negative, ignore any observations before this time.
first_obs_seconds	integer	If negative, don't use. If non-negative, ignore any observations before this time.
last_obs_days	integer	If negative, don't use. If non-negative, ignore any observations after this time.
last_obs_seconds	integer	If negative, don't use. If non-negative, ignore any observations after this time.
obs_window_days	integer	If negative, don't use. If non-negative, reserved for future use.
obs_window_seconds	integer	If negative, don't use. If non-negative, reserved for future use.
trace_execution	logical	True means output very detailed messages about what routines are being called in the main loop. Useful if a job hangs or otherwise doesn't execute as expected.
output_timestamps	logical	True means output timestamps be-

6.182.3 Modules used

```
types_mod
utilities_mod
time_manager_mod
obs_sequence_mod
obs_def_mod
obs_model_mod
assim_model_mod
mpi_utilities_mod
random_seq_mod
ensemble_manager_mod
```

6.182.4 Files

- observation sequence input file; name comes from obs_seq_in_file_name
- observation sequence output file; name comes from obs_seq_out_file_name
- input state vector file; name comes from restart_in_file_name
- output state vector file; name comes from restart_out_file_name
- perfect_model_mod.nml in input.nml

6.182.5 References

- none

6.183 program `obs_selection`

6.183.1 Overview

This specialized tool selects a subset of input observations from an observation sequence file. For a more general purpose observation sequence file tool, see the *program obs_sequence_tool*. This tool takes a selected list of observation types, times, and locations, and extracts only the matching observations out of one or more obs_sequence files. The tool which creates the input selection file is usually *program obs_seq_coverage*. Alternatively, the selection file can be a full observation sequence file, in which case the types, times, and locations of those observations are used as the selection criteria.

This tool processes each observation sequence file listed in the input namelist `filename_seq` or `filename_seq_list`. If the observation type, time and location matches an entry in the selection file, it is copied through to the output. Otherwise it is ignored.

The actions of the `obs_selection` program are controlled by a Fortran namelist, read from a file named `input.nml` in the current directory. A detailed description of each namelist item is described in the namelist section of this document. The names used in this discussion refer to these namelist items.

6.183.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_selection_nml
   filename_seq          = ''
   filename_seq_list     = ''
   filename_out          = 'obs_seq.processed'
   num_input_files       = 0
   selections_file       = 'obsdef_mask.txt'
   selections_is_obs_seq = .false.
   latlon_tolerance      = 0.000001
   match_vertical        = .false.
   surface_tolerance     = 0.0001
   pressure_tolerance    = 0.001
   height_tolerance      = 0.0001
   scaleheight_tolerance = 0.001
   level_tolerance       = 0.00001
   print_only            = .false.
   partial_write         = .false.
   print_timestamps      = .false.
   calendar              = 'Gregorian'
  /
```

Item	Type	Description
filename_seq	character(len=256), dimension(500)	The array of names of the observation sequence files to process, up to a max count of 500 files. (Specify only the actual number of input files. It is not necessary to specify 500 entries.)
filename_seq_list	character(len=256)	An alternative way to specify the list of input files. The name of a text file which contains, one per line, the names of the observation sequence files to process. You can only specify one of filename_seq OR filename_seq_list, not both.
num_input_files	integer	Optional. The number of observation sequence files to process. Maximum of 500. If 0, the length is set by the number of input files given. If non-zero, must match the given input file list length. (Can be used to verify the right number of input files were processed.)
filename_out	character(len=256)	The name of the resulting output observation sequence file. There is only a single output file from this tool. If the input specifies multiple obs_seq input files, the results are concatinated into a single output file.
selections_file	character(len=256)	The name of the input file containing the mask of observation definitions (the textfile output of *program obs_seq_coverage*). Alternatively, this can be the name of a full observation sequence file. In this case, the types, times, and locations are extracted from this file and then used in the same manner as a mask file from the coverage tool.
selections_is_obs_seq	logical	If .TRUE. the filename given for the "selections_file" is a full obs_sequence file and not a text file from the coverage tool.
lat_lon_tolerance	real(r8)	Specified in degrees. For observations to match in the horizontal the difference in degrees for each of latitude and longitude must be less than this threshold. If less than or equal to 0, the values must match exactly.
match_vertical	logical	If .TRUE. the locations of the observations in the input files have to match the selection list not only the horizontal but also in the vertical.
surface_tolerance	real(r8)	Specified in meters. If "match_vertical" is .FALSE. this value is ignored. If "match_vertical" is .TRUE., this applies to observations with a vertical type of VERTISSURFACE. For observations which match in the horizontal, the vertical surface elevation difference must be less than this to be considered the same.
pressure_tolerance	real(r8)	Specified in pascals. If "match_vertical" is .FALSE. this value is ignored. If "match_vertical" is .TRUE., this applies to observations with a vertical type of VERTISPRESSURE. For observations which match in the horizontal, the vertical difference must be less than this to be considered the same.
height_tolerance	real(r8)	Specified in meters. If "match_vertical" is .FALSE. this value is ignored. If "match_vertical" is .TRUE., this applies to observations with a vertical type of VERTISHEIGHT. For observations which match in the horizontal, the vertical difference must be less than this to be considered the same.
scale_height_tolerance	real(r8)	Specified in unitless values. If "match_vertical" is .FALSE. this value is ignored. If "match_vertical" is .TRUE., this applies to observations with a vertical type of VERTISSCALEHEIGHT. For observations which match in the horizontal, the vertical difference must be less than this to be considered the same.
level_tolerance	real(r8)	Specified in fractional model levels. If "match_vertical" is .FALSE. this value is ignored. If "match_vertical" is .TRUE., this applies to observations with a vertical type of VERTISLEVEL. For observations which match in the horizontal, the vertical difference must be less than this to be considered the same. Note that some models only support integer level values, but others support fractional levels. The vertical value in an observation is a floating point/real value, so fractional levels are possible to specify for an observation.
print_only	logical	If .TRUE. do not create an output file, but print a summary of the number and types of each observation in each input file, and then the number of observations and types which would have been created in an output file.
partial_write	logical	Generally only used for debugging problems. After each input obs_seq file is processed, this flag, if .TRUE., causes the code to write out the partial results to the output file. The default is to process all input files (if more than a single file is specified) and write the output file only at the end of the processing.
print_timestamps	logical	Generally only used for debugging very slow execution runs. This flag, if .TRUE., causes the code to output timestamps (wall clock time) at various locations during the processing phases. It may

6.183.3 Building

Most `$DART/models/*/work` directories contain files needed to build this tool along with the other executable programs. It is also possible to build this tool in the `$DART/observations/utilities` directory. In either case the `preprocess` program must be built and run first to define what set of observation types will be supported. See the *PROGRAM preprocess* for more details on how to define the list and run it. The `&preprocess_nml` namelist in the `input.nml` file must contain files with definitions for the combined set of all observation types which will be encountered over all input obs_seq files.

Usually the directories where executables are built will include a `quickbuild.sh` script which builds and runs preprocess and then builds the rest of the executables.

6.183.4 Modules used

```
types_mod
utilities_mod
time_manager_mod
obs_def_mod
obs_sequence_mod
```

6.183.5 Files

- `input.nml`
- The input files specified in the `filename_seq` namelist variable.
- The output file specified in the `filename_out` namelist variable.

6.183.6 References

- none

6.184 program `obs_sequence_tool`

6.184.1 Overview

DART observation sequence files are stored in a proprietary format. This tool makes it easier to manipulate these files, allowing the user to subset or combine one or more files into a single output file.

The tool has many options to select subsets of observations by time, type, data value, and location. The tool also allows the contents of observations to be changed by subsetting and/or reordering the copies and qc entries. Files with equivalent data but with different metadata labels (e.g. 'NCEP QC' vs. 'QC') can now be merged as well. The tool can be run without creating an output file, only printing a summary of the counts of each observation type in the input files, and it can be used to convert from binary to ASCII and back.

The actions of the `obs_sequence_tool` program are controlled by a Fortran namelist, read from a file named `input.nml` in the current directory. A detailed description of each namelist item is described in the namelist section below.

The examples section of this document below has extensive examples of common usages for this tool. Below that are more details about DART observation sequence files, the structure of individual observations, and general background information.

6.184.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_sequence_tool_nml
   filename_seq          = ''
   filename_seq_list     = ''
   filename_out          = 'obs_seq.processed'
   first_obs_days        = -1
   first_obs_seconds     = -1
   last_obs_days         = -1
   last_obs_seconds      = -1
   obs_types             = ''
   keep_types            = .true.
   min_box               = -888888.0
   max_box               = -888888.0
   min_lat               =  -90.0
   max_lat               =   90.0
   min_lon               =    0.0
   max_lon               =  360.0
   copy_metadata         = ''
   min_copy              = -888888.0
   max_copy              = -888888.0
   copy_type             = ''
   edit_copy_metadata    = .false.
   new_copy_metadata     = ''
   edit_copies           = .false.
   new_copy_index        = -1
   new_copy_data         = -888888.0
   qc_metadata           = ''
   min_qc                = -888888.0
   max_qc                = -888888.0
   edit_qc_metadata      = .false.
   new_qc_metadata       = ''
   edit_qcs              = .false.
   new_qc_index          = -1
   new_qc_data           = -888888.0
   synonymous_copy_list  = ''
   synonymous_qc_list    = ''
   print_only            = .false.
   gregorian_cal         = .true.
   min_gps_height        = -888888.0
   /
```

Item	Type	Description
filename_seq	character(len=256), dimension(1000)	The array of names of the observation sequence files to p
filename_seq_list	character(len=256)	The name of a text file which contains, one per line, the n
filename_out	character(len=256)	The name of the resulting output observation sequence fi
first_obs_days	integer	If non-negative, restrict the timestamps of the observatio
first_obs_seconds	integer	If non-negative, restrict the timestamps of the observatio
last_obs_days	integer	If non-negative, restrict the timestamps of the observatio
last_obs_seconds	integer	If non-negative, restrict the timestamps of the observatio
obs_types	character(len=32), dimension(500)	The array of observation type names to process. If any na
keep_types	logical	Ignored unless one or more observation types are specifie
min_box	real(r8)(:)	If the locations are 1D, set a min value here instead of us
max_box	real(r8)(:)	If the locations are 1D, set a max value here instead of us
min_lat	real(r8)	If specified, the minimum latitude, in degrees, of observa
max_lat	real(r8)	If specified, the maximum latitude, in degrees, of observa
min_lon	real(r8)	If specified, the minimum longitude, in degrees, of obser
max_lon	real(r8)	If specified, the maximum longitude, in degrees, of obser
copy_metadata	character	If specified, the metadata string describing one of the dat
min_copy	real	If specified, the minimum value in the data copy field ma
max_copy	real	If specified, the maximum value in the data copy field ma
copy_type	character(len=32)	If specified, the string name of an observation type to be
edit_copy_metadata	logical	If true, replace the output file metadata strings with the li
new_copy_metadata	character(len=*)(:)	List of new metadata strings. Use with care, there is no e
edit_copies	logical	If true, subset or rearrange the actual data copies in the o
new_copy_index	integer(:)	An array of integers, which control how copies in the inp
new_copy_data	real(:)	An array of reals. The length should correspond to the nu
qc_metadata	character	If specified, the metadata string describing one of the qua
min_qc	real	If specified, the minimum qc value in the QC field match
max_qc	real	If specified, the maximum qc value in the QC field match
edit_qc_metadata	logical	If true, replace the output file metadata strings with the li
new_qc_metadata	character(len=*)(:)	List of new metadata strings. Use with care, there is no e
edit_qcs	logical	If true, subset or rearrange the actual data QCs in the out
new_qc_index	integer(:)	An array of integers, which control how QCs in the input
new_qc_data	real(:)	An array of reals. The length should correspond to the nu
synonymous_copy_list	character(len=*)(:)	An array of strings which are to be considered synonyms
synonymous_qc_list	character(len=*)(:)	An array of strings which are to be considered synonyms
print_only	logical	If .TRUE., do not create an output file, but print a summa
gregorian_cal	logical	If .true. the dates of the first and last observations in each
num_input_files	integer	DEPRECATED. The number of observation sequence fil
remove_precomputed_FO_values	character(len=32), dimension(500)	The (case-insensitive) array of observation type names w

6.184.3 Examples

Here are details on how to set up common cases using this tool:

- Merge multiple files
- Subset in Time
- Subset by Observation Type
- Subset by Location
- Binary to ASCII and back
- Merging files with incompatible Metadata
- Altering the number of Copies or QC values
- Printing only
- Subset by Observation or QC Value
- Precomputed Forward Operator Values

Merge multiple files

Either specify a list of input files for `filename_seq`, like:

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101',
                        'qscatL2B_2007_11_01a.out',
                        'obs_seq.gpsro_2007110106',
   filename_out       = 'obs_seq20071101.all',
   gregorian_cal      = .true.
/
```

and all observations in each of the three input files will be merged in time order and output in a single observation sequence file. Or from the command line create a file containing one filename per line, either with 'ls':

```
ls obs_seq_in* > tlist
```

or with a text editor, or any other tool of your choice. Then,

```
&obs_sequence_tool_nml
   filename_seq_list = 'tlist',
   filename_out       = 'obs_seq20071101.all',
   gregorian_cal      = .true.
/
```

will open 'tlist' and read the filenames, one per line, and merge them together. The output file will be named 'obs_seq20071101.all'. Note that the filenames inside the list file should not have delimiters (e.g. single or double quotes) around the filenames.

Subset in time

The observations copied to the output file can be restricted in time by setting the namelist items for the first and last observation timestamps (in days and seconds). It is not an error for some of the input files to have no observations in the requested time range, and multiple input files can have overlapping time ranges. For example:

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101',
                        'qscatL2B_2007_11_01a.out',
                        'obs_seq.gpsro_2007110106',
   filename_out       = 'obs_seq20071101.06hrs',
   first_obs_days     = 148592,
   first_obs_seconds  =  10801,
   last_obs_days      = 148592,
   last_obs_seconds   =  32400,
   gregorian_cal      = .true.
/
```

The time range is inclusive on both ends; observations with times equal to the boundary times will be copied to the output. To split a single input file up into proper subsets (no replicated observations), the first time of the following output sequence should be +1 second from the last time of the previous output sequence. If the goal is to match an observation sequence file with an assimilation window during the execution of the `filter` program, the windows should be centered around the assimilation time starting at minus 1/2 the window time plus 1 second, and ending at exactly plus 1/2 the window time.

Subset by observation type

You specify a list of observation types, by string name, and then specify a logical value to say whether this is the list of observations to keep, or if it's the list of observations to discard. For example,

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101.06hrs',
   filename_out       = 'obs_seq20071101.wind',
   obs_types          = 'RADIOSONDE_U_WIND_COMPONENT',
                        'RADIOSONDE_V_WIND_COMPONENT',
   keep_types         = .true.,
   gregorian_cal      = .true.
/
```

will create an output file which contains only the U and V wind observations from the given input file.

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101.06hrs',
   filename_out       = 'obs_seq20071101.notemp',
   obs_types          = 'RADIOSONDE_TEMPERATURE',
   keep_types         = .false.,
   gregorian_cal      = .true.
/
```

will strip out all the radiosonde temperature observations and leave everything else.

Subset by location

If the observations have locations specified in 3 dimensions, as latitude, longitude, and a vertical coordinate, then it can be subset by specifying the corners of a lat, lon box. There is currently no vertical subsetting option. For example:

```
min_lat            =    0.0,
max_lat            =   20.0,
min_lon            =  230.0,
max_lon            =  260.0,
```

will only output observations between 0 and 20 latitude and 230 to 260 in longitude. Latitude ranges are 90 to 90, longitude can either be specified from 180 to +180, or 0 to 360.

If the observations have 1 dimensional locations, between 0 and 1, then a bounding box can be specified like:

```
min_box = 0.2,
max_box = 0.4,
```

will keep only those observations between 0.2 and 0.4. In all these tests, points on the boundaries are considered inside the box.

Binary to ASCII and back

To convert a (more compact) binary observation sequence file to a (human readable and portable) ASCII file, a single input and single output file can be specified with no selection criteria. The output file format is specified by the `write_binary_obs_sequence` item in the `&obs_sequence_nml` namelist in the `input.nml` file. It is a Fortran logical; setting it to `.TRUE.` will write a binary file, setting it to `.FALSE.` will write an ASCII text file. If you have a binary file, it must be converted on the same kind of platform as it was created on before being moved to another architecture. At this point in time, there are only 2 remaining incompatible platforms: IBM systems based on PowerPC chips, and everything else (which is Intel or AMD).

Any number of input files and selection options can be specified, as well, but for a simple conversion, leave all other input namelist items unset.

Merging files with incompatible metadata

To merge files which have the same number of copies and qc but different labels for what is exactly the same data, you can specify a list of synonym strings that will pass the matching test. For example:

```
&obs_sequence_tool_nml
   filename_seq       = 'qscatL2B_2007_11_01.out',
                        'obs_seq20071101',
                        'obs_seq.gpsro_2007110124',
   filename_out       = 'obs_seq20071101.all',
   gregorian_cal      = .true.
   synonymous_copy_list = 'NCEP BUFR observation', 'AIRS observation', 'observation',
   synonymous_qc_list   = 'NCEP QC index', 'AIRS QC', 'QC flag - wvc quality flag', 'QC',
/
```

will allow any copy listed to match any other copy on that list, and same with the QC values. If the output metadata strings are not specified (see below), then the actual metadata strings from the first file which is used will set the output metadata strings.

To rename or override, with care, existing metadata strings in a file, set the appropriate edit strings to true, and set the same number of copies and/or QC values as will be in the output file. Note that this will replace, without warning, whatever is originally listed as metadata. You can really mangle things here, so use this with caution:

```
&obs_sequence_tool_nml
   filename_seq       = 'qscat_all_qc_305.out', 'qscat_all_qc_306.out',
   filename_out       = 'qscat_1_qc_2007_11.out',
   edit_copy_metadata = .true.,
   new_copy_metadata  = 'observation',
   edit_qc_metadata   = .true.,
   new_qc_metadata    = 'QC', 'DART quality control',
   gregorian_cal      = .true.
/
```

The log file will print out what input strings are being replaced; check this carefully to be sure you are doing what you expect.

If you use both a synonym list and the edit list, the output file will have the specified edit list strings for metadata.

Altering the number of copies or QC values

To delete some of the copies or QC values in each observation, specify the copy or QC index numbers which are to be passed through, and list them in the exact order they should appear in the output:

```
edit_copies = .true.,
new_copy_index = 1, 2, 81, 82,

edit_qcs = .true.,
new_qc_index = 2,
```

This will create an output sequence file with only 4 copies; the original first and second copies, and copies 81 and 82. The original metadata will be retained. It will have only the second QC value from the original file.

If you are editing the copies or QCs and also specifying new metadata strings, use the number and order appropriate to the output file regardless of how many copies or QC values there were in the original input files.

You can use these index lists to reorder copies or QC values by specifying the same number of index values as currently exist but list them in a different order. Index values can be repeated multiple times in a list. This will duplicate both the metadata string as well as the data values for the copy or QC.

To delete all copies or QCs specify -1 as the first (only) entry in the new index list.

```
edit_qcs = .true.,
new_qc_index = -1,
```

To add copies or QCs, use 0 as the index value.

```
edit_copies = .true.,
new_copy_index = 1, 2, 0, 81, 82, 0
new_copy_data = 3.0, 8.0,

edit_qcs = .true.,
new_qc_index = 2, 1, 3, 0,
new_qc_data = 1.0,
```

This will insert 2 new copies in each observation and give them values of 3.0 and 8.0 in all observations. There is no way to insert a different value on a per-obs basis. This example will also reorder the 3 existing QC values and then add 1 new QC value of 1 in all observations. The 'edit_copy_metadata' and 'edit_qc_metadata' flags with the 'new_copy_metadata' and 'new_qc_metadata' lists can be used to set the metadata names of the new copies and QCs.

```
edit_copies = .true.,
new_copy_index = 1, 0, 2, 0,
new_copy_data = 3.0, 8.0,
edit_copy_metadata = .true.,
new_copy_metadata = 'observation', 'new copy 1',
                    'truth',       'new copy 2',

edit_qcs = .true.,
new_qc_index = 0, 2,
new_qc_data = 0.0,
edit_qc_metadata = .true.,
new_qc_metadata = 'dummy QC', 'DART QC',
```

To remove an existing QC value and add a QC value of 0 for all observations, run with:

```
edit_qcs = .true.,
new_qc_index = 0,
new_qc_data = 0.0,
edit_qc_metadata = .true.,
new_qc_metadata = 'dummy QC',
```

to add a constant QC of 0 for all observations, with a metadata label of 'dummy QC'.

It would be useful to allow copies or QCs from one file to be combined, obs by obs, with those from another file. However, it isn't easy to figure out how to ensure the observations in multiple files are in exactly the same order so data from the same obs are being combined. Also how to specify what should be combined is a bit complicated. So this functionality is NOT available in this tool.

Printing only

Note that you can set all the other options and then set print true, and it will do all the work and then just print out how many of each obs type would have been created. It is an easy way to preview what your choices would do without waiting to write an output file. It only prints the type breakdown for output file, but does print a running total of how many obs are being kept from each input file. For example:

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101',
   print_only         =  .true.,
/
```

Subset by observation or QC value

You can specify a min, max data value and/or min, max qc value, and only those within the range will be kept. There is no exclude option. For the data value, you must also specify an observation type since different types have different units and valid ranges. For example:

```
# keep only observations with a DART QC of 0:
   qc_metadata        = 'Dart quality control',
   min_qc             = 0,
   max_qc             = 0,

# keep only radiosonde temp obs between 250 and 300 K:
   copy_metadata      = 'NCEP BUFR observation',
   copy_type          = 'RADIOSONDE_TEMPERATURE',
   min_copy           = 250.0,
   max_copy           = 300.0,
```

Precomputed Forward Operator Values

Precomputed Forward Operator Values are the result of an external program that computes the expected observation values from an ensemble of model states and includes these values as part of the observation metadata (see, for example, the `GSI2DART` observation converter). By default, any observation with precomputed forward operator (FO) values will have those values simply pass through `obs_sequence_tool` just like any other piece of metadata. If the precomputed forward operator values for any or all observation types are not wanted, it is possible to remove the precomputed values and retain the rest of the observation.

Note: observations resulting from `perfect_model_obs` are **not** precomputed forward observation values!

```
# keep all precomputed values from all observations with precomputed values (the␣
↪default):
   remove_precomputed_FO_values = ''

# remove all precomputed values from all observations with precomputed values:
   remove_precomputed_FO_values = 'ALL'

# remove all precomputed values for specific observation types (case does not matter):
# The observations themselves will still be present in the output, given no other
# subsetting processing.
   remove_precomputed_FO_values = 'RADIOSONDE_TEMPERATURE', 'AMDAR_U_WIND_COMPONENT'
```

6.184.4 Discussion

DART observation sequence files are lists of individual observations, each with a type, a time, one or more values (called copies), zero or more quality control flags, a location, and an error estimate. Regardless of the physical order of the observations in the file, they are always processed in increasing time order, using a simple linked list mechanism. This tool reads in one or more input observation sequence files, and creates a single output observation sequence file with all observations sorted into a single, monotonically increasing time ordered output file.

DART observation sequence files contain a header with the total observation count and a table of contents of observation types. The output file from this tool culls out unused observations, and only includes observation types in the table of contents which actually occur in the output file. The table of contents **does not** need to be the same across

multiple files to merge them. Each file has a self-contained numbering system for observation types. However, the `obs_sequence_tool` must be compiled with a list of observation types (defined in the `obs_def` files listed in the `preprocess` namelist) which includes all defined types across all input files. See the building section below for more details on compiling the tool.

The tool can handle observation sequence files at any stage along the processing pipeline: a template file with locations but no data, input files for an assimilation which have observation values only, or output files from an assimilation which then might include the prior and posterior mean and standard deviation, and optionally the output from the forward operator from each ensemble member. In all of these cases, the format of each individual observation is the same. It has zero or more *copies*, which is where the observation value and the means, forward operators, etc are stored. Each observation also has zero or more quality control values, *qc*, which can be associated with the incoming data quality, or can be added by the DART software to indicate how the assimilation processed this observation. Each of the copies and qc entries has an single associated character label at the start of the observation sequence file which describes what each entry is, called the *metadata*.

For multiple observation sequence files to be merged they must have the same number of *copies* and *qc* values, and all associated *metadata* must be identical. To merge multiple files where the numbers do not match exactly, the tool can be used on the individual files to rename, subset, and reorder the *copies* and/or *qc* first, and then the resulting files are mergeable. To merge multiple files where the metadata strings do not match, but the data copy or qc values are indeed the same things, there are options to rename the metadata strings. **This option should be used with care. If the copies or qc values in different files are not really the same, the tool will go ahead and merge them but the resulting file will be very wrong.**

The tool offers an additional option for specifying a list of input files. The user creates an ASCII file by any desired method (e.g. ls > file, editor), with one filename per line. The names on each line in the file should not have any delimiters, e.g. no single or double quotes at the start or end of the filename. They specify this file with the `filename_seq_list` namelist item, and the tool opens the list file and processes each input file in turn. The namelist item `num_input_files` is now DEPRECATED and is ignored. The number of input files is computed from either the explicit list in `filename_seq`, or the contents of the `filename_seq_list` file.

Time is stored inside of DART as a day number and number of seconds, which is the same no matter which calendar is being used. But many real-world observations use the Gregorian calendar for converting between number of days and an actual date. If the `gregorian_cal` namelist item is set to `.TRUE.` then any times will be printed out to the log file will be both in day/seconds and calendar date. If the observation times are not using the Gregorian calendar, then set this value to `.FALSE.` and only days/seconds will be printed.

The most common use of this tool is to process a set of input files into a single output file, or to take one input file and extract a subset of observations into a smaller file. The examples section below outlines several common scenerios.

The tool now also allows the number of copies to be changed, but only to select subsets or reorder them. It is not yet possible to merge copies or QCs from observations in different files into a single observation with more copies.

Observations can also be selected by a given range of quality control values or data values.

Observations can be restricted to a given bounding box, either in latitude and longitude (in the horizontal only), or if the observations have 1D locations, then a single value for min_box and max_box can be specified to restrict the observations to a subset of the space.

6.184.5 Faq

Can i merge files where the observation types are different?

Yes. The numbering in the table of contents at the top of each file is only local to that file. All processing of types is done with the string name, not the numbers. Neither the set of obs types, nor the observation numbers need to match across files.

I get an error about unknown observation types

Look at the `&preprocess_nml` namelist in the input.nml file in the directory where your tool was built. It must have all the observation types you need to handle listed in the `input_files` item.

Can i list more files than necessary in my input file list?

Sure. It will take slightly longer to run, in that the tool must open the file and check the times and observation types. But it is not an error to list files where no observations will be copied to the output file. It is a common task to list a set of observation files and then set the first and last observation times, run the tool to select a shorter time period, then change the first and last times and run again with the same list of files.

6.184.6 Building

Most `$DART/models/*/work` directories will build the tool along with other executable programs. It is also possible to build the tool in the `$DART/observations/utilities` directory. The `preprocess` program must be built and run first, to define what set of observation types will be supported. See the *PROGRAM preprocess* for more details on how to define the list and run it. The combined list of all observation types which will be encountered over all input files must be in the preprocess input list. The other important choice when building the tool is to include a compatible locations module. For the low-order models, the `oned` module should be used; for real-world observations, the `threed_sphere` module should be used.

6.184.7 Modules used

```
types_mod
utilities_mod
time_manager_mod
obs_def_mod
obs_sequence_mod
```

6.184.8 Files

- `input.nml`
- The input files specified in the `filename_seq` namelist variable, or inside the file named in `filename_seq_list`.
- The output file specified in the `filename_out` namelist variable.

6.184.9 References

- none

6.185 PROGRAM integrate_model

6.185.1 Overview

Generic main program which can be compiled with a model-specific model_mod.f90 file. The model must provide an adv_1step() subroutine which advances one copy of the model forward in time.

The executable built by this program can be used by the serial program perfect_model_obs, or either the serial or parallel version of the filter program. This program is called by the default script in the template directory called advance_model.csh, and is selected by setting the corresponding "async = " namelist setting to 2.

This program only advances a single ensemble member per execution and is expected to be run as a serial program. It can be compiled with the MPI wrappers and called with mpirun with more than 1 task, however, it will only call the model advance subroutine from a single task (task 0). This can be useful in testing various scripting options using simpler and smaller models in preparation for running a larger parallel model.

6.185.2 Namelist

There is no namelist for this program.

6.185.3 Modules used

```
types_mod
time_manager_mod
utilities_mod
assim_model_mod
obs_model_mod
ensemble_manager_mod
mpi_utilities_mod
```

6.185.4 Files

- inputfile (temp_ic)
- outputfile (temp_ud)

6.185.5 References

- none

6.186 PROGRAM `obs_diag` (for 1D observations)

6.186.1 Overview/usage

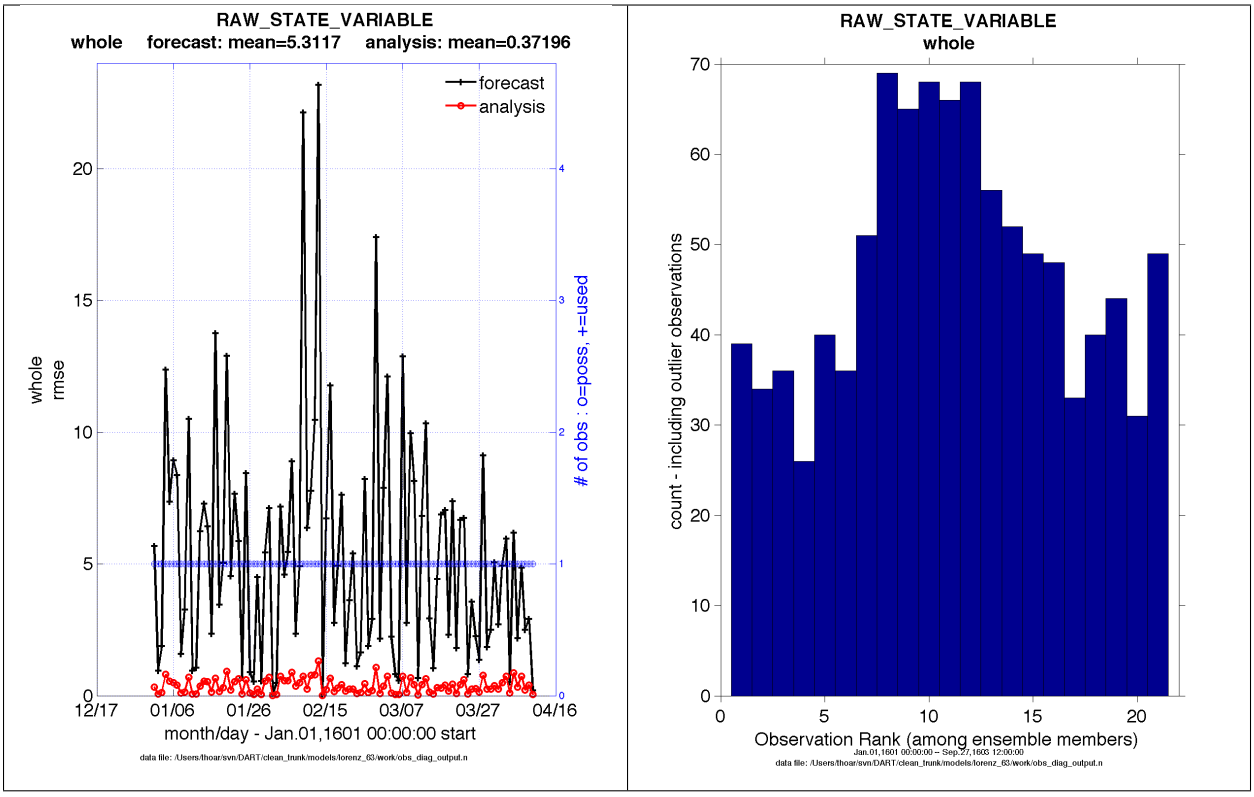
Main program for observation-space diagnostics for the models with 1D locations. 18 quantities are calculated for each region for each temporal bin specified by user input. The result of the code is a netCDF file that contains the 18 quantities of the prior (aka 'guess') and posterior (aka 'analysis') estimates as a function of time and region as well as all the metadata to create meaningful figures. **The 1D version of `obs_diag` has defaults that automatically set the first and last bin center based on the first and last observation time in the set of observations being processed.** This is different behavior than the 3D versions.

Each `obs_seq.final` file contains an observation sequence that has multiple 'copies' of the observation. One copy is the actual observation, another copy is the prior ensemble mean estimate of the observation, one is the spread of the prior ensemble estimate, one may be the prior estimate from ensemble member 1, ... etc. The only observations for the 1D models are generally the result of a 'perfect model' experiment, so there is an additional copy called the 'truth' - the noise-free expected observation given the true model state. Since this copy does not, in general, exist for the high-order models, all comparisons are made with the copy labelled 'observation'. There is also a namelist variable (`use_zero_error_obs`) to compare against the 'truth' instead; the observation error variance is then automatically set to zero.

Each ensemble member applies a forward observation operator to the state to compute the "expected" value of an observation. Please note: the forward observation operator is applied **AFTER** any prior inflation has taken place! Similarly, the forward observation operator is applied AFTER any posterior inflation. This has always been the case. For a detailed look at the relationship between the observation operators and inflation, please look at the Detailed Program Execution Flow section of *PROGRAM filter*.

Given multiple estimates of the observation, several quantities can be calculated. It is possible to compute the expected observations from the state vector before assimilating (the "guess", "forecast", or "prior") or after the assimilation (the "analysis", or "posterior").

Even with `input.nml:filter_nml:num_output_obs_members` set to `0`; the full [prior,posterior] ensemble mean and [prior,posterior] ensemble spread are preserved in the `obs_seq.final` file. Consequently, the ensemble means and spreads are used to calculate the diagnostics. If the `input.nml:filter_nml:num_output_obs_members` is set to `80` (for example); the first 80 ensemble members prior and posterior "expected" values of the observation are also included. In this case, the `obs_seq.final` file contains enough information to calculate a rank histograms, verify forecasts, etc. The ensemble means are still used for many other calculations.



Since this program is fundamentally interested in the response as a function of region, there are three versions of this program; one for each of the `oned, threed_sphere, or threed_cartesian` location modules (`location_mod.f90`). It did not make sense to ask the `lorenz_96` model what part of North America you'd like to investigate or how you would like to bin in the vertical. The low-order models write out similar netCDF files and the Matlab scripts have been updated accordingly. The oned observations have locations conceptualized as being on a unit circle, so only the namelist input variables pertaining to longitude are used.

`obs_diag` is designed to explore the effect of the assimilation in two ways; 1) as a function of time for a particular variable (this is the figure on the left), and sometimes 2) in terms of a rank histogram - "Where does the actual observation rank relative to the rest of the ensemble?" (figure on the right). The figures were created by Matlab® scripts that query the `obs_diag_output.nc` file: *DART/diagnostics/matlab/*plot_evolution.m and plot_rank_histogram.m. Both of these takes as input a file name and a 'quantity' to plot ('rmse','spread','totalspread', ...) and exhaustively plots the quantity (for every variable, every region) in a single matlab figure window - and creates a series of .ps files with multiple pages for each of the figures. The directory gets cluttered with them.

The observation sequence files contain only the time of the observation, nothing of the assimilation interval, etc. - so it requires user guidance to declare what sort of temporal binning for the temporal evolution plots. I do a 'bunch' of arithmetic on the namelist times to convert them to a series of temporal bin edges that are used when traversing the observation sequence. The actual algorithm is that the user input for the start date and bin width set up a sequence that ends in one of two ways ... the last time is reached or the number of bins has been reached. **NOTE:** for the purpose of interpretability, the 1D `obs_diag` routines saves 'dates' as GREGORIAN dates despite the fact these systems have no concept of a calendar.

`obs_diag` reads `obs_seq.final` files and calculates the following quantities (in no particular order) for an arbitrary number of regions and levels. `obs_diag` creates a netCDF file called `obs_diag_output.nc`. It is necessary to query the `CopyMetaData` variable to determine the storage order (i.e. "which copy is what?") if you want to use your own plotting routines.

ncdump -f F -v CopyMetaData obs_diag_output.nc

Nposs	The number of observations available to be assimilated.
Nused	The number of observations that were assimilated.
rmse	The root-mean-squared error (the horizontal wind components are also used to calculate the vector wind velocity and its RMS error).
bias	The simple sum of forecast - observation. The bias of the horizontal wind speed (not velocity) is also computed.
spread	The standard deviation of the univariate obs. DART does not exploit the bivariate nature of U,V winds and so the spread of the horizontal wind is defined as the sum of the spreads of the U and V components.
totalspread	The total standard deviation of the estimate. We pool the ensemble variance of the observation plus the observation error variance and take the square root.
NbadDARTQC	the number of observations that had a DART QC value (> 1 for a prior, > 3 for a posterior)
observation	the mean of the observation values
ens_mean	the ensemble mean of the model estimates of the observation values
N_trusted	the number of implicitly trusted observations, regardless of DART QC
N_DARTqc_0	number of observations that had a DART QC value of 0
N_DARTqc_1	number of observations that had a DART QC value of 1
N_DARTqc_2	number of observations that had a DART QC value of 2
N_DARTqc_3	number of observations that had a DART QC value of 3
N_DARTqc_4	number of observations that had a DART QC value of 4
N_DARTqc_5	number of observations that had a DART QC value of 5
N_DARTqc_6	number of observations that had a DART QC value of 6
N_DARTqc_7	number of observations that had a DART QC value of 7
N_DARTqc_8	number of observations that had a DART QC value of 8

The DART QC flag is intended to provide information about whether the observation was assimilated, evaluated only, whether the assimilation resulted in a 'good' observation, etc. DART QC values **<2** indicate the prior and posteriors are OK. DART QC values **>3** were **not** assimilated or evaluated. Here is the table that should explain things more fully:

DART QC flag value	meaning
0	observation assimilated
1	observation evaluated only (because of namelist settings)
2	assimilated, but the posterior forward operator failed
3	evaluated only, but the posterior forward operator failed
4	prior forward operator failed
5	not used because observation type not listed in namelist
6	rejected because incoming observation QC too large
7	rejected because of a failed outlier threshold test
8	vertical conversion failed
9+	reserved for future use

6.186.2 What is new in the Manhattan release

1. Support for DART QC = 8 (failed vertical conversion). This is provided simply to make the netCDF files as consistent as needed for plotting purposes.
2. Simplified input file specification.
3. Some of the internal variable names have been changed to make it easier to distinguish between variances and standard deviations.

6.186.3 What is new in the Lanai release

`obs_diag` has several improvements:

1. Support for 'trusted' observations. Trusted observation types may be specified in the namelist and all observations of that type will be counted in the statistics despite the DART QC code (as long as the forward observation operator succeeds). See namelist variable `trusted_obs`.
2. Support for 'true' observations (i.e. from an OSSE). If the 'truth' copy of an observation is desired for comparison (instead of the default copy) the observation error variance is set to 0.0 and the statistics are calculated relative to the 'truth' copy (as opposed to the normal 'noisy' or 'observation' copy). See namelist variable `use_zero_error_obs`.
3. discontinued the use of `rat_cri` and `input_qc_threshold` namelist variables. Their functionality was replaced by the DART QC mechanism long ago.
4. The creation of the rank histogram (if possible) is now namelist-controlled by namelist variable `create_rank_histogram`.

6.186.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_diag_nml
   obs_sequence_name     = ''
   obs_sequence_list     = ''
   bin_width_days        = -1
   bin_width_seconds     = -1
   init_skip_days        = 0
   init_skip_seconds     = 0
   max_num_bins          = 9999
   Nregions              = 3
   lonlim1               = 0.0, 0.0, 0.5
   lonlim2               = 1.0, 0.5, 1.0
   reg_names             = 'whole', 'yin', 'yang'
   trusted_obs           = 'null'
   use_zero_error_obs    = .false.
   create_rank_histogram = .true.
   outliers_in_histogram = .true.
   verbose               = .false.
   /
```

The allowable ranges for the region boundaries are: lon [0.0, 1.0). The 1D locations are conceived as the distance around a unit sphere. An observation with a location exactly ON a region boundary cannot 'count' for both regions. The logic used to resolve this is:

if((lon lon1) .and. (lon < lon2)) keeper = .true.

Consequently, if you want to include an observation precisely AT 1.0, (for example), you need to specify something a little larger than 1.0.

You can only specify **either** `obs_sequence_name` **or** `obs_sequence_list` – not both. One of them has to be an empty string … i.e. `''`.

Item	Type	Description
obs_sequence_name	character(len=256), dimension(100)	An array of names of observation sequence files. These may be relative or absolute filenames. If this is set, `obs_sequence_list` must be set to ' ' (empty string).
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, `obs_sequence_name` must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program. If this is set, `obs_sequence_name` must be set to ' ' (empty string).
bin_width_days, bin_width_seconds	integer	Specifies the width of the analysis window. All observations within a window centered at the observation time +/- bin_width_[days,seconds] is used. If both values are 0, half the separation between observation times as defined in the observation sequence file is used for the bin width (i.e. all observations used).
init_skip_days, init_skip_seconds	integer	Ignore all observations before this time. This allows one to skip the 'spinup' or stabilization period of an assimilation.
max_num_bins	integer	This provides a way to restrict the number of temporal bins. If `max_num_bins` is set to '10', only 10 timesteps will be output, provided there are that many.
Nregions	integer	The number of regions for the unit circle for which you'd like observation-space diagnostics. If 3 is not enough increase `MaxRegions` in `obs_diag.f90` and recompile.
lonlim1	real(r8) array of length(Nregions)	starting value of coordinates defining 'regions'. A value of -1 indicates the start of 'no region'.
lonlim2	real(r8) array of length(Nregions)	ending value of coordinates defining 'regions'. A value of -1 indicates the end of 'no region'.
reg_names	character(len=6), dimension(Nregions)	Array of names for each of the regions. The default example has the unit circle as a whole and divided into two equal parts, so there are only three regions.
trusted_obs	character(len=32), dimension(5)	Array of names for observation TYPES that will be included in the statistics if at all possible (i.e. the forward observation operator succeeds). For 1D observations the only choices in the code as distributed are 'RAW_STATE_VARIABLE' and/or 'RAW_STATE_1D_INTEGRAL'. (Additional 1D observation types can be added by the user.)
use_zero_error_obs	logical	if `.true.`, the observation copy used for the statistics calculations will be 'truth'. Only 'perfect' observations (from `perfect_model_obs`) have this copy. The observation error variance will be set to zero.
create_rank_histogram	logical	if `.true.` and there are actual ensemble estimates of the observations in the `obs_seq.final` (i.e. `filter_nml:num_output_obs_members` is larger than zero), a rank histogram will be created.
outliers_in_histogram	logical	if `.true.` the observations that have been rejected by the outlier threshhold mechanism will be *included* in the calculation of the rank histogram.
verbose	logical	switch controlling amount of run-time output.

6.186.5 Modules directly used

```
types_mod
obs_sequence_mod
obs_def_mod
obs_kind_mod
location_mod
time_manager_mod
utilities_mod
sort_mod
random_seq_mod
```

6.186.6 Modules indirectly used

```
assim_model_mod
cov_cutoff_mod
model_mod
null_mpi_utilities_mod
```

6.186.7 Files

- `input.nml` is used for `obs_diag_nml`
- `obs_diag_output.nc` is the netCDF output file
- `dart_log.out` list directed output from the obs_diag.
- `LargeInnov.txt` contains the distance ratio histogram – useful for estimating the distribution of the magnitudes of the innovations.

Discussion of obs_diag_output.nc

Every observation type encountered in the observation sequence file is tracked separately, and aggregated into temporal and spatial bins. There are two main efforts to this program. One is to track the temporal evolution of any of the quantities available in the netCDF file for any possible observation type:

ncdump -v CopyMetaData,ObservationTypes obs_diag_output.nc

The other is to explore the vertical profile of a particular observation kind. By default, each observation kind has a 'guess/prior' value and an 'analysis/posterior' value - which shed some insight into the innovations.

Temporal evolution

The `obs_diag_output.nc` output file has all the metadata I could think of, as well as separate variables for every observation type in the observation sequence file. Furthermore, there is a separate variable for the 'guess/prior' and 'analysis/posterior' estimate of the observation. To distinguish between the two, a suffix is appended to the variable name. An example seems appropriate:

```
...
char CopyMetaData(copy, stringlength) ;
        CopyMetaData:long_name = "quantity names" ;
```

(continues on next page)

(continued from previous page)

```
...
int rank_bins(rank_bins) ;
        rank_bins:long_name = "rank histogram bins" ;
        rank_bins:comment = "position of the observation among the sorted noisy ensemble␣
↪members" ;
float RAW_STATE_VARIABLE_guess(time, copy, region) ;
        RAW_STATE_VARIABLE_guess:_FillValue = -888888.f ;
        RAW_STATE_VARIABLE_guess:missing_value = -888888.f ;
float RAW_STATE_VARIABLE_analy(time, copy, region) ;
        RAW_STATE_VARIABLE_analy:_FillValue = -888888.f ;
        RAW_STATE_VARIABLE_analy:missing_value = -888888.f ;
...
```

Rank histograms

If it is possible to calculate a rank histogram, there will also be :

```
 ...
int RAW_STATE_VARIABLE_guess_RankHist(time, rank_bins, region) ;
 ...
```

as well as some global attributes. The attributes reflect the namelist settings and can be used by plotting routines to provide additional annotation for the histogram.

```
:DART_QCs_in_histogram = 0, 1, 2, 3, 7 ;
:outliers_in_histogram = "TRUE" ;
```

Please note:
netCDF restricts variable names to 40 characters, so '_Rank_Hist' may be truncated.

6.186.8 References

1. none

6.186.9 Private components

N/A

6.187 PROGRAM obs_diag (for observations that use the threed_cartesian location module)

6.187.1 Overview

Main program for evaluating filter performance in observation space. Primarily, the prior or posterior ensemble mean (and spread) are compared to the observation and several quantities are calculated. These quantities are then saved in a netCDF file that has all the metadata to create meaningful figures.

Each `obs_seq.final` file contains an observation sequence that has multiple 'copies' of the observation. One copy is the actual observation, another copy is the prior ensemble mean estimate of the observation, one is the spread of the prior ensemble estimate, one may be the prior estimate from ensemble member 1, ... etc. If the original observation sequence is the result of a 'perfect model' experiment, there is an additional copy called the 'truth' - the noise-free expected observation given the true model state. Since this copy does not, in general, exist for the high-order models, all comparisons are made with the copy labelled 'observation'. There is also a namelist variable (`use_zero_error_obs`) to compare against the 'truth' instead; the observation error variance is then automatically set to zero.

Each ensemble member applies a forward observation operator to the state to compute the "expected" value of an observation. Please note: the forward observation operator is applied **AFTER** any prior inflation has taken place! Similarly, the forward observation operator is applied AFTER any posterior inflation. This has always been the case. For a detailed look at the relationship between the observation operators and inflation, please look at the Detailed Program Execution Flow section of *PROGRAM filter*.

Given multiple estimates of the observation, several quantities can be calculated. It is possible to compute the expected observations from the state vector before assimilating (the "guess", "forecast", or "prior") or after the assimilation (the "analysis", or "posterior").

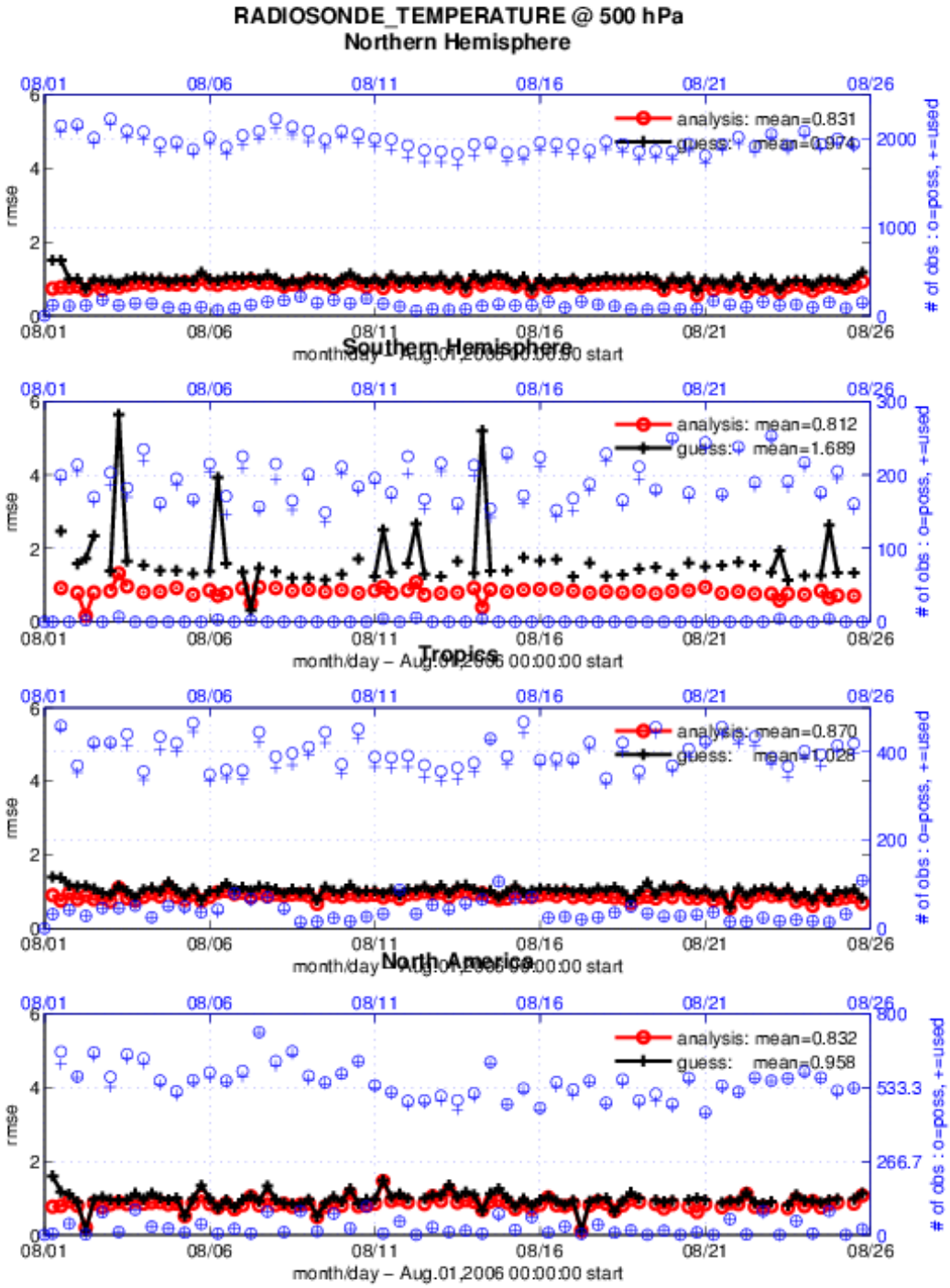
Even with `input.nml:filter_nml:num_output_obs_members` set to `0`; the full [prior,posterior] ensemble mean and [prior,posterior] ensemble spread are preserved in the `obs_seq.final` file. Consequently, the ensemble means and spreads are used to calculate the diagnostics. If the `input.nml:filter_nml:num_output_obs_members` is set to `80` (for example); the first 80 ensemble members prior and posterior "expected" values of the observation are also included. In this case, the `obs_seq.final` file contains enough information to calculate a rank histograms, verify forecasts, etc. The ensemble means are still used for many other calculations.

Since this program is fundamentally interested in the response as a function of region, there are three versions of this program; one for each of the `oned, threed_sphere, or threed_cartesian` location modules (`location_mod.f90`). It did not make sense to ask the `lorenz_96` model what part of North America you'd like to investigate or how you would like to bin in the vertical. The low-order models write out similar netCDF files and the Matlab scripts have been updated accordingly. The oned observations have locations conceptualized as being on a unit circle, so only the namelist input variables pertaining to longitude are used.

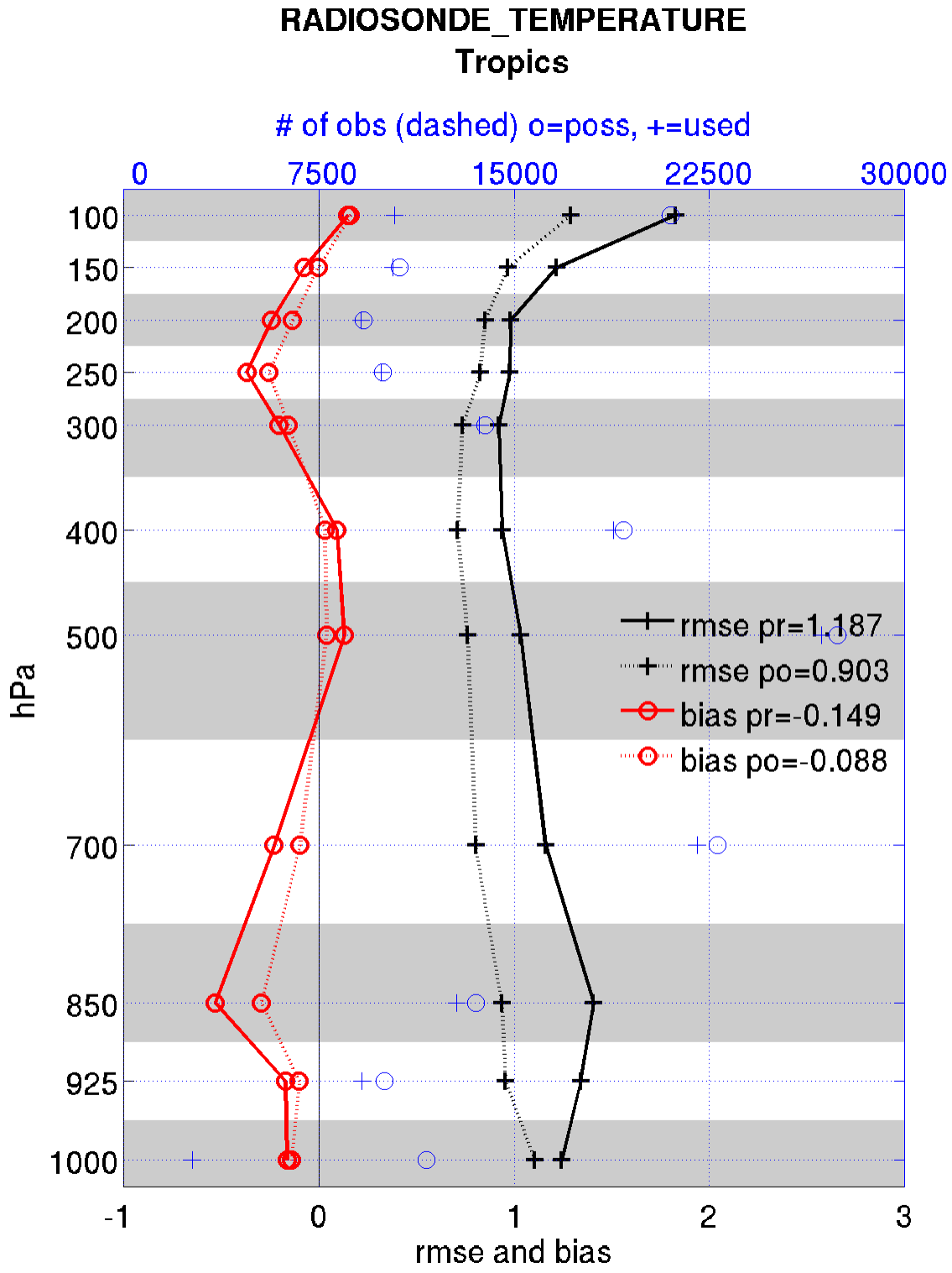
Identity observations (only possible from "perfect model experiments") are already explored with state-space diagnostics, so `obs_diag` simply skips them.

6.187.2 `obs_diag` is designed to explore the effect of the assimilation in three ways:

1) as a function of time for a particular variable and level

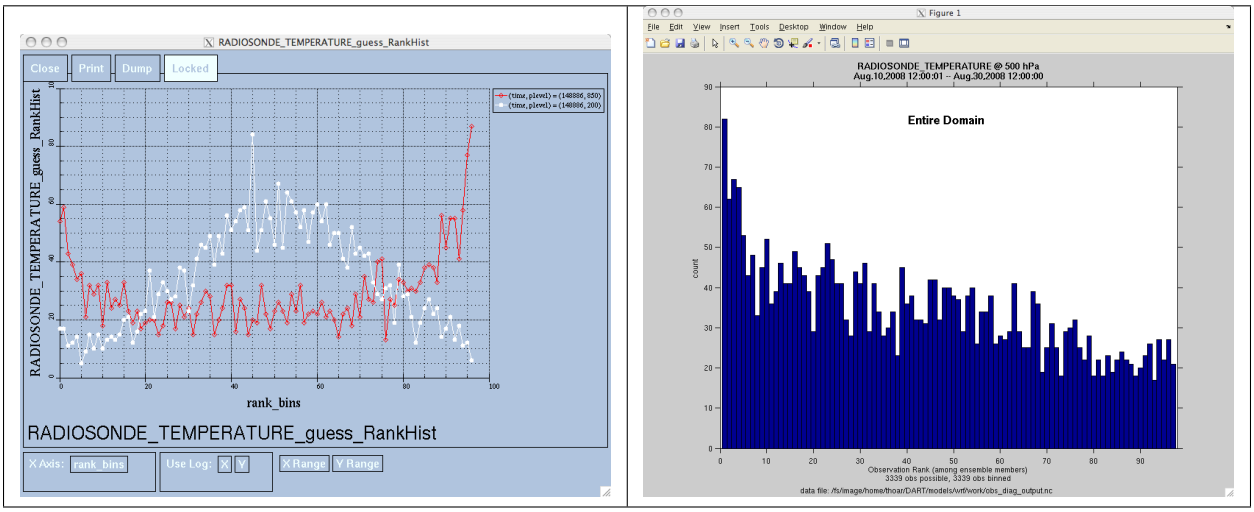


2) as a time-averaged vertical profile



3) and in terms of a rank histogram -

"Where does the actual observation rank relative to the rest of the ensemble?"



The figures in sections 1 and 2 were created by Matlab® scripts that query the `obs_diag_output.nc` file: `DART/diagnostics/matlab/plot_evolution.m` and `plot_profile.m`. Both of these takes as input a file name and a 'quantity' to plot ('rmse','spread','totalspread', ...) and exhaustively plots the quantity (for every variable, every level, every region) in a single matlab figure window - and creates a series of .ps files with multiple pages for each of the figures. The directory gets cluttered with them. The rank histogram information in `obs_diag_output.nc` can easily be plotted with ncview (left), a free third-party piece of software or with `plot_rank_histogram.m` (right). See the *Rank histograms* section for more information and links to instructions.

`obs_diag` can be configured to compare the ensemble estimates against the 'observation' copy or the 'truth' copy based on the setting of the `use_zero_error_obs` namelist variable.

The observation sequence files contain only the time of the observation, nothing of the assimilation interval, etc. - so it requires user guidance to declare what sort of temporal binning for the temporal evolution plots. I do a 'bunch' of arithmetic on the namelist times to convert them to a series of temporal bin edges that are used when traversing the observation sequence. The actual algorithm is that the user input for the start date and bin width set up a sequence that ends in one of two ways ... the last time is reached or the number of bins has been reached.

`obs_diag` reads `obs_seq.final` files and calculates the following quantities (in no particular order) for an arbitrary number of regions and levels. `obs_diag` creates a netCDF file called `obs_diag_output.nc`. It is necessary to query the `CopyMetaData` variable to determine the storage order (i.e. "which copy is what?") if you want to use your own plotting routines.

ncdump -f F -v CopyMetaData obs_diag_output.nc

Nposs	The number of observations available to be assimilated.
Nused	The number of observations that were assimilated.
NbadUV	the number of velocity observations that had a matching component that was not assimilated;
NbadLV	the number of observations that were above or below the highest or lowest model level, respectively;
rmse	The root-mean-squared error (the horizontal wind components are also used to calculate the vector wind velocity and its RMS error).
bias	The simple sum of forecast - observation. The bias of the horizontal wind speed (not velocity) is also computed.
spread	The standard deviation of the univariate obs. DART does not exploit the bivariate nature of U,V winds and so the spread of the horizontal wind is defined as the sum of the spreads of the U and V components.
totalspread	The total standard deviation of the estimate. We pool the ensemble variance of the observation plus the observation error variance and take the square root.
NbadDARTQC	the number of observations that had a DART QC value (> 1 for a prior, > 3 for a posterior)
observation	the mean of the observation values
ens_mean	the ensemble mean of the model estimates of the observation values
N_trusted	the number of implicitly trusted observations, regardless of DART QC
N_DARTqc_0	number of observations that had a DART QC value of 0
N_DARTqc_1	number of observations that had a DART QC value of 1
N_DARTqc_2	number of observations that had a DART QC value of 2
N_DARTqc_3	number of observations that had a DART QC value of 3
N_DARTqc_4	number of observations that had a DART QC value of 4
N_DARTqc_5	number of observations that had a DART QC value of 5
N_DARTqc_6	number of observations that had a DART QC value of 6
N_DARTqc_7	number of observations that had a DART QC value of 7
N_DARTqc_8	number of observations that had a DART QC value of 8

The temporal evolution of the above quantities for every observation type (RADIOSONDE_U_WIND_COMPONENT, AIRCRAFT_SPECIFIC_HUMIDITY, ...) is recorded in the output netCDF file - `obs_diag_output.nc`. This netCDF file can then be loaded and displayed using the Matlab® scripts in `.../DART/diagnostics/matlab`. (which may depend on functions in `.../DART/matlab`). The temporal, geographic, and vertical binning are under namelist control. Temporal averages of the above quantities are also stored in the netCDF file. Normally, it is useful to skip the 'burn-in' period - the amount of time to skip is under namelist control.

The DART QC flag is intended to provide information about whether the observation was assimilated, evaluated only, whether the assimilation resulted in a 'good' observation, etc. DART QC values **<2** indicate the prior and posteriors are OK. DART QC values **>3** were **not** assimilated or evaluated. Here is the table that should explain things more fully:

DART QC flag value	meaning
0	observation assimilated
1	observation evaluated only (because of namelist settings)
2	assimilated, but the posterior forward operator failed
3	evaluated only, but the posterior forward operator failed
4	prior forward operator failed
5	not used because observation type not listed in namelist
6	rejected because incoming observation QC too large
7	rejected because of a failed outlier threshold test
8	vertical conversion failed
9+	reserved for future use

6.187.3 What is new in the Manhattan release

1. Support for DART QC = 8 (failed vertical conversion).
2. Simplified input file specification.
3. Replace namelist integer variable `debug` with logical variable `verbose` to control amount of run-time output.
4. Removed `rat_cri` and `input_qc_threshold` from the namelists. They had been deprecated for quite some time.
5. Some of the internal variable names have been changed to make it easier to distinguish between variances and standard deviations.

6.187.4 What is new in the Lanai release

`obs_diag` has several improvements:

1. Improved vertical specification. Namelist variables `[h,p,m]level_edges` allow fine-grained control over the vertical binning. It is not allowed to specify both the edges and midpoints for the vertical bins.
2. Improved error-checking for input specification, particularly the vertical bins. Repeated values are squeezed out.
3. Support for 'trusted' observations. Trusted observation types may be specified in the namelist and all observations of that type will be counted in the statistics despite the DART QC code (as long as the forward observation operator succeeds). See namelist variable `trusted_obs`. For more details, see the section on Trusted observations.
4. Support for 'true' observations (i.e. from an OSSE). If the 'truth' copy of an observation is desired for comparison (instead of the default copy) the observation error variance is set to 0.0 and the statistics are calculated relative to the 'truth' copy (as opposed to the normal 'noisy' or 'observation' copy). See namelist variable `use_zero_error_obs`.
5. discontinued the use of `rat_cri` and `input_qc_threshold` namelist variables. Their functionality was replaced by the DART QC mechanism long ago.
6. The creation of the rank histogram (if possible) is now namelist-controlled by namelist variable `create_rank_histogram`.

6.187.5 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_diag_nml
   obs_sequence_name     = ''
   obs_sequence_list     = ''
   first_bin_center      =  2003, 1, 1, 0, 0, 0
   last_bin_center       =  2003, 1, 2, 0, 0, 0
   bin_separation        =     0, 0, 0, 6, 0, 0
   bin_width             =     0, 0, 0, 6, 0, 0
   time_to_skip          =     0, 0, 0, 6, 0, 0
   max_num_bins          = 1000
   hlevel                = -888888.0
   hlevel_edges          = -888888.0
   Nregions              = 0
```

(continues on next page)

(continued from previous page)

```
   xlim1                = -888888.0
   xlim2                = -888888.0
   ylim1                = -888888.0
   ylim2                = -888888.0
   reg_names            = 'null'
   trusted_obs          = 'null'
   create_rank_histogram = .true.
   outliers_in_histogram = .false.
   use_zero_error_obs   = .false.
   verbose              = .false.
   /
```

The date-time integer arrays in this namelist have the form (YYYY, MM, DY, HR, MIN, SEC).
The allowable ranges for the region boundaries are: latitude [-90.,90], longitude [0.,Inf.]

You can only specify **either** `obs_sequence_name` **or** `obs_sequence_list` – not both. One of them has to be an empty string … i.e. `''`.

Item	Type	Description
obs_sequence_name	character(len=256), dimension(100)	An array of names of observation sequence files. These may be relative or absolute filenames. If this is set, `obs_sequence_list` must be set to ' ' (empty string).
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, `obs_sequence_name` must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program. If this is set, `obs_sequence_name` must be set to ' ' (empty string).
first_bin_center	integer, dimension(6)	first timeslot of the first obs_seq.final file to process. The six integers are: year, month, day, hour, hour, minute, second, in that order. `obs_diag` has improved run-time output that reports the time and date of the first and last observations in every observation sequence file. Look for the string 'First observation date' in the logfile. If the `verbose` is 'true', it is also written to the screen.
last_bin_center	integer, dimension(6)	last timeslot of interest. (reminder: the last timeslot of day 1 is hour 0 of day 2) The six integers are: year, month, day, hour, hour, minute, second, in that order. This does not need to be exact, the values from `first_bin_center` and `bin_separation` are used to populate the time array and stop on or before the time defined by `last_bin_center`. See also `max_num_bins`.
bin_separation	integer, dimension(6)	Time between bin centers. The year and month values *must* be zero.
bin_width	Integer, dimension(6)	Time span around bin centers in which obs will be compared. The year and month values *must* be zero. Frequently, but not required to be, the same as the values for bin_separation. 0
time_to_skip	integer, dimension(6)	Time span at the beginning to skip when calculating vertical profiles of rms error and bias. The year and month values *must* be zero. Useful because it takes some time for the assimilation to settle down from the climatological spread at the start. `time_to_skip` is an amount of time AFTER the first edge of the first bin.
max_num_bins	integer	This provides an alternative way to declare the `last_bin_center`. If `max_num_bins` is set to '10', only 10 timesteps will be output - provided `last_bin_center` is set to some later date.
hlevel	real, dimension(50)	Same, but for observations that have height(m) or depth(m) as the vertical coordinate. An example of defining the midpoints is: `hlev el = 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000,`
hlevel_edges	real, dimension(51)	The edges defining the height (or depth) levels for the vertical binning. You may specify either `hlevel` or `hlevel_edges`, but not both. An example of defining the edges is: `hlevel_edges = 0, 1500, 2500, 3500, 4500, 5500, 6500,`
Nregions	integer	Number of regions of the globe for which obs space diagnostics are computed separately. Must be between [1,50]. If 50 is not enough, increase `obs_diag.f90MaxRegions` and recompile.
xlim1	real, dimension(50)	western extent of each of the regions.
xlim2	real, dimension(50)	eastern extent of each of the regions.
ylim1	real, dimension(50)	southern extent of the regions.
ylim2	real, dimension(50)	northern extent of the regions.
reg_names	character(len=129), dimension(50)	Array of names for the regions to be analyzed. Will be used for plot titles.
trusted_obs	character(len=32), dimension(50)	list of observation types that **must** participate in the calculation of the statistics, regardless of the DART QC (provided that the forward observation operator can still be applied without failure). e.g. 'RADIOSONDE_TEMPERATURE', ... For more details, see the section on Trusted observations.
use_zero_error_obs	logical	if `.true.`, the observation copy used for the statistics calculations will be 'truth'. Only 'perfect'

6.187.6 Other modules used

```
obs_sequence_mod
obs_kind_mod
obs_def_mod (and possibly other obs_def_xxx mods)
assim_model_mod
random_seq_mod
model_mod
location_mod
types_mod
time_manager_mod
utilities_mod
sort_mod
```

6.187.7 Files

- `input.nml` is used for `obs_diag_nml`
- `obs_diag_output.nc` is the netCDF output file
- `dart_log.out` list directed output from the obs_diag.
- `LargeInnov.txt` contains the distance ratio histogram – useful for estimating the distribution of the magnitudes of the innovations.

Obs_diag may require a model input file from which to get grid information, metadata, and links to modules providing the models expected observations. It all depends on what's needed by the `model_mod.f90`

Discussion of obs_diag_output.nc

Every observation type encountered in the observation sequence file is tracked separately, and aggregated into temporal and 3D spatial bins. There are two main efforts to this program. One is to track the temporal evolution of any of the quantities available in the netCDF file for any possible observation type:

ncdump -v CopyMetaData,ObservationTypes obs_diag_output.nc

The other is to explore the vertical profile of a particular observation kind. By default, each observation kind has a 'guess/prior' value and an 'analysis/posterior' value - which shed some insight into the innovations.

Temporal evolution

The `obs_diag_output.nc` output file has all the metadata I could think of, as well as separate variables for every observation type in the observation sequence file. Furthermore, there is a separate variable for the 'guess/prior' and 'analysis/posterior' estimate of the observation. To distinguish between the two, a suffix is appended to the variable name. An example seems appropriate:

```
...
char CopyMetaData(copy, stringlength) ;
        CopyMetaData:long_name = "quantity names" ;
char ObservationTypes(obstypes, stringlength) ;
        ObservationTypes:long_name = "DART observation types" ;
        ObservationTypes:comment = "table relating integer to observation type string" ;
float RADIOSONDE_U_WIND_COMPONENT_guess(time, copy, hlevel, region) ;
```

(continues on next page)

(continued from previous page)

```
        RADIOSONDE_U_WIND_COMPONENT_guess:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_guess:missing_value = -888888.f ;
float RADIOSONDE_V_WIND_COMPONENT_guess(time, copy, hlevel, region) ;
        RADIOSONDE_V_WIND_COMPONENT_guess:_FillValue = -888888.f ;
        RADIOSONDE_V_WIND_COMPONENT_guess:missing_value = -888888.f ;
...
float MARINE_SFC_ALTIMETER_guess(time, copy, surface, region) ;
        MARINE_SFC_ALTIMETER_guess:_FillValue = -888888.f ;
        MARINE_SFC_ALTIMETER_guess:missing_value = -888888.f ;
...
float RADIOSONDE_WIND_VELOCITY_guess(time, copy, hlevel, region) ;
        RADIOSONDE_WIND_VELOCITY_guess:_FillValue = -888888.f ;
        RADIOSONDE_WIND_VELOCITY_guess:missing_value = -888888.f ;
...
float RADIOSONDE_U_WIND_COMPONENT_analy(time, copy, hlevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_analy:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_analy:missing_value = -888888.f ;
float RADIOSONDE_V_WIND_COMPONENT_analy(time, copy, hlevel, region) ;
        RADIOSONDE_V_WIND_COMPONENT_analy:_FillValue = -888888.f ;
        RADIOSONDE_V_WIND_COMPONENT_analy:missing_value = -888888.f ;
...
```

There are several things to note:

1. the 'WIND_VELOCITY' component is nowhere 'near' the corresponding U,V components.
2. all of the 'guess' variables come before the matching 'analy' variables.
3. surface variables (i.e. `MARINE_SFC_ALTIMETER` have a coordinate called 'surface' as opposed to 'hlevel' for the others in this example).

Vertical profiles

Believe it or not, there are another set of netCDF variables specifically for the vertical profiles, essentially duplicating the previous variables but **without the 'time' dimension**. These are distinguished by the suffix added to the observation kind - 'VPguess' and 'VPanaly' - 'VP' for Vertical Profile.

```
...
float SAT_WIND_VELOCITY_VPguess(copy, hlevel, region) ;
        SAT_WIND_VELOCITY_VPguess:_FillValue = -888888.f ;
        SAT_WIND_VELOCITY_VPguess:missing_value = -888888.f ;
...
float RADIOSONDE_U_WIND_COMPONENT_VPanaly(copy, hlevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_VPanaly:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_VPanaly:missing_value = -888888.f ;
...
```

Observations flagged as 'surface' do not participate in the vertical profiles (Because surface variables cannot exist on any other level, there's not much to plot!). Observations on the lowest level DO participate. There's a difference!

Rank histograms

If it is possible to calculate a rank histogram, there will also be :

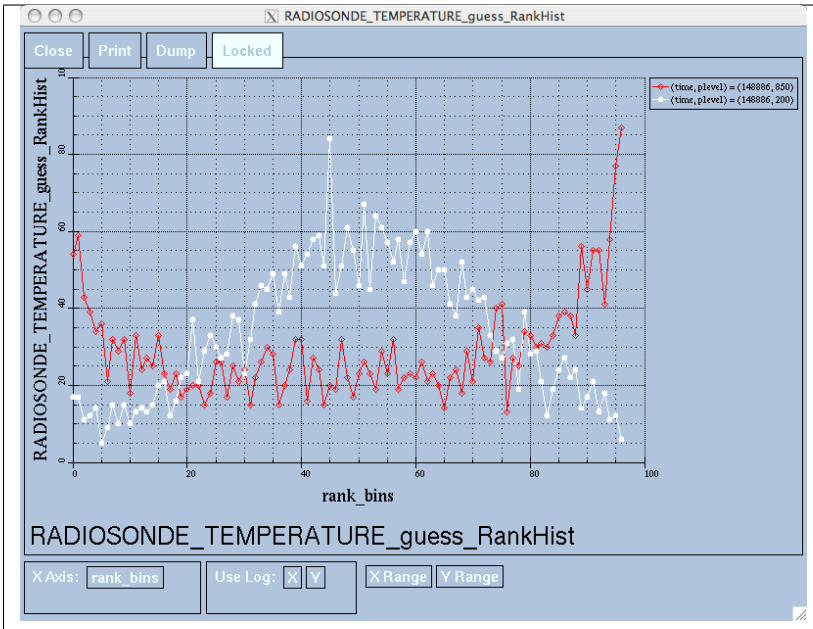
```
...
int RADIOSONDE_U_WIND_COMPONENT_guess_RankHi(time, rank_bins, hlevel, region) ;
...
int RADIOSONDE_V_WIND_COMPONENT_guess_RankHi(time, rank_bins, hlevel, region) ;
...
int MARINE_SFC_ALTIMETER_guess_RankHist(time, rank_bins, surface, region) ;
...
```

as well as some global attributes. The attributes reflect the namelist settings and can be used by plotting routines to provide additional annotation for the histogram.

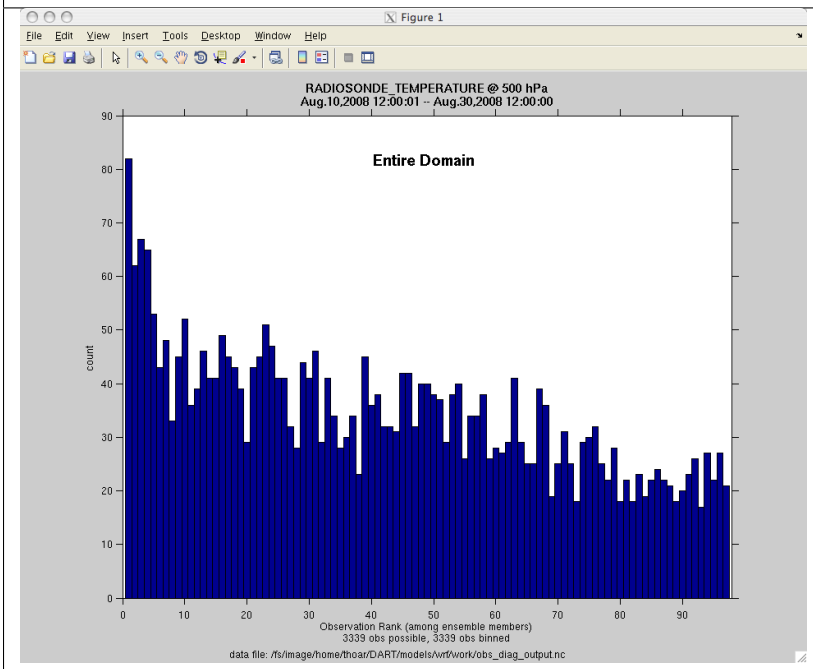
```
:DART_QCs_in_histogram = 0, 1, 2, 3, 7 ;
:outliers_in_histogram = "TRUE" ;
```

Please note:

1. netCDF restricts variable names to 40 characters, so '_Rank_Hist' may be truncated.
2. It is sufficiently vague to try to calculate a rank histogram for a velocity derived from the assimilation of U,V components such that NO rank histogram is created for velocity. A run-time log message will inform as to which variables are NOT having a rank histogram variable preserved in the `obs_diag_output.nc` file - IFF it is possible to calculate a rank histogram in the first place.



Instructions for viewing the rank histogram with ncview.



Instructions for viewing the rank histogram with Matlab.

"trusted" observation types

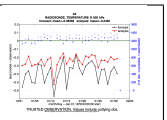
This needs to be stated up front: `obs_diag` is a post-processor; it cannot influence the assimilation. One interpretation of a TRUSTED observation is that the assimilation should **always** use the observation, even if it is far from the ensemble. At present (23 Feb 2015), the filter in DART does not forcibly assimilate any one observation and selectively assimilate the others. Still, it is useful to explore the results using a set of 'trusted type' observations, whether they were assimilated, evaluated, or rejected by the outlier threshhold. This is the important distinction. The diagnostics can be calculated differently for each *observation type*.

The normal diagnostics calculate the metrics (rmse, bias, etc.) only for the 'good' observations - those that were assimilated or evaluated. The `outlier_threshold` essentially defines what observations are considered too far from the ensemble **prior** to be useful. These observations get a DART QC of 7 and are not assimilated. The observations with a DART QC of 7 do not contribute the the metrics being calculated. Similarly, if the forward observation operator fails, these observations cannot contribute. When the operator fails, the 'expected' observation value is 'MISSING', and there is no ensemble mean or spread.

'Trusted type' observation metrics are calculated using all the observations that were assimilated or evaluated **AND** the observations that were rejected by the outlier threshhold. `obs_diag` can post-process the DART QC and calculate the metrics appropriately for **observation types** listed in the `trusted_obs` namelist variable. If there are trusted observation types specified for `obs_diag`, the `obs_diag_output.nc` has global metadata to indicate that a different set of criteria were used to calculate the metrics. The individual variables also have an extra attribute. In the following output, `input.nml:obs_diag_nml:trusted_obs` was set: `trusted_obs = 'RADIOSONDE_TEMPERATURE', 'RADIOSONDE_U_WIND_COMPONENT'`

```
  ...
        float RADIOSONDE_U_WIND_COMPONENT_guess(time, copy, hlevel, region) ;
                RADIOSONDE_U_WIND_COMPONENT_guess:_FillValue = -888888.f ;
                RADIOSONDE_U_WIND_COMPONENT_guess:missing_value = -888888.f ;
                RADIOSONDE_U_WIND_COMPONENT_guess:TRUSTED = "TRUE" ;
        float RADIOSONDE_V_WIND_COMPONENT_guess(time, copy, hlevel, region) ;
                RADIOSONDE_V_WIND_COMPONENT_guess:_FillValue = -888888.f ;
                RADIOSONDE_V_WIND_COMPONENT_guess:missing_value = -888888.f ;
  ...
// global attributes:
  ...
                :trusted_obs_01 = "RADIOSONDE_TEMPERATURE" ;
                :trusted_obs_02 = "RADIOSONDE_U_WIND_COMPONENT" ;
                :obs_seq_file_001 = "cam_obs_seq.1978-01-01-00000.final" ;
                :obs_seq_file_002 = "cam_obs_seq.1978-01-02-00000.final" ;
                :obs_seq_file_003 = "cam_obs_seq.1978-01-03-00000.final" ;
  ...
                :MARINE_SFC_ALTIMETER = 7 ;
                :LAND_SFC_ALTIMETER = 8 ;
                :RADIOSONDE_U_WIND_COMPONENT--TRUSTED = 10 ;
                :RADIOSONDE_V_WIND_COMPONENT = 11 ;
                :RADIOSONDE_TEMPERATURE--TRUSTED = 14 ;
                :RADIOSONDE_SPECIFIC_HUMIDITY = 15 ;
                :AIRCRAFT_U_WIND_COMPONENT = 21 ;
  ...
```

The Matlab scripts try to ensure that the trusted observation graphics clarify that the metrics plotted are somehow 'different' than the normal processing stream. Some text is added to indicate that the values include the outlying observations. **IMPORTANT:** The interpretation of the number of observations 'possible' and 'used' still reflects what was used **in the assimilation!** The number of observations rejected by the outlier threshhold is not explicilty plotted. To reinforce this, the text for the observation axis on all graphics has been changed to "`o=possible, *=assimilated`". In short, the distance between the number of observations possible and the number assimilated still reflects the number of observations rejected by the outlier threshhold and the number of failed forward observation operators.



There is ONE ambiguous case for trusted observations. There may be instances in which the observation fails the outlier threshhold test (which is based on the prior) and the posterior forward operator fails. DART does not have a QC that explicilty covers this case. The current logic in `obs_diag` correctly handles these cases **except** when trying to use 'trusted' observations. There is a section of code in `obs_diag` that may be enabled if you are encountering this ambiguous case. As `obs_diag` runs, a warning message is issued and a summary count is printed if the ambiguous case is encountered. What normally happens is that if that specific observation type is trusted, the posterior values include a MISSING value in the calculation which makes them inaccurate. If the block of code is enabled, the DART QC is recast as the PRIOR forward observation operator fails. This is technically incorrect, but for the case of trusted observations, it results in only calculating statistics for trusted observations that have a useful prior and posterior. **This should not be used unless you are willing to intentionally disregard 'trusted' observations that were rejected by the outlier threshhold.** Since the whole point of a trusted observation is to *include* observations potentially rejected by the outlier threshhold, you see the problem. Some people like to compare the posteriors. *THAT* can be the problem.

```
if ((qc_integer == 7) .and. (abs(posterior_mean(1) - MISSING_R8) < 1.0_r8)) then
            write(string1,*)'WARNING ambiguous case for obs index ',obsindex
            string2 = 'obs failed outlier threshhold AND posterior operator failed.'
            string3 = 'Counting as a Prior QC == 7, Posterior QC == 4.'
            if (trusted) then
! COMMENT      string3 = 'WARNING changing DART QC from 7 to 4'
! COMMENT      qc_integer = 4
            endif
            call error_handler(E_MSG,'obs_diag',string1,text2=string2,text3=string3)
            num_ambiguous = num_ambiguous + 1
         endif
```

6.187.8 Usage

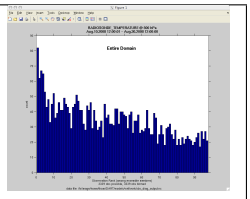
`obs_diag` is built in .../DART/models/*your_model*/work, in the same way as the other DART components.

Multiple observation sequence files

There are two ways to specify input files for `obs_diag`. You can either specify the name of a file containing a list of files (in `obs_sequence_list`), or you may specify a list of files via `obs_sequence_name`.

Example: observation sequence files spanning 30 days

In this example, we will be accumulating metrics for 30 days. The `obs_diag_output.nc` file will have exactly ONE timestep in it (so it won't be much use for the `plot_evolution` functions) - but the `plot_profile` functions and the `plot_rank_histogram` function will be used to explore the assimilation. By way of an example, we will NOT be using outlier observations in the rank histogram. Lets presume that all your `obs_seq.final` files are in alphabetically-nice directories:



```
/Exp1/Dir01/obs_seq.final
/Exp1/Dir02/obs_seq.final
/Exp1/Dir03/obs_seq.final
...
/Exp1/Dir99/obs_seq.final
```

The first step is to create a file containing the list of observation sequence files you want to use. This can be done with the unix command 'ls' with the -1 option (that's a number one) to put one file per line.

ls -1 /Exp1/Dir*/obs_seq.final > obs_file_list.txt

It is necessary to turn on the verbose option to check the first/last times that will be used for the histogram. Then, the namelist settings for 2008 07 31 12Z through 2008 08 30 12Z are:

```
&obs_diag_nml
   obs_sequence_name     = ''
   obs_sequence_list     = 'obs_file_list.txt'
   first_bin_center      =  2008, 8,15,12, 0, 0
   last_bin_center       =  2008, 8,15,12, 0, 0
   bin_separation        =     0, 0,30, 0, 0, 0
   bin_width             =     0, 0,30, 0, 0, 0
   time_to_skip          =     0, 0, 0, 0, 0, 0
   max_num_bins          = 1000
   Nregions              = 1
   xlim1                 = -1.0
   xlim2                 = 1000000.0
   ylim1                 = -1.0
   ylim2                 = 1000000.0
   reg_names             = 'Entire Domain'
   create_rank_histogram = .true.
   outliers_in_histogram = .false.
   verbose               = .true.
   /
```

then, simply run obs_diag in the usual manner - you may want to save the run-time output to a file. Here is a portion of the run-time output:

```
...
Region  1 Entire Domain                    (WESN):     0.0000   360.0000   -90.0000   ␣
↪90.0000
 Requesting            1  assimilation periods.

epoch      1  start day=148865, sec=43201
epoch      1 center day=148880, sec=43200
epoch      1    end day=148895, sec=43200
epoch      1  start 2008 Jul 31 12:00:01
epoch      1 center 2008 Aug 15 12:00:00
epoch      1    end 2008 Aug 30 12:00:00
...
MARINE_SFC_HORIZONTAL_WIND_guess_RankHis has            0 "rank"able observations.
SAT_HORIZONTAL_WIND_guess_RankHist       has            0 "rank"able observations.
...
```

Discussion: It should be pretty clear that there is exactly 1 assimilation period, it may surprise you that the start is 1 second past 12Z. This is deliberate and reflects the DART convention of starting intervals 1 second after the end of the previous interval. The times in the netCDF variables reflect the defined start/stop of the period, regardless of the time of the first/last observation.

Please note that none of the 'horizontal_wind' variables will have a rank histogram, so they are not written to the netCDF file. ANY variable that does not have a rank histogram with some observations will NOT have a rank histogram variable in the netCDF file.

Now that you have the `obs_diag_output.nc`, you can explore it with `plot_profile.m`, `plot_bias_xxx_profile.m`, or `plot_rmse_xxx_profile.m`, rank histograms with ncview or `plot_rank_histogram.m`.

6.187.9 References

1. none

6.187.10 Private components

N/A

6.188 PROGRAM `obs_diag` (for observations that use the threed_sphere location module)

6.188.1 Overview

Main program for evaluating filter performance in observation space. Primarily, the prior or posterior ensemble mean (and spread) are compared to the observation and several quantities are calculated. These quantities are then saved in a netCDF file that has all the metadata to create meaningful figures.

Each `obs_seq.final` file contains an observation sequence that has multiple 'copies' of the observation. One copy is the actual observation, another copy is the prior ensemble mean estimate of the observation, one is the spread of the prior ensemble estimate, one may be the prior estimate from ensemble member 1, … etc. If the original observation sequence is the result of a 'perfect model' experiment, there is an additional copy called the 'truth' - the noise-free expected observation given the true model state. Since this copy does not, in general, exist for the high-order models, all comparisons are made with the copy labelled 'observation'. There is also a namelist variable (`use_zero_error_obs`) to compare against the 'truth' instead; the observation error variance is then automatically set to zero.

Each ensemble member applies a forward observation operator to the state to compute the "expected" value of an observation. Please note: the forward observation operator is applied **AFTER** any prior inflation has taken place! Similarly, the forward observation operator is applied AFTER any posterior inflation. This has always been the case. For a detailed look at the relationship between the observation operators and inflation, please look at the Detailed Program Execution Flow section of *PROGRAM filter*.

Given multiple estimates of the observation, several quantities can be calculated. It is possible to compute the expected observations from the state vector before assimilating (the "guess", "forecast", or "prior") or after the assimilation (the "analysis", or "posterior").

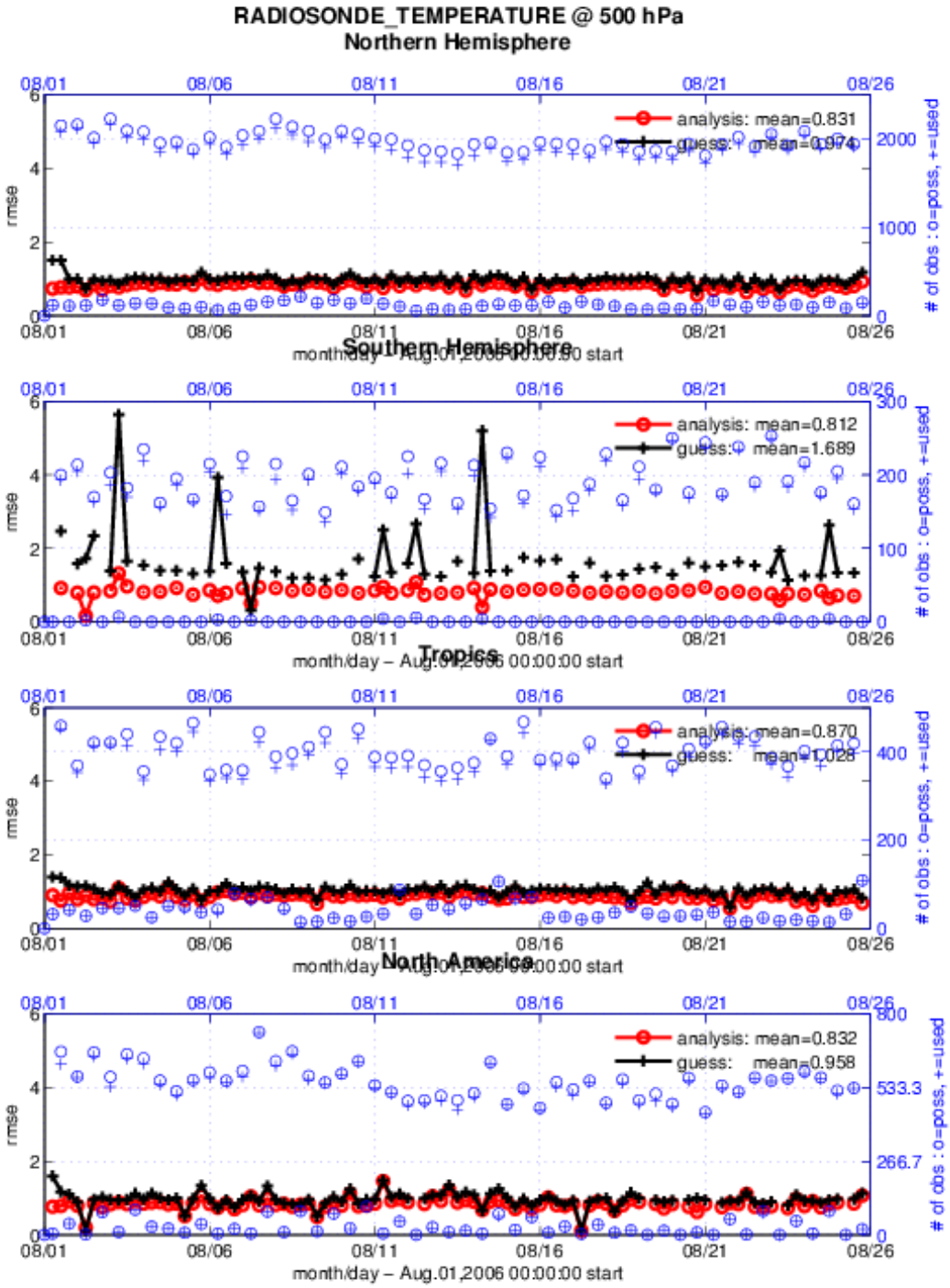
Even with `input.nml:filter_nml:num_output_obs_members` set to `0`; the full [prior,posterior] ensemble mean and [prior,posterior] ensemble spread are preserved in the `obs_seq.final` file. Consequently, the ensemble means and spreads are used to calculate the diagnostics. If the `input.nml:filter_nml:num_output_obs_members` is set to `80` (for example); the first 80 ensemble members prior and posterior "expected" values of the observation are also included. In this case, the `obs_seq.final` file contains enough information to calculate a rank histograms, verify forecasts, etc. The ensemble means are still used for many other calculations.

Since this program is fundamentally interested in the response as a function of region, there are three versions of this program; one for each of the `oned, threed_sphere, or threed_cartesian` location modules (`location_mod.f90`). It did not make sense to ask the `lorenz_96` model what part of North America you'd like to investigate or how you would like to bin in the vertical. The low-order models write out similar netCDF files and the Matlab scripts have been updated accordingly. The oned observations have locations conceptualized as being on a unit circle, so only the namelist input variables pertaining to longitude are used.

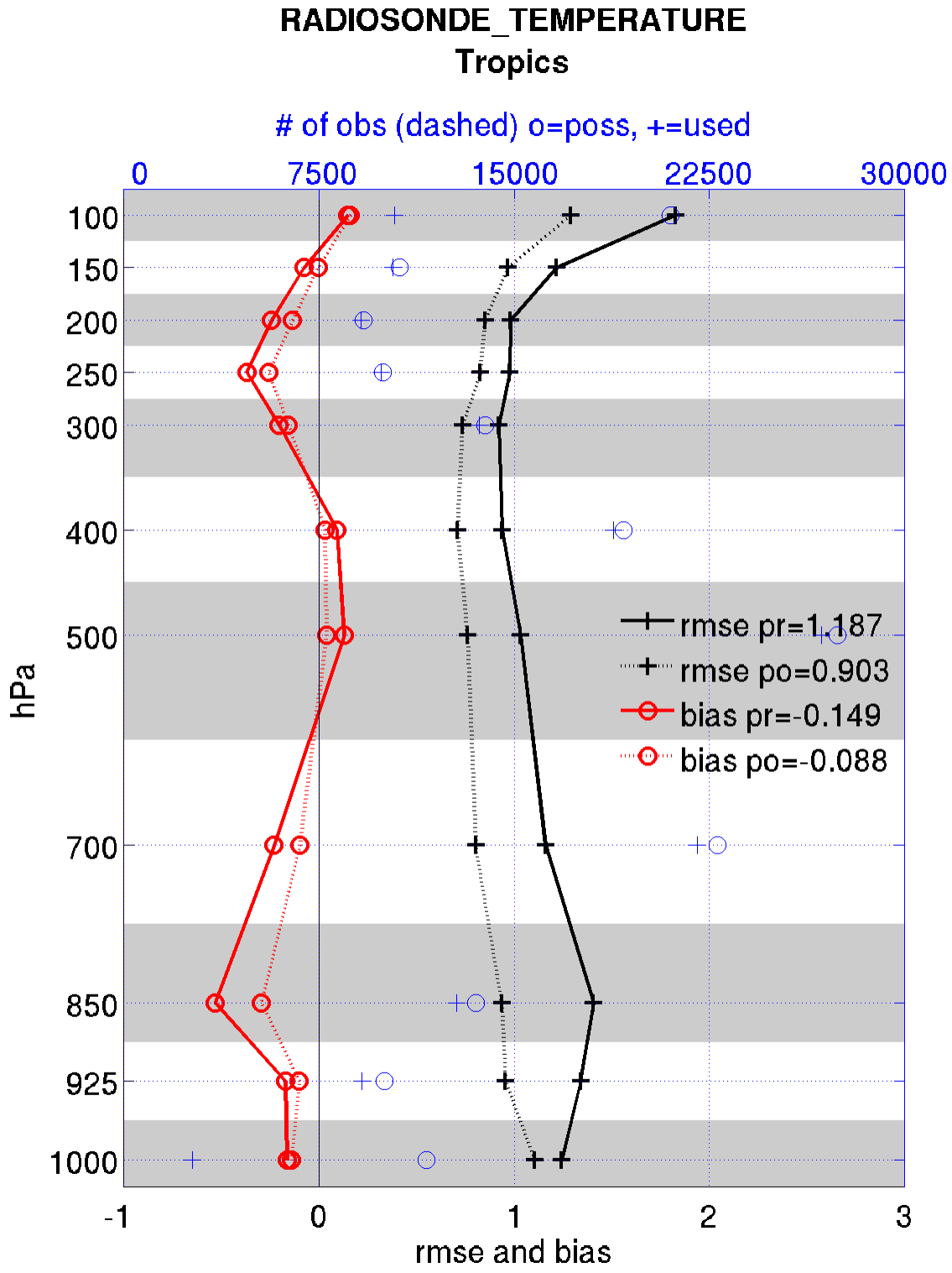
Identity observations (only possible from "perfect model experiments") are already explored with state-space diagnostics, so `obs_diag` simply skips them. The notable exception to this is a program specifically written for streamflow observations taken at gauge locations as represented by the 'channel-only' configuration of WRF-Hydro. There is a separate program *DART/assimilation_code/programs/obs_diag/stream_flow/streamflow_obs_diag.f90* specifically for those observations, since the model is designed to run **at** the USGS gauge locations.

6.188.2 `obs_diag` is designed to explore the effect of the assimilation in three ways:

1) as a function of time for a particular variable and level

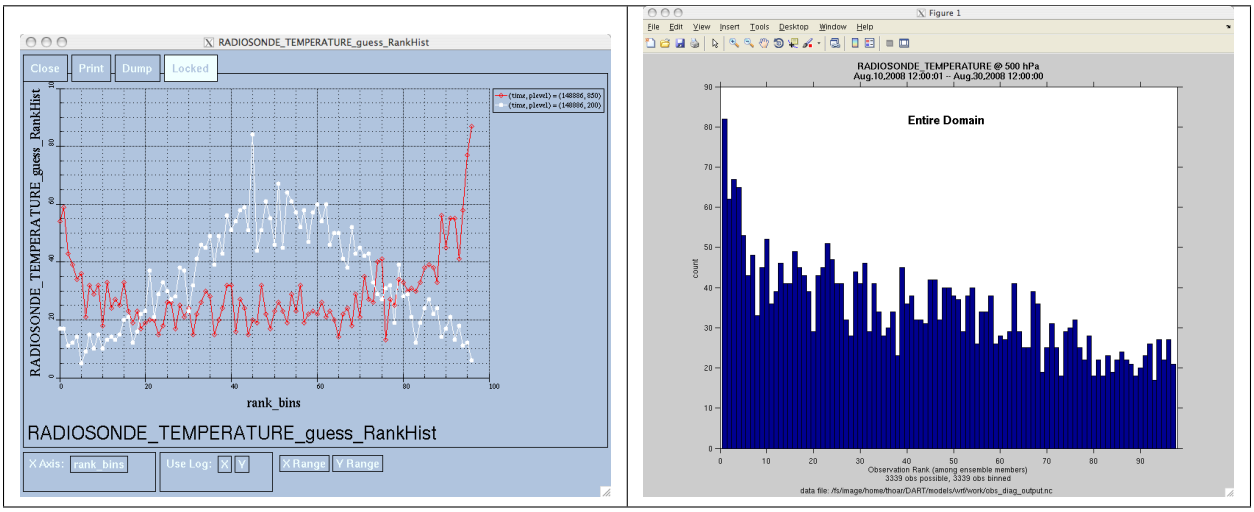


2) as a time-averaged vertical profile



3) and in terms of a rank histogram -

"Where does the actual observation rank relative to the rest of the ensemble?"



The figures in sections 1 and 2 were created by Matlab® scripts that query the `obs_diag_output.nc` file: `DART/diagnostics/matlab/plot_evolution.m` and `plot_profile.m`. Both of these takes as input a file name and a 'quantity' to plot ('rmse','spread','totalspread', ...) and exhaustively plots the quantity (for every variable, every level, every region) in a single matlab figure window - and creates a series of .ps files with multiple pages for each of the figures. The directory gets cluttered with them. The rank histogram information in `obs_diag_output.nc` can easily be plotted with ncview (left), a free third-party piece of software or with `plot_rank_histogram.m` (right). See the *Rank histograms* section for more information and links to instructions.

`obs_diag` can be configured to compare the ensemble estimates against the 'observation' copy or the 'truth' copy based on the setting of the `use_zero_error_obs` namelist variable.

The observation sequence files contain only the time of the observation, nothing of the assimilation interval, etc. - so it requires user guidance to declare what sort of temporal binning for the temporal evolution plots. I do a 'bunch' of arithmetic on the namelist times to convert them to a series of temporal bin edges that are used when traversing the observation sequence. The actual algorithm is that the user input for the start date and bin width set up a sequence that ends in one of two ways ... the last time is reached or the number of bins has been reached.

`obs_diag` reads `obs_seq.final` files and calculates the following quantities (in no particular order) for an arbitrary number of regions and levels. `obs_diag` creates a netCDF file called `obs_diag_output.nc`. It is necessary to query the `CopyMetaData` variable to determine the storage order (i.e. "which copy is what?") if you want to use your own plotting routines.

ncdump -f F -v CopyMetaData obs_diag_output.nc

Nposs	The number of observations available to be assimilated.
Nused	The number of observations that were assimilated.
NbadUV	the number of velocity observations that had a matching component that was not assimilated;
NbadLV	the number of observations that were above or below the highest or lowest model level, respectively;
rmse	The root-mean-squared error (the horizontal wind components are also used to calculate the vector wind velocity and its RMS error).
bias	The simple sum of forecast - observation. The bias of the horizontal wind speed (not velocity) is also computed.
spread	The standard deviation of the univariate obs. DART does not exploit the bivariate nature of U,V winds and so the spread of the horizontal wind is defined as the sum of the spreads of the U and V components.
total-spread	The total standard deviation of the estimate. We pool the ensemble variance of the observation plus the observation error variance and take the square root.
Nbad-DARTQC	the number of observations that had a DART QC value (> 1 for a prior, > 3 for a posterior)
obser-vation	the mean of the observation values
ens_mean	the ensemble mean of the model estimates of the observation values
N_trusted	the number of implicitly trusted observations, regardless of DART QC
N_DARTqc_0	number of observations that had a DART QC value of 0
N_DARTqc_1	number of observations that had a DART QC value of 1
N_DARTqc_2	number of observations that had a DART QC value of 2
N_DARTqc_3	number of observations that had a DART QC value of 3
N_DARTqc_4	number of observations that had a DART QC value of 4
N_DARTqc_5	number of observations that had a DART QC value of 5
N_DARTqc_6	number of observations that had a DART QC value of 6
N_DARTqc_7	number of observations that had a DART QC value of 7
N_DARTqc_8	number of observations that had a DART QC value of 8

The temporal evolution of the above quantities for every observation type (RADIOSONDE_U_WIND_COMPONENT, AIRCRAFT_SPECIFIC_HUMIDITY, ...) is recorded in the output netCDF file - `obs_diag_output.nc`. This netCDF file can then be loaded and displayed using the Matlab® scripts in `.../DART/diagnostics/matlab`. (which may depend on functions in `.../DART/matlab`). The temporal, geographic, and vertical binning are under namelist control. Temporal averages of the above quantities are also stored in the netCDF file. Normally, it is useful to skip the 'burn-in' period - the amount of time to skip is under namelist control.

The DART QC flag is intended to provide information about whether the observation was assimilated, evaluated only, whether the assimilation resulted in a 'good' observation, etc. DART QC values **<2** indicate the prior and posteriors are OK. DART QC values **>3** were **not** assimilated or evaluated. Here is the table that should explain things more fully:

DART QC flag value	meaning
0	observation assimilated
1	observation evaluated only (because of namelist settings)
2	assimilated, but the posterior forward operator failed
3	evaluated only, but the posterior forward operator failed
4	prior forward operator failed
5	not used because observation type not listed in namelist
6	rejected because incoming observation QC too large
7	rejected because of a failed outlier threshold test
8	vertical conversion failed
9+	reserved for future use

6.188.3 What is new in the Manhattan release

1. Support for DART QC = 8 (failed vertical conversion).
2. Simplified input file specification.
3. Removed `rat_cri` and `input_qc_threshold` from the namelists. They had been deprecated for quite some time.
4. Some of the internal variable names have been changed to make it easier to distinguish between variances and standard deviations.

6.188.4 What is new in the Lanai release

`obs_diag` has several improvements:

1. Improved vertical specification. Namelist variables `[h,p,m]level_edges` allow fine-grained control over the vertical binning. It is not allowed to specify both the edges and midpoints for the vertical bins.
2. Improved error-checking for input specification, particularly the vertical bins. Repeated values are squeezed out.
3. Support for 'trusted' observations. Trusted observation types may be specified in the namelist and all observations of that type will be counted in the statistics despite the DART QC code (as long as the forward observation operator succeeds). See namelist variable `trusted_obs`. For more details, see the section on Trusted observations.
4. Support for 'true' observations (i.e. from an OSSE). If the 'truth' copy of an observation is desired for comparison (instead of the default copy) the observation error variance is set to 0.0 and the statistics are calculated relative to the 'truth' copy (as opposed to the normal 'noisy' or 'observation' copy). See namelist variable `use_zero_error_obs`.
5. discontinued the use of `rat_cri` and `input_qc_threshold` namelist variables. Their functionality was replaced by the DART QC mechanism long ago.
6. The creation of the rank histogram (if possible) is now namelist-controlled by namelist variable `create_rank_histogram`.

6.188.5 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_diag_nml
   obs_sequence_name     = ''
   obs_sequence_list     = ''
   first_bin_center      =  2003, 1, 1, 0, 0, 0
   last_bin_center       =  2003, 1, 2, 0, 0, 0
   bin_separation        =     0, 0, 0, 6, 0, 0
   bin_width             =     0, 0, 0, 6, 0, 0
   time_to_skip          =     0, 0, 1, 0, 0, 0
   max_num_bins          = 1000
   plevel                = -888888.0
   hlevel                = -888888.0
   mlevel                = -888888
   plevel_edges          = -888888.0
```

(continues on next page)

(continued from previous page)

```
   hlevel_edges          = -888888.0
   mlevel_edges          = -888888
   Nregions              = 0
   lonlim1               = -888888.0
   lonlim2               = -888888.0
   latlim1               = -888888.0
   latlim2               = -888888.0
   reg_names             = 'null'
   trusted_obs           = 'null'
   create_rank_histogram = .true.
   outliers_in_histogram = .false.
   use_zero_error_obs    = .false.
   verbose               = .false.
   /
```

The date-time integer arrays in this namelist have the form (YYYY, MM, DY, HR, MIN, SEC).
The allowable ranges for the region boundaries are: latitude [-90.,90], longitude [0.,Inf.]

You can only specify **either** `obs_sequence_name` **or** `obs_sequence_list` – not both. One of them has to be an empty string … i.e. `''`.

Item	Type	Description
obs_sequence_name	character(len=256), dimension(100)	An array of names of observation sequence files. These may be relative or absolute filenames. If this is set, `obs_sequence_list` must be set to ' ' (empty string).
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, `obs_sequence_name` must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program. If this is set, `obs_sequence_name` must be set to ' ' (empty string).
first_bin_center	integer, dimension(6)	first timeslot of the first obs_seq.final file to process. The six integers are: year, month, day, hour, hour, minute, second, in that order. `obs_diag` has improved run-time output that reports the time and date of the first and last observations in every observation sequence file. Look for the string 'First observation date' in the logfile. If the `verbose` is 'true', it is also written to the screen.
last_bin_center	integer, dimension(6)	last timeslot of interest. (reminder: the last timeslot of day 1 is hour 0 of day 2) The six integers are: year, month, day, hour, hour, minute, second, in that order. This does not need to be exact, the values from `first_bin_center` and `bin_separation` are used to populate the time array and stop on or before the time defined by `last_bin_center`. See also `max_num_bins`.
bin_separation	integer, dimension(6)	Time between bin centers. The year and month values *must* be zero.
bin_width	Integer, dimension(6)	Time span around bin centers in which obs will be compared. The year and month values *must* be zero. Frequently, but not required to be, the same as the values for bin_separation. 0
time_to_skip	integer, dimension(6)	Time span at the beginning to skip when calculating vertical profiles of rms error and bias. The year and month values *must* be zero. Useful because it takes some time for the assimilation to settle down from the climatological spread at the start. `time_to_skip` is an amount of time AFTER the first edge of the first bin.
max_num_bins	integer	This provides an alternative way to declare the `last_bin_center`. If `max_num_bins` is set to '10', only 10 timesteps will be output - provided `last_bin_center` is set to some later date.
plevel	real, dimension(50)	The midpoints defining the pressure levels for the vertical binning. There is no specification of bin width - a continuum is used. If a single midpoint value is entered, the bin edges are +/- 10% of the midpoint value. If you'd like to change that see the routine *Rmidpoints2edges()*. You may specify either `plevel` or `plevel_edges`, but not both.
plevel_edges	real, dimension(51)	The edges defining the pressure levels for the vertical binning. You may specify either `plevel` or `plevel_edges`, but not both.
hlevel	real, dimension(50)	Same, but for observations that have height(m) or depth(m) as the vertical coordinate.
hlevel_edges	real, dimension(51)	The edges defining the height (or depth) levels for the vertical binning. You may specify either `hlevel` or `hlevel_edges`, but not both.
mlevel	real, dimension(50)	Same, but for observations that have model level as the vertical coordinate.
mlevel_edges	real, dimension(51)	The edges defining the model levels for the vertical binning. You may specify either `mlevel` or `mlevel_edges`, but not both.
Nregions	integer	Number of regions of the globe for which obs space diagnostics are computed separately. Must be between [1,50]. If 50 is not enough, increase `obs_diag.f90MaxRegions` and recompile.
lonlim1	real, dimension(50)	Westernmost longitudes of each of the regions.
lonlim2	real, dimension(50)	Easternmost longitudes of each of the regions. *If any of these values is* **less than** *the westernmost values, it defines a region that spans the prime meridian.* e.g. a specification of `lonlim1 = 330 , lonlim2 = 50` could identify a region like "Africa".
latlim1	real, dimen-	Southernmost latitudes of the regions.

6.188.6 Other modules used

```
obs_sequence_mod
obs_kind_mod
obs_def_mod (and possibly other obs_def_xxx mods)
assim_model_mod
random_seq_mod
model_mod
location_mod
types_mod
time_manager_mod
utilities_mod
sort_mod
```

6.188.7 Files

- `input.nml` is used for `obs_diag_nml`
- `obs_diag_output.nc` is the netCDF output file
- `dart_log.out` list directed output from the obs_diag.
- `LargeInnov.txt` contains the distance ratio histogram – useful for estimating the distribution of the magnitudes of the innovations.

Obs_diag may require a model input file from which to get grid information, metadata, and links to modules providing the models expected observations. It all depends on what's needed by the `model_mod.f90`

Discussion of obs_diag_output.nc

Every observation type encountered in the observation sequence file is tracked separately, and aggregated into temporal and 3D spatial bins. There are two main efforts to this program. One is to track the temporal evolution of any of the quantities available in the netCDF file for any possible observation type:

ncdump -v CopyMetaData,ObservationTypes obs_diag_output.nc

The other is to explore the vertical profile of a particular observation kind. By default, each observation kind has a 'guess/prior' value and an 'analysis/posterior' value - which shed some insight into the innovations.

Temporal evolution

The `obs_diag_output.nc` output file has all the metadata I could think of, as well as separate variables for every observation type in the observation sequence file. Furthermore, there is a separate variable for the 'guess/prior' and 'analysis/posterior' estimate of the observation. To distinguish between the two, a suffix is appended to the variable name. An example seems appropriate:

```
...
char CopyMetaData(copy, stringlength) ;
        CopyMetaData:long_name = "quantity names" ;
char ObservationTypes(obstypes, stringlength) ;
        ObservationTypes:long_name = "DART observation types" ;
        ObservationTypes:comment = "table relating integer to observation type string" ;
float RADIOSONDE_U_WIND_COMPONENT_guess(time, copy, plevel, region) ;
```

(continues on next page)

(continued from previous page)

```
        RADIOSONDE_U_WIND_COMPONENT_guess:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_guess:missing_value = -888888.f ;
float RADIOSONDE_V_WIND_COMPONENT_guess(time, copy, plevel, region) ;
        RADIOSONDE_V_WIND_COMPONENT_guess:_FillValue = -888888.f ;
        RADIOSONDE_V_WIND_COMPONENT_guess:missing_value = -888888.f ;
...
float MARINE_SFC_ALTIMETER_guess(time, copy, surface, region) ;
        MARINE_SFC_ALTIMETER_guess:_FillValue = -888888.f ;
        MARINE_SFC_ALTIMETER_guess:missing_value = -888888.f ;
...
float RADIOSONDE_WIND_VELOCITY_guess(time, copy, plevel, region) ;
        RADIOSONDE_WIND_VELOCITY_guess:_FillValue = -888888.f ;
        RADIOSONDE_WIND_VELOCITY_guess:missing_value = -888888.f ;
...
float RADIOSONDE_U_WIND_COMPONENT_analy(time, copy, plevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_analy:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_analy:missing_value = -888888.f ;
float RADIOSONDE_V_WIND_COMPONENT_analy(time, copy, plevel, region) ;
        RADIOSONDE_V_WIND_COMPONENT_analy:_FillValue = -888888.f ;
        RADIOSONDE_V_WIND_COMPONENT_analy:missing_value = -888888.f ;
...
```

There are several things to note:

1. the 'WIND_VELOCITY' component is nowhere 'near' the corresponding U,V components.
2. all of the 'guess' variables come before the matching 'analy' variables.
3. surface variables (i.e. `MARINE_SFC_ALTIMETER` have a coordinate called 'surface' as opposed to 'plevel' for the others in this example).

Vertical profiles

Believe it or not, there are another set of netCDF variables specifically for the vertical profiles, essentially duplicating the previous variables but **without the 'time' dimension**. These are distinguished by the suffix added to the observation kind - 'VPguess' and 'VPanaly' - 'VP' for Vertical Profile.

```
...
float SAT_WIND_VELOCITY_VPguess(copy, plevel, region) ;
        SAT_WIND_VELOCITY_VPguess:_FillValue = -888888.f ;
        SAT_WIND_VELOCITY_VPguess:missing_value = -888888.f ;
...
float RADIOSONDE_U_WIND_COMPONENT_VPanaly(copy, plevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_VPanaly:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_VPanaly:missing_value = -888888.f ;
...
```

Observations flagged as 'surface' do not participate in the vertical profiles (Because surface variables cannot exist on any other level, there's not much to plot!). Observations on the lowest level DO participate. There's a difference!

Rank histograms

If it is possible to calculate a rank histogram, there will also be :

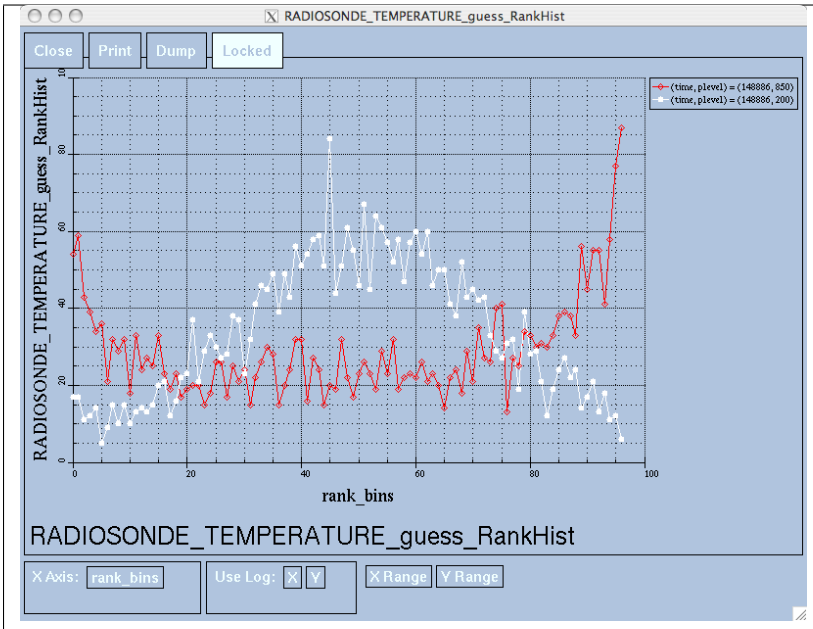
```
...
int RADIOSONDE_U_WIND_COMPONENT_guess_RankHi(time, rank_bins, plevel, region) ;
...
int RADIOSONDE_V_WIND_COMPONENT_guess_RankHi(time, rank_bins, plevel, region) ;
...
int MARINE_SFC_ALTIMETER_guess_RankHist(time, rank_bins, surface, region) ;
...
```

as well as some global attributes. The attributes reflect the namelist settings and can be used by plotting routines to provide additional annotation for the histogram.

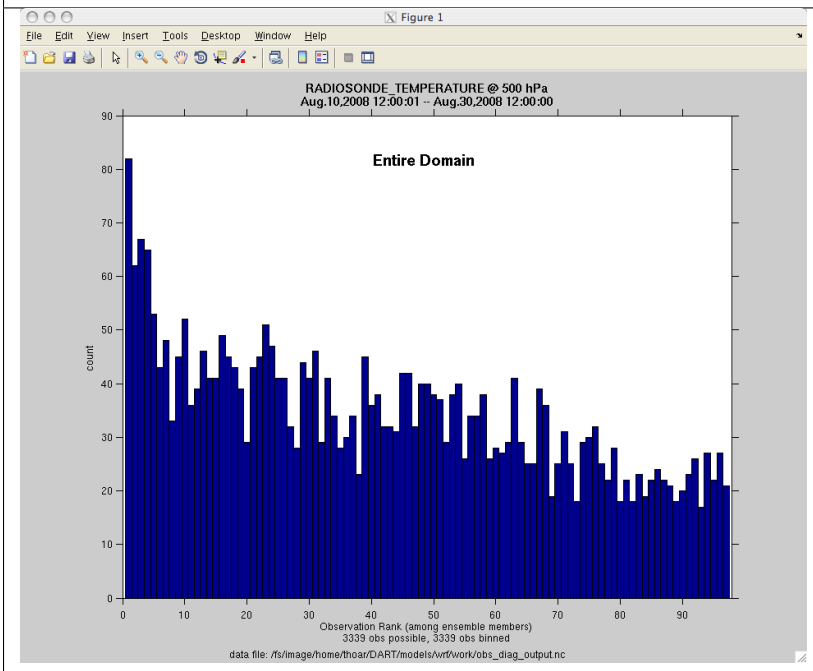
```
:DART_QCs_in_histogram = 0, 1, 2, 3, 7 ;
:outliers_in_histogram = "TRUE" ;
```

Please note:

1. netCDF restricts variable names to 40 characters, so '_Rank_Hist' may be truncated.
2. It is sufficiently vague to try to calculate a rank histogram for a velocity derived from the assimilation of U,V components such that NO rank histogram is created for velocity. A run-time log message will inform as to which variables are NOT having a rank histogram variable preserved in the `obs_diag_output.nc` file - IFF it is possible to calculate a rank histogram in the first place.



Instructions for viewing the rank histogram with ncview.



Instructions for viewing the rank histogram with Matlab.

"trusted" observation types

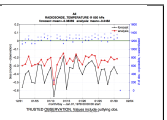
This needs to be stated up front: `obs_diag` is a post-processor; it cannot influence the assimilation. One interpretation of a TRUSTED observation is that the assimilation should **always** use the observation, even if it is far from the ensemble. At present (23 Feb 2015), the filter in DART does not forcibly assimilate any one observation and selectively assimilate the others. Still, it is useful to explore the results using a set of 'trusted type' observations, whether they were assimilated, evaluated, or rejected by the outlier threshhold. This is the important distinction. The diagnostics can be calculated differently for each *observation type*.

The normal diagnostics calculate the metrics (rmse, bias, etc.) only for the 'good' observations - those that were assimilated or evaluated. The `outlier_threshold` essentially defines what observations are considered too far from the ensemble **prior** to be useful. These observations get a DART QC of 7 and are not assimilated. The observations with a DART QC of 7 do not contribute the the metrics being calculated. Similarly, if the forward observation operator fails, these observations cannot contribute. When the operator fails, the 'expected' observation value is 'MISSING', and there is no ensemble mean or spread.

'Trusted type' observation metrics are calculated using all the observations that were assimilated or evaluated **AND** the observations that were rejected by the outlier threshhold. `obs_diag` can post-process the DART QC and calculate the metrics appropriately for **observation types** listed in the `trusted_obs` namelist variable. If there are trusted observation types specified for `obs_diag`, the `obs_diag_output.nc` has global metadata to indicate that a different set of criteria were used to calculate the metrics. The individual variables also have an extra attribute. In the following output, `input.nml:obs_diag_nml:trusted_obs` was set: `trusted_obs = 'RADIOSONDE_TEMPERATURE', 'RADIOSONDE_U_WIND_COMPONENT'`

```
  ...
        float RADIOSONDE_U_WIND_COMPONENT_guess(time, copy, plevel, region) ;
                RADIOSONDE_U_WIND_COMPONENT_guess:_FillValue = -888888.f ;
                RADIOSONDE_U_WIND_COMPONENT_guess:missing_value = -888888.f ;
                RADIOSONDE_U_WIND_COMPONENT_guess:TRUSTED = "TRUE" ;
        float RADIOSONDE_V_WIND_COMPONENT_guess(time, copy, plevel, region) ;
                RADIOSONDE_V_WIND_COMPONENT_guess:_FillValue = -888888.f ;
                RADIOSONDE_V_WIND_COMPONENT_guess:missing_value = -888888.f ;
  ...
// global attributes:
  ...
                :trusted_obs_01 = "RADIOSONDE_TEMPERATURE" ;
                :trusted_obs_02 = "RADIOSONDE_U_WIND_COMPONENT" ;
                :obs_seq_file_001 = "cam_obs_seq.1978-01-01-00000.final" ;
                :obs_seq_file_002 = "cam_obs_seq.1978-01-02-00000.final" ;
                :obs_seq_file_003 = "cam_obs_seq.1978-01-03-00000.final" ;
  ...
                :MARINE_SFC_ALTIMETER = 7 ;
                :LAND_SFC_ALTIMETER = 8 ;
                :RADIOSONDE_U_WIND_COMPONENT--TRUSTED = 10 ;
                :RADIOSONDE_V_WIND_COMPONENT = 11 ;
                :RADIOSONDE_TEMPERATURE--TRUSTED = 14 ;
                :RADIOSONDE_SPECIFIC_HUMIDITY = 15 ;
                :AIRCRAFT_U_WIND_COMPONENT = 21 ;
  ...
```

The Matlab scripts try to ensure that the trusted observation graphics clarify that the metrics plotted are somehow 'different' than the normal processing stream. Some text is added to indicate that the values include the outlying observations. **IMPORTANT:** The interpretation of the number of observations 'possible' and 'used' still reflects what was used **in the assimilation!** The number of observations rejected by the outlier threshhold is not explicilty plotted. To reinforce this, the text for the observation axis on all graphics has been changed to "`o=possible, *=assimilated`". In short, the distance between the number of observations possible and the number assimilated still reflects the number of observations rejected by the outlier threshhold and the number of failed forward observation operators.



There is ONE ambiguous case for trusted observations. There may be instances in which the observation fails the outlier threshhold test (which is based on the prior) and the posterior forward operator fails. DART does not have a QC that explicilty covers this case. The current logic in `obs_diag` correctly handles these cases **except** when trying to use 'trusted' observations. There is a section of code in `obs_diag` that may be enabled if you are encountering this ambiguous case. As `obs_diag` runs, a warning message is issued and a summary count is printed if the ambiguous case is encountered. What normally happens is that if that specific observation type is trusted, the posterior values include a MISSING value in the calculation which makes them inaccurate. If the block of code is enabled, the DART QC is recast as the PRIOR forward observation operator fails. This is technically incorrect, but for the case of trusted observations, it results in only calculating statistics for trusted observations that have a useful prior and posterior. **This should not be used unless you are willing to intentionally disregard 'trusted' observations that were rejected by the outlier threshhold.** Since the whole point of a trusted observation is to *include* observations potentially rejected by the outlier threshhold, you see the problem. Some people like to compare the posteriors. *THAT* can be the problem.

```
if ((qc_integer == 7) .and. (abs(posterior_mean(1) - MISSING_R8) < 1.0_r8)) then
            write(string1,*)'WARNING ambiguous case for obs index ',obsindex
            string2 = 'obs failed outlier threshhold AND posterior operator failed.'
            string3 = 'Counting as a Prior QC == 7, Posterior QC == 4.'
            if (trusted) then
! COMMENT      string3 = 'WARNING changing DART QC from 7 to 4'
! COMMENT      qc_integer = 4
            endif
            call error_handler(E_MSG,'obs_diag',string1,text2=string2,text3=string3)
            num_ambiguous = num_ambiguous + 1
         endif
```

6.188.8 Usage

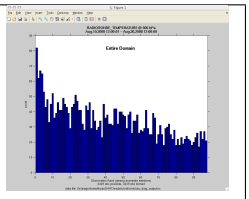
`obs_diag` is built in .../DART/models/*your_model*/work, in the same way as the other DART components.

Multiple observation sequence files

There are two ways to specify input files for `obs_diag`. You can either specify the name of a file containing a list of files (in `obs_sequence_list`), or you may specify a list of files via `obs_sequence_name`.

Example: observation sequence files spanning 30 days

In this example, we will be accumulating metrics for 30 days. The `obs_diag_output.nc` file will have exactly ONE timestep in it (so it won't be much use for the `plot_evolution` functions) - but the `plot_profile` functions and the `plot_rank_histogram` function will be used to explore the assimilation. By way of an example, we will NOT be using outlier observations in the rank histogram. Lets presume that all your `obs_seq.final` files are in alphabetically-nice directories:



```
/Exp1/Dir01/obs_seq.final
/Exp1/Dir02/obs_seq.final
/Exp1/Dir03/obs_seq.final
...
/Exp1/Dir99/obs_seq.final
```

The first step is to create a file containing the list of observation sequence files you want to use. This can be done with the unix command 'ls' with the -1 option (that's a number one) to put one file per line.

ls -1 /Exp1/Dir*/obs_seq.final > obs_file_list.txt

It is necessary to turn on the verbose option to check the first/last times that will be used for the histogram. Then, the namelist settings for 2008 07 31 12Z through 2008 08 30 12Z are:

```
&obs_diag_nml
   obs_sequence_name     = ''
   obs_sequence_list     = 'obs_file_list.txt'
   first_bin_center      =  2008, 8,15,12, 0, 0
   last_bin_center       =  2008, 8,15,12, 0, 0
   bin_separation        =     0, 0,30, 0, 0, 0
   bin_width             =     0, 0,30, 0, 0, 0
   time_to_skip          =     0, 0, 0, 0, 0, 0
   max_num_bins          = 1000
   Nregions              = 1
   lonlim1               =   0.0
   lonlim2               = 360.0
   latlim1               = -90.0
   latlim2               =  90.0
   reg_names             = 'Entire Domain'
   create_rank_histogram = .true.
   outliers_in_histogram = .false.
   verbose               = .true.
   /
```

then, simply run `obs_diag` in the usual manner - you may want to save the run-time output to a file. Here is a portion of the run-time output:

```
...
Region  1 Entire Domain                    (WESN):     0.0000   360.0000   -90.0000   ␣
↪90.0000
 Requesting            1  assimilation periods.

epoch      1  start day=148865, sec=43201
epoch      1 center day=148880, sec=43200
epoch      1    end day=148895, sec=43200
epoch      1  start 2008 Jul 31 12:00:01
epoch      1 center 2008 Aug 15 12:00:00
epoch      1    end 2008 Aug 30 12:00:00
...
MARINE_SFC_HORIZONTAL_WIND_guess_RankHis has            0 "rank"able observations.
SAT_HORIZONTAL_WIND_guess_RankHist       has            0 "rank"able observations.
...
```

Discussion: It should be pretty clear that there is exactly 1 assimilation period, it may surprise you that the start is 1 second past 12Z. This is deliberate and reflects the DART convention of starting intervals 1 second after the end of the previous interval. The times in the netCDF variables reflect the defined start/stop of the period, regardless of the time of the first/last observation.

Please note that none of the 'horizontal_wind' variables will have a rank histogram, so they are not written to the netCDF file. ANY variable that does not have a rank histogram with some observations will NOT have a rank histogram variable in the netCDF file.

Now that you have the `obs_diag_output.nc`, you can explore it with `plot_profile.m`, `plot_bias_xxx_profile.m`, or `plot_rmse_xxx_profile.m`, rank histograms with ncview or `plot_rank_histogram.m`.

6.188.9 References

1. none

6.188.10 Private components

N/A

6.189 PROGRAM `fill_inflation_restart`

6.189.1 Overview

Utility program to create inflation restart files with constant values.

These files can be used as input for the first step of a multi-step assimilation when adaptive inflation is being used. This allows the namelist items `inf_initial_from_restart` and `inf_sd_initial_from_restart` in the `&filter_nml` namelist to be `.TRUE.` for all steps of the assimilation including the very first one. (These items control whether inflation values are read from an input file or read from constants in the namelist.)

Adaptive inflation restart files are written at the end of a `filter` run and are needed as input for the next timestep. This program creates files that can be used for the initial run of filter when no inflation restart files have been created by filter but are required to be read as input.

This program reads the inflation values to use from the `&fill_inflation_restart_nml` namelist for setting the prior inflation mean and standard deviation, and/or the posterior inflation mean and standard deviation. It does not use the inflation values in the `&filter` namelist.

This program uses the information from the model_mod code to determine the number of items in the state vector. It must be compiled with the right model's model_mod, and if the items in the state vector are selectable by namelist options, the namelist when running this program must match exactly the namelist used during the assimilation run.

It creates files with names consistent with the input names expected by filter:

```
input_priorinf_mean.nc
input_priorinf_sd.nc
input_postinf_mean.nc
input_postinf_sd.nc
```

An older (and deprecated) alternative to running `fill_inflation_restart` is to create inflation netcdf files by using one of the NCO utilities like "`ncap2`" on a copy of another restart file to set the initial inflation mean, and another for the initial inflation standard deviation. Inflation mean and sd values look exactly like restart values, arranged by variable type like T, U, V, etc.

Depending on your version of the NCO utilities, you can use `ncap2` to set the T,U and V inf values using one of two syntaxes:

```
ncap2 -s 'T=1.0;U=1.0;V=1.0' wrfinput_d01 input_priorinf_mean.nc
ncap2 -s 'T=0.6;U=0.6;V=0.6' wrfinput_d01 input_priorinf_sd.nc
-or-
ncap2 -s 'T(:,:,:)=1.0;U(:,:,:)=1.0;V(:,:,:)=1.0' wrfinput_d01 input_priorinf_mean.nc
ncap2 -s 'T(:,:,:)=0.6;U(:,:,:)=0.6;V(:,:,:)=0.6' wrfinput_d01 input_priorinf_sd.nc
```

Some versions of the NCO utilities change the full 3D arrays into a single scalar. If that's your result (check your output with `ncdump -h`) use the alternate syntax or a more recent version of the NCO tools.

6.189.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&fill_inflation_restart_nml

   write_prior_inf    = .FALSE.
   prior_inf_mean     = -88888.8888
   prior_inf_sd       = -88888.8888

   write_post_inf     = .FALSE.
   post_inf_mean      = -88888.8888
   post_inf_sd        = -88888.8888

   single_file        = .FALSE.
   input_state_files  = ''
```

(continues on next page)

(continued from previous page)

```
   verbose             = .FALSE.
/
```

The namelist controls which files are created and what values are written to the restart files.

Item	Type	Description
write_prior_inf	logical	Setting this to .TRUE. writes both the prior inflation mean and standard deviation files: `input_priorinf_mean.nc`, `input_priorinf_sd.nc`.
prior_inf_mean	real(r8)	Prior inflation mean value.
prior_inf_sd	real(r8)	Prior inflation standard deviation value.
write_post_inf	logical	Setting this to .TRUE. writes both the posterior inflation mean and standard deviation files `input_postinf_mean.nc`, `input_postinf_sd.nc`.
post_inf_mean	real(r8)	Posterior inflation mean value.
post_inf_sd	real(r8)	Posterior inflation standard deviation value.
single_file	logical	Currently not supported, but would be used in the case where you have a single restart file that contains all of the ensemble members. Must be .false.
input_state_files	character(:)	List one per domain, to be used as a template for the output inflation files.
verbose	logical	Setting this to .TRUE. gives more output, and is generally used for debugging

Here is an example of a typical namelist for `fill_inflation_restart` :

```
&fill_inflation_restart_nml

   write_prior_inf    = .TRUE.
   prior_inf_mean     = 1.01
   prior_inf_sd       = 0.6

   write_post_inf     = .FALSE.
   post_inf_mean      = 1.0
   post_inf_sd        = 0.6

   single_file        = .FALSE.
   input_state_files  = ''
   verbose            = .FALSE.
/
```

6.189.3 Files

Creates:

```
input_priorinf_mean.nc
input_priorinf_sd.nc
input_postinf_mean.nc
input_postinf_sd.nc
```

based on the template file from the specific model this code is compiled for.

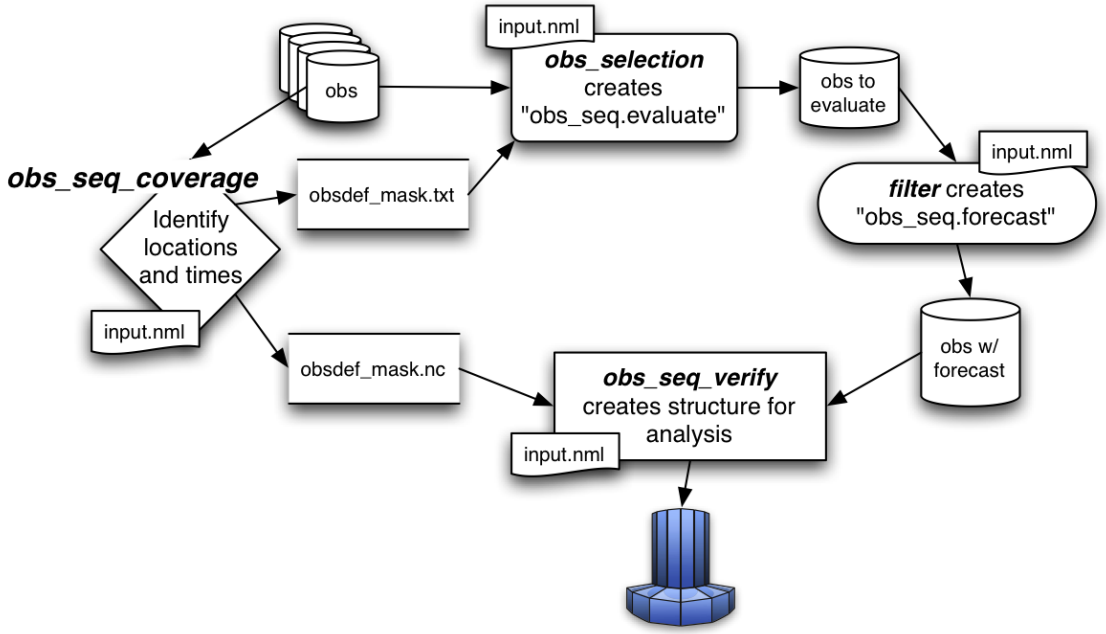
6.189.4 References

- none

6.190 program obs_seq_coverage

6.190.1 Overview

`obs_seq_coverage` queries a set of observation sequence files to determine which observation locations report frequently enough to be useful for a verification study. The big picture is to be able to pare down a large set of observations into a compact observation sequence file to run through *PROGRAM filter* with all of the intended observation types flagged as *evaluate_only*. DART's forward operators then get applied and all the forecasts are preserved in a standard `obs_seq.final` file - perhaps more appropriately called `obs_seq.forecast`! Paring down the input observation sequence file cuts down on the unnecessary application of the forward operator to create observation copies that will not be used anyway ...



`obs_seq_coverage` results in two output files:

- `obsdef_mask.txt` contains the list of observation definitions (but not the observations themselves) that are desired. The observation definitions include the locations and times for each of the desired observation types.

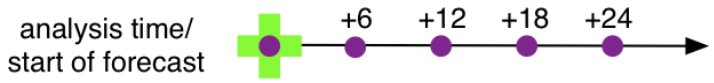
This file is read by *program obs_selection* and combined with the raw observation sequence files to create the observation sequence file appropriate for use in a forecast.

- `obsdef_mask.nc` contains information needed to be able to plot the times and locations of the observations in a manner to help explore the design of the verification locations/network. `obsdef_mask.nc` is *required* by *program obs_seq_verify*, the program that reorders the observations into a structure that makes it easy to calculate statistics like ROC, etc.

The following section explains the strategy and requirements for determining what observations will be used to verify a forecast. Since it is 'standard practice' to make several forecasts to build statistical strength, it is important to use the SAME set of observation locations for all the forecasts that will be verified together. To make the discussion easier, let's define the *verification network* as the set of locations and times for a particular observation type.

The entire discussion about finding locations that are repeatedly observed through time boils down to the simple statement that if the observation is within about 500cm of a previous observation, they are treated as co-located observations. For some very high resolution applications, this may be insufficient, but there it is. For observations at pressure levels, see the Word about vertical levels.

The only complicated part of determining the verification network is the temporal component. The initial time (usually an *analysis time* from a previous assimilation), the *verification interval*, and the *forecast length* completely specify the temporal aspect of a forecast. The following example has a verification interval of 6 hours and a forecast length of 24 hours. We adopt the convention of also including the initial conditions (a "nowcast") in the "forecast", so there are 5 times of interest - which we will call *verification times* and are represented by ●. The candidate observation sequence files are scanned to select all the observations that are **closest** to the verification times. The difference in time between the "nowcast" and the "forecast" is the *forecast lead*.

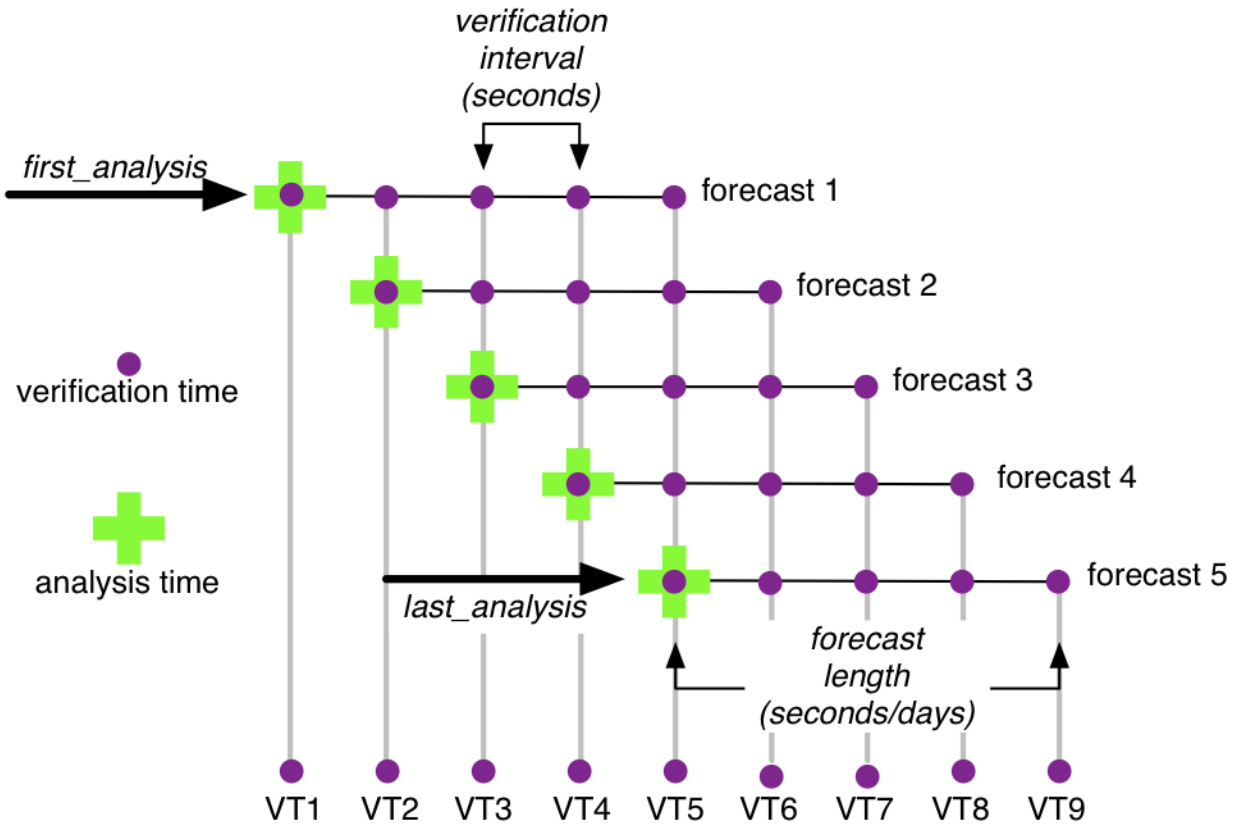


So - that is simple enough if there is only one forecast, but this is rarely the case. Let's say we have a second forecast. Ideally, we'd like to verify at exactly the same locations and forecast leads - otherwise we're not really comparing the same things. If the second verification network happens to be at locations that are easy to predict, we're comparing apples and oranges. The *fair* way to proceed is to determine the verification network that is the same for all forecasts. This generally results in a pretty small set of observations - a problem we will deal with later.

The diagram below illustrates the logic behind determining the list of verification times for a pretty common scenario: a 24-hour forecast with a forecast lead of 6 hours, repeated the next day. The *first_analysis* is at VT1 - let's call it 00Z day 1. We need to have observations available at:

VT1 (00Z day1), VT2 (06Z day1), VT3 (12Z day1), VT4 (18Z day1), and VT5 (24Z day1 / 00Z day2). The *last_analysis* starts at VT5 00Z day 2 and must verify at

VT5 (00Z day2), VT6 (06Z day2), VT7 (12Z day2), VT8 (18Z day2), and VT9 (24Z day2 / 00Z day3).



Note that, if you wanted to, you could launch forecasts at VT2, VT3, and VT4 without adding extra constraints on the verification network. `obs_seq_coverage` simply provides these possible forecasts "for free", there is no assumption about **needing** them. We will use the variable *verification_times* to describe the complete set of times for all possible forecasts. In our example above, there are 5 possible forecasts, each forecast consisting of 5 verification times (the analysis time and the 4 forecast lead times). As such, there are 9 unique verification times.

Note that no attempt is made at checking the QC value of the candidate observations. One of the common problems is that the region definition does not mesh particularly well with the model domain and the DART forward operator fails because it would have to extrapolate (which is not allowed). Without checking the QC value, this can mean there are a lot of 'false positives'; observations that seemingly could be used to validate, but are actually just outside the model domain. I'm working on that ....

The USAGE section has more on the actual use of `obs_seq_coverage`.

6.190.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_seq_coverage_nml
   obs_sequences      = ''
   obs_sequence_list  = ''
   obs_of_interest    = ''
   textfile_out       = 'obsdef_mask.txt'
   netcdf_out         = 'obsdef_mask.nc'
   calendar           = 'Gregorian'
```

(continues on next page)

(continued from previous page)

```
  first_analysis     =  2003, 1, 1, 0, 0, 0
  last_analysis      =  2003, 1, 2, 0, 0, 0
  forecast_length_days          = 1
  forecast_length_seconds       = 0
  verification_interval_seconds = 21600
  temporal_coverage_percent     = 100.0
  lonlim1                       =  -888888.0
  lonlim2                       =  -888888.0
  latlim1                       =  -888888.0
  latlim2                       =  -888888.0
  verbose                       = .false.
  debug                         = .false.
 /
```

Note that -888888.0 is not a useful number. To use the defaults delete these lines from the namelist, or set them to 0.0, 360.0 and -90.0, 90.0.

The date-time integer arrays in this namelist have the form (YYYY, MM, DD, HR, MIN, SEC).

The allowable ranges for the region boundaries are: latitude [-90.,90], longitude [0.,Inf.]

You can specify **either** *obs_sequences* **or** *obs_sequence_list* – not both. One of them has to be an empty string . . . i.e. ‘’.

Item	Type	Description
obs_sequences	character(len=256)	Name of the observation sequence file(s). This may be a relative or absolute filename. If the filename contains a '/', the filename is considered to be comprised of everything to the right, and a directory structure to the left. The directory structure is then queried to see if it can be incremented to handle a sequence of observation files. The default behavior of `obs_seq_coverage` is to look for additional files to include until the files are exhausted or an `obs_seq.final` file is found that contains observations beyond the timeframe of interest. e.g. 'obsdir_001/obs_seq.final' will cause `obs_seq_coverage` to look for 'obsdir_002/obs_seq.final', and so on. If this is set, *obs_sequence_list* must be set to ' '.
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, *obs_sequences* must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program.
obs_of_interest	character(len=32), dimension(:)	These are the observation types that will be verified. It is an array of character strings that must match the standard DART observation types. Simply add as many or as few observation types as you need. Could be 'METAR_U_10_METER_WIND', 'METAR_V_10_METER_WIND',..., for example.
textfile_out	character(len=256)	The name of the file that will contain the observation definitions of the verfication observations. Only the metadata from the observations (location, time, obs_type) are preserved in this file. They are in no particular order. *program obs_selection* will use this file as a 'mask' to extract the real observations from the candidate observation sequence files.
netcdf_out	character(len=256)	The name of the file that will contain the observation definitions of the unique locations that match **any** of the verification times. This file is used in conjunction with *program obs_diag* to match the `obs_seq.forecast` into a structure that will facilitate calculating the statistics and scores of the forecasts.

For example:

```
&obs_seq_coverage_nml
   obs_sequences     = ''
   obs_sequence_list = 'obs_coverage_list.txt'
   obs_of_interest   = 'METAR_U_10_METER_WIND',
                       'METAR_V_10_METER_WIND'
   textfile_out      = 'obsdef_mask.txt'
   netcdf_out        = 'obsdef_mask.nc'
   calendar          = 'Gregorian'
   first_analysis    =  2003, 1, 1, 0, 0, 0
   last_analysis     =  2003, 1, 2, 0, 0, 0
   forecast_length_days          = 1
   forecast_length_seconds       = 0
   verification_interval_seconds = 21600
   temporal_coverage_percent     = 100.0
   lonlim1    =    0.0
   lonlim2    =  360.0
   latlim1    =  -90.0
   latlim2    =   90.0
   verbose    = .false.
   /
```

6.190.3 Other modules used

```
assim_model_mod
types_mod
location_mod
model_mod
null_mpi_utilities_mod
obs_def_mod
obs_kind_mod
obs_sequence_mod
random_seq_mod
time_manager_mod
utilities_mod
```

6.190.4 Files

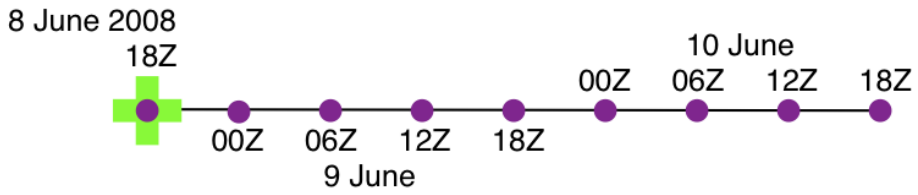
- `input.nml` is used for *obs_seq_coverage_nml*
- A text file containing the metadata for the observations to be used for forecast evaluation is created. This file is subsequently required by *program obs_selection* to subset the set of input observation sequence files into a single observation sequence file (`obs_seq.evaluate`) for the forecast step. (`obsdef_mask.txt` is the default name)
- A netCDF file containing the metadata for a much larger set of observations that may be used is created. This file is subsequently required by *program obs_seq_coverage* to define the desired times and locations for the verification. (`obsdef_mask.nc` is the default name)

6.190.5 Usage

`obs_seq_coverage` is built in .../DART/models/*your_model*/work, in the same way as the other DART components. There is no requirement on the reporting time/frequence of the candidate voxels. Once the verification times have been defined, the observation **closest in time** to the verification time is selected, the others are ignored. Only observations within half the verification interval are eligible to be considered "close".

A word about vertical levels. If the desired observation type has UNDEFINED or SURFACE for the vertical coordinate system, there is no concern about trying to match the vertical. If the desired observation types use PRESSURE; the following 14 levels are used as the standard levels: 1000, 925, 850, 700, 500, 400, 300, 250, 200, 150, 100, 70, 50, 10 (all hPa). **No other vertical coordinate system is supported.**

Example: a single 48-hour forecast that is evaluated every 6 hours



In this example, we are generating an `obsdef_mask.txt` file for a single forecast. All the required input observation sequence filenames will be contained in a file referenced by the *obs_sequence_list* variable. We'll also restrict the observations to a specific rectangular (in Lat/Lon) region at a particular level. It is convenient to turn on the verbose option the first time to get a feel for the logic. Here are the namelist settings if you want to verify the METAR_U_10_METER_WIND and METAR_V_10_METER_WIND observations over the entire globe every 6 hours for 2 days starting 18Z 8 Jun 2008:

```
&obs_seq_coverage_nml
   obs_sequences      = ''
   obs_sequence_list  = 'obs_file_list.txt'
   obs_of_interest    = 'METAR_U_10_METER_WIND',
                        'METAR_V_10_METER_WIND'
   textfile_out       = 'obsdef_mask.txt'
   netcdf_out         = 'obsdef_mask.nc'
   calendar           = 'Gregorian'
   first_analysis     =  2008, 6, 8, 18, 0, 0
   last_analysis      =  2008, 6, 8, 18, 0, 0
   forecast_length_days          = 2
   forecast_length_seconds       = 0
   verification_interval_seconds = 21600
   temporal_coverage_percent     = 100.0
   lonlim1            =    0.0
   lonlim2            =  360.0
   latlim1            =  -90.0
   latlim2            =   90.0
   verbose            = .true.
   /
```

The first step is to create a file containing the list of observation sequence files you want to use. This can be done with the unix command 'ls' with the -1 option (that's a number one) to put one file per line, particularly if the files are organized in a nice fashion. If your observation sequence are organized like this:

```
/Exp1/Dir20080101/obs_seq.final
/Exp1/Dir20080102/obs_seq.final
/Exp1/Dir20080103/obs_seq.final
...
/Exp1/Dir20081231/obs_seq.final
```

then

ls -1 /Exp1/Dir*/obs_seq.final > obs_file_list.txt

creates the desired file. Then, simply run obs_seq_coverage - you may want to save the run-time output to a file. It is convenient to turn on the verbose option the first time. Here is a portion of the run-time output:

```
[thoar@mirage2 work]$ ./obs_seq_coverage | & tee my.log
 Starting program obs_seq_coverage
 Initializing the utilities module.
 Trying to log to unit           10
 Trying to open file dart_log.out

 ---------------------------------------
 Starting ... at YYYY MM DD HH MM SS =
                 2011  2 22 13 15  2
 Program obs_seq_coverage
 ---------------------------------------

 set_nml_output Echo NML values to log file only
 Trying to open namelist log dart_log.nml
 location_mod: Ignoring vertical when computing distances; horizontal only
 ---------------------------------------------------


 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RADIOSONDE_TEMPERATURE
 RADIOSONDE_U_WIND_COMPONENT
 RADIOSONDE_V_WIND_COMPONENT
 SAT_U_WIND_COMPONENT
 SAT_V_WIND_COMPONENT
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 RADIOSONDE_SPECIFIC_HUMIDITY
 ---------------------------------------------------

 METAR_U_10_METER_WIND is type           36
 METAR_V_10_METER_WIND is type           37

 There are            9  verification times per forecast.
 There are            1  supported forecasts.
 There are            9  total times we need observations.

 At least           9  observations times are required at:
 verification #            1 at 2008 Jun 08 18:00:00
 verification #            2 at 2008 Jun 09 00:00:00
 verification #            3 at 2008 Jun 09 06:00:00
 verification #            4 at 2008 Jun 09 12:00:00
 verification #            5 at 2008 Jun 09 18:00:00
```

(continues on next page)

(continued from previous page)

```
 verification #            6 at 2008 Jun 10 00:00:00
 verification #            7 at 2008 Jun 10 06:00:00
 verification #            8 at 2008 Jun 10 12:00:00
 verification #            9 at 2008 Jun 10 18:00:00

 obs_seq_coverage  opening obs_seq.final.2008060818
 QC index           1  NCEP QC index
 QC index           2  DART quality control

First observation time day=148812, sec=64380
First observation date 2008 Jun 08 17:53:00
 Processing obs        10000  of        84691
 Processing obs        20000  of        84691
 Processing obs        30000  of        84691
 Processing obs        40000  of        84691
 Processing obs        50000  of        84691
 Processing obs        60000  of        84691
 Processing obs        70000  of        84691
 Processing obs        80000  of        84691
 obs_seq_coverage  doneDONEdoneDONE does not exist. Finishing up.

 There were          442  voxels matching the input criterion.
...
```

Discussion

Note that the values of `ASSIMILATE_THESE_OBS_TYPES` and `EVALUATE_THESE_OBS_TYPES` are completely irrelevant - since we're not actually doing an assimilation. The **BIG** difference between the two output files is that `obsdef_mask.txt` contains the metadata for just the matching observations while `obsdef_mask.nc` contains the metadata for all candidate locations as well as a lot of information about the desired verification times. It is possible to explore `obsdef_mask.nc` to review the selection criteria to include observations/"voxels" that do not perfectly match the original selection criteria.

Now that you have the `obsdef_mask.nc`, you can explore it with ncdump.

```
netcdf obsdef_mask {
dimensions:
        voxel = UNLIMITED ; // (512 currently)
        time = 9 ;
        analysisT = 1 ;
        forecast_lead = 9 ;
        nlevels = 14 ;
        linelen = 256 ;
        nlines = 446 ;
        stringlength = 32 ;
        location = 3 ;
variables:
        int voxel(voxel) ;
                voxel:long_name = "desired voxel flag" ;
                voxel:description = "1 == good voxel" ;
```

(continues on next page)

(continued from previous page)

```
        double time(time) ;
                time:long_name = "verification time" ;
                time:units = "days since 1601-1-1" ;
                time:calendar = "GREGORIAN" ;
        double analysisT(analysisT) ;
                analysisT:long_name = "analysis (start) time of each forecast" ;
                analysisT:units = "days since 1601-1-1" ;
                analysisT:calendar = "GREGORIAN" ;
        int forecast_lead(forecast_lead) ;
                forecast_lead:long_name = "current forecast length" ;
                forecast_lead:units = "seconds" ;
        double verification_times(analysisT, forecast_lead) ;
                verification_times:long_name = "verification times during each forecast␣
↪run" ;
                verification_times:units = "days since 1601-1-1" ;
                verification_times:calendar = "GREGORIAN" ;
                verification_times:rows = "each forecast" ;
                verification_times:cols = "each verification time" ;
        float mandatory_level(nlevels) ;
                mandatory_level:long_name = "mandatory pressure levels" ;
                mandatory_level:units = "Pa" ;
        char namelist(nlines, linelen) ;
                namelist:long_name = "input.nml contents" ;
        char obs_type(voxel, stringlength) ;
                obs_type:long_name = "observation type string at this voxel" ;
        double location(voxel, location) ;
                location:description = "location coordinates" ;
                location:location_type = "loc3Dsphere" ;
                location:long_name = "threed sphere locations: lon, lat, vertical" ;
                location:storage_order = "Lon Lat Vertical" ;
                location:units = "degrees degrees which_vert" ;
        int which_vert(voxel) ;
                which_vert:long_name = "vertical coordinate system code" ;
                which_vert:VERTISUNDEF = -2 ;
                which_vert:VERTISSURFACE = -1 ;
                which_vert:VERTISLEVEL = 1 ;
                which_vert:VERTISPRESSURE = 2 ;
                which_vert:VERTISHEIGHT = 3 ;
                which_vert:VERTISSCALEHEIGHT = 4 ;
        int ntimes(voxel) ;
                ntimes:long_name = "number of observation times at this voxel" ;
        double first_time(voxel) ;
                first_time:long_name = "first valid observation time at this voxel" ;
                first_time:units = "days since 1601-1-1" ;
                first_time:calendar = "GREGORIAN" ;
        double last_time(voxel) ;
                last_time:long_name = "last valid observation time at this voxel" ;
                last_time:units = "days since 1601-1-1" ;
                last_time:calendar = "GREGORIAN" ;
        double ReportTime(voxel, time) ;
                ReportTime:long_name = "time of observation" ;
                ReportTime:units = "days since 1601-1-1" ;
```

(continues on next page)

(continued from previous page)

```
                ReportTime:calendar = "GREGORIAN" ;
                ReportTime:missing_value = 0. ;
                ReportTime:_FillValue = 0. ;

// global attributes:
                :creation_date = "YYYY MM DD HH MM SS = 2011 03 01 09 28 40" ;
                :obs_seq_coverage_source = "$URL$" ;
                :obs_seq_coverage_revision = "$Revision$" ;
                :obs_seq_coverage_revdate = "$Date$" ;
                :min_steps_required = 9 ;
                :forecast_length_days = 2 ;
                :forecast_length_seconds = 0 ;
                :verification_interval_seconds = 21600 ;
                :obs_of_interest_001 = "METAR_U_10_METER_WIND" ;
                :obs_of_interest_002 = "METAR_V_10_METER_WIND" ;
                :obs_seq_file_001 = "obs_seq.final.2008060818" ;
data:

 time = 148812.75, 148813, 148813.25, 148813.5, 148813.75, 148814, 148814.25,
    148814.5, 148814.75 ;

 forecast_lead = 0, 21600, 43200, 64800, 86400, 108000, 129600, 151200, 172800 ;
}
```

The first thing to note is that there are more voxels (512) than reported during the run-time output (442). Typically, there will be many more voxels in the netCDF file than will meet the selection criteria - but this is just an example. Some of the voxels in the netCDF file do not meet the selection criteria - meaning they do not have observations at all 9 required times. Furthermore, there are 512 locations for ALL of the desired observation types. In keeping with the DART philosophy of scalar observations, each observation type gets a separate voxel. There are **not** 512 METAR_U_10_METER_WIND observations and 512 METAR_V_10_METER_WIND observations. There are N METAR_U_10_METER_WIND observations and M METAR_V_10_METER_WIND observations where N+M = 512. And only 442 of them have observations at all the times required for the verification. Dump the *obs_type* variable to see what voxel has what observation type.

The *voxel* variable is fundamentally a flag that indicates if the station has all of the desired verification times. Combine that information with the *obs_type* and *location* to determine where your verifications of any particular observation type will take place.

Now that you have the `obsdef_mask.txt`, you can run *program obs_selection* to subset the observation sequence files into one compact file to use in your ensemble forecast.

6.190.6 References

- none - but this seems like a good place to start: The Centre for Australian Weather and Climate Research - Forecast Verification Issues, Methods and FAQ

6.191 PROGRAM advance_time

6.191.1 Overview

Provides a shell-scripting-friendly way to increment and decrement calendar dates and times. The code uses the standard DART time manager for all time calculations.

A date, an increment or decrement, and an optional output formatting flag are read from standard input. Increments can be days, hours, minutes, or seconds. The accuracy is to the second. The resulting output time string is echoed to standard output. For example:

```
echo 2007073012 12 | advance_time
```

will output the string 2007073100. It uses the Gregorian calendar and will roll over month and year boundaries, both going forward and backwards in time. See the Usage section below for more examples of use.

The program is general purpose, but based on a time program distributed with the WRF model. This is the reason there are a few WRF specific options, for example the '-w' flag outputs a date string in a WRF-specific format, useful for creating WRF filenames.

The program does require that an 'input.nml' namelist file exist in the current directory, and at least a &utilities_nml namelist (which can be empty) exists.

6.191.2 Usage

Interface identical to the `wrf/WRF_DART_utilities/advance_cymdh`, except for reading the arg line from standard input, to be more portable since `iargc()` is nonstandard across different fortran implementations.

- default numeric increment is hours
- has accuracy down to second
- can use day/hour/minute/second (with/without +/- sign) to advance time
- can digest various input date format if it still has the right order (ie. cc yy mm dd hh nn ss)
- can digest flexible time increment
- can output in wrf date format (ccyy-mm-dd_hh:nn:ss)
- can specify output date format
- can output Julian day
- can output Gregorian days and seconds (since year 1601)

Some examples:

```
advance 12 h:
  echo 20070730      12         | advance_time

back 1 day 2 hours 30 minutes and 30 seconds:
  echo 2007073012   -1d2h30m30s | advance_time

back 3 hours 30 minutes less 1 second:
  echo 2007073012    1s-3h30m   | advance_time

advance 2 days and 1 second, output in wrf date format :
  echo 200707301200  2d1s -w    | advance_time
  echo 2007-07-30_12:00:00 2d1s -w  | advance_time
  echo 200707301200  2d1s -f ccyy-mm-dd_hh:nn:ss | advance_time

advance 120 h, and print year and Julian day:
  echo 2007073006    120 -j     | advance_time

advance 120 h, print year, Julian day, hour, minute and second:
  echo 2007073006    120 -J     | advance_time

print Gregorian day and second (since year 1601):
  echo 2007073006    0 -g       | advance_time
```

6.191.3 Modules used

```
utilities_mod
time_manager_mod
parse_args_mod
```

6.191.4 Namelist

No namelist is currently defined for `advance_time`.

6.191.5 Files

- input.nml

6.192 program `model_mod_check`

6.192.1 Overview

`model_mod_check` tests some of the more fundamental routines in any `model_mod`. This is intended to be used when adding a new model to DART - test the pieces as they are written. As such, this program is meant to be hacked up and customized to your own purpose. Right now, it reads in model netCDF file(s) - one per domain/nest/whatever - and writes out files, queries the metdata, etc. It also exercises `static_init_model()`, which is the first routine to get right ...

6.192.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_mod_check_nml
   num_ens                = 1
   single_file            = .FALSE.
   input_state_files      = 'null'
   output_state_files     = 'null'
   all_metadata_file      = 'metadata.txt'

   test1thru              = 7
   run_tests              = -1

   x_ind                  = -1
   loc_of_interest        = -1.0, -1.0, -1.0
   quantity_of_interest   = 'NONE'

   interp_test_dlon       = 10.0
   interp_test_dlat       = 10.0
   interp_test_dvert      = 10.0

   interp_test_lonrange   = 0.0, 120.0
   interp_test_latrange   = 0.0, 120.0
   interp_test_vertrange  = 0.0, 100.0

   interp_test_dx         = -888888.0
   interp_test_dy         = -888888.0
   interp_test_dz         = -888888.0

   interp_test_xrange     = -888888.0, -888888.0
   interp_test_yrange     = -888888.0, -888888.0
   interp_test_zrange     = -888888.0, -888888.0

   interp_test_vertcoord  = 'VERTISHEIGHT'
   verbose                = .FALSE.
   /
```

Item	Type	Description
num_ens	integer	Provided for future use. Must be 1. Ultimately, The number of ensemble members you would like to read in for testing.
single_file	logical	If .TRUE. all members are stored in a single restart file.
input_state_files(:)	character(len=256)	The name(s) of the NetCDF file(s) containing the model states, one per domain. If num_ens > 1 and not single_file, specify a filename for each ensemble member (num_ens). If you have both multiple ensemble members in separate files AND multiple domains, specify all the ensemble member filenames for domain 1, then all the ensemble member filenames for domain 2, etc.
output_state_files(:)	character(len=256)	The name(s) of the output NetCDF file(s) for testing IO, one per domain. If num_ens > 1 and not single_file, specify a filename for each ensemble member (num_ens). If you have both multiple ensemble members in separate files AND multiple domains, specify all the ensemble member filenames for domain 1, then all the ensemble member filenames for domain 2, etc.
all_metadata_file	character(len=256)	Test 6 produces an exhaustive list of metadata for EVERY element in the DART state vector. The metadata get written to this file name.
x_ind	integer(i8)	An integer index into the DART state vector. This will be used to test the metadata routines. Answers questions about location, what variable type is stored there, etc.
loc_of_interest	real(r8), dimension(3)	The lat/lon/level for a **particular** location. Used in Test 4, the single-point interpolation test. Indirectly tests the routine to find the closest gridpoint.
quantity_of_interest	character(len=32)	Specifies the QUANTITY of the model state to use in Tests 4, 5, and 7.
interp_test_dlon	real(r8)	The distance (measured in degrees) on the longitude interpolation grid. Ignored if interpolating with cartesian coordinates. Used in Test 5.
interp_test_dlat	real(r8)	The distance (measured in degrees) on the latitude interpolation grid. Ignored if interpolating with cartesian coordinates. Used in Test 5.
interp_test_dvert	real(r8)	The distance (measured in interp_vertcoord) on the vertical interpolation grid. Ignored if interpolating with cartesian coordinates. Used in Test 5.

A more typical namelist for a single ensemble member for a model with an outer grid and a single nested grid is shown below.

```
&model_mod_check_nml
   input_state_files     = 'dart_vector1.nc','dart_vector2.nc'
   output_state_files    = 'check_me1.nc', 'check_me2.nc'
   all_metadata_file     = 'metadata.txt'
   verbose               = .TRUE.
   test1thru             = 5
   run_tests             = -1
   loc_of_interest       = 243.72386169, 52.78578186, 10.0
   x_ind                 = 12666739
   quantity_of_interest  = 'QTY_POTENTIAL_TEMPERATURE'
   interp_test_lonrange  = 144.0, 326.0
   interp_test_dlon      = 1.0
   interp_test_latrange  = -5.0, 80.0
   interp_test_dlat      = 1.0
   interp_test_vertrange = 100.0, 11000.0
   interp_test_dvert     = 200.0
   interp_test_vertcoord = 'VERTISHEIGHT'
  /
```

6.192.3 Other modules used

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/location/utilities/default_location_mod.f90
assimilation_code/location/utilities/location_io_mod.f90
assimilation_code/modules/assimilation/adaptive_inflate_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/assimilation/assim_tools_mod.f90
assimilation_code/modules/assimilation/cov_cutoff_mod.f90
assimilation_code/modules/assimilation/filter_mod.f90
assimilation_code/modules/assimilation/obs_model_mod.f90
assimilation_code/modules/assimilation/quality_control_mod.f90
assimilation_code/modules/assimilation/reg_factor_mod.f90
assimilation_code/modules/assimilation/sampling_error_correction_mod.f90
assimilation_code/modules/assimilation/smoother_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/io/direct_netcdf_mod.f90
assimilation_code/modules/io/io_filenames_mod.f90
assimilation_code/modules/io/state_structure_mod.f90
assimilation_code/modules/io/state_vector_io_mod.f90
assimilation_code/modules/observations/forward_operator_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/distributed_state_mod.f90
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/netcdf_utilities_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/null_win_mod.f90
assimilation_code/modules/utilities/obs_impact_mod.f90
assimilation_code/modules/utilities/options_mod.f90
```

(continues on next page)

(continued from previous page)

```
assimilation_code/modules/utilities/parse_args_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
assimilation_code/programs/model_mod_check/model_mod_check.f90
models/your_model_here/model_mod.f90
models/model_mod_tools/test_interpolate_threed_sphere.f90
models/model_mod_tools/model_check_utilities_mod.f90
models/utilities/default_model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/forward_operators/obs_def_utilities_mod.f90
```

Items highlighted may change based on which model is being tested.

6.192.4 Files

- `input.nml` is used for `model_mod_check_nml`
- The `"input_state_files"` can either be a single file containing multiple restart files, or a single NetCDF restart file. One file per domain.
- The `"output_state_files"` is the output netCDF files from Test 2. Check the attributes, values, etc.
- `check_me_interptest.nc` and `check_me_interptest.m` are the result of Test 5.
- `"all_metadata_file"` is the run-time output of Test 6.

6.192.5 Usage

Normal circumstances indicate that you are trying to put a new model into DART, so to be able to build and run `model_mod_check`, you will need to create a `path_names_model_mod_check` file with the following contents:

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/location/utilities/default_location_mod.f90
assimilation_code/location/utilities/location_io_mod.f90
assimilation_code/modules/assimilation/adaptive_inflate_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/assimilation/assim_tools_mod.f90
assimilation_code/modules/assimilation/cov_cutoff_mod.f90
assimilation_code/modules/assimilation/filter_mod.f90
assimilation_code/modules/assimilation/obs_model_mod.f90
assimilation_code/modules/assimilation/quality_control_mod.f90
assimilation_code/modules/assimilation/reg_factor_mod.f90
assimilation_code/modules/assimilation/sampling_error_correction_mod.f90
assimilation_code/modules/assimilation/smoother_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/io/direct_netcdf_mod.f90
assimilation_code/modules/io/io_filenames_mod.f90
assimilation_code/modules/io/state_structure_mod.f90
assimilation_code/modules/io/state_vector_io_mod.f90
```

(continues on next page)

(continued from previous page)

```
assimilation_code/modules/observations/forward_operator_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/distributed_state_mod.f90
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/netcdf_utilities_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/null_win_mod.f90
assimilation_code/modules/utilities/obs_impact_mod.f90
assimilation_code/modules/utilities/options_mod.f90
assimilation_code/modules/utilities/parse_args_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
assimilation_code/programs/model_mod_check/model_mod_check.f90
models/your_model_here/model_mod.f90
models/model_mod_tools/test_interpolate_threed_sphere.f90
models/utilities/default_model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/forward_operators/obs_def_utilities_mod.f90
```

as well as a `mkmf_model_mod_check` script. You should be able to look at any other `mkmf_xxxx` script and figure out what to change. Once they exist:

```
[~/DART/models/yourmodel/work] % csh mkmf_model_mod_check
[~/DART/models/yourmodel/work] % make
[~/DART/models/yourmodel/work] % ./model_mod_check
```

Unlike other DART components, you are expected to modify `model_mod_check.f90` to suit your needs as you develop your `model_mod`. The code is roughly divided into the following categories:

1. Check the geometry information,
2. Read/write a restart file,
3. Check the construction of the state vector … i.e. the metadata,
4. Interpolate at a single point,
5. Interpolate for a range of points.

Test 0. mandatory

The first test in `model_mod_check` reads the namelist and runs `static_init_model` - which generally sets the geometry of the grid, the number of state variables and their shape, etc. Virtually everything requires knowledge of the grid and state vector, so this block cannot be skipped.

Test 1. checking the geometry information

The first test in `model_mod_check` exercises a basic required interface `get_model_size()`. This also generates a report on the geometry of the grid, the number of state variables and their shape, etc. as well as the total number of elements in the DART state vector.

Test 2. read/writing a restart file

This directly reads and write state variables from the model netCDF file. This is a nice sanity check to make sure that the DART state vector is being read in properly.

Test 3. check the construction of the state vector

It is critical to return the correct metadata for any given index into the DART state vector. This code block tests the two most common features of the metadata. As a bonus, this routine is also quite useful to determine EXACTLY where to place your first test observation. If you test precisely at a grid location, you should be able to really get a handle on debugging your `model_interpolate()` routine.

Test 4. test interpolation on a single point

This tests your model's interpolation routine on a single point and returns the interpolated value. This requires that Test 2 works - it needs a valid model state with data. Test 2 is automatically run if this test is selected.

Test 5. test interpolation on a range of values

This tests your model's interpolation routine on a range of values returns the interpolated grid in `check_me_interptest.nc` and `check_me_interptest.m` which can be read in Matlab and used to visualize the result. This requires that Test 2 works - it needs a valid model state with data. Test 2 is automatically run if this test is selected.

Test 6. exhaustively test the construction of the state vector

This can be a long test, depending on the size of your state vector. This returns the same data as in Test 3 - but *for every element* in the state vector. The metadata are written to a file specified by `all_metadata_file` and `check_me_interptest.m` which can be read in Matlab and used to visualize the result.

Test 7. find the closest gridpoint to a test location

This is a good test to verify that *get_state_meta_data()* and the grid information are correct. Typically, one would put in a location that is actually **on** the grid and see if the correct gridpoint index is returned. Repeat the test with slightly different locations until the next gridpoint is closer. Repeat ...

6.192.6 References

- none

6.193 PROGRAM `closest_member_tool`

6.193.1 Overview

Utility program to compare the ensemble mean to an ensemble of restart files, which can now be run in parallel. The program prints out a sorted order of which members are 'closest' to the mean, where the method used to determine 'close' is selectable by namelist option. It also creates a file with a single number or character string in it, for ease in scripting, which identifies the closest member.

The ensemble mean is computed from the input ensemble. The difference is computed point by point across the ensemble members. There is an option to restrict the computation to just a subset of the entire state vector by listing one or more generic quantities. In this case, only state vector items matching one of these quantities will contribute to the total difference value.

Available methods are:

1 - simple absolute difference:
The absolute value of the difference between each item in the mean vector and the corresponding item in each ensemble member, accumulated over the entire state vector.

2 - normalized absolute difference:
The absolute value of the difference between each item in the mean vector and the corresponding item in each ensemble member normalized by the mean value, accumulated over the entire state vector.

3 - simple RMS difference:
The square root of the accumulated sum of the square of the difference between each item in the mean vector and the corresponding item in each ensemble member.

4 - normalized RMS difference:
The square root of the accumulated sum of the square of the normalized difference between each item in the mean vector and the corresponding item in each ensemble member.

This program could be used to select one or more ensemble members to run a free model forecast forward in time after the assimilation is finished. Each member is an equally likely representation of the model state. Using the ensemble mean may not be the best choice since the mean may not have self-consistent fine-scale structures in the data.

In addition to printing out data about all members to both the console and to the dart log file, this program creates a single output file containing information about the closest member. If the input restart data is in a single file, the output file 'closest_restart' contains a single number which is the ensemble member number. If the input restart data is in separate files, the output file contains the full filename of the closest member, e.g. 'filter_restart.0004' if member 4 is closest. For scripting the contents of this file can be used to copy the corresponding member data and convert it to the model input format for a free forecast, for example.

6.193.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&closest_member_tool_nml
   input_restart_files     = ''
   input_restart_file_list = ''
   output_file_name        = 'closest_restart'
   ens_size                = 1
   difference_method       = 4
   use_only_qtys           = ''
   single_restart_file_in  = .false.
  /
```

Item	Type	Description
input_restart_files	character(len=256), dimension(ens_size x num_domains)	An array of filenames each containing a list DART restart data.
input_restart_file_list	character(len=256), dimension(num_domains)	A file containing a list of filenames for DART restart data, one for each domain.
output_file_name	character(len=256)	This is a file containing the member number that is closest to the ensemble mean.
ens_size	integer	Total number of ensemble members.
difference_method	integer	Select which method is used to compute 'distance' from mean: • 1 = simple absolute difference • 2 = absolute difference normalized by the mean • 3 = simple RMS difference • 4 = RMS of the normalized difference
use_only_quantities	character(len=32)	If unspecified, all items in the state vector contribute to the total difference. If one or more quantities are listed here, only items in the state vector of these quantities contribute to the total difference. These are the generic quantities, such as QTY_TEMPERATURE, QTY_U_WIND_COMPONENT, QTY_DENSITY, etc. and not specific types like RADIOSONDE_TEMPERATURE. Consult the model interface code to determine which possible quantities are returned by the get_state_meta_data() routine.
single_restart_file_in	logical	**Not supported yet.** Contact dart@ucar.edu if you are interested in using this tool with files that contain all ensemble members in a single file.

Below is an example of a typical namelist for the closest_member_tool.

```
&closest_member_tool_nml
   input_restart_files     = ''
   input_restart_file_list = 'restart_list.txt'
   output_file_name        = 'closest_restart.txt'
   ens_size                = 3
   single_restart_file_in  = .false.
   difference_method       = 4
   use_only_qtys           = ''
  /
```

where `restart_list.txt` contains

```
cam_restart_0001.nc
cam_restart_0002.nc
cam_restart_0003.nc
```

Currently `single_restart_file_in` is not supported. This is typically used for simpler models that have built in model advances such as `lorenz_96`.

6.193.3 Files

- inputfile.####.nc (list of restarts to find closest member) -or-
- `restart_list.txt` (a file containing a list of restart files) and,
- `input.nml`

6.193.4 References

- none

6.194 PROGRAM `filter`

6.194.1 Overview

Main program for driving ensemble filter assimilations.

`filter` is a Fortran 90 program, and provides a large number of options for controlling execution behavior and parameter configuration that are driven from its namelist. See the namelist section below for more details. The number of assimilation steps to be done is controlled by the input observation sequence and by the time-stepping capabilities of the model being used in the assimilation.

This overview includes these subsections:

- Program Flow
- Filter Types
- Getting Started
- Free Model Run after Assimilation
- Evaluate a Model State against Observations
- Compare Results with and without Assimilation
- DART Quality Control Values on Output
- Description of Inflation Options
- Detailed Program Flow

See *Welcome to the Data Assimilation Research Testbed* for more documentation, including a discussion of the capabilities of the assimilation system, a diagram of the entire execution cycle, the options and features.

Program flow

The basic execution loop is:

- Read in model initial conditions, observations, set up and initialize
- Until out of observations:
  - Run multiple copies of the model to get forecasts of model state
  - Assimilate all observations in the current time window
  - Repeat
- Write out diagnostic files, restart files, final observation sequence file

The time of the observations in the input observation sequence file controls the length of execution of filter.

For large, parallel models, the execution loop is usually wrapped in an external script which does these additional steps:

- Link to an observation sequence file which contains only observation times within the next assimilation window
- Link any output inflation files from the previous step to be the input files for this step
- Run filter, which will exit after doing the assimilation without trying to advance the model
- Save the output diagnostic files for later
- Advance the N copies of the model using the model scripts or whatever method is appropriate
- Repeat until all data is assimilated

For large models filter is almost always compiled to be a parallel MPI program, and most large models are themselves a parallel program using OpenMP, MPI, or both. MPI programs usually cannot start other MPI programs, so the external script submits both the filter job and the N model advances to a batch system so all run as independent parallel jobs.

The same source code is used for all applications of filter. The code specific to the types of observations and the interface code for the computational model is configured at compile time. The top level directory has been simplified from previous versions to look like :

- `README`
- `COPYRIGHT`
- *assimilation_code*
- *build_templates*
- *diagnostics*
- *documentation*
- *models*
- *observations*

the *assimilation_code* contains all *module* and *program* source code for all of the main programs including filter. Specifically in the modules directory there is a `filter_mod.f90` which contains the source for the filter main program. Each model has a separate directory under DART/models, and under each model is a work directory where the code is compiled and can be run for testing. Generally when a full-size experiment is done the executables are copied to a different location - e.g. scratch space on a large filesystem - since the data files for 10s to 100s of copies of a model can get very large.

Directories expected to be modified

DART is distributed as a toolkit/library/facility that can be used as-is with the existing models and observations, but is also designed so that users can add new models, new observation types and forward operators, and new assimilation algorithms.

The locations in the DART code tree which are intended to be modified by users are:

New Models
Add a new directory in the `models` subdirectory. Copy (recursively, e.g. `cp -r`) the contents of the `template` directory and modify from there. Note that the `model_mod.f90` file in the template dir is appropriate for small models; for large geophysical models see the `full_model_mod.f90` file and also examine other model directories for ideas. See additional documentation in the *MODULE model_mod* documentation, and the DART web pages on adding new models.

New Observation Platforms
To convert observations from other formats to DART format, add a new directory in the `observations/obs_converters` subdirectory and populate it with converter code.

New Observation Types and Forward Operators
Define a new type (a measurement from an observing platform) via a file in the `observations/forward_operators` subdirectory. If the forward operator is more complicated than directly interpolating a field in the model state, this is where the code for that goes. See additional documentation in the *MODULE obs_def_mod* documentation, and the DART web pages on adding new types. Adding a new type may require adding a new `generic kind`, which is documented in *MODULE obs_kind_mod*.

New Assimilation Algorithms
If you want to try out a different filter type modify the filter code in the `assim_tools_mod.f90` file. See the *MODULE assim_tools_mod* documentation.

Detailed program execution flow

The Manhattan release of DART includes state space output expanded from the previous two stages (Prior and Posterior) to up to six (input, forecast, preassim, postassim, analysis, and output). This makes it possible to examine the states with and without either kind of inflation, as described below. In addition, the state space vectors are each written to a separate NetCDF file: `${stage}_mean.nc, ${stage}_sd.nc, ${stage}_member_####.nc` . The detailed execution flow inside the filter program is:

- Read in observations.
- Read in state vectors from model netcdf restart files.
- Initialize inflation fields, possibly reading netcdf restart files.
- If requested, initialize and write to "input" netcdf diagnostic files.
- Trim off any observations if start/stop times specified.
- Begin main assimilation loop:
  - Check model time vs observation times:
    - If current assimilation window is earlier than model time, error.
    - If current assimilation window includes model time, begin assimilating.
    - If current assimilation window is later than model time, advance model:
      - Write out current state vectors for all ensemble members.
      - Advance the model by subroutine call or by shell script:

            - Tell the model to run up to the requested time.
            - Read in new state vectors from netcdf files for all ensemble members.
        - Apply prior inflation if requested.
        - Compute ensemble of prior observation values with forward operators.
        - If requested, compute and write the "preassim" netcdf diagnostic files. This is AFTER any prior inflation has been applied.
        - Compute prior observation space diagnostics.
        - Assimilate all observations in this window:
            * Get all obs locations and kinds.
            * Get all state vector locations and kinds.
            * For each observation:
                - Compute the observation increments.
                - Find all other obs and states within localization radius.
                - Compute the covariance between obs and state variables.
                - Apply increments to state variables weighted by correlation values.
                - Apply increments to any remaining unassimilated observations.
                - Loop until all observations in window processed.
        - If requested, compute and write the "postassim" netcdf diagnostic files (members, mean, spread). This is BEFORE any posterior inflation has been applied.
        - Apply posterior inflation if requested.
        - Compute ensemble of posterior observation values with forward operators.
        - Compute posterior observation space diagnostics.
        - If requested, compute and write out the "output" netcdf diagnostic files (members, mean, spread). This is AFTER any posterior inflation has been applied.
        - Loop until all observations in input file processed.
- Close diagnostic files.
- Write out final observation sequence file.
- Write out inflation restart files if requested.
- Write out final state vectors to model restart files if requested.
- Release memory for state vector and observation ensemble members.

6.194.2 Namelist

See the filter namelist page for a detailed description of all `&filter_nml` variables. This namelist is read from the file `input.nml`.

6.194.3 Modules used

```
mpi_utilities_mod
filter_mod
```

Note that filter_mod.f90 uses many more modules.

6.194.4 Files

See Detailed Program Flow for a short description of DART's new 'stages'. In addition, the Manhattan release simplifies some namelists by replacing many user-settable file names with hardwired filenames. Files can then be renamed in the run scripts to suit the user's needs.

- input ensemble member states; from *&filter_nml :: input_state_files* or *input_state_file_list*
- output ensemble member states; to *&filter_nml :: output_state_files* or *output_state_file_list*
- input observation sequence file; from `&filter_nml :: obs_sequence_in_name`
- output observation sequence file; from `&filter_nml :: obs_sequence_out_name`
- output state space diagnostics files; `${stage}_mean.nc, ${stage}_sd.nc,` where stage = {input,forecast,preassim,postassim,analysis,output}
- input state space inflation data (if enabled); from `input_{prior,post}inf_{mean,sd}.nc.`
- output state space inflation data (if enabled); to `${stage}_{prior,post}inf_{mean,sd}.nc.`, where stage "input"
- input.nml, to read &filter_nml

6.194.5 References

- Anderson, J. L., 2001: An Ensemble Adjustment Kalman Filter for Data Assimilation. Mon. Wea. Rev., 129, 2884-2903. doi: 10.1175/1520-0493(2001)129<2884:AEAKFF>2.0.CO;2
- Anderson, J. L., 2003: A Local Least Squares Framework for Ensemble Filtering. Mon. Wea. Rev., 131, 634-642. doi: 10.1175/1520-0493(2003)131<0634:ALLSFF>2.0.CO;2
- Anderson, J. L., 2007: An adaptive covariance inflation error correction algorithm for ensemble filters. Tellus A, 59, 210-224. doi: 10.1111/j.1600-0870.2006.00216.x
- Anderson, J. L., 2007: Exploring the need for localization in ensemble data assimilation using a hierarchical ensemble filter. Physica D, 230, 99-111. doi:10.1016/j.physd.2006.02.011
- Anderson, J., Collins, N., 2007: Scalable Implementations of Ensemble Filter Algorithms for Data Assimilation. Journal of Atmospheric and Oceanic Technology, 24, 1452-1463. doi: 10.1175/JTECH2049.1
- Anderson, J. L., 2009: Spatially and temporally varying adaptive covariance inflation for ensemble filters. Tellus A, 61, 72-83. doi: 10.1111/j.1600-0870.2008.00361.x

- Anderson, J., T. Hoar, K. Raeder, H. Liu, N. Collins, R. Torn, and A. Arellano, 2009: The Data Assimilation Research Testbed: A Community Facility. Bull. Amer. Meteor. Soc., 90, 1283-1296. doi: 10.1175/2009BAMS2618.1
- Anderson, J. L., 2010: A Non-Gaussian Ensemble Filter Update for Data Assimilation. Mon. Wea. Rev., 139, 4186-4198. doi: 10.1175/2010MWR3253.1
- Anderson, J. L., 2011: Localization and Sampling Error Correction in Ensemble Kalman Filter Data Assimilation. Submitted for publication, Jan 2011. Contact author.

6.195 program obs_keep_a_few

6.195.1 Overview

This program creates an output observation sequence (obs_seq) file that is shorter than the input obs_seq file. There are two ways to restrict the number of observations copied to the output: the total number of observations regardless of observation type, or up to N observations of each type. Observations in an obs_seq file are processed in time order so the observations with the earliest timestamps will be copied.

Set either limit to -1 to disable it. If both the maximum count per type and maximum total count are given the copying stops when the first limit is reached.

If you want to subset an obs_seq file starting at a later time see the *program obs_sequence_tool* for subsetting by time and then use this tool on the output. That tool also allows you to subset by obs type, location, data value, and a variety of other options.

The `obs_keep_a_few` program only subsets by numbers of observations. It is expected to be useful when prototyping experiments so the run time is short, or for debugging or testing. Setting a limit per type ensures you have up to N of each type of observation present in the output file.

Identity observations are all considered to be the same identity "observation type" by this tool.

6.195.2 Other modules used

```
types_mod
utilities_mod
location_mod
obs_def_mod
obs_kind_mod
time_manager_mod
obs_sequence_mod
```

6.195.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_keep_a_few_nml
   filename_in        = ''
   filename_out       = ''
   max_count_per_type = 10
```

(continues on next page)

(continued from previous page)

```
   max_total_count     = -1
   print_only          = .false.
   calendar            = 'Gregorian'
   /
```

Item	Type	Description
filename_in	character(len=256)	Name of the observation sequence file to read.
filename_out	character(len=256)	Name of the observation sequence file to create. An existing file will be overwritten.
max_count_per_type	integer	The first N observations of each different type will be copied to the output file. Observation sequence files are processed in time order so these will be the ones with the earliest time stamps relative to other observations of this same type. Set to -1 to disable this limit.
max_total_count	integer	If greater than 0, sets the upper limit on the total number of observations to be copied to the output file regardless of type. The program quits when either this limit is reached or when there are N of each different obs type in the output. Set to -1 to disable.
print_only	logical	If true, does all the work and prints out what the output file would have in it (timestamps and counts of each obs type) but doesn't create the output file.
calendar	character(len=256)	Name of the DART calendar type to use. Generally 'Gregorian' or 'No calendar'. See the DART time manager for more options. Only controls the formatting of how the times in the output summary messages are displayed.

6.195.4 Files

- filename_in is read.
- filename_out is written.

6.195.5 References

- none

6.196 program create_obs_sequence

6.196.1 Overview

This program creates an observation sequence file using values read from standard input. It is typically used to create synthetic observations, or shorter sequences of observations (although there is no limit on the number of observations). For creating observation sequence files directly from large, real-world observation datasets, see the observations directory.

This program can be run interactively (input from a terminal), or input files can be created with a text editor, perl or matlab script, or any other convenient method, and then run with standard input redirected from this file. The latter method is most commonly used to create larger observation sequence files for perfect model applications.

The program can create complete observation sequences ready to be assimilated, or it can create observations with only partial data which is later filled in by another program. Each observation needs to have a type, location, time, expected error, and optionally a data value and/or a quality control indicator. For perfect model applications, it is usually convenient to define 0 quality control fields and 0 copies of the data for each observation. The output of create_obs_sequence can be read by *program perfect_model_obs* which will then create a synthetic (perfect_model) observation sequence complete with two copies of the data for each observation: the observed value and the 'true' value.

Another common approach for perfect model applications is to use create_obs_sequence to define a set of observation locations and types, and where observations will be repeatedly sampled in time. When running create_obs_sequence, specify a single observation for each different location and type, with 0 copies of data and giving all the observations the same time. Then the program *program create_fixed_network_seq* can read the output of create_obs_sequence and create an observation sequence file that will contain the set of input observations at a number of different times. This models a fixed observation station, observing the system at some frequency in time.

This program can also create what are called "identity observations". These are observations located directly at one of the state variables, so that computing the value requires no model interpolation but simply returns the actual state variable value. To specify these types of observations, the convention is to put in the negative index number for the offset of that state variable in the state vector. By specifying the index both the observation kind and location are defined by the kind and location of that state variable.

The types of observations which can be created by this program is controlled by the observation types built into the source files created by the *PROGRAM preprocess* program. The preprocess namelist sets the available observation types, and must be run each time it is changed, and then the create_obs_sequence program must be recompiled to incorporate the updated source files.

6.196.2 Other modules used

```
utilities_mod
obs_sequence_mod
assim_model_mod
```

6.196.3 Namelist

This program does not use a namelist. All user input is prompted for at the command line.

6.196.4 Files

- A file containing the output sequence is created. (`set_def.out` is the recommended name)

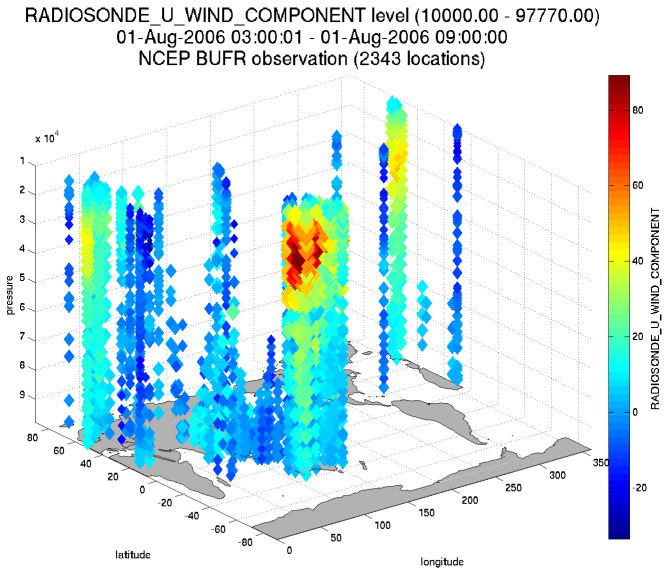
6.196.5 References

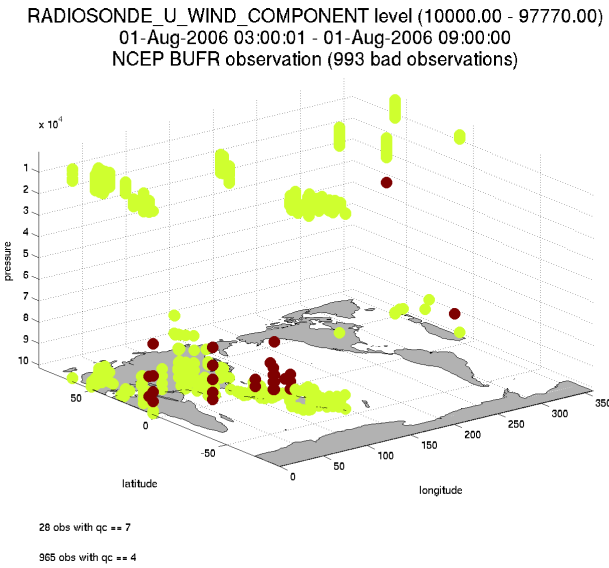
- none

6.197 PROGRAM `obs_seq_to_netcdf`

6.197.1 Overview

`obs_seq_to_netcdf` is a routine to extract the observation components from observation sequence files and write out netCDF files that can be easily digested by other applications. This routine will allow you to plot the spatial distribution of the observations and be able to discern which observations were assimilated or rejected, for example. Here are some graphics from `DART/diagnostics/matlab/plot_obs_netcdf.m`.





The intent is that user input is queried and a series of output files - one per assimilation cycle - will contain the observations for that cycle. It is hoped this will be useful for experiment design or, perhaps, debugging. This routine is also

the first to use the new `schedule_mod` module which will ultimately control the temporal aspects of the assimilations (i.e. the assimilation schedule).

There is also a facility for exploring the spatial distributions of quantities like bias between the ensemble mean and the observations: `DART/diagnostics/matlab/plot_obs_netcdf_diffs.m`. Required namelist interfaces `&obs_seq_to_netcdf` and `&schedule_nml` are read from file `input.nml`.

What's on the horizon ..

`obs_seq_to_netcdf` is a step toward encoding our observations in netCDF files. *The dependence on the* `threed_sphere/location_mod.f90` *has been removed. This program will work with any* `location_mod.f90`. Also, this program no longer tries to construct 'wind' observations from horizontal components since the program really should be faithful to preserving exactly what is in the input file. i.e. We're not making stuff up.

There are several Matlab scripts that understand how to read and plot observation data in netcdf format. See the `link_obs.m` script that creates several linked figures with the ability to 'brush' data in one view and have those selected data (and attributes) get highlighted in the other views.

6.197.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_seq_to_netcdf_nml
   obs_sequence_name = 'obs_seq.final',
   obs_sequence_list = '',
   append_to_netcdf  = .false.,
   lonlim1    =    0.0,
   lonlim2    =  360.0,
   latlim1    =  -90.0,
   latlim2    =   90.0,
   verbose    = .false.
/
```

The allowable ranges for the region boundaries are: latitude [-90.,90], longitude [0.,360.] ... but it is possible to specify a region that spans the dateline by specifying the `lonlim2` to be less than `lonlim1`.

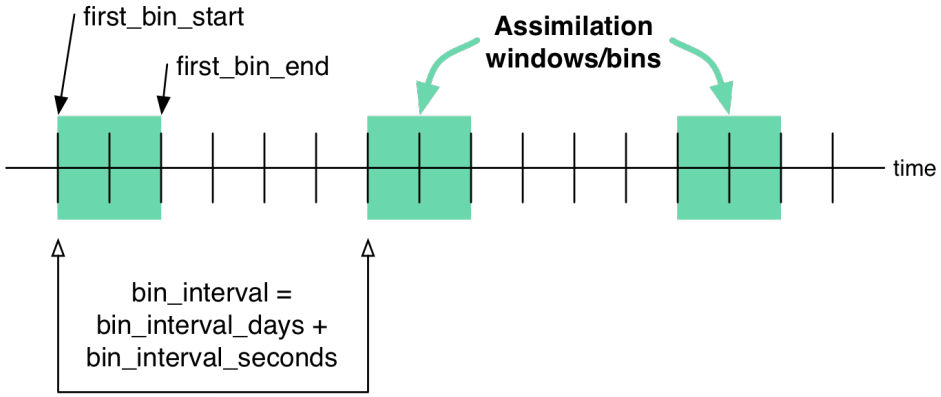
You can only specify **either** `obs_sequence_name` **or** `obs_sequence_list` – not both.

Item	Type	Description
obs_sequence_name	character(len=256)	Name of an observation sequence file(s). This may be a relative or absolute filename. If the filename contains a '/', the filename is considered to be comprised of everything to the right, and a directory structure to the left. The directory structure is then queried to see if it can be incremented to handle a sequence of observation files. The default behavior of `obs_seq_to_netcdf` is to look for additional files to include until the files are exhausted or an `obs_seq.final` file is found that contains observations beyond the timeframe of interest. e.g. 'obsdir_001/obs_seq.final' will cause `obs_seq_to_netcdf` to look for 'obsdir_002/obs_seq.final', and so on. If this is specified, 'obs_sequence_list' must be set to ' '.
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, 'obs_sequence_name' must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program.
append_to_netcdf	logical	This gives control over whether to overwrite or append to an existing netcdf output file. It is envisioned that you may want to combine multiple observation sequence files into one netcdf file (i.e. `append_to_netcdf=.true.`) to explore the effects on data coverage, etc. The default behavior is to create a new `obs_epoch_xxx.nc` file with every execution.
lonlim1	real	Westernmost longitude of the region in degrees.
lonlim2	real	Easternmost longitude of the region in degrees. *If* `lonlim2 < lonlim1` *, it defines a region that spans the prime meridian.* It is perfectly acceptable to specify `lonlim1 = 330` , `lonlim2 = 50` to identify a region like "Africa".
latlim1	real	Southernmost latitude of the region in degrees.
latlim2	real	Northernmost latitude of the region in degrees.
verbose	logical	Print extra info about the obs_seq_to_netcdf run.

The schedule namelist

The default values specify one giant 'bin'.

If the `print_table` variable is 'true' a summary of the assimilation schedule will be written to the screen.




```
&schedule_nml
   calendar        = 'Gregorian',
   first_bin_start =  1601,  1,  1,  0,  0,  0,
   first_bin_end   =  2999,  1,  1,  0,  0,  0,
   last_bin_end    =  2999,  1,  1,  0,  0,  0,
   bin_interval_days    = 1000000,
   bin_interval_seconds = 0,
```

(continues on next page)

(continued from previous page)

```
   max_num_bins          = 1000,
   print_table           = .true.
  /
```

Item	Type	Description
calendar	character(len=32)	Type of calendar to use to interpret dates. May be any type supported by the `time_manager_mod`. The string is case-insensitive.
first_bin_start	integer, dimension(6)	the first time of the first assimilation period. The six integers are: year, month, day, hour, hour, minute, second – in that order.
first_bin_end	integer, dimension(6)	the end of the first assimilation period. The six integers are: year, month, day, hour, hour, minute, second – in that order.
last_bin_end	integer, dimension(6)	the approximate end of the last assimilation period. The six integers are: year, month, day, hour, hour, minute, second – in that order. This does not need to be exact, the values from `last_bin_end`, `bin_interval_days`, and `bin_interval_seconds` are used to derive the assimilation schedule. The assimilation periods are repeated and will stop on or before the time defined by `last_bin_end`. See also `max_num_bins`.
bin_interval_days, bin_interval_seconds	integer	Collectively, `bin_interval_days` and `bin_interval_seconds` define the time between the start of successive assimilation windows. It is not possible to define a bin_interval such that there are overlapping bins (i.e. you can't use the same observations more than once).
max_num_bins	integer	An alternate way to specify the maximum number of assimilation periods. The assimilation bin is repeated by the bin_interval until one of two things happens: either the last time of interest is encountered (defined by `last_bin_end`) or the maximum number of assimilation periods has been reached (defined by `max_num_bins`).
print_table	logical	Prints the assimilation schedule.

Example

The following example illustrates the fact the `last_bin_end` does not have to be a 'perfect' bin end - and it gives you an idea of an assimilation schedule table. Note that the user input defines the last bin to end at 09 Z, but the last bin in the table ends at 06 Z.

```
&schedule_nml
   calendar        = 'Gregorian',
   first_bin_start =  2006, 8, 1, 0, 0, 0 ,
```

(continues on next page)

(continued from previous page)

```
   first_bin_end   =  2006, 8, 1, 6, 0, 0 ,
   last_bin_end    =  2006, 8, 2, 9, 0, 0 ,
   bin_interval_days    = 0,
   bin_interval_seconds = 21600,
   max_num_bins         = 1000,
   print_table          = .true.
   /
```

This is the 'table' part of the run-time output:

```
Requesting  5  assimilation periods.

epoch      1  start day=148135, sec=1
epoch      1    end day=148135, sec=21600
epoch      1  start 2006 Aug 01 00:00:01
epoch      1    end 2006 Aug 01 06:00:00

epoch      2  start day=148135, sec=21601
epoch      2    end day=148135, sec=43200
epoch      2  start 2006 Aug 01 06:00:01
epoch      2    end 2006 Aug 01 12:00:00

epoch      3  start day=148135, sec=43201
epoch      3    end day=148135, sec=64800
epoch      3  start 2006 Aug 01 12:00:01
epoch      3    end 2006 Aug 01 18:00:00

epoch      4  start day=148135, sec=64801
epoch      4    end day=148136, sec=0
epoch      4  start 2006 Aug 01 18:00:01
epoch      4    end 2006 Aug 02 00:00:00

epoch      5  start day=148136, sec=1
epoch      5    end day=148136, sec=21600
epoch      5  start 2006 Aug 02 00:00:01
epoch      5    end 2006 Aug 02 06:00:00
```

Notice that the leading edge of an assimilation window/bin/epoch/period is actually 1 second **after** the specified start time. This is consistent with the way DART has always worked. If you specify assimilation windows that fully occupy the temporal continuum, there has to be some decision at the edges. An observation precisely ON the edge should only participate in one assimilation window. Historically, DART has always taken observations precisely on an edge to be part of the subsequent assimilation cycle. The smallest amount of time representable to DART is 1 second, so the smallest possible delta is added to one of the assimilation edges.

6.197.3 Other modules used

```
location_mod
netcdf
obs_def_mod
obs_kind_mod
obs_sequence_mod
schedule_mod
time_manager_mod
typeSizes
types_mod
utilities_mod
```

Naturally, the program must be compiled with support for the observation types contained in the observation sequence files, so `preprocess` must be run to build appropriate `obs_def_mod` and `obs_kind_mod` modules - which may need specific `obs_def_?????.f90` files.

6.197.4 Files

Run-time

- `input.nml` is used for `obs_seq_to_netcdf_nml` and `schedule_nml`.
- `obs_epoch_xxx.nc` is a netCDF output file for assimilation period 'xxx'. Each observation copy is preserved - as are any/all QC values/copies.
- `dart_log.out` list directed output from the obs_seq_to_netcdf.

Related Matlab functions

- `diagnostics/matlab/read_obs_netcdf.m` reads the netcdf files and returns a structure with easy-to-plot components. More on that in the 'Usage' section below.
- `diagnostics/matlab/plot_obs_netcdf.m` may be used to explore the spatial distribution of observations and their values. More on that in the 'Usage' section below.
- `diagnostics/matlab/plot_obs_netcdf_diffs.m` will take the difference between any two observation copies and plot the spatial distribution and value of the difference. Useful for exploring the bias between 'observation' and 'prior ensemble mean', for example. Again, more on that in the 'Usage' section below.

Discussion of obs_epoch_xxx.nc structure

This might be a good time to review the basic observation sequence file structure. The only thing missing in the netcdf files is the 'shared' metadata for observations (e.g. GPS occultations). The observation locations, values, qc flags, error variances, etc., are all preserved in the netCDF files. The intent is to provide everything you need to make sensible plots of the observations. Some important aspects are highlighted.

```
[shad] % ncdump -v QCMetaData,CopyMetaData,ObsTypesMetaData obs_epoch_001.nc
netcdf obs_epoch_001 {
dimensions:
        linelen = 129 ;
        nlines = 104 ;
        stringlength = 32 ;
```

(continues on next page)

(continued from previous page)

```
        copy = 7 ;
        qc_copy = 2 ;
        location = 3 ;
        ObsTypes = 58 ;
        ObsIndex = UNLIMITED ; // (4752 currently)
variables:
        int copy(copy) ;
                copy:explanation = "see CopyMetaData" ;
        int qc_copy(qc_copy) ;
                qc_copy:explanation = "see QCMetaData" ;
        int ObsTypes(ObsTypes) ;
                ObsTypes:explanation = "see ObsTypesMetaData" ;
        char ObsTypesMetaData(ObsTypes, stringlength) ;
                ObsTypesMetaData:long_name = "DART observation types" ;
                ObsTypesMetaData:comment = "table relating integer to observation type␣
↪string" ;
        char QCMetaData(qc_copy, stringlength) ;
                QCMetaData:long_name = "quantity names" ;
        char CopyMetaData(copy, stringlength) ;
                CopyMetaData:long_name = "quantity names" ;
        char namelist(nlines, linelen) ;
                namelist:long_name = "input.nml contents" ;
        int ObsIndex(ObsIndex) ;
                ObsIndex:long_name = "observation index" ;
                ObsIndex:units = "dimensionless" ;
        double time(ObsIndex) ;
                time:long_name = "time of observation" ;
                time:units = "days since 1601-1-1" ;
                time:calendar = "GREGORIAN" ;
                time:valid_range = 1.15740740740741e-05, 0.25 ;
        int obs_type(ObsIndex) ;
                obs_type:long_name = "DART observation type" ;
                obs_type:explanation = "see ObsTypesMetaData" ;
                location:units = "deg_Lon deg_Lat vertical" ;
        double observations(ObsIndex, copy) ;
                observations:long_name = "org observation, estimates, etc." ;
                observations:explanation = "see CopyMetaData" ;
                observations:missing_value = 9.96920996838687e+36 ;
        int qc(ObsIndex, qc_copy) ;
                qc:long_name = "QC values" ;
                qc:explanation = "see QCMetaData" ;
        double location(ObsIndex, location) ;
                location:long_name = "location of observation" ;
                location:storage_order = "Lon Lat Vertical" ;
                location:units = "degrees degrees which_vert" ;
        int which_vert(ObsIndex) ;
                which_vert:long_name = "vertical coordinate system code" ;
                which_vert:VERTISUNDEF = -2 ;
                which_vert:VERTISSURFACE = -1 ;
                which_vert:VERTISLEVEL = 1 ;
                which_vert:VERTISPRESSURE = 2 ;
                which_vert:VERTISHEIGHT = 3 ;
```

(continues on next page)

(continued from previous page)

```
// global attributes:
                :creation_date = "YYYY MM DD HH MM SS = 2009 05 01 16 51 18" ;
                :obs_seq_to_netcdf_source = "$url: http://subversion.ucar.edu/DAReS/DART/
↪trunk/obs_sequence/obs_seq_to_netcdf.f90 $" ;
                :obs_seq_to_netcdf_revision = "$revision: 4272 $" ;
                :obs_seq_to_netcdf_revdate = "$date: 2010-02-12 14:26:40 -0700 (Fri, 12␣
↪Feb 2010) $" ;
                :obs_seq_file_001 = "bgrid_solo/work/01_01/obs_seq.final" ;
data:

 ObsTypesMetaData =
  "RADIOSONDE_U_WIND_COMPONENT     ",
  "RADIOSONDE_V_WIND_COMPONENT     ",
  "RADIOSONDE_SURFACE_PRESSURE     ",
  "RADIOSONDE_TEMPERATURE          ",
  "RADIOSONDE_SPECIFIC_HUMIDITY    ",
  ...
  yeah, yeah, yeah ... we're very impressed ...
  ...
  "VORTEX_PMIN                     ",
  "VORTEX_WMAX                     " ;

 QCMetaData =
  "Quality Control                 ",
  "DART quality control            " ;

 CopyMetaData =
  "observations                    ",
  "truth                           ",
  "prior ensemble mean             ",
  "posterior ensemble mean         ",
  "prior ensemble spread           ",
  "posterior ensemble spread       ",
  "observation error variance      " ;
}
```

So, first off, the UNLIMITED dimension is not 'time'. It's simply the number of observations - a coordinate variable called `ObsIndex`. The `observations` variable is a 2D array - each column is a 'copy' of the observation. The interpretation of the column is found in the `CopyMetaData` variable. Same thing goes for the `qc` variable - each column is defined by the `QCMetaData` variable.

The `Obs_Type` variable is crucial. Each observation has an integer code to define the specific ... DART observation type. In our example - lets assume that observation number 10 (i.e. ObsIndex == 10) has an `obs_type` of 3 [i.e. obs_type(10) = 3]. Since `ObsTypesMetaData(3) == "RADIOSONDE_SURFACE_PRESSURE"`, we know that any/all quantities where ObsIndex == 10 pertain to a radiosonde surface pressure observation.

6.197.5 Usage

Obs_seq_to_netcdf

`obs_seq_to_netcdf` is built and run in `/DART/observations/utilities/threed_sphere` or `/DART/observations/utilities/oned` or in the same way as the other DART components. That directory is intentionally designed to hold components that are model-insensitive. Essentially, we avoid having to populate every `model` directory with identical `mkmf_obs_seq_to_netcdf` and `path_names_obs_seq_to_netcdf` files. After the program has been run, `/DART/observations/utilities/threed_sphere/plot_obs_netcdf.m` can be run to plot the observations. Be aware that the `ObsTypesMetaData` list is all known observation types and not only the observation types in the netCDF file.

Example

```
&schedule_nml
   calendar        = 'Gregorian',
   first_bin_start =  2006, 8, 1, 3, 0, 0 ,
   first_bin_end   =  2006, 8, 1, 9, 0, 0 ,
   last_bin_end    =  2006, 8, 3, 3, 0, 0 ,
   bin_interval_days    = 0,
   bin_interval_seconds = 21600,
   max_num_bins         = 1000,
   print_table          = .true.
   /

&obs_seq_to_netcdf_nml
   obs_sequence_name = '',
   obs_sequence_list = 'olist',
   append_to_netcdf  = .false.,
   lonlim1    =    0.0,
   lonlim2    =  360.0,
   latlim1    =  -80.0,
   latlim2    =   80.0,
   verbose    = .false.
   /
```

> *cat olist* /users/thoar/temp/obs_0001/obs_seq.final /users/thoar/temp/obs_0002/obs_seq.final /users/thoar/temp/obs_0003/obs_seq.final

Here is the pruned run-time output. Note that multiple input observation sequence files are queried and the routine ends (in this case) when the first observation time in a file is beyond the last time of interest.

```
 ---------------------------------------
 Starting ... at YYYY MM DD HH MM SS =
                 2009  5 15  9  0 23
 Program obs_seq_to_netcdf
 ---------------------------------------

 Requesting            8  assimilation periods.

epoch      1  start day=148135, sec=10801
epoch      1    end day=148135, sec=32400
```

(continues on next page)

(continued from previous page)

```
epoch      1  start 2006 Aug 01 03:00:01
epoch      1    end 2006 Aug 01 09:00:00

epoch      2  start day=148135, sec=32401
epoch      2    end day=148135, sec=54000
epoch      2  start 2006 Aug 01 09:00:01
epoch      2    end 2006 Aug 01 15:00:00

epoch      3  start day=148135, sec=54001
epoch      3    end day=148135, sec=75600
epoch      3  start 2006 Aug 01 15:00:01
epoch      3    end 2006 Aug 01 21:00:00

epoch      4  start day=148135, sec=75601
epoch      4    end day=148136, sec=10800
epoch      4  start 2006 Aug 01 21:00:01
epoch      4    end 2006 Aug 02 03:00:00

epoch      5  start day=148136, sec=10801
epoch      5    end day=148136, sec=32400
epoch      5  start 2006 Aug 02 03:00:01
epoch      5    end 2006 Aug 02 09:00:00

epoch      6  start day=148136, sec=32401
epoch      6    end day=148136, sec=54000
epoch      6  start 2006 Aug 02 09:00:01
epoch      6    end 2006 Aug 02 15:00:00

epoch      7  start day=148136, sec=54001
epoch      7    end day=148136, sec=75600
epoch      7  start 2006 Aug 02 15:00:01
epoch      7    end 2006 Aug 02 21:00:00

epoch      8  start day=148136, sec=75601
epoch      8    end day=148137, sec=10800
epoch      8  start 2006 Aug 02 21:00:01
epoch      8    end 2006 Aug 03 03:00:00

 obs_seq_to_netcdf  opening /users/thoar/temp/obs_0001/obs_seq.final

 num_obs_in_epoch (           1 ) =       103223
 InitNetCDF  obs_epoch_001.nc is fortran unit            5
 num_obs_in_epoch (           2 ) =       186523
 InitNetCDF  obs_epoch_002.nc is fortran unit            5
 num_obs_in_epoch (           3 ) =       110395
 InitNetCDF  obs_epoch_003.nc is fortran unit            5
 num_obs_in_epoch (           4 ) =       191957
 InitNetCDF  obs_epoch_004.nc is fortran unit            5

 obs_seq_to_netcdf  opening /users/thoar/temp/obs_0002/obs_seq.final

 num_obs_in_epoch (           5 ) =        90683
```

(continues on next page)

(continued from previous page)

```
 InitNetCDF  obs_epoch_005.nc is fortran unit            5
 num_obs_in_epoch (           6 ) =       186316
 InitNetCDF  obs_epoch_006.nc is fortran unit            5
 num_obs_in_epoch (           7 ) =       109465
 InitNetCDF  obs_epoch_007.nc is fortran unit            5
 num_obs_in_epoch (           8 ) =       197441
 InitNetCDF  obs_epoch_008.nc is fortran unit            5

 obs_seq_to_netcdf  opening /users/thoar/temp/obs_0003/obs_seq.final

 ---------------------------------------
 Finished ... at YYYY MM DD HH MM SS =
                 2009  5 15  9  2 56
 ---------------------------------------
```

Matlab setup

You will need the 'normal' `DART/diagnostics/matlab` functions available to Matlab, so be sure your MATLABPATH is set such that you have access to `plot_obs_netcdf`
You can do this with the following Matlab command :

```
>> addpath('replace_this_with_the_real_path_to/DART/diagnostics/matlab')
```

As is standard practice, the instructions for using the Matlab scripts `plot_obs_netcdf` and `plot_obs_netcdf_diffs` are available by using the Matlab 'help' facility (i.e. *help plot_obs_netcdf*). A quick discussion of them here still seems appropriate. If you run the following Matlab commands with an `obs_sequence_001.nc` file you cannot possibly have:

```
>> help plot_obs_netcdf
fname         = 'obs_sequence_001.nc';
ObsTypeString = 'RADIOSONDE_U_WIND_COMPONENT';
region        = [0 360 -90 90 -Inf Inf];
CopyString    = 'NCEP BUFR observation';
QCString      = 'DART quality control';
maxQC         = 2;
verbose       = 1;

obs = plot_obs_netcdf(fname, ObsTypeString, region, CopyString, QCString, maxQC,␣
↪verbose);

>> fname         = 'obs_sequence_001.nc';
>> ObsTypeString = 'RADIOSONDE_U_WIND_COMPONENT';
>> region        = [0 360 -90 90 -Inf Inf];
>> CopyString    = 'NCEP BUFR observation';
>> QCString      = 'DART quality control';
>> maxQC         = 2;
```

(continues on next page)

(continued from previous page)

```
>> verbose       = 1;
>> obs = plot_obs_netcdf(fname, ObsTypeString, region, CopyString, QCString, maxQC,␣
↪verbose);

N =  3336 RADIOSONDE_U_WIND_COMPONENT  obs (type   1) between levels 550.00 and 101400.00
N =  3336 RADIOSONDE_V_WIND_COMPONENT  obs (type   2) between levels 550.00 and 101400.00
N =    31 RADIOSONDE_SURFACE_PRESSURE  obs (type   3) between levels 0.00 and 1378.00
N =  1276 RADIOSONDE_TEMPERATURE       obs (type   4) between levels 550.00 and 101400.00
N =   691 RADIOSONDE_SPECIFIC_HUMIDITY obs (type   5) between levels 30000.00 and 101400.
↪00
N = 11634 AIRCRAFT_U_WIND_COMPONENT    obs (type   6) between levels 17870.00 and 99510.
↪00
N = 11634 AIRCRAFT_V_WIND_COMPONENT    obs (type   7) between levels 17870.00 and 99510.
↪00
N =  8433 AIRCRAFT_TEMPERATURE         obs (type   8) between levels 17870.00 and 76710.
↪00
N =  6993 ACARS_U_WIND_COMPONENT       obs (type  10) between levels 17870.00 and 76680.
↪00
N =  6993 ACARS_V_WIND_COMPONENT       obs (type  11) between levels 17870.00 and 76680.
↪00
N =  6717 ACARS_TEMPERATURE            obs (type  12) between levels 17870.00 and 76680.
↪00
N = 20713 SAT_U_WIND_COMPONENT         obs (type  22) between levels 10050.00 and 99440.
↪00
N = 20713 SAT_V_WIND_COMPONENT         obs (type  23) between levels 10050.00 and 99440.
↪00
N =   723 GPSRO_REFRACTIVITY           obs (type  46) between levels 220.00 and 12000.00
NCEP BUFR observation is copy   1
DART quality control is copy   2
Removing 993 obs with a DART quality control value greater than 2.000000
```

you get the plots at the top of this document. If you have a relatively new version of Matlab, you can dynamically rotate the 3D view ... coooool. Even spiffier, if you click on the observations (try the BAD observations), Matlab reports the lat/lon/level of these observations. At least R2008b does, I haven't tried it with all the other variants.

The vertical levels are reported so you can restrict the area of interest with the 'region' variable [minlon maxlon minlat maxlat minlevel maxlevel]. Only the observations with a QC value less than or equal to 'maxQC' are plotted in 'Figure 1'. Note the values of 'QCString' and 'CopyString' must match some value of `QCMetaData` and `CopyMetaData`, respectively. If you're not so keen on a 3D plot, simply change the view to be directly 'overhead':

```
>> view(0,90)
```

And if you act today, we'll throw in a structure containing the selected data AT NO EXTRA CHARGE.

```
>> obs
obs =
            fname: 'obs_sequence_001.nc'
    ObsTypeString: 'RADIOSONDE_U_WIND_COMPONENT'
           region: [0 360 -90 90 -Inf Inf]
       CopyString: 'NCEP BUFR observation'
```

(continues on next page)

(continued from previous page)

```
        QCString: 'DART quality control'
           maxQC: 2
         verbose: 1
            lons: [2343x1 double]
            lats: [2343x1 double]
               z: [2343x1 double]
             obs: [2343x1 double]
            Ztyp: [2343x1 double]
              qc: [2343x1 double]
       numbadqc: 993
         badobs: [1x1 struct]
```

If there are observations with QC values above that defined by `maxQC` there will be a `badobs` structure as a component in the `obs` structure.

6.197.6 References

1. none

6.197.7 Private components

N/A

6.198 program `obs_common_subset`

6.198.1 Overview

This specialized tool allows you to select subsets of observations from two or more observation sequence files output from `filter`. It creates a new set of output observation sequence files containing only the observations which were successfully assimilated in all experiments.

Experiments using the same input observation sequence file but with different configurations (e.g. different inflation values, different localization radii, etc) can assimilate different numbers of the available observations. In that case there will be differences in the diagnostic plots which are not directly relatable to the differences in the quality of the assimilation. If this tool is run on the `obs_seq.final` files from all the experiments and then the diagnostics are generated, only the observations which were assimilated in all experiments will contribute to the summary statistics. A more direct comparison can be made and improvements can be correctly attributed to the differences in the experimental parameters.

This tool is intended to be used when comparing the results from a group of related experiments in which **the exact same input observation sequence file** is used for all runs. The tool cannot process observation sequence files which differ in anything other than whether an observation was successfully assimilated/evaluated or not. Note that it is fine to add or remove observation types from the `assimilate_these_obs_types` or `evaluate_these_obs_types` namelist items for different experiments. The output observation sequence files will still contain an identical list of observations, with some marked with a DART QC indicating 'not assimilated because of namelist control'.

See the *"two_experiment" diagnostic plots in* documentation for Matlab scripts supplied with DART to directly compare the observation diagnostic output from multiple experiments (it does more than two, the script has a poor name).

This is one of a set of tools which operate on observation sequence files. For a more general purpose tool see the *program obs_sequence_tool*, and for a more flexible selection tool see the obs_selection_tool.

Creating an input filelist

One of the inputs to this tool is a list of filenames to compare. The filenames can be directly in the namelist file, or they can be in a set of separate text files. The latter may be easier when there are more than just a few files to compare.

For experiments where there are multiple job steps, and so multiple output observation sequence files per experiment, the input to this tool would then be a list of lists of filenames. Each set of names must be put into a text file with each filename on a separate line.

If each experiment was run in a different set of directories, and if a list of observation sequence filenames was made with the `ls` command:

```
> ls exp1/*/obs_seq.final > exp1list
> cat exp1list
exp1/step1/obs_seq.final
exp1/step2/obs_seq.final
exp1/step3/obs_seq.final
exp1/step4/obs_seq.final
> ls exp2/*/obs_seq.final > exp2list
> cat exp2list
exp2/step1/obs_seq.final
exp2/step2/obs_seq.final
exp2/step3/obs_seq.final
exp2/step4/obs_seq.final
> ls exp3/*/obs_seq.final > exp3list
> cat exp2list
exp3/step1/obs_seq.final
exp3/step2/obs_seq.final
exp3/step3/obs_seq.final
exp3/step4/obs_seq.final
```

Then the namelist entries would be:

```
filename_seq = ''
filename_seq_list = 'exp1list', 'exp2list', exp3list'
num_to_compare_at_once = 3
```

6.198.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_common_subset_nml
 num_to_compare_at_once = 2,
 filename_seq           = '',
 filename_seq_list      = '',
 filename_out_suffix    = '.common' ,
 print_every            = 10000,
 dart_qc_threshold      = 3,
 calendar               = 'Gregorian',
 print_only             = .false.,
 eval_and_assim_can_match = .false.,
/
```

Item	Type	Description
num_to_compare_at_once	integer	Number of observation sequence files to compare together at a time. Most commonly the value is 2, but can be any number. If more than this number of files are listed as inputs, the tool will loop over the list N files at a time.
filename_seq	character(len=256), dimension(5000)	The array of names of the observation sequence files to process. If more than N files (where N is num_to_compare_at_once) are listed, they should be ordered so the first N files are compared together, followed by the next set of N files, etc. You can only specify one of filename_seq OR filename_seq_list, not both.
filename_seq_list	character(len=256), dimension(100)	An alternative way to specify the list of input observation sequence files. Give a list of N filenames which contain, one per line, the names of the observation sequence files to process. There should be N files specified (where N is num_to_compare_at_once), and the first observation sequence filename listed in each file will be compared together, then the second, until the lists are exhausted. You can only specify one of filename_seq OR filename_seq_list, not both.
filename_out_suffix	character(len=32)	A string to be appended to each of the input observation sequence file names to create the output filenames.
print_every	integer	To indicate progress, a count of the successfully processed observations is printed every Nth set of obs. To decrease the output volume set this to a larger number. To disable this output completely set this to -1.
dart_qc_threshold	integer	Observations with a DART QC value larger than this threshold will be discarded. Note that this is the QC value set by `filter` to indicate the outcome of trying to assimilate an observation. This is not related to the incoming data QC. For an observation which was successfully assimilated or evaluated in both the Prior and Posterior this should be set to 1. To also include observations which were successfully processed in the Prior but not the Posterior, set to 3. To ignore the magnitude of the DART QC values and keep observations only if the DART QCs match, set this to any value higher than 7.
calendar	character(len=32)	Set to the name of the calendar; only controls the printed output for the dates of the first and last observations in the file. Set this to "no_calendar" if the observations are not using any calendar.
print_only	logical	If .TRUE. do not create the output files, but print a summary of the number and types of each observation in each of the input and output files.
eval_and_assim_can_match	logical	Normally FALSE. . If .TRUE. then observations which were either successfully evaluated OR assimilated will match and are kept.

6.198.3 Building

Most `$DART/models/*/work` directories will build the tool along with other executable programs. It is also possible to build the tool in the `$DART/observations/utilities` directory. The `preprocess` program must be built and run first, to define what set of observation types will be supported. See the *PROGRAM preprocess* for more details on how to define the list and run it. The combined list of all observation types which will be encountered over all input files must be in the preprocess input list. The other important choice when building the tool is to include a compatible locations module. For the low-order models, the `oned` module should be used; for real-world observations, the `threed_sphere` module should be used.

Generally the directories where executables are built will include a "quickbuild.sh" script which will build and run preprocess and then build the rest of the executables. The "input.nml" namelists will need to be edited to include all the required observation types first.

6.198.4 Modules used

```
types_mod
utilities_mod
time_manager_mod
obs_def_mod
obs_sequence_mod
```

6.198.5 Files

- `input.nml`
- The input files specified in the `filename_seq` or `filename_seq_list` namelist variable.
- The output files are specified by appending the string from the `filename_out_suffix` namelist item to the input filenames.

6.198.6 References

- none

6.199 MODULE ensemble_manager_mod

6.199.1 Overview

Manages storage and a number of operations for multiple copies of a vector. The most obvious use is to manage ensembles of model state vectors. In this case, the number of copies stored for each state vector element is the ensemble size plus one or more additional copies like the mean, variance, associated inflation values, etc. The ensemble_manager provides routines to compute the mean and variance of a subset of the copies, to track the time associated with the copies, and to write and read restart files. Most importantly, it provides a capability to do transposes between two storage representations of an ensemble. In one representation, each process stores all copies of a subset of the state variables while in the other, each process stores all of the state variables for a subset of copies. The ensemble manager is also used to manage ensembles of observation priors and quality control and ensembles of forward observation operator error status.

The ensemble manager interacts strongly with the multiple process capability of the Message Passing Interface (MPI) libraries. It is used to partition the data so each MPI process stores only a subset of the copies and variables, dividing

the data as evenly as possible across the processes. At no time during the execution does any one process have to store the entire dataset for all ensemble members (unless running in serial mode without MPI, or if running with 1 MPI task).

The ensemble manager is set of general purpose data management routines. For run-time efficiency, the derived type information is not marked private which means other modules can directly manipulate the data arrays. However it means much care must be taken to access the most recently updated representation of the data, either the copies or variables arrays.

A set of sanity check routines have been added to track the last modified version of the data: the copies array or the vars array. Before directly reading or writing these arrays call one of the 'prepare' routines to indicate what kind of data access you are about to make. If the most recently updated data is not as expected an error message will occur. After the direct access if the following operations detect that the data they are operating on is not the most recently updated they will print an error message. Routines inside the ensemble manager that alter the copies or vars will set the state automatically so these routines are only necessary to call if you are directly accessing the copies or vars arrays from outside the ensemble manager.

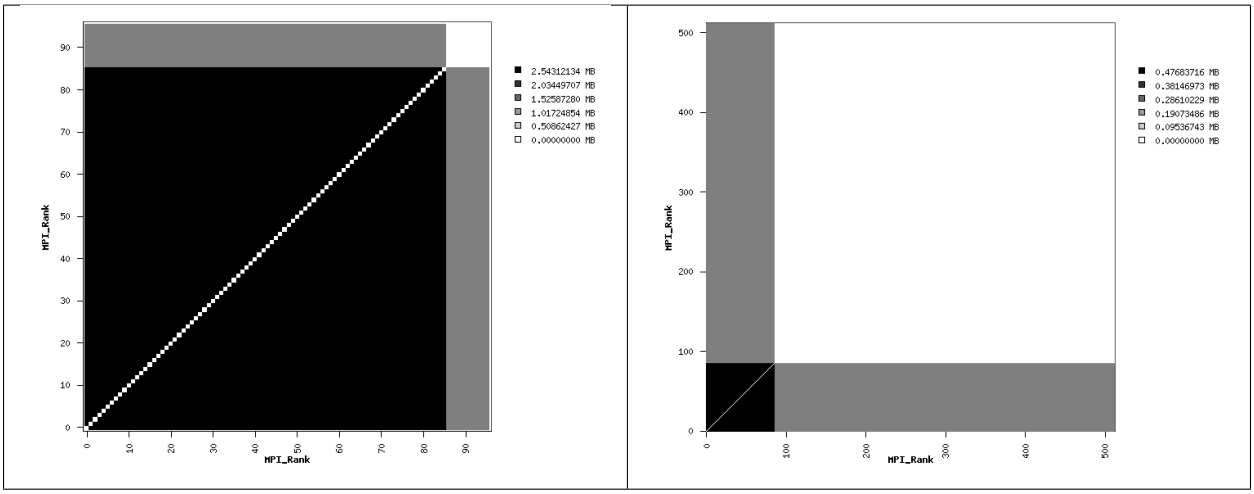
6.199.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&ensemble_manager_nml
   layout                = 1
   tasks_per_node        = 1
   communication_configuration = 1
   debug                 = .false.
  /
```

Item	Type	Description
layout	integer	Determines the logical process (PE) layout across MPI tasks. 1 is PE = MPI task. 2 is a round-robin layout around the nodes. Layout 2 results in a more even usage of memory across nodes. This may allow you to run with a larger state vector without hitting the memory limit of the node. It may give a slight (5%) increase in performance, but this is machine dependent. It has no effect on serial runs.
tasks_per_node	integer	The number of MPI tasks per hardware node is generally fixed when a batch job is submitted. This namelist item tells the ensemble manager what the user selected at that time. Once a program is running the code has no control to change how MPI tasks are assigned to physical CPUs. This number is used only if layout = 2, and it allows the code spread high-memory-use PEs to different hardware nodes by assigning them in a round-robin order. The job will still run if this number does not match the real "tasks_per_node" at the hardware level, but it may run out of memory if the mismatch causes multiple high-memory-use tasks to be run on the same node.
communication_configuration	integer	For most users, the default value of 1 is the best choice. However there are multiple strategies for the internal MPI communication patterns (see *Note below). Values from 1 to 4 select different options; try the various options to see if one might be faster than the others.
debug	logical	If true print debugging information.

**Note about MPI communication flags:*
The communication_configuration flags select various combinations of the internal settings for use_copy2var_send_loop and use_var2copy_rec_loop. These flags change the order of the MPI send and MPI receives in the the routines all_copies_to_all_vars and all_vars_to_all_copies. The figures below show the data transferred between tasks for an 80 member ensemble. The left figure is using 96 tasks, the right figure is using 512 tasks. As the number of tasks increases, the 'all to all' data transfer becomes a 'some to all, all to some' transfer and the order of MPI send and MPI receives becomes increasingly important. The default values give a performance advantage as the number of tasks becomes much greater than the the ensemble size. However, for small numbers of tasks, i.e. less than the ensemble size, changing the default values may improve performance.



6.199.3 Other modules used

```
types_mod
utilities_mod
assim_model_mod
time_manager_mod
random_seq_mod
mpi_utilities_mod
sort_mod
```

6.199.4 Public interfaces

use ensemble_manager_mod, only :	init_ensemble_manager
	read_ensemble_restart
	write_ensemble_restart
	get_copy
	put_copy
	broadcast_copy
	set_ensemble_time
	get_ensemble_time
	end_ensemble_manager
	duplicate_ens
	get_my_num_copies
	get_my_copies
	get_my_num_vars
	get_my_vars
	get_copy_owner_index
	get_var_owner_index
	all_vars_to_all_copies
	all_copies_to_all_vars
	compute_copy_mean
	compute_copy_mean_sd
	compute_copy_mean_var
	prepare_to_write_to_vars
	prepare_to_write_to_copies
	prepare_to_read_from_vars
	prepare_to_read_from_copies
	prepare_to_update_vars
	prepare_to_update_copies
	print_ens_handle
	map_pe_to_task

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

```
type ensemble_type
   !DIRECT ACCESS INTO STORAGE IS ALLOWED; BE CAREFUL
   integer :: num_copies
   integer :: num_vars
   integer :: my_num_copies
   integer :: my_num_vars
   integer, pointer :: my_copies(:)
   integer, pointer :: my_vars(:)
   ! Storage in next line is to be used when each PE has all copies of subset of vars
   real(r8), pointer :: copies(:, :)  ! Dimensioned (num_copies, my_num_vars)
   ! Storage on next line is used when each PE has subset of copies of all vars
   real(r8), pointer :: vars(:, :)    ! Dimensioned (num_vars, my_num_copies)
   ! Time is only related to var complete
   type(time_type), pointer :: time(:)
   integer :: distribution_type
   integer :: valid     ! copies modified last, vars modified last, both same
   integer :: id_num
   integer, allocatable :: task_to_pe_list(:) ! List of tasks
   integer, allocatable :: pe_to_task_list(:) ! List of tasks
   ! Flexible my_pe, layout_type which allows different task layouts for different␣
↪ensemble handles
   integer :: my_pe
   integer :: layout_type
end type ensemble_type
```

Provides a handle for an ensemble that manages copies of a vector. For efficiency, the type internals are not private and direct access to the storage arrays is used throughout DART.

Component	Description
num_copies	Global number of copies of the vector.
num_vars	Global number of elements (variables) in the vector.
my_num_copies	Number of copies stored by this process.
my_num_vars	Number of variables stored by this process.
my_copies	Dimensioned to size my_num_copies. Contains a list of the global indices of copies stored by this process.
my_vars	Dimensioned to size my_num_vars. Contains a list of the global indices of variables stored by this process.
copies	Dimensioned (num_copies, my_num_vars). Storage for all copies of variables stored by this process.
vars	Dimensioned (num_vars, my_num_copies). Storage for all variables of copies stored by this process.
time	Dimensioned my_num_copies. A time_type that stores time associated with a given copy of the vector.
distribution_type	Does nothing at present. Can be used for future releases to control the layout of different copies and variables in storage.
valid	Flag to track whether the copies array has the most recently updated data, the vars array is most recently modified, or if both the arrays have identical data, like after a transpose.
id_num	Internal number unique to each ensemble handle, used for debugging purposes.
task_to_pe_list	Mapping from MPI task number to logical Processing Element (PE) number. Enables different assignment of MPI tasks to PEs. If the number of MPI tasks is larger than the number of copies of the vector, when the ensemble is var complete then the first N MPI tasks have allocated 'vars' arrays and the remaining ones do not. Assigning the MPI tasks round-robin to multi-processor nodes can make the memory usage more uniform across nodes, which may allow more MPI tasks per node than the standard layout.
pe_to_task_list	Inverse PE to MPI task mapping. See above for more description.
my_pe	The logical PE number for the MPI task.
layout_type	Controls the mapping type between MPI tasks and PEs. Currently type 1 is the standard layout (one-to-one mapping) and type 2 is a round-robin mapping where each node gets a task in turn before assigning a second task to each node, until all tasks are assigned.

call init_ensemble_manager(ens_handle, num_copies, num_vars [, distribution_type_in] [, layout_type])

```
type(ensemble_type), intent(out) :: ens_handle
integer,             intent(in)  :: num_copies
integer,             intent(in)  :: num_vars
integer, optional,   intent(in)  :: distribution_type_in
integer, optional,   intent(in)  :: layout_type
```

Initializes an instance of an ensemble. Storage is allocated and the size descriptions in the ensemble_type are initialized.

ens_handle	Handle for the ensemble being initialized
num_copies	Number of copies of vector.
num_vars	Number of variables in the vector.
distribution_type_in	Controls layout of storage on PEs. Currently only option 1 is supported.
layout_type	Controls layout of MPI tasks on PEs. Type 1 is the default, where MPI tasks are assigned to PEs on a one-to-one basis. Type 2 is a round-robin assignment where each node gets one task before the nodes are assigned a second task. If running with more MPI tasks than num_copies, this can result in a more uniform usage of memory across the nodes.

call read_ensemble_restart(ens_handle, start_copy, end_copy, start_from_restart, file_name [, init_time] [, force_single_file])

```
type(ensemble_type),       intent(inout) :: ens_handle
integer,                   intent(in)    :: start_copy
integer,                   intent(in)    :: end_copy
logical,                   intent(in)    :: start_from_restart
character(len=*),          intent(in)    :: file_name
type(time_type), optional, intent(in)    :: init_time
logical, optional,         intent(in)    :: force_single_file
```

Read in a set of copies of a vector from file file_name. The copies read are place into global copies start_copy:end_copy in the ens_handle. If start_from_restart is false, then only a single copy of the vector is read from the file and then it is perturbed using routines in assim_model_mod to generate the required number of copies. The read can be from a single file that contains all needed copies or from a different file for each copy. This choice is controlled by the namelist entry single_restart_file_in. However, the optional argument force_single_file forces the read to be from a single file if it is present and true. This is used for ensembles that contain the inflation values for state space inflation. If multiple files are to be read, the file names are generated by appending integers to the input file_name. If the input is a single file all reads are done sequentially by process 0 and then shipped to the PE that stores that copy. If the input is multiple files each MPI task reads the copies it stores directly and independently.

ens_handle	Handle of ensemble.
start_copy	Global index of first of continguous set of copies to be read.
end_copy	Global index of last of contiguous set of copies to be read, copies(start_copy:end_copy).
start_from_restart	If true, read all copies from file. If false, read one copy and perturb to get required number.
file_name	Name of file from which to read.
init_time	If present, set time of all copies read to this value.
force_single_file	If present and true, force the read to be from a single file which contains all copies.

call write_ensemble_restart(ens_handle, file_name, start_copy, end_copy [, force_single_file])

```
type(ensemble_type), intent(inout) :: ens_handle
character(len=*),    intent(in)    :: file_name
```

(continues on next page)

(continued from previous page)

```
integer,             intent(in)    :: start_copy
integer,             intent(in)    :: end_copy
logical, optional,   intent(in)    :: force_single_file
```

Writes a set of copies of a vector to file file_name. The copies written are from global copies start_copy:end_copy in the ens_handle. The write can be to a single file or to a different file for each copy. This choice is controlled by the namelist entry single_restart_file_out. However, the optional argument force_single_file forces the write to be to a single file if it is present and true. This is used for ensembles that contain the inflation values for state space inflation. If multiple files are to be written, the file names are generated by appending integers to the input file_name. If the output is a single file all copies are shipped from the PE that stores that copy to process 0, and then written out sequentially. If the output is to multiple files each MPI task writes the copies it stores directly and independently.

ens_handle	Handle of ensemble.
file_name	Name of file from which to read.
start_copy	Global index of first of continguous set of copies to be written.
end_copy	Global index of last of contiguous set of copies to be written, copies(start_copy:end_copy).
force_single_file	If present and true, force the write to be to a single file which contains all copies.

call get_copy(receiving_pe, ens_handle, copy, vars [, mtime])

```
integer,                   intent(in)  :: receiving_pe
type(ensemble_type),       intent(in)  :: ens_handle
integer,                   intent(in)  :: copy
real(r8), dimension(:),    intent(out) :: vars
type(time_type), optional, intent(out) :: mtime
```

Retrieves a copy of the state vector, indexed by the global index copy. The process that is to receive the copy is receiving_pe and the copy is returned in the one dimensional array vars. The time of the copy is also returned if mtime is present. This is generally used for operations, like IO, that require a single processor to do things with the entire state vector. Data is only returned in vars on the receiving PE; vars on all other PEs is unset.

receiving_pe	This process ends up with the requested copy of the state vector.
ens_handle	Handle for ensemble.
copy	The global index of the copy of the state vector that is to be retrieved.
vars	One dimensional array in which the requested copy of the state vector is returned. Data is only returned in vars on the receiving PE; vars on all other PEs is unset.
mtime	If present returns the time of the requested copy.

call put_copy(sending_pe, ens_handle, copy, vars [, mtime])

```
integer,                   intent(in)    :: sending_pe
type(ensemble_type),       intent(inout) :: ens_handle
integer,                   intent(in)    :: copy
```

(continues on next page)

(continued from previous page)

```
real(r8), dimension(:),    intent(in)    :: vars
type(time_type), optional, intent(in)    :: mtime
```

Sends a state vector, in vars, from the given process to the process storing the global index copy. The time of the copy is also sent if mtime is present. This is generally used for operations, like IO, that require a single processor to do things with the entire state vector. For instance, if a single process reads in a state vector, it can be shipped to the storing process by this subroutine. Only the data in vars on the sending PE is processed; vars on all other PEs is ignored.

sending_pe	This process sends the copy of the state vector.
ens_handle	Handle for ensemble.
copy	The global index of the copy of the state vector that is to be sent.
vars	One dimensional array in which the requested copy of the state vector is located. Only the data in vars on the sending PE is processed; vars on all other PEs is ignored.
mtime	If present send the time of the copy.

call broadcast_copy(ens_handle, copy, arraydata)

```
type(ensemble_type),    intent(in)   :: ens_handle
integer,                intent(in)   :: copy
real(r8), dimension(:), intent(out)  :: arraydata
```

Finds which PE has the global index copy and broadcasts that copy to all PEs. `arraydata` is an output on all PEs, even on the PE which is the owner if it is separate storage from the vars array in the ensemble handle. This is a collective routine, which means it must be called by all processes in the job.

ens_handle	Handle for ensemble.
copy	The global index of the copy of the state vector that is to be sent.
arraydata	One dimensional array into which the requested copy of the state vector will be copied on all PEs, including the sending PE.

call set_ensemble_time(ens_handle, indx, mtime)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)    :: indx
type(time_type),     intent(in)    :: mtime
```

Set the time of a copy to the given value. `indx` in this case is the local copy number for a specific task. get_copy_owner_index() can be called to see if you are the owning task for a given global copy number, and to get the local index number for that copy.

ens_handle	Handle for ensemble.
indx	The local index of the copy of the state vector that is to be set.
mtime	The time to set for this copy.

call get_ensemble_time(ens_handle, indx, mtime)

```
type(ensemble_type), intent(in)   :: ens_handle
integer,             intent(in)   :: indx
type(time_type),     intent(out)  :: mtime
```

Get the time associated with a copy. `indx` in this case is the local copy number for a specific task. get_copy_owner_index() can be called to see if you are the owning task for a given global copy number, and to get the local index number for that copy.

`ens_handle`	Handle for ensemble.
`indx`	The local index of the copy to retrieve the time from.
`mtime`	The returned time value.

call end_ensemble_manager(ens_handle)

```
type(ensemble_type), intent(in)  :: ens_handle
```

Frees up storage associated with an ensemble.

`ens_handle`	Handle for an ensemble.

call duplicate_ens(ens1, ens2, duplicate_time)

```
type(ensemble_type), intent(in)    :: ens1
type(ensemble_type), intent(inout) :: ens2
logical, intent(in)                :: duplicate_time
```

Copies the contents of the vars array from ens1 into ens2. If the num_copies and num_vars are not consistent or if the distribution_type is not consistent, fails with an error. If duplicate_time is true, the times from ens1 are copied over the times of ens2. Only the vars array data is copied from the source to the destination. Transpose the data after duplication if you want to access the copies.

`ens1`	Ensemble handle of ensemble to be copies into ens2. Data from the vars array will be replicated.
`ens2`	Ensemble handle of ensemble into which ens1 vars data will be copied.
`duplicate_time`	If true, copy the times from ens1 into ens2, else leave ens2 times unchanged.

var = get_my_num_copies(ens_handle)

```
integer                           :: get_my_num_copies
type(ensemble_type), intent(in)   :: ens_handle
```

Returns number of copies stored by this process when storing all variables for a subset of copies. Same as num_copies if running with only a single process.

var	Returns the number of copies stored by this process when storing all variables for a subset of copies.
ens_handle	Handle for an ensemble.

var = get_my_num_vars(ens_handle)

```
integer                           :: get_my_num_vars
type(ensemble_type), intent(in) :: ens_handle
```

Returns number of variables stored by this process when storing all copies of a subset of variables. Same as num_vars if running with only a single process.

var	Returns the number of vars stored by this process when storing all copies of a subset of variables.
ens_handle	Handle for an ensemble.

call get_my_copies(ens_handle, copies)

```
type(ensemble_type), intent(in) :: ens_handle
integer, intent(out)            :: copies(:)
```

Returns a list of the global copy numbers stored on this process when storing subset of copies of all variables.

ens_handle	Handle for an ensemble.
copies	List of all copies stored by this process when storing subset of copies of all variables.

call get_my_vars(ens_handle, vars)

```
type(ensemble_type), intent(in) :: ens_handle
integer, intent(out)            :: vars(:)
```

Returns a list of the global variable numbers stored on this process when storing all copies of a subset of variables.

ens_handle	Handle for an ensemble.
vars	List of all variables stored on this process when storing all copies of a subset of variables.

call get_copy_owner_index(copy_number, owner, owners_index)

```
integer, intent(in)  :: copy_number
integer, intent(out) :: owner
integer, intent(out) :: owners_index
```

Given the global index of a copy number, returns the PE that stores this copy when all variables of a subset of copies are stored and the local storage index for this copy on that process.

copy_number	Global index of a copy from an ensemble.
owner	Process Element (PE) that stores this copy when each has all variables of a subset of copies.
owners_index	Local storage index for this copy on the owning process.

call get_var_owner_index(var_number, owner, owners_index)

```
integer, intent(in)  :: var_number
integer, intent(out) :: owner
integer, intent(out) :: owners_index
```

Given the global index of a variable in the vector, returns the PE that stores this variable when all copies of a subset of variables are stored and the local storage index for this variable on that process.

var_number	Global index of a variable in the vector from an ensemble.
owner	Process Element (PE) that stores this variable when each has all copies of subset of variables.
owners_index	Local storage index for this variable on the owning process.

call all_vars_to_all_copies(ens_handle, label)

```
type(ensemble_type), intent(inout)        :: ens_handle
character(len=*),    intent(in), optional :: label
```

Transposes data from a representation in which each PE has a subset of copies of all variables to one in which each has all copies of a subset of variables. In the current implementation, storage is not released so both representations are always available. However, one representation may be current while the other is out of date.

Different different numbers of copies, different lengths of the vectors, different numbers of PEs and different implementations of the MPI parallel libraries can have very different performance characteristics. The namelist item communication_configuration controls one of four possible combinations of the operation order during the transposes. If performance is an issue the various settings on this namelist item can be explored. See the namelist section for more details.

The transpose routines make both representations of the data equivalent until the next update to either the copies or the vars arrays, so either can be used as a data source.

ens_handle	The handle of the ensemble being transposed.
label	A character string label. If present, a timestamp with this label is printed at the start and end of the transpose.

call all_copies_to_all_vars(ens_handle, label)

```
type(ensemble_type), intent(inout)        :: ens_handle
character(len=*),    intent(in), optional :: label
```

Transposes data from a representation in which each processor has all copies of a subset of variables to one in which each has a subset of copies of all variables. In the current implementation, storage is not released so both representations are always available. However, one representation may be current while the other is out of date.

Different different numbers of copies, different lengths of the vectors, different numbers of PEs and different implementations of the MPI parallel libraries can have very different performance characteristics. The namelist item communication_configuration controls one of four possible combinations of the operation order during the transposes. If performance is an issue the various settings on this namelist item can be explored. See the namelist section for more details.

The transpose routines make both representations of the data equivalent until the next update to either the copies or the vars arrays, so either can be used as a data source.

ens_handle	The handle of the ensemble being transposed.
label	A character string label. If present, a timestamp with this label is printed at the start and end of the transpose.

call compute_copy_mean(ens_handle, start_copy, end_copy, mean_copy)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)    :: start_copy
integer,             intent(in)    :: end_copy
integer,             intent(in)    :: mean_copy
```

Computes the mean of a contiguous subset of copies starting with global index start_copy and ending with global index ens_copy. Mean is written to global index mean_copy.

When this routine is called the ensemble must have all copies of a subset of the vars. It updates the copies array with the mean, so after this call the copies array data is more current and the vars data is stale.

ens_handle	Handle for an ensemble.
start_copy	Global index of first copy in mean and sd computation.
end_copy	Global index of last copy in mean and sd computation.
mean_copy	Global index of copy into which mean is written.

call compute_copy_mean_sd(ens_handle, start_copy, end_copy, mean_copy, sd_copy)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)    :: start_copy
integer,             intent(in)    :: end_copy
integer,             intent(in)    :: mean_copy
integer,             intent(in)    :: sd_copy
```

Computes the mean and standard deviation of a contiguous subset of copies starting with global index start_copy and ending with global index ens_copy. Mean is written to index mean_copy and standard deviation to index sd_copy.

When this routine is called the ensemble must have all copies of a subset of the vars. It updates the copies arrays with the mean and sd, so after this call the copies array data is more current and the vars data is stale.

ens_handle	Handle for an ensemble.
start_copy	Global index of first copy in mean and sd computation.
end_copy	Global index of last copy in mean and sd computation.
mean_copy	Global index of copy into which mean is written.
sd_copy	Global index of copy into which standard deviation is written.

call compute_copy_mean_var(ens_handle, start_copy, end_copy, mean_copy, var_copy)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)  :: start_copy
integer,             intent(in)  :: end_copy
integer,             intent(in)  :: mean_copy
integer,             intent(in)  :: var_copy
```

Computes the mean and variance of a contiguous subset of copies starting with global index start_copy and ending with global index ens_copy. Mean is written to index mean_copy and variance to index var_copy.

When this routine is called the ensemble must have all copies of a subset of the vars. It updates the copies arrays with the mean and variance, so after this call the copies array data is more current and the vars data is stale.

ens_handle	Handle for an ensemble.
start_copy	Global index of first copy in mean and sd computation.
end_copy	Global index of last copy in mean and sd computation.
mean_copy	Global index of copy into which mean is written.
var_copy	Global index of copy into which variance is written.

call prepare_to_update_vars(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the ens_handle%vars array when the data is going to be updated, and the incoming vars array should have the most current data representation.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

`ens_handle`	Handle for the ensemble being accessed directly.

call prepare_to_update_copies(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the `ens_handle%copies` array when the data is going to be updated, and the incoming copies array should have the most current data representation.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

`ens_handle`	Handle for the ensemble being accessed directly.

call prepare_to_read_from_vars(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the `ens_handle%vars` array for reading only, when the incoming vars array should have the most current data representation.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated

representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

ens_handle	Handle for the ensemble being accessed directly.

call prepare_to_read_from_copies(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the ens_handle%copies array for reading only, when the incoming copies array should have the most current data representation.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (all_vars_to_all_copies() or all_copies_to_all_vars()) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

ens_handle	Handle for the ensemble being accessed directly.

call prepare_to_write_to_vars(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the ens_handle%vars array for writing. This routine differs from the 'update' version in that it doesn't care what the original data state is. This routine might be used in the case where an array is being filled for the first time and consistency with the data in the copies array is not an issue.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (all_vars_to_all_copies() or all_copies_to_all_vars()) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

ens_handle	Handle for the ensemble being accessed directly.

call prepare_to_write_to_copies(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the ens_handle%copies array for writing. This routine differs from the 'update' version in that it doesn't care what the original data state is. This routine might be used in the case where an array is being filled for the first time and consistency with the data in the vars array is not an issue.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (all_vars_to_all_copies() or all_copies_to_all_vars()) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

ens_handle	Handle for the ensemble being accessed directly.

6.199.5 Private interfaces

	assign_tasks_to_pes
	calc_tasks_on_each_node
	create_pe_to_task_list
	get_copy_list
	get_max_num_copies
	get_max_num_vars
	get_var_list
	round_robin
	set_up_ens_distribution
	simple_layout
	sort_task_list
	timestamp_message

var = get_max_num_copies(num_copies)

```
integer              :: get_max_num_copies
integer, intent(in)  :: num_copies
```

Returns the largest number of copies that are on any pe when var complete. Depends on distribution_type with only option 1 currently implemented. Used to get size for creating storage to receive a list of the copies on a PE.

`var`	Returns the largest number of copies any an individual PE when var complete.
`num_copies`	Total number of copies in the ensemble.

var = get_max_num_vars(num_vars)

```
integer              :: get_max_num_vars
integer, intent(in)  :: num_vars
```

Returns the largest number of vars that are on any pe when copy complete. Depends on distribution_type with only option 1 currently implemented. Used to get size for creating storage to receive a list of the vars on a PE.

var	Returns the largest number of vars any an individual PE when copy complete.
num_copies	Total number of vars in an ensemble vector.

call set_up_ens_distribution(ens_handle)

```
type(ensemble_type), intent(inout) :: ens_handle
```

Figures out how to lay out the copy complete and vars complete distributions. The distribution_type identifies different options. Only distribution_type 1 is implemented. This puts every Nth var or copy on a given processor where N is the total number of processes.

ens_handle	Handle for an ensemble.

call get_var_list(num_vars, pe, var_list, pes_num_vars)

```
integer,   intent(in)     :: num_vars
integer,   intent(in)     :: pe
integer,   intent(out)    :: var_list(:)
integer,   intent(out)    :: pes_num_vars
```

Returns a list of the vars stored by process pe when copy complete and the number of these vars. var_list must be dimensioned large enough to hold all vars. Depends on distribution_type with only option 1 currently implemented.

call get_copy_list(num_copies, pe, copy_list, pes_num_copies)

```
integer,   intent(in)     :: num_copies
integer,   intent(in)     :: pe
integer,   intent(out)    :: copy_list(:)
integer,   intent(out)    :: pes_num_copies
```

Returns a list of the copies stored by process pe when var complete and the number of these copies. copy_list must be dimensioned large enough to hold all copies. Depends on distribution_type with only option 1 currently implemented.

call timestamp_message(msg [, sync] [, alltasks])

```
character(len=*), intent(in)           :: msg
logical,          intent(in), optional :: sync
logical,          intent(in), optional :: alltasks
```

Write current time and message to stdout and log file. If sync is present and true, sync mpi jobs before printing time. If alltasks is present and true, all tasks print the time. The default is only task 0 prints a timestamp.

msg	character string to prepend to the time info
sync	if present and true, execute an MPI_Barrier() to sync all MPI tasks before printing the time. this means the time will be the value of the slowest of the tasks to reach this point.
alltasks	if present and true, have all tasks print out a timestamp. the default is for just task 0 to print. the usual combination is either sync=true and alltasks=false, or sync=false and alltasks=true.

call print_ens_handle(ens_handle, force, label)

```
type(ensemble_type),        intent(in) :: ens_handle
logical,          optional, intent(in) :: force
character(len=*), optional, intent(in) :: label
```

For debugging use, dump the contents of an ensemble handle derived type. If the debug namelist item is true, this will print in any case. If debug is false, set force to true to force printing. The optional string label can help provide context for the output.

ens_handle	The derived type to print information about.
force	If the debug namelist item is false, set this to true to enable printing.
label	Optional string label to print to provide context for the output.

call assign_tasks_to_pes(ens_handle, nEns_members, layout_type)

```
type(ensemble_type), intent(inout)    :: ens_handle
integer,             intent(in)       :: nEns_members
integer,             intent(inout)    :: layout_type
```

Calulate the task layout based on the tasks per node and the total number of tasks. Allows the user to spread out the ensemble members as much as possible to balance memory usage between nodes. Possible options: 1. Standard task layout - first n tasks have the ensemble members my_pe = my_task_id() 2. Round-robin on the nodes

ens_handle	Handle for an ensemble.

call round_robin(ens_handle)

```
type(ensemble_type), intent(inout)    :: ens_handle
```

Round-robin MPI task layout starting at the first node. Starting on the first node forces pe 0 = task 0. The smoother code assumes task 0 has an ensemble member. If you want to break the assumption that pe 0 = task 0, this routine is a good place to start. Test with the smoother.

ens_handle	Handle for an ensemble.

call create_pe_to_task_list(ens_handle)

```
type(ensemble_type), intent(inout)    :: ens_handle
```

Creates the ens_handle%pe_to_task_list. ens_handle%task_to_pe_list must have been assigned first, otherwise this routine will just return nonsense.

ens_handle	Handle for an ensemble.

call calc_tasks_on_each_node(nodes, last_node_task_number)

```
integer, intent(out)  :: last_node_task_number
integer, intent(out)  :: nodes
```

Finds the of number nodes and how many tasks are on the last node, given the number of tasks and the tasks_per_node (ptile). The total number of tasks is num_pes = task_count() The last node may have fewer tasks, for example, if ptile = 16 and the number of mpi tasks = 17

call simple_layout(ens_handle, n)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)    :: n
```

assigns the arrays task_to_pe_list and pe_to_task list for the simple layout where my_pe = my_task_id()

ens_handle
: Handle for an ensemble.

n
: size

call sort_task_list(i, idx, n)

```
integer, intent(in)    :: n
integer, intent(inout) :: x(n)   ! array to be sorted
integer, intent(out)   :: idx(n) ! index of sorted array
```

sorts an array and returns the sorted array, and the index of the original array

n
size

x(n)
array to be sorted

idx(n)
index of sorted array

call map_pe_to_task(ens_handle, p)

```
type(ensemble_type), intent(in) :: ens_handle
integer,             intent(in) :: p
```

Return the physical task for my_pe

ens_handle	Handle for an ensemble.
p	The MPI task corresponding to the given PE number

call map_task_to_pe(ens_handle, t)

```
type(ensemble_type), intent(in) :: ens_handle
integer,             intent(in) :: t
```

Return my_pe corresponding to the physical task

ens_handle	Handle for an ensemble.
t	Return the PE corresponding to the given MPI task number.

Files

- input.nml
- State vector restart files, either one for all copies or one per copy.
- State vector output files, either one for all copies or one per copy.

References

1. none

Private components

N/A

6.200 MODULE random_seq_mod

6.200.1 Overview

Provides access to any number of reproducible random sequences. Can sample from uniform, gaussian, two-dimensional gaussian, gamma, inverse gamma, and exponential distributions.

The current random sequence generator is a Fortran version of the GNU Library implementation of the Mersenne Twister algorithm. The original code is in the C language and the conversion to Fortran was done by the DART team.

There are test programs in the `developer_tests/random_seq` directory which show examples of calling these routines. Build and run these tests in the `test` subdirectory.

6.200.2 Other modules used

```
types_mod
utilities_mod
```

6.200.3 Public interfaces

use random_seq_mod, only :	random_seq_type
	init_random_seq
	random_uniform
	random_gaussian
	several_random_gaussians
	twod_gaussians
	random_gamma
	random_inverse_gamma
	random_exponential

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

```
type random_seq_type
   private

   integer     :: mti
   integer(i8) :: mt(624)
   real(r8)    :: lastg
   logical     :: gset

end type random_seq_type
```

This type is used to uniquely identify a sequence. Keeps the state history of the linear congruential number generator. In this implementation it is based on the Mersenne Twister from the GNU Scientific Library.

call init_random_seq(r, [, seed])

```
type(random_seq_type),           intent(inout) :: r
integer,               optional, intent(in)    :: seed
```

Initializes a random sequence for use. This must be called before any random numbers can be generated from this sequence. Any number of independent, reproducible random sequences can be generated by having multiple instances of a random_seq_type. A specified integer seed, optional, can produce a specific 'random' sequence.

r	A random sequence type to be initialized.
seed	A seed for a random sequence.

var = random_uniform(r)

```
real(r8)                              :: random_uniform
type(random_seq_type), intent(inout) :: r
```

Returns a random draw from a uniform distribution on interval [0,1].

random_uniform	A random draw from a Uniform[0,1] distribution.
r	An initialized random sequence type.

var = random_gaussian(r, mean, standard_deviation)

```
real(r8)                              :: random_gaussian
type(random_seq_type), intent(inout) :: r
```

(continues on next page)

(continued from previous page)

```
real(r8),              intent(in)    :: mean
real(r8),              intent(in)    :: standard_deviation
```

Returns a random draw from a Gaussian distribution with the specified mean and standard deviation.

See this Wikipedia page for more explanation about this function.

random_gaussian	A random draw from a gaussian distribution.
r	An initialized random sequence type.
mean	Mean of the gaussian.
standard_deviation	Standard deviation of the gaussian.

call several_random_gaussians(r, mean, standard_deviation, n, rnum)

```
type(random_seq_type),  intent(inout) :: r
real(r8),               intent(in)    :: mean
real(r8),               intent(in)    :: standard_deviation
integer,                intent(in)    :: n
real(r8), dimension(:), intent(out)   :: rnum
```

Returns n random samples from a gaussian distribution with the specified mean and standard deviation. Array rnum must be at least size n.

r	An initialized random sequence type.
mean	Mean of the Gaussian to be sampled.
standard_deviation	Standard deviation of the Gaussian.
n	Number of samples to return
rnum	The random samples of the Gaussian.

call twod_gaussians(r, mean, cov, rnum)

```
type(random_seq_type),    intent(inout) :: r
real(r8), dimension(2),   intent(in)    :: mean
real(r8), dimension(2,2), intent(in)    :: cov
real(r8), dimension(2),   intent(out)   :: rnum
```

Returns a random draw from a 2D gaussian distribution with the specified mean and covariance.

The algorithm used is from Knuth, exercise 13, section 3.4.1. See this Wikipedia page for more explanation about this function.

r	An initialized random sequence type.
mean	Mean of 2D gaussian distribution.
cov	Covariance of 2D gaussian.
rnum	Returned random draw from gaussian.

var = random_gamma(r, rshape, rscale)

```
real(r8)                              :: random_gamma
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: rshape
real(r8),              intent(in)    :: rscale
```

Returns a random draw from a Gamma distribution with specified `rshape` and `rscale`. Both must be positive.

Note that there are three different parameterizations in common use:

1. With shape parameter (kappa) and scale parameter (theta).
2. With shape parameter (alpha) and rate parameter (beta). Alpha is the same as kappa, and beta is an inverse scale parameter so = 1/.
3. With shape parameter (kappa) and mean parameter (mu). = /, so = /.

This form uses the first parameterization, shape () and scale (). The distribution mean is and the variance is (^{2}). This routine is based on the Gamma(a,b) generator from the GNU Scientific library. See this Wikipedia page for more explanation of the various parameterizations of this function.

`random_gamma`	A random draw from a gamma distribution.
`r`	An initialized random sequence type.
`rshape`	Shape parameter. Often written as either alpha or kappa.
`rscale`	Scale parameter. Often written as theta. If you have a rate parameter (often beta) pass in (1/rate) for scale.

var = random_inverse_gamma(r, rshape, rscale)

```
real(r8)                              :: random_inverse_gamma
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: rshape
real(r8),              intent(in)    :: rscale
```

Returns a random draw from an inverse Gamma distribution with the specified `shape` and `scale`. Both must be positive. If you have 'rate' instead of 'scale' pass in (1/rate) for scale.

See this Wikipedia page for more explanation about this function.

`random_inverse_gamma`	A random draw from an inverse gamma distribution.
`r`	An initialized random sequence type.
`rshape`	Shape parameter. Often written as either alpha or kappa.
`rscale`	Scale parameter. Often written as theta. If you have a rate parameter (often beta) pass in (1/rate) for scale.

var = random_exponential(r, rate)

```
real(r8)                              :: random_exponential
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: rate
```

Returns a random draw from an exponential distribution with the specified `rate`. If you have a scale parameter (which is the same as the mean, the standard deviation, and the survival parameter), specify (1/scale) for rate.

See this Wikipedia page for more explanation about this function.

`random_exponential`	A random draw from an exponential distribution.
`r`	An initialized random sequence type.
`rate`	Rate parameter. Often written as lambda. If you have a scale parameter pass in (1/scale) for rate.

6.200.4 Namelist

This module has no namelist input.

6.200.5 Files

- NONE

6.200.6 References

1. Knuth, Vol 2.
2. GNU Scientific Library Reference Manual

6.200.7 Private components

	init_ran
	ran_unif
	ran_gauss
	ran_gamma

call init_ran(s, seed)

```
type(random_seq_type), intent(out) :: s
integer,               intent(in)  :: seed
```

Initializes a random sequence with an integer. Any sequence initialized with the same integer will produce the same sequence of pseudo-random numbers.

s	A random sequence to be initialized
seed	An integer seed to start the sequence.

var = ran_unif(s)

```
real(r8)                           :: ran_unif
type(random_seq_type), intent(inout) :: s
```

Generate the next uniform [0, 1] random number in the sequence.

ran_unif	Next uniformly distributed [0, 1] number in sequence.
s	A random sequence.

var = ran_gauss(s)

```
real(r8)                           :: ran_gauss
type(random_seq_type), intent(inout) :: s
```

Generates a random draw from a standard gaussian.

ran_gauss	A random draw from a standard gaussian.
s	A random sequence.

var = ran_gamma(r, rshape, rscale)

```
real(r8)                           :: ran_gamma
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: rshape
real(r8),              intent(in)    :: rscale
```

Generates a random draw from a Gamma distribution. See notes in the random_gamma() section about (alpha,beta) vs (kappa,theta) vs (kappa,mu) parameterizations. This is transcribed from C code in the GNU Scientific library and keeps the (shape,scale) interface.

ran_gamma	A random draw from a Gamma distribution.
r	A random sequence.
rshape	Shape parameter.
rscale	Scale parameter. (This is the inverse of a rate parameter.)

6.201 MODULE mpi_utilities_mod

6.201.1 Overview

This module provides subroutines which utilize the MPI (Message Passing Interface) parallel communications library. DART does **not** require MPI; to compile without using MPI substitute the `null_mpi_utilities_mod.f90` file for this one. That file contains the same module name and public entry points as this one but implements a serial version of all the routines. However, to be able to run most larger models with a reasonable number of ensemble members (e.g. 30-100) MPI will be needed.

Several DART executables can be compiled and run as either a serial program or a parallel program. Most work directories in the DART distribution source tree have a `quickbuild.sh` script. To build DART without MPI use `./quickbuild.sh nompi`. No source code changes are required to switch between using mpi or no mpi.

A parallel program generally runs faster and requires less memory per CPU than the serial code. It requires an implementation of the MPI library and run-time system to pass data between different nodes on a parallel cluster or supercomputer. There is a lot of information about MPI on the web. See here for an intro to MPI and parallel programming, and here for downloads and technical help.

Most of the larger models need to be compiled and run with MPI because of limitations on total memory accessible by a single executable. The smaller models (e.g. any of the Lorenz models) can generally be run as a serial program without needing MPI.

The MPI distributions usually include a module named `mpi` which defines the public entry points and the types and names of the routine arguments. However there are build-time options and older distributions which only supply an `mpi.h` include file. If you get a compile-time error about the mpi module being missing, edit the source code in `mpi_utilities/mpi_utilities_mod.f90` and comment out the `use mpi` line and comment in the `include 'mpi.h'` line. The 'use' line must be before the 'contains' line, while the 'include' line must be after, so do not move the existing lines. Just comment them in or out depending on which one you need to use.

To preserve backwards compatibility this code does not require a namelist. However there is a namelist defined in the source file which contains some useful run-time options. To enable it edit the source file in `mpi_utilities/mpi_utilities_mod.f90` and set `use_namelist` to .TRUE. and recompile. The code will then read the namelist described below. Messages printed to the nml output log file will confirm whether the defaults are being used or if the namelist is being read in.

6.201.2 Namelist

The source code defines a namelist, but for backwards compatibility it is not read in unless the source code in `mpi_utilities/mpi_utilities_mod.f90` is edited, the module global variable `use_namelist` is changed from .FALSE. to .TRUE., and then all executables are recompiled.

If enabled, this namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&mpi_utilities_nml
    reverse_task_layout        = .false.
    all_tasks_print            = .false.
    verbose                    = .false.
    async2_verbose             = .false.
    async4_verbose             = .false.
    shell_name                 = ''
    separate_node_sync         = .false.
    create_local_comm          = .true.
    make_copy_before_sendrecv  = .false.
   /
```

Item	Type	Description
reverse_task_layout	logical	The synchronizing mechanism between the job script and the parallel filter in async=4 mode relies on the script and task 0 running on the same node (in the same memory space if the nodes have multiple processors). Some MPI implementations (OpenMPI being the most commonly used one) lay the tasks out so that the last task is on the same node as the script. If the async 4 model advance never starts but there are no error messages, try setting this to .TRUE. before running. See also the 'async4_verbose' flag below.
all_tasks_print	logical	In the parallel filter, informational messages only print from task 0 to avoid N copies of the same messages. Error messages and warnings print no matter which task they occur in. If this variable is set to true, even messages will print from all tasks.
verbose	logical	USE WITH CAUTION! This flag enables debugging print messages for every MPI call - sends, receives, barriers - and is very, very verbose. In most cases the size of the output file will exceed the filesystem limits or will cause the executable to run so slowly that it will not be useful. However in small testcases this can be useful to trace problems.
async2_verbose	logical	Print out messages about the handshaking between filter and the advance model scripts when running in async=2 mode. Not anywhere as verbose as the flag above; in most cases the output volume is reasonable.
async4_verbose	logical	Print out messages about the handshaking between filter and the run script when running in async=4 mode. Not anywhere as verbose as the flag above; in most cases the output volume is reasonable.
shell_name	character(len=129)	If running on compute nodes which do not have the expected default shell for async=2 or async=4 mode, specify the full pathname of the shell to execute the script. Not normally needed on most systems we run on. (However, at least one type of Cray system has this need.)
separate_node_async	logical	Not supported yet. Will use files to handshake between the filter executable and the run script in async=4 mode when the launch script is not running on any of the same nodes as the filter tasks.
create_local_comm	logical	The DART MPI routines normally create a separate local MPI communicator instead of using MPI_COMM_WORLD. This keeps DART communications separate from any other user code. To use the default world communicator set this to .FALSE. . Normal use should leave this true.
make_copy_before_sends	logical	Workaround for old MPI bug. Should be .false.

6.201.3 Other modules used

```
types_mod
utilities_mod
time_manager_mod
mpi  (or mpif.h if mpi module not available)
```

6.201.4 Public interfaces

use mpi_utilities_mod, only :	initialize_mpi_utilities
	finalize_mpi_utilities
	task_count
	my_task_id
	task_sync
	block_task
	restart_task
	array_broadcast
	send_to
	receive_from
	iam_task0
	broadcast_send
	broadcast_recv
	shell_execute
	sleep_seconds
	sum_across_tasks
	get_dart_mpi_comm
	exit_all

call initialize_mpi_utilities([progname] [, alternatename])

```
character(len=*), intent(in), optional :: progname
character(len=*), intent(in), optional :: alternatename
```

Initializes the MPI library, creates a private communicator, stores the total number of tasks and the local task number for later use, and registers this module. This routine calls `initialize_utilities()` internally before returning, so the calling program need only call this one routine to initialize the DART internals.

On some implementations of MPI (in particular some variants of MPICH) it is best to initialize MPI before any I/O is done from any of the parallel tasks, so this routine should be called as close to the process startup as possible.

It is not an error to try to initialize the MPI library more than once. It is still necessary to call this routine even if the application itself has already initialized the MPI library. Thise routine creates a private communicator so internal communications are shielded from any other communication called outside the DART libraries.

It is an error to call any of the other routines in this file before calling this routine.

progname	If given, written to the log file to document which program is being started.
alternatename	If given, use this name as the log file instead of the default dart_log.out.

call finalize_mpi_utilities([callfinalize] [, async])

```
logical, intent(in), optional  :: callfinalize
integer, intent(in), optional  :: async
```

Frees the local communicator, and shuts down the MPI library unless callfinalize is specified and is .FALSE.. On some hardware platforms it is problematic to try to call print or write from the parallel tasks after finalize has been executed, so this should only be called immediately before the process is ready to exit. This routine does an MPI_Barrier() call before calling MPI_Finalize() to ensure all tasks are finished writing.

If the application itself is using MPI the callfinalize argument can be used to defer closing the MPI library until the application does it itself. This routine does close the DART log file and releases the local communicator even if not calling MPI_Finalize, so no other DART routines which might generate output can be used after calling this routine.

It is an error to call any of the other routines in this file after calling this routine.

callfinalize	If false, do not call the MPI_Finalize() routine.
async	If the model advance mode (selected by the async namelist value in the filter_nml section) requires any synchronization or actions at shutdown, this is done. Currently async=4 requires an additional set of actions at shutdown time.

var = task_count()

```
integer         :: task_count
```

Returns the total number of MPI tasks this job was started with. Note that MPI task numbers start at 0, but this is a count. So a 4-task job will return 4 here, but the actual task numbers will be from 0 to 3.

var	Total number of MPI tasks in this job.

var = my_task_id()

```
integer         :: my_task_id
```

Returns the local MPI task number. This is one of the routines in which all tasks can make the same function call but each returns a different value. The return can be useful in creating unique filenames or otherwise distinguishing resources which are not shared amongst tasks. MPI task numbers start at 0, so valid task id numbers for a 4-task job will be 0 to 3.

var	My unique MPI task id number.

call task_sync()

Synchronize tasks. This call does not return until all tasks have called this routine. This ensures all tasks have reached the same place in the code before proceeding. All tasks must make this call or the program will hang.

call send_to(dest_id, srcarray [, time])

```
integer,                   intent(in) :: dest_id
real(r8), dimension(:),    intent(in) :: srcarray
type(time_type), optional, intent(in) :: time
```

Use the MPI library to send a copy of an array of data from one task to another task. The sending task makes this call; the receiving task must make a corresponding call to `receive_from()`.

If `time` is specified, it is also sent to the receiving task. The receiving call must match this sending call regarding this argument; if `time` is specified here it must also be specified in the receive; if not given here it cannot be given in the receive.

The current implementation uses `MPI_Ssend()` which does a synchronous send. That means this routine will not return until the receiving task has called the receive routine to accept the data. This may be subject to change; MPI has several other non-blocking options for send and receive.

dest_id	The MPI task id of the receiver.
srcarray	The data to be copied to the receiver.
time	If specified, send the time as well.

The send and receive subroutines must be used with care. These calls must be used in pairs; the sending task and the receiving task must make corresponding calls or the tasks will hang. Calling them with different array sizes will result in either a run-time error or a core dump. The optional time argument must either be given in both calls or in neither or one of the tasks will hang. (Executive summary: There are lots of ways to go wrong here.)

call receive_from(src_id, destarray [, time])

```
integer, intent(in)                    :: src_id
real(r8), dimension(:), intent(out)    :: destarray
type(time_type), intent(out), optional :: time
```

Use the MPI library to receive a copy of an array of data from another task. The receiving task makes this call; the sending task must make a corresponding call to `send_to()`. Unpaired calls to these routines will result in the tasks hanging.

If `time` is specified, it is also received from the sending task. The sending call must match this receiving call regarding this argument; if `time` is specified here it must also be specified in the send; if not given here it cannot be given in the send.

The current implementation uses `MPI_Recv()` which does a synchronous receive. That means this routine will not return until the data has arrived in this task. This may be subject to change; MPI has several other non-blocking options for send and receive.

`src_id`	The MPI task id of the sender.
`destarray`	The location where the data from the sender is to be placed.
`time`	If specified, receive the time as well.

See the notes section of `send_to()`.

call exit_all(exit_code)

```
integer, intent(in)   :: exit_code
```

A replacement for calling the Fortran intrinsic `exit`. This routine calls `MPI_Abort()` to kill all MPI tasks associated with this job. This ensures one task does not exit silently and leave the rest hanging. This is not the same as calling `finalize_mpi_utilities()` which waits for the other tasks to finish, flushes all messages, closes log files cleanly, etc. This call immediately and abruptly halts all tasks associated with this job.

Depending on the MPI implementation and job control system, the exit code may or may not be passed back to the calling job script.

`exit_code`	A numeric exit code.

This routine is now called from the standard error handler. To avoid circular references this is NOT a module routine. Programs which are compiled without the mpi code must now compile with the `null_mpi_utilities_mod.f90` file to satisfy the call to this routine in the error handler.

call array_broadcast(array, root)

```
real(r8), dimension(:), intent(inout) :: array
integer, intent(in)                   :: root
```

All tasks must make this call together, but the behavior in each task differs depending on whether it is the `root` or not. On the task which has a task id equal to `root` the contents of the array will be sent to all other tasks. On any task which has a task id *not* equal to `root` the array is the location where the data is to be received into. Thus `array` is intent(in) on root, and intent(out) on all other tasks.

When this routine returns, all tasks will have the contents of the root array in their own arrays.

`array`	Array containing data to send to all other tasks, or the location in which to receive data.
`root`	Task ID which will be the data source. All others are destinations.

This is another of the routines which must be called by all tasks. The MPI call used here is synchronous, so all tasks block here until everyone has called this routine.

var = iam_task0()

```
logical                         :: iam_task0
```

Returns `.TRUE.` if called from the task with MPI task id 0. Returns `.FALSE.` in all other tasks. It is frequently the case that some code should execute only on a single task. This allows one to easily write a block surrounded by `if (iam_task0()) then ...` .

`var`	Convenience function to easily test and execute code blocks on task 0 only.

call broadcast_send(from, array1 [, array2] [, array3] [, array4] [, array5] [, scalar1] [, scalar2] [, scalar3] [, scalar4] [, scalar5])

```
integer, intent(in)                   :: from
real(r8), dimension(:), intent(inout) :: array1
real(r8), dimension(:), intent(inout), optional :: array2
real(r8), dimension(:), intent(inout), optional :: array3
real(r8), dimension(:), intent(inout), optional :: array4
real(r8), dimension(:), intent(inout), optional :: array5
real(r8), intent(inout), optional :: scalar1
real(r8), intent(inout), optional :: scalar2
real(r8), intent(inout), optional :: scalar3
real(r8), intent(inout), optional :: scalar4
real(r8), intent(inout), optional :: scalar5
```

Cover routine for `array_broadcast()`. This call must be matched with the companion call `broadcast_recv()`. This routine should only be called on the task which is the root of the broadcast; it will be the data source. All other tasks must call `broadcast_recv()`. This routine sends up to 5 data arrays and 5 scalars in a single call. A common pattern in the DART filter code is sending 2 arrays, but other combinations exist. This routine ensures that `from` is the same as the current task ID. The arguments to this call must be matched exactly in number and type with the companion call to `broadcast_recv()` or an error (or hang) will occur.

In reality the data here are `intent(in)` only but this routine will be calling `array_broadcast()` internally and so must be `intent(inout)` to match.

`from`	Current task ID; the root task for the data broadcast.
`array1`	First data array to be broadcast.
array2	If given, second data array to be broadcast.
array3	If given, third data array to be broadcast.
array4	If given, fourth data array to be broadcast.
array5	If given, fifth data array to be broadcast.
scalar1	If given, first data scalar to be broadcast.
scalar2	If given, second data scalar to be broadcast.
scalar3	If given, third data scalar to be broadcast.
scalar4	If given, fourth data scalar to be broadcast.
scalar5	If given, fifth data scalar to be broadcast.

This is another of the routines which must be called consistently; only one task makes this call and all other tasks call the companion `broadcast_recv` routine. The MPI call used here is synchronous, so all tasks block until everyone has called one of these two routines.

call broadcast_recv(from, array1 [, array2] [, array3] [, array4] [, array5] [, scalar1] [, scalar2] [, scalar3] [, scalar4] [, scalar5])

```
integer, intent(in)                    :: from
real(r8), dimension(:), intent(inout) :: array1
real(r8), dimension(:), intent(inout), optional :: array2
real(r8), dimension(:), intent(inout), optional :: array3
real(r8), dimension(:), intent(inout), optional :: array4
real(r8), dimension(:), intent(inout), optional :: array5
real(r8), intent(inout), optional :: scalar1
real(r8), intent(inout), optional :: scalar2
real(r8), intent(inout), optional :: scalar3
real(r8), intent(inout), optional :: scalar4
real(r8), intent(inout), optional :: scalar5
```

Cover routine for `array_broadcast()`. This call must be matched with the companion call `broadcast_send()`. This routine must be called on all tasks which are *not* the root of the broadcast; the arguments specify the location in which to receive data from the root. (The root task should call `broadcast_send()`.) This routine receives up to 5 data arrays and 5 scalars in a single call. A common pattern in the DART filter code is receiving 2 arrays, but other combinations exist. This routine ensures that `from` is *not* the same as the current task ID. The arguments to this call must be matched exactly in number and type with the companion call to `broadcast_send()` or an error (or hang) will occur.

In reality the data arrays here are `intent(out)` only but this routine will be calling `array_broadcast()` internally and so must be `intent(inout)` to match.

from	The task ID for the data broadcast source.
array1	First array location to receive data into.
array2	If given, second data array to receive data into.
array3	If given, third data array to receive data into.
array4	If given, fourth data array to receive data into.
array5	If given, fifth data array to receive data into.
scalar1	If given, first data scalar to receive data into.
scalar2	If given, second data scalar to receive data into.
scalar3	If given, third data scalar to receive data into.
scalar4	If given, fourth data scalar to receive data into.
scalar5	If given, fifth data scalar to receive data into.

This is another of the routines which must be called consistently; all tasks but one make this call and exactly one other task calls the companion broadcast_send routine. The MPI call used here is synchronous, so all tasks block until everyone has called one of these two routines.

call sum_across_tasks(addend, sum)

```
integer, intent(in)                  :: addend
integer, intent(out)                 :: sum
```

All tasks call this routine, each with their own different addend. The returned value in sum is the total of the values summed across all tasks, and is the same for each task.

addend	Single input value per task to be summed up.
sum	The sum.

This is another of those calls which must be made from each task, and the calls block until this is so.

call block_task()

Create a named pipe (fifo) and read from it to block the process in such a way that it consumes no CPU time. Beware that once you put yourself to sleep you cannot wake yourself up. Some other MPI program must call restart_task() on the same set of processors the original program was distributed over.

Even though fifos appear to be files, in reality they are implemented in the kernel. The write into the fifo must be executed on the same node as the read is pending on. See the man pages for the mkfifo(1) command for more details.

call restart_task()

Write into the pipe to restart the reading task. Note that this must be an entirely separate executable from the one which called block_task(), because it is asleep like Sleeping Beauty and cannot wake itself. See filter and wakeup_filter for examples of a program pair which uses these calls in async=4 mode.

Even though fifos appear to be files, in reality they are implemented in the kernel. The write into the fifo must be executed on the same node as the read is pending on. See the man pages for the mkfifo(1) command for more details.

call finished_task(async)

```
integer, intent(in) :: async
```

For async=4 and task id = 0, write into the main filter-to-script fifo to tell the run script that filter is exiting. Does nothing else otherwise.

Even though fifos appear to be files, in reality they are implemented in the kernel. The write into the fifo must be executed on the same node as the read is pending on. See the man pages for the mkfifo(1) command for more details.

rc = shell_execute()

```
integer                         :: shell_execute
character(len=*), intent(in)  :: execute_string
logical, intent(in), optional :: serialize
```

Wrapper routine around the system() library function to execute shell level commands from inside the Fortran program. Will wait for the command to execute and will return the error code. 0 means ok, any other number indicates error.

rc	Return code from the shell exit after the command has been executed.
execute_string	Command to be executed by the shell.
serialize	If specified and if .TRUE. run the command from each PE in turn, waiting for each to complete before beginning the next. The default is .FALSE. and does not require that all tasks call this routine. If given and .TRUE. then all tasks must make this call.

call sleep_seconds(naplength)

```
real(r8), intent(in) :: naplength
```

Wrapper routine for the sleep command. Argument is a real in seconds. Some systems have different lower resolutions for the minimum time it will sleep. This routine can round up to even seconds if a smaller than 1.0 time is given.

naplength	Number of seconds to sleep as a real value.

The amount of time this routine will sleep is not precise and might be in units of whole seconds on some platforms.

comm = get_dart_mpi_comm()

```
integer    :: get_dart_mpi_comm
```

This code creates a private communicator for DART MPI calls, in case other code in the executable is using the world communicator. This routine returns the private communicator. If it is called before the internal setup work is completed it returns MPI_COMM_WORLD. If it is called before MPI is initialized, it returns 0.

comm	The private DART communicator.

6.201.5 Files

- mpi module or
- mpif.h

Depending on the implementation of MPI, the library routines are either defined in an include file (`mpif.h`) or in a proper Fortran 90 module (`use mpi`). If it is available the module is preferred; it allows for better argument checking and optional arguments support in the MPI library calls.

6.201.6 References

- MPI: The Complete Reference; Snir, Otto, Huss-Lederman, Walker, Dongarra; MIT Press, 1996, ISBN 0-262-69184-1
- http://www-unix.mcs.anl.gov/mpi/

6.201.7 Private components

N/A

6.202 MODULE time_manager_mod

6.202.1 Overview

Provides a set of routines to manipulate both time and calendars of various types.

Time intervals are stored and defined in terms of integer number of days and integer seconds. The minimum time resolution is 1 second. Mathematical operations (e.g. addition, subtraction, multiplication) are defined on these intervals. Seconds which roll over 86400 (the number of seconds in a day) are converted into days.

Calendars interpret time intervals in terms of years, months, days. Various calendars commonly in use in the scientific community are supported.

6.202.2 Other modules used

```
types_mod
utilities_mod
```

6.202.3 Public interfaces

use time_manager_mod, only :	time_type
	operator(+)
	operator(-)
	operator(*)
	operator(/)
	operator(>)
	operator(>=)
	operator(==)
	operator(/=)
	operator(<)
	operator(<=)
	operator(//)
	set_time
	set_time_missing
	increment_time
	decrement_time
	get_time
	interval_alarm
	repeat_alarm
	THIRTY_DAY_MONTHS
	JULIAN
	GREGORIAN
	NOLEAP
	NO_CALENDAR
	GREGORIAN_MARS
	set_calendar_type
	get_calendar_type
	get_calendar_string
	set_date
	get_date
	increment_date
	decrement_date
	days_in_month
	leap_year
	length_of_year
	days_in_year
	month_name
	julian_day
	time_manager_init
	print_time
	print_date
	write_time

continues on next page

Table 7 – continued from previous page

	read_time
	interactive_time

var = set_time(seconds [, days])

```
type(time_type)               :: set_time
integer,           intent(in) :: seconds
integer, optional, intent(in) :: days
```

Fills a time type. If seconds are > 86400, they are converted into the appropriate number of days. Note that seconds are specified first.

seconds	Number of seconds. If larger than 86400, they are converted into the appropriate number of days.
days	Number of days. Default is 0.

var = set_time_missing()

```
type(time_type)                         :: set_time_missing
```

Set a time type to a missing value. The resulting time value will cause an error if used for an arithmetic operation or if get_time() is called.

var = increment_time(time, seconds [, days])

```
type(time_type)               :: increment_time
type(time_type),   intent(in) :: time
integer,           intent(in) :: seconds
integer, optional, intent(in) :: days
```

Adds the specified number of seconds and optionally, days, to the given time and returns the new time. Increments cannot be negative (see decrement_time below).

time	time value to be incremented.
seconds	number of seconds to add to given time.
days	optionally a number of days to add to the given time.

var = decrement_time(time, seconds [, days])

```
type(time_type)                        :: decrement_time
type(time_type), intent(in)            :: time
integer,         intent(in)            :: seconds
integer,         intent(in), optional  :: days
```

Subtract the specified number of seconds and optionally, days, to the given time and returns the new time. Decrements cannot be negative (see increment_time above).

time	time value to be decremented.
seconds	number of seconds to subtract from the given time.
days	optionally a number of days to subtract from the given time.

var = interval_alarm(time, time_interval, alarm, alarm_interval)

```
logical                        :: interval_alarm
type(time_type), intent(in)    :: time
type(time_type), intent(in)    :: time_interval
type(time_type), intent(inout) :: alarm
type(time_type), intent(in)    :: alarm_interval
```

Supports a commonly used type of test on times for models. Given the current time, and a time for an alarm, determines if this is the closest time to the alarm time given a time step of time_interval. If this is the closest time (alarm - time <= time_interval/2), the function returns true and the alarm is incremented by the alarm_interval. Watch for problems if the new alarm time is less than time + time_interval.

time	Current time.
time_interval	Bin size for determining if alarm time is close enough to now.
alarm	When alarm next goes off next. Updated by this routine.
alarm_interval	How often alarm goes off.

var = repeat_alarm(time, alarm_frequency, alarm_length)

```
type(time_type)                :: repeat_alarm
type(time_type), intent(in)    :: time
type(time_type), intent(in)    :: alarm_frequency
type(time_type), intent(in)    :: alarm_length
```

Repeat_alarm supports an alarm that goes off with alarm_frequency and lasts for alarm_length. If the nearest occurence of an alarm time is less than half an alarm_length from the input time, repeat_alarm is true. For instance, if the alarm_frequency is 1 day, and the alarm_length is 2 hours, then repeat_alarm is true from time 2300 on day n to time 0100 on day n + 1 for all n.

time	Current time.
alarm_frequency	How often the alarm goes off.
alarm_length	How long the alarm is true.

var = get_calendar_type()

```
integer :: get_calendar_type
```

Returns default calendar type for mapping from time to date. Calendar types are public integer parameters that define various calendars. See elsewhere in this file for the list.

var = set_date(year, month, day [, hours, minutes, seconds])

```
type(time_type)                :: set_date
integer, intent(in)            :: year
integer, intent(in)            :: month
integer, intent(in)            :: day
integer, intent(in), optional  :: hours
integer, intent(in), optional  :: minutes
integer, intent(in), optional  :: seconds
```

Given a date interpreted using the current calendar type, compute the corresponding time.

year	Integer year.
month	Integer month number.
day	Integer day number.
hours	Integer hour. Default is 0.
minutes	Integer minutes. Default is 0.
seconds	Integer seconds. Default is 0.

var = increment_date(time [, years, months, days, hours, minutes, seconds])

```
type(time_type)                :: increment_date
type(time_type), intent(in)    :: time
integer, intent(in), optional  :: years
integer, intent(in), optional  :: months
integer, intent(in), optional  :: days
integer, intent(in), optional  :: hours
integer, intent(in), optional  :: minutes
integer, intent(in), optional  :: seconds
```

Given a time and some date increment, compute a new time. The interpretation of the date depends on the currently selected calendar type.

`time`	Current time.
year	Integer years to add. Default is 0.
month	Integer months to add. Default is 0.
day	Integer days to add. Default is 0.
hours	Integer hours to add. Default is 0.
minutes	Integer minutes to add. Default is 0.
seconds	Integer seconds to add. Default is 0.

var = decrement_date(time [, years, months, days, hours, minutes, seconds])

```
type(time_type)                 :: decrement_date
type(time_type), intent(in)     :: time
integer, intent(in), optional   :: years
integer, intent(in), optional   :: months
integer, intent(in), optional   :: days
integer, intent(in), optional   :: hours
integer, intent(in), optional   :: minutes
integer, intent(in), optional   :: seconds
```

Given a time and some date decrement, compute a new time. The interpretation of the date depends on the currently selected calendar type.

`time`	Current time.
year	Integer years to subtract. Default is 0.
month	Integer months to subtract. Default is 0.
day	Integer days to subtract. Default is 0.
hours	Integer hours to subtract. Default is 0.
minutes	Integer minutes to subtract. Default is 0.
seconds	Integer seconds to subtract. Default is 0.

var = days_in_month(time)

```
integer                         :: days_in_month
type(time_type), intent(in)     :: time
```

Given a time, determine the month based on the currently selected calendar type and return the numbers of days in that month.

`time`	Current time.

var = leap_year(time)

```
logical                         :: leap_year
type(time_type),intent(in)      :: time
```

Given a time, determine if the current year is a leap year in the currently selected calendar type.

time	Current time.

var = length_of_year()

```
integer                        :: length_of_year
```

For the currently selected calendar type, return the number of days in a year if that value is fixed (e.g. there are not leap years). For other calendar types, see days_in_year() which takes a time argument to determine the current year.

var = days_in_year(time)

```
integer                         :: days_in_year
type(time_type), intent(in)     :: time
```

Given a time, determine the year based on the currently selected calendar type and return the numbers of days in that year.

time	Current time.

var = month_name(n)

```
character(len=9)                :: month_name
integer,         intent(in)     :: n
```

Return a character string containing the month name corresponding to the given month number.

n	Month number. Must be between 1 and 12, inclusive.

var = julian_day(year, month, day)

```
integer                         :: julian_day
integer,        intent(in)      :: year
integer,        intent(in)      :: month
integer,        intent(in)      :: day
```

Given a date in year/month/day format, compute the day number from the beginning of the year. The currently selected calendar type must be GREGORIAN.

year	Year number in the Gregorian calendar.
month	Month number in the Gregorian calendar.
day	Day of month in the Gregorian calendar.

var = read_time(file_unit [, form, ios_out])

```
type(time_type)                         :: read_time
integer,          intent(in)            :: file_unit
character(len=*), intent(in),  optional :: form
integer,          intent(out), optional :: ios_out
```

Read a time from the given file unit number. The unit must already be open. The default format is ascii/formatted. If an error is encountered and ios_out is specified, the error status will be returned to the caller; otherwise the error is fatal.

file_unit	Integer file unit number of an already open file.
form	Format to read the time. Options are 'formatted' or 'unformatted'. Default is 'formatted'.
ios_out	On error, if specified, the error status code is returned here. If not specified, an error calls the standard error_handler and exits.

call get_time(time, seconds [, days])

```
type(time_type), intent(in)             :: time
integer,         intent(out)            :: seconds
integer,         intent(out), optional  :: days
```

Returns days and seconds (< 86400) corresponding to a time. If the optional 'days' argument is not given, the days are converted to seconds and the total time is returned as seconds. Note that seconds preceeds days in the argument list.

time	Time to convert into seconds and days.
seconds	If days is specified, number of seconds in the current day. Otherwise, total number of seconds in time.
days	If specified, number of days in time.

call set_calendar_type(mytype) or *call set_calendar_type(calstring)*

```
integer, intent(in)               :: mytype
 or
character(len=*), intent(in)      :: calstring
```

Selects the current calendar type, for converting between time and year/month/day. The argument can either be one of the predefined calendar integer parameter types (see elsewhere in this file for the list of types), or a string which matches the name of the integer parameters. The string interface is especially suitable for namelist use.

mytype	Integer parameter to select the calendar type.

or

calstring	Character string to select the calendar type. Valid strings match the names of the integer parameters.

call get_calendar_string(mystring)

```
character(len=*), intent(out)     :: mystring
```

Return the character string corresponding to the currently selected calendar type.

mystring	Character string corresponding to the current calendar type.

call get_date(time, year, month, day, hour, minute, second)

```
type(time_type), intent(in)       :: time
integer, intent(out)              :: year
integer, intent(out)              :: month
integer, intent(out)              :: day
integer, intent(out)              :: hour
integer, intent(out)              :: minute
integer, intent(out)              :: second
```

Given a time, compute the corresponding date given the currently selected calendar type.

time	Input time.
year	Corresponding calendar year.
month	Corresponding calendar month.
day	Corresponding calendar day.
hour	Corresponding hour.
minute	Corresponding minute.
second	Corresponding second.

call time_manager_init()

Initializes any internal data needed by the time manager code. Does not need to be called before using any of the time manager routines; it will be called internally before executing any of the other routines.

call print_time(time [, str, iunit])

```
type(time_type),  intent(in)           :: time
character(len=*), intent(in), optional :: str
integer,          intent(in), optional :: iunit
```

Print the time as days and seconds. If the optional str argument is specified, print that string as a label. If iunit is specified, write output to that unit; otherwise write to standard output/terminal.

`time`	Time to be printed as days/seconds.
str	String label to print before days/seconds. Default: 'TIME: '.
iunit	Unit number to write output on. Default is standard output/terminal (unit 6).

call print_date(time [, str, iunit])

```
type(time_type),  intent(in)           :: time
character(len=*), intent(in), optional :: str
integer,          intent(in), optional :: iunit
```

Print the time as year/month/day/hour/minute/second, as computed from the currently selected calendar type. If the optional str argument is specified, print that string as a label. If iunit is specified, write output to that unit; otherwise write to standard output/terminal.

`time`	Time to be printed as a calendar date/time.
str	String label to print before date. Default: 'DATE: '.
iunit	Unit number to write output on. Default is standard output/terminal (unit 6).

call write_time(file_unit, time [, form, ios_out])

```
integer,          intent(in)              :: file_unit
type(time_type),  intent(in)              :: time
character(len=*), intent(in),  optional   :: form
integer,          intent(out), optional   :: ios_out
```

Write a time to an already open file unit. The optional 'form' argument controls whether it is formatted or unformatted. On error, the optional 'ios_out' argument returns the error code; otherwise a fatal error is triggered.

`file_unit`	Integer unit number for an already open file.
`time`	Time to write to the file.
form	String format specifier; either 'unformatted' or 'formatted'. Defaults to 'formatted'.
ios_out	If specified, on error the i/o status error code is returned here. Otherwise, the standard error handler is called and the program exits.

call interactive_time(time)

```
type(time_type), intent(inout) :: time
```

Prompt the user for a time as a calendar date, based on the currently selected calendar type. Writes prompt to standard output and reads from standard input.

`time`	Time type to be returned.

```
type time_type
   private
   integer :: seconds
   integer :: days
end type time_type
```

This type is used to define a time interval.

```
integer :: NO_CALENDAR
integer :: GREGORIAN
integer :: GREGORIAN_MARS
integer :: JULIAN
integer :: THIRTY_DAY_MONTHS
integer :: NOLEAP
```

The public integer parameters which define different calendar types. The same names defined as strings can be used to set the calendar type.

```
operator(+)
operator(-)
operator(*)
operator(/)
operator(>)
operator(>=)
operator(==)
operator(/=)
operator(<)
operator(<=)
operator(//)
```

Arithmetic operations are defined for time types, so expressions like

```
t3 = t1 + t2
```

can be constructed. To use these operators, they must be listed on the module use statement in the form specified above.

Multiplication is one time and one scalar.

Division with a single slash is integer, and returns the largest integer for which time1 >= time2 * n. Division with a double slash returns a double precision quotient of the two times.

6.202.4 Namelist

No namelist is currently defined for the time manager code.

6.202.5 Files

- none

6.202.6 References

1. none

6.202.7 Private components

N/A

6.203 MODULE utilities_mod

6.203.1 Overview

Provides a number of tools used by most DART modules including tools for file IO, diagnostic tools for registering modules and recording namelist arguments, and an error handler.

6.203.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&utilities_nml
   TERMLEVEL      = 2,
   logfilename    = 'dart_log.out',
   nmlfilename    = 'dart_log.nml',
   module_details = .true.,
   print_debug    = .false.,
   write_nml      = 'file'
/
```

The namelist controls how the logging, namelist, messages, and general utility routines behave.

Item	Type	Description
TERMLEVEL	integer	Level at which calls to error manager terminate program. The default setting is warnings and errors terminate the program. Setting this to 2 (E_ERR) means only errors terminate. Setting this to 3 means even errors do not cause an exit (which is not a good idea).
logfilename	character(len=256)	File to which the log messages are written.
nmlfilename	character(len=256)	File to which the namelist output is written. Can be the same name as the logfile.
module_details	logical	Each source code module can write out the repository version number and filename to the logfile. Verbose, but useful for knowing what version of the code was used during the run.
print_debug	logical	Setting this to .true. causes additional debug messages to print. These can be very verbose and by default are turned off.
write_nml	character(len=32)	String which controls where to write the namelist values that are being used for this execution. Valid values are: 'none', 'file', 'terminal', 'both'. 'none' turns off this write. 'file' writes a copy only to the `nmlfilename`. Writes are always in append mode, so the most recent information will be at the end of an existing file. 'terminal' will write to the job's standard output. 'both' will write both to the nml file and the standard output unit.

6.203.3 Other modules used

```
types_mod
netCDF
```

6.203.4 Public interfaces

use utilities, only :	file_exist
	get_unit
	open_file
	close_file
	timestamp
	register_module
	error_handler
	to_upper
	nc_check
	logfileunit
	nmlfileunit
	initialize_utilities
	finalize_utilities
	dump_unit_attributes
	find_namelist_in_file
	check_namelist_read
	find_textfile_dims
	file_to_text
	is_longitude_between
	get_next_filename
	set_filename_list
	set_tasknum
	set_output
	do_output
	E_DBG, DEBUG
	E_MSG, MESSAGE
	E_WARN, WARNING
	E_ERR, FATAL

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

var = file_exist(file_name)

```
logical                       :: file_exist
character(len=*), intent(in) :: file_name
```

Returns true if file_name exists in the working directory, else false.

var	True if file_name exists in working directory.
file_name	Name of file to look for.

var = get_unit()

```
integer :: get_unit
```

Returns an unused unit number for IO.

var	An unused unit number.

var = open_file(fname [, form, action])

```
integer                                 :: open_file
character(len=*), intent(in)            :: fname
character(len=*), optional, intent(in) :: form
character(len=*), optional, intent(in) :: action
```

Returns a unit number that is opened to the file fname. If form is not present or if form is "formatted" or "FORMATTED", file is opened for formatted IO. Otherwise, it is unformatted. The action string is the standard action string for Fortran IO (see F90 language description).

var	Unit number opened to file fname.
fname	Name of file to be opened.
form	Format: 'formatted' or 'FORMATTED' give formatted, anything else is unformatted. Default is formatted.
action	Standard fortran string description of requested file open action.

call timestamp([string1, string2, string3,] pos)

```
character(len=*), optional, intent(in) :: string1
character(len=*), optional, intent(in) :: string2
character(len=*), optional, intent(in) :: string3
character(len=*), intent(in)           :: pos
```

Prints the message 'Time is YYYY MM DD HH MM SS' to the logfile along with three optional message strings. If the pos argument is 'end', the message printed is 'Finished... at YYYY MM DD HH MM SS' and the logfile is closed.

string1	An optional message to be printed.
string2	An optional message to be printed.
string3	An optional message to be printed.
pos	If 'end' terminates log_file output.

call close_file(iunit)

```
integer, intent(in) :: iunit
```

Closes the given unit number. If the unit is not open, nothing happens.

iunit	File unit to be closed.

call register_module(src, rev, rdate)

```
character(len=*), intent(in) :: src
character(len=*), optional, intent(in) :: rev
character(len=*), optional, intent(in) :: rdate
```

Writes the source name to both the logfileunit and to standard out. The rev and revdate are deprecated as they are unsupported by git.

src	source file name.
rev	ignored
rdate	ignored

call error_handler(level, routine, text, src, rev, rdate [, aut, text2, text3])

```
integer, intent(in)                    :: level
character(len=*), intent(in)           :: routine
```

(continues on next page)

(continued from previous page)

```
character(len=*), intent(in)           :: text
character(len=*), intent(in)           :: src
character(len=*), intent(in)           :: rev
character(len=*), intent(in)           :: rdate
character(len=*), optional, intent(in) :: aut
character(len=*), optional, intent(in) :: text2
character(len=*), optional, intent(in) :: text3
```

Prints an error message to standard out and to the logfileunit. The message contains the routine name, an error message, the source file, revision and revision date, and optionally the author. The level of severity is message, debug, warning, or error. If the level is greater than or equal to the TERMLEVEL (set in the namelist), execution is terminated. The default TERMLEVEL only stops for ERRORS.

`level`	Error severity (message, debug, warning, error). See below for specific ations.
`routine`	Name of routine generating error.
`text`	Error message.
`src`	Source file containing routine generating message.
`rev`	Revision number of source file.
`rdate`	Revision date of source file.
aut	Author of routine.
text2	If specified, the second line of text for the error message.
text3	If specified, the third line of text for the error message.

call find_namelist_in_file(namelist_file_name, nml_name, iunit, [,write_to_logfile_in])

```
character(len=*),  intent(in)          :: namelist_file_name
character(len=*),  intent(in)          :: nml_name
integer,           intent(out)         :: iunit
logical, optional, intent(in)          :: write_to_logfile_in
```

Opens the file namelist_file_name if it exists on unit iunit. A fatal error occurs if the file does not exist (DART requires an input.nml to be available, even if it contains no values). Searches through the file for a line containing ONLY the string &nml_name (for instance &filter_nml if nml_name is "filter_nml"). If this line is found, the file is rewound and the routine returns. Otherwise, a fatal error message is issued.

`namelist`	Name of file assumed to hold the namelist.
`nml_name`	Name of the namelist to be searched for in the file, for instance, filter_nml.
`iunit`	Channel number on which file is opened.
write_to_logfile_in	While the namelist for the utilities module is read, the logfile has not yet been open because its name is in the namelist. If errors are found, have to write to standard out. So, when utilities module calls this internally, this optional argument is set to false. For all other applications, it is normally not used (default is false).

call check_namelist_read(iunit, iostat_in, nml_name, [, write_to_logfile_in])

```
integer, intent(in)                    :: iunit
integer, intent(in)                    :: iostat_in
character(len=*), intent(in)           :: nml_name
logical, optional, intent(in)          :: write_to_logfile_in
```

Once a namelist has been read from an opened namelist file, this routine checks for possible errors in the read. If the namelist read was successful, the file opened on iunit is closed and the routine returns. If iostat is not zero, an attempt is made to rewind the file on iunit and read the last line that was successfully read. If this can be done, this last line is printed with the preamble "INVALID NAMELIST ENTRY". If the attempt to read the line after rewinding fails, it is assumed that the original read (before the call to this subroutine) failed by reaching the end of the file. An error message stating that the namelist started but was never terminated is issued.

iunit	Channel number on which file is opened.
iostat_in	Error status return from an attempted read of a namelist from this file.
nml_name	The name of the namelist that is being read (for instance filter_nml).
write_to_logfile_in	While the namelist for the utilities module is read, the logfile has not yet been open because its name is in the namelist. If errors are found, have to write to standard out. So, when utilities module calls this internally, this optional argument is set to false. For all other applications, it is normally not used (default is false).

call find_textfile_dims (fname, nlines, linelen)

```
character(len=*), intent (IN)  :: fname
integer,          intent (OUT) :: nlines
integer,          intent (OUT) :: linelen
```

Determines the number of lines and maximum line length of an ASCII text file.

fname	input, character string file name
nlines	output, number of lines in the file
linelen	output, length of longest line in the file

call file_to_text (fname, textblock)

```
character(len=*),               intent (IN)  :: fname
character(len=*), dimension(:), intent (OUT) :: textblock
```

Opens the given filename and reads ASCII text lines into a character array.

fname	input, character string file name
textblock	output, character array of text in the file

var = is_longitude_between(lon, minlon, maxlon [, doradians])

```
real(r8), intent(in)           :: lon
real(r8), intent(in)           :: minlon
real(r8), intent(in)           :: maxlon
logical,  intent(in), optional :: doradians
logical                        :: is_longitude_between
```

Uniform way to test longitude ranges, in degrees, on a globe. Returns true if lon is between min and max, starting at min and going EAST until reaching max. Wraps across 0 longitude. If min equals max, all points are inside. Includes endpoints. If optional arg doradians is true, do computation in radians between 0 and 2*PI instead of default 360. There is no rejection of input values based on range; they are all converted to a known range by calling modulo() first.

var	True if lon is between min and max.
lon	Location to test.
minlon	Minimum longitude. Region will start here and go east.
maxlon	Maximum longitude. Region will end here.
doradians	Optional argument. Default computations are in degrees. If this argument is specified and is .true., do the computation in radians, and wrap across the globe at 2 * PI. All inputs must then be specified in radians.

var = get_next_filename(listname, lineindex)

```
character(len=*),  intent(in) :: listname
integer,           intent(in) :: lineindex
character(len=128)            :: get_next_filename
```

Returns the specified line of a text file, given a filename and a line number. It returns an empty string when the line number is larger than the number of lines in a file.

Intended as an easy way to process a list of files. Use a command like 'ls > out' to create a file containing the list, in order, of files to be processed. Then call this function with an increasing index number until the return value is empty.

var	An ascii string, up to 128 characters long, containing the contents of line `lineindex` of the input file.
listname	The filename to open and read lines from.
lineindex	Integer line number, starting at 1. If larger than the number of lines in the file, the empty string '' will be returned.

var = set_filename_list(name_array, listname, caller_name)

```
character(len=*),  intent(inout) :: name_array
character(len=*),  intent(in)    :: listname
```

(continues on next page)

(continued from previous page)

```
character(len=*),  intent(in)    :: caller_name
integer                          :: var
```

Returns the count of filenames specified. Verifies that one of either the name_array or the listname was specified but not both. If the input was a listname copy the names into the name_array so when this routine returns all the filenames are in name_array(). Verifies that no more than the allowed number of names was specified if the input was a listname file.

var	The count of input files specified.
name_array	Array of input filename strings. Either this item or the listname must be specified, but not both.
listname	The filename to open and read filenames from, one per line. Either this item or the name_array must be specified but not both.
caller_name	Calling subroutine name, used for error messages.

call to_upper(string)

```
character(len=*), intent (INOUT) :: string
```

Converts the character string to UPPERCASE - in place. The input string **is** modified.

string	any character string

call nc_check(istatus, subr_name [, context])

```
integer, intent(in)                    :: istatus
character(len=*), intent(in)           :: subr_name
character(len=*), optional, intent(in) :: context
```

Check the return code from a netcdf call. If no error, return without taking any action. If an error is indicated (in the istatus argument) then call the error handler with the subroutine name and any additional context information (e.g. which file or which variable was being processed at the time of the error). All errors are currently hardcoded to be FATAL and this routine will not return.

This routine calls a netCDF library routine to construct the text error message corresponding to the error code in the first argument. An example use of this routine is:

```
call nc_check(nf90_create(path = trim(ncFileID%fname), cmode = nf90_share, ncid =␣
↪ncFileID%ncid), &
              'init_diag_output', 'create '//trim(ncFileID%fname))
```

istatus	The return value from any netCDF call.
subr_name	String name of the current subroutine, used in case of error.
context	Additional text to be used in the error message, for example to indicate which file or which variable is being processed.

call set_tasknum(tasknum)

```
integer, intent(in)                :: tasknum
```

Intended to be used in the MPI multi-task case. Sets the local task number, which is then prepended to subsequent messages.

tasknum	Task number returned from MPI_Comm_Rank(). MPI task numbers are 0 based, so for a 4-task job these numbers are 0-3.

call set_output(doflag)

```
logical, intent(in)                :: doflag
```

Set the status of output. Can be set on a per-task basis if you are running with multiple tasks. If set to false only warnings and fatal errors will write to the log. The default in the multi-task case is controlled by the MPI module initialization code, which sets task 0 to .TRUE. and all other tasks to .FALSE.

doflag	Sets, on a per-task basis, whether messages are to be written to the logfile or standard output. Warnings and errors are always output.

var = do_output()

```
logical                        :: do_output
```

Returns true if this task should write to the log, false otherwise. Set by the `set_output()` routine. Defaults to true for the single task case. Can be used in code like so:

```
if (do_output()) then
 write(*,*) 'At this point in the code'
endif
```

var	True if this task should write output.

call initialize_utilities([progname] [, alternatename])

```
character(len=*), intent(in), optional :: progname
character(len=*), intent(in), optional :: alternatename
```

Reads the namelist and opens the logfile. Records the values of the namelist and registers this module.

progname	If given, use in the timestamp message in the log file to say which program is being started.
alternatename	If given, log filename to use instead of the value in the namelist. This permits, for example, different programs sharing the same input.nml file to have different logs. If not given here and no value is specified in the namelist, this defaults to dart_log.out

call finalize_utilities()

Closes the logfile; using utilities after this call is a bad idea.

call dump_unit_attributes(iunit)

```
integer, intent(in) :: iunit
```

Writes all information about the status of the IO unit to the error handler with error level message.

`iunit`	Unit about which information is requested.

```
integer :: E_DBG, DEBUG
integer :: E_MSG, MESSAGE
integer :: E_WARN, WARNING
integer :: E_ERR, FATAL
```

	Severity levels to be passed to error handler. Levels are debug, message, warning and fatal. The namelist parameter TERMLEVEL can be used to control at which level program termination should occur.

```
integer :: logfileunit
```

`logfileunit`	Unit opened to file for diagnostic output.

```
integer :: nmlfileunit
```

nmlfileunit	Unit opened to file for diagnostic output of namelist files. Defaults to same as `logfileunit`. Provides the flexibility to log namelists to a separate file, reducing the clutter in the log files and perhaps increasing readability.

6.203.5 Files

- assim_model_mod.nml in input.nml
- logfile, name specified in namelist

6.203.6 References

- none

6.203.7 Error codes and conditions

Routine	Message	Comment
get_unit	No available units	Unable to open enough IO channels
check_nml_error	WARNING: error reading namelist _____	Fatal error reading namelist. This could be caused by having an entry in the namelist input file that is not in the namelist, by having illegal values for namelist variables, or by a variety of other compiler dependent problems.
find_namelist_in_file	Namelist entry &____ must exist in namelist_nml.	There must be an entry for the required namelist, for instance &filter_nml, in the input.nml namelist file. Even if no values are to be changed from the default, an entry like &filter_nml followed by a line containing only / is required.
find_namelist_in_file	Namelist input file: input.nml must exist	The namelist input file (usually input.nml) must exist.
check_namelist_read	INVALID NAMELIST ENTRY: ___ in namelist ____	While reading the namelist, either a bad entry was found or an end of file was encountered. The most confusing case is when a namelist is being read successfully but is not appropriately terminated with a /. The line printed out by the error message will be the start of the next namelist in the input.nml file in this case.

6.203.8 Private components

N/A

6.204 MODULE types_mod

6.204.1 Overview

Provides some commonly used mathematical constants, and a set of Fortran integer and real kinds, to be used to select the right variable size (e.g. 4 bytes, 8 bytes) to match the rest of the DART interfaces. (DART does not depend on compiler flags to set precision, but explicitly specifies a kind for each variable in the public interfaces.)

6.204.2 Other modules used

none

6.204.3 Public interfaces

This routine provides the following constants, but no routines of any kind.

The constants defined here *may* or *may not* be declared the same as constants used in non-DART pieces of code. It would seem like a good idea to match the DART definition of 'gas_constant' to the WRF equivalent if you are going to be running WRF/DART experiments (for example).

use types_mod, only :	i4
	i8
	r4
	r8
	c4
	c8
	digits12
	PI
	DEG2RAD
	RAD2DEG
	SECPERDAY
	MISSING_R4
	MISSING_R8
	MISSING_I
	MISSING_DATA
	metadatalength
	obstypelength
	t_kelvin
	es_alpha
	es_beta
	es_gamma
	gas_constant_v
	gas_constant
	L_over_Rv
	ps0
	earth_radius
	gravity

```
integer, parameter :: i4
integer, parameter :: i8
integer, parameter :: r4
integer, parameter :: r8
integer, parameter :: c4
integer, parameter :: c8
integer, parameter :: digits12
```

These kinds are used when declaring variables, like:

```
real(r8)    :: myvariable
integer(i4) :: shortint
```

All DART public interfaces use types on the real values to ensure they are consistent across various compilers and compile-time options. The digits12 is generally only used for reals which require extra precision.
Some models are able to run with single precision real values, which saves both memory when executing and file space when writing and reading restart files. To accomplish this, the users edit this file, redefine r8 to equal r4, and then rebuild all of DART.

```
real(KIND=R8), parameter :: PI
real(KIND=R8), parameter :: DEG2RAD
real(KIND=R8), parameter :: RAD2DEG
real(KIND=R8), parameter :: SECPERDAY
```

Some commonly used math constants, defined here for convenience.

```
real(KIND=R4), parameter :: MISSING_R4
real(KIND=R8), parameter :: MISSING_R8
integer,       parameter :: MISSING_I
integer,       parameter :: MISSING_DATA
```

Numeric constants used in the DART code when a numeric value is required, but the data is invalid or missing. These are typically defined as negative and a series of 8's, so they are distinctive when scanning a list of values.

```
integer, parameter :: metadatalength
integer, parameter :: obstypelength
```

Some common string limits used system-wide by DART code. The obstypelength is limited by the Fortran-imposed maximum number of characters in a parameter; the metadatalength was selected to be long enough to allow descriptive names but short enough to keep printing to less than a single line.

```
real(KIND=R8), parameter :: t_kevin
real(KIND=R8), parameter :: es_alpha
real(KIND=R8), parameter :: es_beta
real(KIND=R8), parameter :: es_gamma
real(KIND=R8), parameter :: gas_constant_v
real(KIND=R8), parameter :: gas_constant
real(KIND=R8), parameter :: L_over_Rv
real(KIND=R8), parameter :: ps0
real(KIND=R8), parameter :: earth_radius
real(KIND=R8), parameter :: gravity
```

A set of geophysical constants, which could be argued do not belong in a DART-supplied file since they are quite probably specific to a model or a particular forward operator.

Best case would be if we could engineer the code so these constants were provided by the model and then used when compiling the forward operator files. But given that Fortran use statements cannot be circular, this poses a problem. Perhaps we could work out how the obs_def code could define these constants and then they could be used by the model code. For now, they are defined here but it is up to the model and obs_def code writers whether to use these or not.

6.204.4 Namelist

There is no namelist for this module.

6.204.5 Files

None.

6.204.6 References

1. none

6.204.7 Private components

N/A

6.205 MODULE schedule_mod

6.205.1 Overview

Provides a set of routines to generate a regular pattern of time windows. This module is only used for converting observation sequences files to netCDF format. If it stands the test of time, it will likely be used to create an assimilation schedule independent of the observation sequence file.

6.205.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&schedule_nml
   first_bin_start      =  1601,  1,  1,  0,  0,  0
   first_bin_end        =  2999,  1,  1,  0,  0,  0
   last_bin_end         =  2999,  1,  1,  0,  0,  0
   bin_interval_days    = 1000000
   bin_interval_seconds = 0
   max_num_bins         = 1000
   calendar             = 'Gregorian'
   print_table          = .true.
  /
```

The default values will cause (pretty much) all possible observations to be put into one output file.

Item	Type	Description
first_bin_start	integer, dimension(6)	Date/time specification for starting time of first bin.
first_bin_end	integer, dimension(6)	Date/time specification for ending time of first bin. Sets the bin width.
last_bin_end	integer, dimension(6)	Date/time specification for ending time of last bin. Sets the length of the overall time of the schedule.
bin_interval_days	integer	Sets the time between bins. Must be larger or equal to the bin width.
bin_interval_seconds	integer	Sets the time between bins. Must be larger or equal to the bin width.
max_num_bins	integer	Upper limit on the number of bins.
calendar	character(len=32)	String calendar type. Valid types are listed in the time_manager_mod file.
print_table	logical	If .TRUE., print out information about the schedule each time set_regular_schedule() is called.

6.205.3 Other modules used

```
types_mod
utilities_mod
time_manager_mod
```

6.205.4 Public interfaces

use schedule_mod, only :	schedule_type
	set_regular_schedule
	get_time_from_schedule
	get_schedule_length

Namelist `&schedule_mod_nml` may be read from file `input.nml`.

call set_regular_schedule(schedule)

```
type(schedule_type), intent(out) :: schedule
```

Uses the namelist information to compute and fill a schedule_type variable.

schedule	Fills this derived type with the information needed to generate a series of regularly spaced time windows.

call get_time_from_schedule(mytime, schedule, iepoch [, edge])

```
type(time_type),     intent(out) :: mytime
 or
real(digits12),     intent(out) :: mytime
type(schedule_type), intent(in)  :: schedule
integer,             intent(in)  :: iepoch
integer, optional,   intent(in)  :: edge
```

Returns either the leading or trailing time for the specified bin/epoch number for the given schedule. The time can be returned in one of two formats, depending on the variable type specified for the first argument: either a DART derived time_type, or a real of kind digits12 (defined in the types_mod).

mytime	Return value with the leading or trailing edge time for the requested bin. There are two supported return formats, either as a standard DART time_type, or as a real value which will contain the number of days plus any fraction.
schedule	Schedule type to extract information from.
iepoch	The bin number, or epoch number, to return a time for. Unless edge is specified and requests the ending time, the time returned is the starting time for this bin.
edge	If specified, and if edge is larger than 1, the trailing edge time of the bin is returned. Any other value, or if this argument is not specified, returns the leading edge time of the bin.

var = get_schedule_length()

```
integer                              :: get_schedule_length
type(schedule_type), intent(in)      :: schedule
```

Return the total number of intervals/bins/epochs defined by this schedule.

schedule	Return number of time intervals in this schedule.

```
type schedule_type
   private
   integer :: num_bins
   integer :: current_bin
   logical :: last_bin
   integer :: calendar
   character(len=32) :: calendarstring
   type(time_type)          :: binwidth
   type(time_type)          :: bininterval
   type(time_type), pointer :: binstart(   :) => NULL()
   type(time_type), pointer :: binend(     :) => NULL()
   real(digits12),  pointer :: epoch_start(:) => NULL()
   real(digits12),  pointer :: epoch_end(  :) => NULL()
end type schedule_type
```

This type is used to define a schedule.

6.205.5 Files

filename	purpose
input.nml	to read the schedule_mod namelist

6.205.6 References

- none

6.205.7 Private components

N/A

6.206 MODULE obs_kind_mod

6.206.1 Overview

Introduction

This module provides definitions of specific observation types and generic variable quantities, routines for mapping between integer identifiers and string names, routines for reading and writing this information, and routines for determining whether and how to process observations from an observation sequence file.

The distinction between quantities and types is this: `Quantities` apply both to observations and to state vector variables. Knowing the type of an observation must be sufficient to compute the correct forward operator. The quantity associated with an observation must be sufficient to identify which variable in the state vector should be used to compute the expected value. `Types` only apply to observations, and are usually observation-platform dependent. Making distinctions between different observation sources by using different types allows users to selectively assimilate, evaluate, or ignore them.

Examples and use

Generic quantities are associated with an observation type or with a model state variable. An example quantity is `QTY_U_WIND_COMPONENT`. Multiple different specific observation types can be associated with this generic quantity, for instance `RADIOSONDE_U_WIND_COMPONENT`, `ACARS_U_WIND_COMPONENT`, and `SAT_U_WIND_COMPONENT`. Generic quantities are defined via an integer parameter statement at the start of this module. As new generic quantities are needed they are added to this list. Generic quantity integer parameters are required to start with `QTY_` and observation types are NOT allowed to start with `QTY_`.

Typically quantities are used by model-interface files `models/xx/model_mod.f90`, observation forward operator files `observations/forward_operators/obs_def_xx_mod.f90`, and observation converter programs `observations/obs_converters/xx/xx.f90`.

The obs_kind module being described here is created by the program `preprocess` from two categories of input files. First, a DEFAULT obs_kind module (normally called `DEFAULT_obs_kind_mod.F90` and documented in this directory) is used as a template into which the preprocessor incorporates information from zero or more special obs_def modules (such as `obs_def_1d_state_mod.f90` or `obs_def_reanalysis_bufr_mod.f90`) which are documented in the obs_def directory. If no special obs_def files are included in the preprocessor namelist, a minimal `obs_kind_mod.f90` is created which can only support identity forward observation operators.

All of the build scripts in DART remove the existing `obs_kind_mod.f90` file and regenerate it using the `preprocess` program. Do not add new quantities to `obs_kind_mod.f90`, because these changes will not be kept when you run *quickbuild.sh*.

Adding additional quantities

New quantities should be added to a quantity file, for example a new ocean quantity should be added to `ocean_quantities_mod.f90`. The quantity files are in `assimilation_code/modules/observations/`.

Every line in a quantity file between the start and end markers must be a comment or a quantity definition (QTY_string). Multiple name-value pairs can be specified for a quantity but are not required. For example, temperature may be defined: `! QTY_TEMPERATURE units="K" minval=0.0`. Comments are allowed between quantity definitions or on the same line as the definition. The code snippet below shows acceptable formats for quantity definitions

```
! BEGIN DART PREPROCESS QUANTITY DEFINITIONS
!
! Formats accepted:
!
! QTY_string
! QTY_string name=value
! QTY_string name=value name2=value2
!
! QTY_string ! comments
!
!
! comment
!
! END DART PREPROCESS QUANTITY DEFINITIONS
```

Implementation details

The obs_kind module contains an automatically-generated list of integer parameters, derived from the obs_def files, an integer parameter `max_defined_types_of_obs`, and an automatically-generated list of initializers for the `obs_type_type` derived type that defines the details of each observation type that has been created by the preprocess program. Each entry contains the integer index of the observation type, the string name of the observation type (which is identical to the F90 identifier), the integer index of the associated generic quantities, and three logicals indicating whether this observation type is to be assimilated, evaluated only (forward operator is computed but not assimilated), assimilated but has externally computed forward operator values in the input observation sequence file, or ignored entirely. The logicals initially default to .false. and are set to .true. via the `&obs_kind_nml` namelist. A second derived type `obs_qty_type` maps the integer parameter for a quantity to the quantity name (a string), and stores any additional pair-value metadata for that quantity.

6.206.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_kind_nml
   assimilate_these_obs_types          = 'null',
   evaluate_these_obs_types            = 'null'
   use_precomputed_FOs_these_obs_types = 'null'
 /
```

Controls what observation types are to be assimilated, evaluated, or ignored. For each entry, a list of observation type names can be specified. Any name in the obs_type_type table is eligible. Specifying a name that is not in the table results in an error. Specifying the same name for both namelist entries also results in an error. Observation types specified in the list for assimilate_these_obs_types are assimilated. Those in the evaluate_these_obs_types list have their forward operators computed and included in diagnostic files but are not assimilated. An observation type that is specified in neither list is ignored. Identity observations, however, are always assimilated if present in the obs_seq.out file.

Item	Type	Description
assimilate_these_obs_types	character(len=31), dimension(:)	Names of observation types to be assimilated.
evaluate_these_obs_types	character(len=31), dimension(:)	Names of observation types to be evaluated only.
use_precomputed_FOs_these_obs_types	character(len=31), dimension(:)	If the forward operator values have been precomputed outside of filter, for example for radiances or other compute intensive computations, the ensemble of forward operator values can be stored in the observation sequence file. For any type listed here, the forward operator interpolation code will not be called and the values in the file will be used instead.

For example:

```
&obs_kind_nml
   assimilate_these_obs_types = 'RADIOSONDE_TEMPERATURE',
                                'RADIOSONDE_U_WIND_COMPONENT',
                                'RADIOSONDE_V_WIND_COMPONENT',
   evaluate_these_obs_types   = 'RADIOSONDE_SURFACE_PRESSURE',
  use_precomputed_FOs_these_obs_types = 'RADIANCE'
/
```

would assimilate temperature and wind observations, but only compute the forward operators for surface pressure obs. Radiance observations have precomputed values for each ensemble member in the input observation sequence file which would be used instead of calling the forward operator code.

6.206.3 Modules used

```
utilities_mod
```

6.206.4 Public interfaces

use obs_def_mod, only :	max_defined_types_of_obs
	get_num_types_of_obs
	get_num_quantities
	get_name_for_type_of_obs
	get_name_for_quantity
	get_index_for_type_of_obs
	get_index_for_quantity
	assimilate_this_type_of_obs
	evaluate_this_type_of_obs
	get_quantity_for_type_of_obs
	write_type_of_obs_table
	read_type_of_obs_table
	get_type_of_obs_from_menu
	map_type_of_obs_table
	GENERIC_QTY_DEFINITIONS
	OBSERVATION_TYPES

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

integer, parameter :: max_defined_types_of_obs

The total number of available observation types in the obs_type_type table. This value is added by the preprocess program and depends on which `obs_def_xxx_mod.f90` files are listed in the &preprocess_nml namelist.

There is also a function interface which is an alternate method to get this value. In some cases the code requires a parameter value known at compile time (for declaring a fixed length array, for example). For an array allocated at run time the size can be returned by the function interface.

var = get_num_types_of_obs()

```
integer :: get_num_types_of_obs
```

Returns the number of different specific observation types (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_SPECIFIC_HUMIDITY) defined in the obs_kind_mod.f90 file. This file is generated by the preprocess program. This is the same value as the public 'max_defined_types_of_obs' above.

var	Integer count of the total number of specific types defined in the obs_kind_mod.f90 file.

var = get_num_quantities()

```
integer :: get_num_quantities
```

Returns the number of different generic quantities (e.g. QTY_TEMPERATURE, QTY_SPECIFIC_HUMIDITY) defined in the obs_kind_mod.f90 file. This file is generated by the preprocess program.

var	Integer count of the total number of generic quantities defined in the obs_kind_mod.f90 file.

var = get_name_for_type_of_obs(obs_type_ind)

```
character(len=32)              :: get_name_for_type_of_obs
integer, intent(in)            :: obs_type_ind
```

Given an integer index return the string name of the corresponding specific observation type (e.g. "RADIOSONDE_TEMPERATURE", "AIRCRAFT_SPECIFIC_HUMIDITY"). This string is the same as the F90 identifier associated with the integer index.

var	Name string associated with this entry in the obs_type_type table.
obs_type_ind	An integer index into the obs_type_type table.

var = get_name_for_quantity(obs_qty_ind)

```
character(len=32)              :: get_name_for_quantity
integer, intent(in)            :: obs_qty_ind
```

Given an integer index return the string name of the corresponding generic quantity (e.g. "QTY_TEMPERATURE", "QTY_SPECIFIC_HUMIDITY"). This string is the same as the F90 identifier associated with the integer index.

var	Name string associated with this entry in the obs_qty_type table.
obs_qty_ind	An integer index into the obs_qty_type table.

var = get_index_for_type_of_obs(obs_type_name)

```
integer                        :: get_index_for_type_of_obs
character(len=*), intent(in)   :: obs_type_name
```

Given the name of a specific observation type (e.g. "RADIOSONDE_TEMPERATURE", "AIRCRAFT_SPECIFIC_HUMIDITY"), returns the index of the entry in the obs_type_type table with this name. If the name is not found in the table, a -1 is returned. The integer returned for a successful search is the value of the integer parameter with the same identifier as the name string.

get_index_for_type_of_obs	Integer index into the obs_type_type table entry with name string corresponding to obs_type_name.
obs_type_name	Name of specific observation type found in obs_type_type table.

var = get_index_for_quantity(obs_qty_name)

```
integer                        :: get_index_for_quantity
character(len=32), intent(in)  :: obs_qty_name
```

Given the name of a generic quantity (e.g. "QTY_TEMPERATURE", "QTY_SPECIFIC_HUMIDITY"), returns the index of the entry in the obs_qty_type table with this name. If the name is not found in the table, a -1 is returned. The integer returned for a successful search is the value of the integer parameter with the same identifier as the name string.

get_index_for_quantity	Integer index into the obs_qty_type table entry with name string corresponding to obs_qty_name.
obs_qty_name	Name of generic kind found in obs_qty_type table.

var = assimilate_this_type_of_obs(obs_type_ind)

```
logical              :: assimilate_this_type_of_obs
integer, intent(in)  :: obs_type_ind
```

Given the integer index associated with a specific observation type (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_SPECIFIC_HUMIDITY), return true if this observation type is to be assimilated, otherwise false. The parameter defined by this name is used as an integer index into the obs_type_type table to return the status of this type.

var	Returns true if this entry in the obs_type_type table is to be assimilated.
obs_type_ind	An integer index into the obs_type_type table.

var = evaluate_this_type_of_obs(obs_type_ind)

```
logical              :: evaluate_this_type_of_obs
integer, intent(in)  :: obs_type_ind
```

Given the integer index associated with a specific observation type (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_SPECIFIC_HUMIDITY), return true if this observation type is to be evaluated only, otherwise false. The parameter defined by this name is used as an integer index into the obs_type_type table to return the status of this type.

var	Returns true if this entry in the obs_type_type table is to be evaluated.
obs_type_ind	An integer index into the obs_type_type table.

var = get_quantity_for_type_of_obs(obs_type_ind)

```
integer              :: get_quantity_for_type_of_obs
integer, intent(in)  :: obs_type_ind
```

Given the integer index associated with a specific observation type (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_SPECIFIC_HUMIDITY), return the generic quantity associated with this type (e.g. QTY_TEMPERATURE, QTY_SPECIFIC_HUMIDITY). The parameter defined by this name is used as an integer index into the obs_type_type table to return the generic quantity associated with this type.

var	Returns the integer GENERIC quantity index associated with this obs type.
obs_type_ind	An integer index into the obs_type_type table.

call write_type_of_obs_table(ifile [, fform, use_list])

```
integer,                    intent(in) :: ifile
character(len=*), optional, intent(in) :: fform
integer,          optional, intent(in) :: use_list(:)
```

Writes out information about all defined observation types from the obs_type_type table. For each entry in the table, the integer index of the observation type and the associated string are written. These appear in the header of an obs_sequence file. If given, the *use_list(:)* must be the same length as the max_obs_specific count. If greater than 0,

the corresponding index will be written out; if 0 this entry is skipped. This allows a table of contents to be written which only includes those types actually being used.

ifile	Unit number of output observation sequence file being written.
fform	Optional format for file. Default is FORMATTED.
use_list	Optional integer array the same length as the number of specific types (from get_num_types_of_obs() or the public max_defined_types_of_obs). If value is larger than 0, the corresponding type information will be written out. If 0, it will be skipped. If this argument is not specified, all values will be written.

call read_type_of_obs_table(ifile, pre_I_format [, fform])

```
integer,                    intent(in) :: ifile
logical,                    intent(in) :: pre_I_format !(deprecated)
character(len=*), optional, intent(in) :: fform
```

Reads the mapping between integer indices and observation type names from the header of an observation sequence file and prepares mapping to convert these to values defined in the obs_type_type table. If pre_I_format is true, there is no header in the observation sequence file and it is assumed that the integer indices for observation types in the file correspond to the storage order of the obs_type_type table (integer index 1 in the file corresponds to the first table entry, etc.) Support for pre_I_format is deprecated and may be dropped in future releases of DART.

ifile	Unit number of output observation sequence file being written.
pre_I_format	True if the file being read has no obs type definition header (deprecated).
fform	Optional format for file. Default is FORMATTED.

var = get_type_of_obs_from_menu()

```
integer              :: get_type_of_obs_from_menu
```

Interactive input of observation type. Prompts user with list of available types and validates entry before returning.

var	Integer index of observation type.

var = map_type_of_obs_table(obs_def_index)

```
integer              :: map_type_of_obs_table
integer, intent(in)  :: obs_def_index
```

Maps from the integer observation type index in the header block of an input observation sequence file into the corresponding entry in the obs_type_type table. This allows observation sequences that were created with different obs_kind_mod.f90 versions to be used with the current obs_kind_mod.

var	Index of this observation type in obs_type_type table.
obs_def_index	Index of observation type from input observation sequence file.

```
integer, parameter ::  QTY_.....
```

All generic quantities available are public parameters that begin with QTY_.

integer, parameter :: SAMPLE_OBS_TYPE

A list of all observation types that are available is provided as a set of integer parameter statements. The F90 identifiers are the same as the string names that are associated with this identifier in the obs_type_type table.

6.206.5 Files

- &obs_kind_nml in input.nml
- Files containing input or output observation sequences.

6.206.6 References

- none

6.207 MODULE DEFAULT_obs_kind_mod

6.207.1 Overview

DART provides capabilities to assimilate a multitude of different observation types. Since most DA applications only need to assimilate a subset of the observation types that DART is capable of assimilating, the observation types supported by the programs in your application are defined when you compile them. You only need to include the observation types you are interested in.

DEFAULT_obs_kind_mod.F90 is the input template file which is read by the *PROGRAM preprocess* to create *MODULE obs_kind_mod*. Information from zero or more special obs_def modules (such as

MODULE obs_def_1d_state_mod) and obs_quantities modules (such as `DART/assimilation_code/modules/observations/oned_quantities_mod.f90`) are incorporated into the template provided by DEFAULT_obs_def_kind.

If you don't include any specific obs_def files in the preprocessor namelist, `preprocess` will create a minimal `obs_kind_mod.f90` file which can only support identity forward observation operators.

To add a *new* specific observation type, see the *MODULE obs_def_mod* documentation.

To add a *new* specific observation quantity, see the *MODULE obs_kind_mod* documentation.

6.208 MODULE obs_sequence_mod

6.208.1 Overview

Provides interfaces to the observation type and observation sequence type. An observation contains everything there is to know about an observation including all metadata contained in the observation definition and any number of copies of data associated with the observation (for instance an actual observation, an ensemble of first guess values, etc). An observation sequence is a time-ordered set of observations that is defined by a linked list so that observations can be easily added or deleted. A number of commands to extract observations depending on the times at which they were taken are provided. For now, the observations are only ordered by time, but the ability to add extra sort keys could be added.

These routines are commonly used in conversion programs which read observation data from various formats and create a DART observation sequence in memory, and then write it out to a file. See the observations directory for examples of programs which create and manipulate observations using this routines.

6.208.2 Other modules used

```
types_mod
location_mod (depends on model_choice)
obs_def_mod
time_manager_mod
utilities_mod
obs_kind_mod
```

6.208.3 Public interfaces

use obs_sequence_mod, only :	obs_sequence_type
	init_obs_sequence
	interactive_obs_sequence
	get_num_copies
	get_num_qc
	get_num_obs
	get_max_num_obs
	get_copy_meta_data

continues on next page

Table 8 – continued from previous page

	get_qc_meta_data
	get_next_obs
	get_prev_obs
	get_next_obs_from_key
	get_prev_obs_from_key
	insert_obs_in_seq
	delete_obs_from_seq
	set_copy_meta_data
	set_qc_meta_data
	get_first_obs
	get_last_obs
	add_copies
	add_qc
	write_obs_seq
	read_obs_seq
	append_obs_to_seq
	get_obs_from_key
	get_obs_time_range
	set_obs
	get_time_range_keys
	get_num_times
	static_init_obs_sequence
	destroy_obs_sequence
	read_obs_seq_header
	get_expected_obs
	delete_seq_head
	delete_seq_tail
	LINKS BELOW FOR OBS_TYPE INTERFACES
	obs_type
	init_obs
	destroy_obs
	get_obs_def
	set_obs_def
	get_obs_values
	set_obs_values
	replace_obs_values
	get_qc
	set_qc
	replace_qc
	write_obs
	read_obs
	interactive_obs
	copy_obs
	assignment(=)

```
type obs_sequence_type
   private
   integer                        :: num_copies
   integer                        :: num_qc
   integer                        :: num_obs
   integer                        :: max_num_obs
   character(len=64), pointer     :: copy_meta_data(:)
   character(len=64), pointer     :: qc_meta_data(:)
   integer                        :: first_time
   integer                        :: last_time
   type(obs_type), pointer        :: obs(:)
end type obs_sequence_type
```

The obs_sequence type represents a series of observations including multiple copies of data and quality control fields and complete metadata about the observations. The sequence is organized as an integer pointer linked list using a fixed array of storage for obs (type obs_type). Each observation points to the previous and next observation in time order (additional sort keys could be added if needed) and has a unique integer key (see obs_type below). The maximum number of observations in the sequence is represented in the type as max_num_obs, the current number of observations is in num_obs. The number of quality control (qc) fields per observation is num_qc and the number of data values associated with each observation is num_copies. Metadata for each copy of the data is in copy_meta_data and metadata for the qc fields is in qc_meta_data. The first and last pointers into the time linked list are in first_time and last_time. A capability to write and read an obs_sequence structure to disk is available. At present, the entire observation sequence is read in to core memory. An on-disk implementation may be necessary for very large observational datasets.

Component	Description
num_copies	Number of data values associated with each observation.
num_qc	Number of qc fields associated with each observation.
num_obs	Number of observations currently in sequence.
max_num_obs	Upper bounds on number of observations in sequence.
copy_meta_data	Text describing each copy of data associated with observations.
qc_meta_data	Text describing each quality control field.
first_time	Location of first observation in sequence.
last_time	Location of last observation in sequence.
obs	Storage for all of the observations in the sequence.

```
type obs_type
   private
   integer            :: key
   type(obs_def_type) :: def
   real(r8), pointer  :: values(:)
   real(r8), pointer  :: qc(:)
   integer            :: prev_time
   integer            :: next_time
   integer            :: cov_group
end type obs_type
```

Structure to represent everything known about a given observation and to help with storing the observation in the observation sequence structure (see above). The prev_time and next_time are integer pointers that allow a linked list

sorted on time to be constructed. If needed, other sort keys could be introduced (for instance by time available?). Each observation in a sequence has a unique key and each observation has an obs_def_type that contains all the definition and metadata for the observation. A set of values is associated with the observation along with a set of qc fields. The cov_group is not yet implemented but will allow non-diagonal observation error covariances in a future release.

Component	Description
key	Unique integer key when in an obs_sequence.
def	The definition of the observation (see obs_def_mod).
values	Values associated with the observation.
qc	Quality control fields associated with the observation.
prev_time	When in an obs_sequence, points to previous time sorted observation.
next_time	When in an obs_sequence, points to next time sorted observation.
cov_group	Not currently implemented.

call init_obs_sequence(seq, num_copies, num_qc, expected_max_num_obs)

```
type(obs_sequence_type), intent(out) :: seq
integer,                 intent(in)  :: num_copies
integer,                 intent(in)  :: num_qc
integer,                 intent(in)  :: expected_max_num_obs
```

Constructor to create a variable of obs_sequence_type. This routine must be called before using an obs_sequence_type. The number of copies of the data to be associated with each observation (for instance the observation from an instrument, an ensemble of prior guesses, etc.) and the number of quality control fields associated with each observation must be specified. Also, an estimated upper bound on the number of observations to be stored in the sequence is helpful in making creation of the sequence efficient.

seq	The observation sequence being constructed
num_copies	Number of copies of data to be associated with each observation
num_qc	Number of quality control fields associated with each observation
expected_max_num_obs	An estimate of the largest number of observations the sequence might contain

var = interactive_obs_sequence()

```
type(obs_sequence_type) :: interactive_obs_sequence
```

Uses input from standard in to create an observation sequence. Initialization of the sequence is handled by the function.

var	An observation sequence created from standard input

var = get_num_copies(seq)

```
integer                                 :: get_num_copies
type(obs_sequence_type), intent(in) :: seq
```

Returns number of copies of data associated with each observation in an observation sequence.

var	Returns number of copies of data associated with each observation in sequence
seq	An observation sequence

var = get_num_qc(seq)

```
integer                                 :: get_num_qc
type(obs_sequence_type), intent(in) :: seq
```

Returns number of quality control fields associated with each observation in an observation sequence.

var	Returns number of quality control fields associated with each observation in sequence
seq	An observation sequence

var = get_num_obs(seq)

```
integer                                 :: get_num_obs
type(obs_sequence_type), intent(in) :: seq
```

Returns number of observations currently in an observation sequence.

var	Returns number of observations currently in an observation sequence
seq	An observation sequence

var = get_max_num_obs(seq)

```
integer                                 :: get_max_num_obs
type(obs_sequence_type), intent(in) :: seq
```

Returns maximum number of observations an observation sequence can hold.

var	Returns maximum number of observations an observation sequence can hold
seq	An observation sequence

var = get_copy_meta_data(seq, copy_num)

```
character(len=64)                  :: get_copy_meta_data
type(obs_sequence_type), intent(in) :: seq
integer,                 intent(in) :: copy_num
```

Returns metadata associated with a given copy of data in an observation sequence.

var	Returns metadata associated with a copy of data in observation sequence
seq	An observation sequence
copy_num	Return metadata for this copy

var = get_qc_meta_data(seq,qc_num)

```
character(len=64)                  :: get_qc_meta_data
type(obs_sequence_type), intent(in) :: seq
integer,                 intent(in) :: qc_num
```

Returns metadata associated with a given copy of quality control fields associated with observations in an observation sequence.

var	Returns metadata associated with a given qc copy
seq	An observation sequence
qc_num	Return metadata for this copy

call get_next_obs(seq, obs, next_obs, is_this_last)

```
type(obs_sequence_type), intent(in)  :: seq
type(obs_type),          intent(in)  :: obs
type(obs_type),          intent(out) :: next_obs
logical,                 intent(out) :: is_this_last
```

Given an observation in a sequence, returns the next observation in the sequence. If there is no next observation, is_this_last is set to true.

seq	An observation sequence
obs	Find the next observation after this one
next_obs	Return the next observation here
is_this_last	True if obs is the last obs in sequence

call get_prev_obs(seq, obs, prev_obs, is_this_first)

```
type(obs_sequence_type), intent(in)  :: seq
type(obs_type),          intent(in)  :: obs
type(obs_type),          intent(out) :: prev_obs
logical,                 intent(out) :: is_this_first
```

Given an observation in a sequence, returns the previous observation in the sequence. If there is no previous observation, is_this_first is set to true.

seq	An observation sequence
obs	Find the previous observation before this one
prev_obs	Return the previous observation here
is_this_first	True if obs is the first obs in sequence

call get_next_obs_from_key(seq, last_key_used, next_obs, is_this_last)

```
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: last_key_used
type(obs_type),          intent(out) :: next_obs
logical,                 intent(out) :: is_this_last
```

Given the last key used in a sequence, returns the next observation in the sequence. If there is no next observation, is_this_last is set to true.

seq	An observation sequence
last_key_used	Find the next observation after this key
next_obs	Return the next observation here
is_this_last	True if obs is the last obs in sequence

call get_prev_obs_from_key(seq, last_key_used, prev_obs, is_this_first)

```
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: last_key_used
type(obs_type),          intent(out) :: prev_obs
logical,                 intent(out) :: is_this_first
```

Given the last key used in a sequence, returns the previous observation in the sequence. If there is no previous observation, is_this_first is set to true.

seq	An observation sequence
last_key_used	Find the previous observation before this key
prev_obs	Return the previous observation here
is_this_first	True if obs is the first obs in sequence

call get_obs_from_key(seq, key, obs)

```
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: key
type(obs_type),          intent(out) :: obs
```

Each entry in an observation sequence has a unique integer key. This subroutine returns the observation given an integer key.

seq	An observation sequence
key	Return the observation with this key
obs	The returned observation

call insert_obs_in_seq(seq, obs [, prev_obs])

```
type(obs_sequence_type),  intent(inout) :: seq
type(obs_type),           intent(inout) :: obs
type(obs_type), optional, intent(in)    :: prev_obs
```

Inserts an observation in a sequence in appropriate time order. If the optional argument prev_obs is present, the new observation is inserted directly after the prev_obs. If an incorrect prev_obs is provided so that the sequence is no longer time ordered, bad things will happen.

seq	An observation sequence
obs	An observation to be inserted in the sequence
prev_obs	If present, says the new observation belongs immediately after this one

call delete_obs_from_seq(seq, obs)

```
type(obs_sequence_type), intent(inout) :: seq
type(obs_type),          intent(inout) :: obs
```

Given an observation and a sequence, removes the observation with the same key from the observation sequence.

seq	An observation sequence
obs	The observation to be deleted from the sequence

call set_copy_meta_data(seq, copy_num, meta_data)

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: copy_num
character(len=64),       intent(in)    :: meta_data
```

Sets the copy metadata for this copy of the observations in an observation sequence.

seq	An observation sequence
copy_num	Set metadata for this copy of data
meta_data	The metadata

call set_qc_meta_data(seq, qc_num, meta_data)

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: qc_num
character(len=64),       intent(in)    :: meta_data
```

Sets the quality control metadata for this copy of the qc in an observation sequence.

seq	An observation sequence
qc_num	Set metadata for this quality control field
meta_data	The metadata

var = get_first_obs(seq, obs)

```
logical                            :: get_first_obs
type(obs_sequence_type), intent(in)  :: seq
type(obs_type),          intent(out) :: obs
```

Returns the first observation in a sequence. If there are no observations in the sequence, the function returns false, else true.

var	Returns false if there are no obs in sequence
seq	An observation sequence
obs	The first observation in the sequence

var = get_last_obs(seq, obs)

```
logical                            :: get_last_obs
type(obs_sequence_type), intent(in)  :: seq
type(obs_type),          intent(out) :: obs
```

Returns the last observation in a sequence. If there are no observations in the sequence, the function returns false, else true.

var	Returns false if there are no obs in sequence
seq	An observation sequence
obs	The last observation in the sequence

call add_copies(seq, num_to_add)

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: num_to_add
```

Increases the number of copies of data associated with each observation by num_to_add. The current implementation re-creates the entire observation sequence by deallocating and reallocating each entry with a larger size.

seq	An observation sequence
num_to_add	Number of copies of data to add

call add_qc(seq, num_to_add)

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: num_to_add
```

Increases the number of quality control fields associated with each observation by num_to_add. The current implementation re-creates the entire observation sequence by deallocating and reallocating each entry with a larger size.

seq	An observation sequence
num_to_add	Number of quality control fields to add

call read_obs_seq(file_name, add_copies, add_qc, add_obs, seq)

```
character(len=*),        intent(in)  :: file_name
integer,                 intent(in)  :: add_copies
integer,                 intent(in)  :: add_qc
integer,                 intent(in)  :: add_obs
type(obs_sequence_type), intent(out) :: seq
```

Read an observation sequence from file_name. The sequence will have enough space for the number of observations in the file plus any additional space requested by the "add_xx" args. It is more efficient to allocate the additional space at create time rather than try to add it in later. The arguments can specify that the caller wants to add additional data copies associated with each observation, or to add additional quality control fields, or to add space for additional

observations. The format of the file (`formatted` vs. `unformatted`) has been automatically detected since the I release. The obs_sequence file format with I and later releases has a header that associates observation type strings with an integer which was not present in previous versions. I format files are no longer supported.

`file_name`	Read from this file
`add_copies`	Add this number of copies of data to the obs_sequence on file
`add_qc`	Add this number of qc fields to the obs_sequence on file
`add_obs`	Add space for this number of additional observations to the obs_sequence on file
`seq`	The observation sequence read in with any additional space

call write_obs_seq(seq, file_name)

```
type(obs_sequence_type), intent(in) :: seq
character(len=*),        intent(in) :: file_name
```

Write the observation sequence to file file_name. The format is controlled by the namelist parameter write_binary_obs_sequence.

`seq`	An observation sequence
`file_name`	Write the sequence to this file

call set_obs(seq,obs [, key_in])

```
type(obs_sequence_type), intent(inout) :: seq
type(obs_type),          intent(in)    :: obs
integer, optional,       intent(in)    :: key_in
```

Given an observation, copies this observation into the observation sequence using the key specified in the observation. If the optional key_in argument is present, the observation is instead copied into this element of the observation sequence (and the key is changed to be key_in).

`seq`	An observation sequence
`obs`	Observation to be put in sequence
key_in	If present, the obs is copied into this key of the sequence

call append_obs_to_seq(seq, obs)

```
type(obs_sequence_type), intent(inout) :: seq
type(obs_type),          intent(inout) :: obs
```

Append an observation to an observation sequence. An error results if the time of the observation is not equal to or later than the time of the last observation currently in the sequence.

seq	An observation sequence
obs	Append this observation to the sequence

call get_obs_time_range(seq, time1, time2, key_bounds, num_keys, out_of_range [, obs])

```
type(obs_sequence_type),  intent(in)  :: seq
type(time_type),          intent(in)  :: time1
type(time_type),          intent(in)  :: time2
integer, dimension(2),    intent(out) :: key_bounds
integer,                  intent(out) :: num_keys
logical,                  intent(out) :: out_of_range
type(obs_type), optional, intent(in)  :: obs
```

Given a time range specified by a beginning and ending time, find the keys that bound all observations in this time range and the number of observations in the time range. The routine get_time_range_keys can then be used to get a list of all the keys in the range if desired. The logical out_of_range is returned as true if the beginning time of the time range is after the time of the latest observation in the sequence. The optional argument obs can increase the efficiency of the search through the sequence by indicating that all observations before obs are definitely at times before the start of the time range.

seq	An observation sequence
time1	Lower time bound
time2	Upper time bound
key_bounds	Lower and upper bounds on keys that are in the time range
num_keys	Number of keys in the time range
out_of_range	Returns true if the time range is entirely past the time of the last obs in sequence
obs	If present, can start search for time range from this observation

call get_time_range_keys(seq, key_bounds, num_keys, keys)

```
type(obs_sequence_type),      intent(in)  :: seq
integer, dimension(2),        intent(in)  :: key_bounds
integer,                      intent(in)  :: num_keys
integer, dimension(num_keys), intent(out) :: keys
```

Given the keys of the observations at the start and end of a time range and the number of observations in the time range (these are returned by `get_obs_time_range()`), return a list of the keys of all observations in the time range. Combining the two routines allows one to get a list of all observations in any time range by key. The `keys` array must be at least `num_keys` long to hold the return values.

seq	An observation sequence
key_bounds	Keys of first and last observation in a time range
num_keys	Number of obs in the time range
keys	Output list of keys of all obs in the time range

var = get_num_times(seq)

```
integer                              :: get_num_times
type(obs_sequence_type), intent(in) :: seq
```

Returns the number of unique times associated with observations in an observation sequence.

var	Number of unique times for observations in a sequence
seq	An observation sequence

var = get_num_key_range(seq, key1, key2)

```
integer                              :: get_num_key_range
type(obs_sequence_type), intent(in) :: seq
integer, optional,       intent(in) :: key1, key2
```

Returns the number of observations between the two given keys. The default key numbers are the first and last in the sequence file. This routine can be used to count the actual number of observations in a sequence and will be accurate even if the sequence has been trimmed with delete_seq_head() or delete_seq_tail().

var	Number of unique times for observations in a sequence
seq	An observation sequence
key1	The starting key number. Defaults to the first observation in the sequence.
key2	The ending key number. Defaults to the last observation in the sequence.

call static_init_obs_sequence()

Initializes the obs_sequence module and reads namelists. This MUST BE CALLED BEFORE USING ANY OTHER INTERFACES.

call destroy_obs_sequence(seq)

```
type(obs_sequence_type), intent(inout) :: seq
```

Releases all allocated storage associated with an observation sequence.

seq	An observation sequence

call read_obs_seq_header(file_name, num_copies, num_qc, num_obs, max_num_obs, file_id, read_format, pre_I_format [, close_the_file])

```
character(len=*),   intent(in)  :: file_name
integer,            intent(out) :: num_copies
integer,            intent(out) :: num_qc
integer,            intent(out) :: num_obs
integer,            intent(out) :: max_num_obs
integer,            intent(out) :: file_id
character(len=*),   intent(out) :: read_format
logical,            intent(out) :: pre_I_format
logical, optional,  intent(in)  :: close_the_file
```

Allows one to see the global metadata associated with an observation sequence that has been written to a file without reading the whole file.

file_name	File contatining an obs_sequence
num_copies	Number of copies of data associated with each observation
num_qc	Number of quality control fields associated with each observation
num_obs	Number of observations in sequence
max_num_obs	Maximum number of observations sequence could hold
file_id	File channel/descriptor returned from opening the file
read_format	Either the string 'unformatted' or 'formatted'
pre_I_format	Returns .true. if the file was written before the observation type string/index number table was added to the standard header starting with the I release.
close_the_file	If specified and .TRUE. close the file after the header has been read. The default is to leave the file open.

call init_obs(obs, num_copies, num_qc)

```
type(obs_type), intent(out) :: obs
integer,        intent(in)  :: num_copies
integer,        intent(in)  :: num_qc
```

Initializes an obs_type variable. This allocates storage for the observation type and creates the appropriate obs_def_type and related structures. IT IS ESSENTIAL THAT OBS_TYPE VARIABLES BE INITIALIZED BEFORE USE.

obs	An obs_type data structure to be initialized
num_copies	Number of copies of data associated with observation
num_qc	Number of qc fields associated with observation

call destroy_obs(obs)

```
type(obs_type), intent(inout) :: obs
```

Destroys an observation variable by releasing all associated storage.

obs	An observation variable to be destroyed

call get_obs_def(obs, obs_def)

```
type(obs_type),     intent(in)  :: obs
type(obs_def_type), intent(out) :: obs_def
```

Extracts the definition portion of an observation.

obs	An observation
obs_def	The definition portion of the observation

call set_obs_def(obs, obs_def)

```
type(obs_type),     intent(out) :: obs
type(obs_def_type), intent(in)  :: obs_def
```

Given an observation and an observation definition, insert the definition in the observation structure.

obs	An observation whose definition portion will be updated
obs_def	The observation definition that will be inserted in obs

call get_obs_values(obs, values [, copy_indx])

```
type(obs_type),          intent(in)  :: obs
real(r8), dimension(:), intent(out) :: values
integer, optional,       intent(in)  :: copy_indx
```

Extract copies of the data from an observation. If *copy_indx* is present extract a single value indexed by *copy_indx* into `values(1)`. *copy_indx* must be between 1 and `num_copies`, inclusive. If *copy_indx* is not present extract all copies of data into the `values` array which must be `num_copies` long (See `get_num_copies`.)

`obs`	Observation from which to extract values
`values`	The values extracted
copy_indx	If present extract only this copy, otherwise extract all copies

call get_qc(obs, qc [, qc_indx])

```
type(obs_type),          intent(in)  :: obs
real(r8), dimension(:), intent(out) :: qc
integer, optional,       intent(in)  :: qc_indx
```

Extract quality control fields from an observation. If *qc_indx* is present extract a single field indexed by *qc_indx* into `qc(1)`. *qc_indx* must be between 1 and `num_qc`, inclusive. If *qc_indx* is not present extract all quality control fields into the `qc` array which must be `num_qc` long (See `get_num_qc`.)

`obs`	Observation from which to extract qc field(s)
`qc`	Extracted qc fields
qc_indx	If present extract only this field, otherwise extract all qc fields

call set_obs_values(obs, values [, copy_indx])

```
type(obs_type),          intent(out) :: obs
real(r8), dimension(:), intent(in)  :: values
integer, optional,       intent(in)  :: copy_indx
```

Set value(s) of data in this observation. If *copy_indx* is present set the single value indexed by *copy_indx* to `values(1)`. *copy_indx* must be between 1 and `num_copies`, inclusive. If *copy_indx* is not present set all copies of data from the `values` array which must be `num_copies` long (See `get_num_copies`.)

`obs`	Observation whose values are being set
`values`	Array of value(s) to be set
copy_indx	If present set only this copy of data, otherwise set all copies

call replace_obs_values(seq, key, values [, copy_indx])

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: key
real(r8), dimension(:),  intent(in)    :: values
integer, optional,       intent(in)    :: copy_indx
```

Set value(s) of data in the observation from a sequence with the given `key`. If *copy_indx* is present set the single value indexed by *copy_indx* to `values(1)`. *copy_indx* must be between 1 and `num_copies`, inclusive. If *copy_indx* is not present set all copies of data from the `values` array which must be `num_copies` long (See `get_num_copies`.)

`seq`	Sequence which contains observation to update
`key`	Key to select which observation
`values`	Array of value(s) to be set
copy_indx	If present set only this copy of data, otherwise set all copies

call set_qc(obs, qc [, qc_indx])

```
type(obs_type),         intent(out) :: obs
real(r8), dimension(:), intent(in)  :: qc
integer, optional,      intent(in)  :: qc_indx
```

Sets the quality control fields in an observation. If *qc_indx* is present set a single field indexed by *qc_indx* to `qc(1)`. *qc_indx* must be between 1 and `num_qc`, inclusive. If *qc_indx* is not present set all quality control fields from the `qc` array which must be `num_qc` long (See `get_num_qc`.)

`obs`	Observation having its qc fields set
`qc`	Input values of qc fields
qc_indx	If present update only this field, otherwise update all qc fields

call replace_qc(seq, key, qc [, qc_indx])

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: key
real(r8), dimension(:),  intent(in)    :: qc
integer, optional,       intent(in)    :: qc_indx
```

Set value(s) of the quality control fields in the observation from a sequence with the given `key`. If *qc_indx* is present set the single value indexed by *qc_indx* to `qc(1)`. *qc_indx* must be between 1 and `num_qc`, inclusive. If *qc_indx* is not present set all quality control fields from the `qc` array which must be `num_qc` long (See `get_num_qc`.)

`seq`	Observation sequence containing observation to update
`key`	Key to select which observation
`qc`	Input values of qc fields
qc_indx	If present, only update single qc field, else update all qc fields

call write_obs(obs, file_id, num_copies, num_qc)

```
type(obs_type), intent(in) :: obs
integer,        intent(in) :: file_id
integer,        intent(in) :: num_copies
integer,        intent(in) :: num_qc
```

Writes an observation and all its associated metadata to a disk file that has been opened with a format consistent with the namelist parameter `write_binary_obs_sequence`.

`obs`	Observation to be written to file
`file_id`	Channel open to file for writing
`num_copies`	The number of copies of data associated with the observation to be output
`num_qc`	The number of qc fields associated with the observation to be output

call read_obs(file_id, num_copies, add_copies, num_qc, add_qc, key, obs, read_format [, max_obs])

```
integer,            intent(in)    :: file_id
integer,            intent(in)    :: num_copies
integer,            intent(in)    :: add_copies
integer,            intent(in)    :: num_qc
integer,            intent(in)    :: add_qc
integer,            intent(in)    :: key
type(obs_type),     intent(inout) :: obs
character(len=*),   intent(in)    :: read_format
integer, optional,  intent(in)    :: max_obs
```

Reads an observation from an obs_sequence file. The number of copies of data and the number of qc values associated with each observation must be provided. If additional copies of data or additional qc fields are needed, arguments allow them to be added. WARNING: The key argument is no longer used and should be removed.

`file_id`	Channel open to file from which to read
`num_copies`	Number of copies of data associated with observation in file
`add_copies`	Number of additional copies of observation to be added
`num_qc`	Number of qc fields associated with observation in file
`add_qc`	Number of additional qc fields to be added
`key`	No longer used, should be deleted
`obs`	The observation being read in
`read_format`	Either the string `'formatted'` or `'unformatted'`
max_obs	If present, specifies the largest observation key number in the sequence. This is used only for additional error checks on the next and previous obs linked list values.

call interactive_obs(num_copies, num_qc, obs, key)

```
integer,         intent(in)    :: num_copies
integer,         intent(in)    :: num_qc
type(obs_type), intent(inout) :: obs
integer,         intent(in)    :: key
```

Use standard input to create an observation. The number of values, number of qc fields, and an observation type-specific key associated with the observation are input. (Note that the key here is not the same as the key in an observation sequence.)

num_copies	Number of copies of data to be associated with observation
num_qc	Number of qc fields to be associated with observation
obs	Observation created via standard input
key	An observation type-specific key can be associated with each observation for use by the obs_def code.

call copy_obs(obs1, obs2)

```
type(obs_type), intent(out) :: obs1
type(obs_type), intent(in)  :: obs2
```

Copies the observation type obs2 to obs1. If the sizes of obs fields are not compatible, the space in obs1 is deallocated and reallocated with the appropriate size. This is overloaded to assignment(=).

obs1	Copy obs2 to here (destination)
obs2	Copy into obs1 (source)

call get_expected_obs_from_def_distrib_state(state_handle, ens_size, copy_indices, key, & obs_def, obs_kind_ind, state_time, isprior, assimilate_this_ob, evaluate_this_ob, expected_obs, & istatus)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: ens_size
integer,             intent(in)  :: copy_indices(ens_size)
integer,             intent(in)  :: key
type(obs_def_type),  intent(in)  :: obs_def
integer,             intent(in)  :: obs_kind_ind
type(time_type),     intent(in)  :: state_time
logical,             intent(in)  :: isprior
integer,             intent(out) :: istatus(ens_size)
logical,             intent(out) :: assimilate_this_ob, evaluate_this_ob
real(r8),            intent(out) :: expected_obs(ens_size)
```

Used to compute the expected value of a set of observations in an observation sequence given a model state vector. Also returns a status variable that reports on problems taking forward operators. This version returns forward operator

values for the entire ensemble in a single call.

state_handle	An observation sequence
keys	List of integer keys that specify observations in seq
ens_index	The ensemble number for this state vector
state	Model state vector
state_time	The time of the state data
obs_vals	Returned expected values of the observations
istatus	Integer error code for use in quality control (0 means no error)
assimilate_this_ob	Returns true if this observation type is being assimilated
evaluate_this_ob	Returns true if this observation type is being evaluated but not assimilated

call delete_seq_head(first_time, seq, all_gone)

```
type(time_type),          intent(in)    :: first_time
type(obs_sequence_type), intent(inout) :: seq
logical,                  intent(out)   :: all_gone
```

Deletes all observations in the sequence with times before first_time. If no observations remain, return all_gone as .true. If no observations fall into the time window (e.g. all before first_time or empty sequence to begin with), no deletions are done and all_gone is simply returned as .true.

first_time	Delete all observations with times before this
seq	An observation sequence
all_gone	Returns true if there are no valid observations remaining in the sequence after first_time

call delete_seq_tail(last_time, seq, all_gone)

```
type(time_type),          intent(in)    :: last_time
type(obs_sequence_type), intent(inout) :: seq
logical,                  intent(out)   :: all_gone
```

Deletes all observations in the sequence with times after last_time. If no observations remain, return all_gone as .true. If no observations fall into the time window (e.g. all after last_time or empty sequence to begin with), no deletions are done and all_gone is simply returned as .true.

last_time	Delete all observations with times after this
seq	An observation sequence
all_gone	Returns true if there are no valid observations remaining in the sequence before last_time

6.208.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_sequence_nml
   write_binary_obs_sequence = .false.
   read_binary_file_format   = 'native'
  /
```

Item	Type	Description
write_binary_obs_sequence	logical	If true, write binary obs_sequence files. If false, write ascii obs_sequence files.
read_binary_file_format	character(len=32)	The 'endian'ness of binary obs_sequence files. May be 'native' (endianness matches hardware default), 'big-endian', 'little-endian', and possibly 'cray'. Ignored if observation sequence files are ASCII.

6.208.5 Files

- obs_sequence_mod.nml in input.nml
- Files for reading and writing obs_sequences and obs specified in filter_nml.

6.208.6 References

- none

6.208.7 Private components

N/A

6.209 MODULE smoother_mod

Attention: The DART smoother works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `smoother_mod` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

6.209.1 Overview

Implements a fixed lag ensemble smoother as part of the filter. For now, this is done inefficiently with a separate call to `assim_tools_mod:filter_assim()` for each lag.

To enable the smoother, set the number of lags (num_lags) to something larger than 0 in the `smoother_nml` section of your `input.nml` file and run `filter` as before.

```
&smoother_nml
   num_lags              = 10,
   start_from_restart    = .false.,
   output_restart        = .true.,
   restart_in_file_name  = "ics",
   restart_out_file_name = "restart"  /
```

In the low order models, 10 is a plausible number.

In addition to generating `preassim.nc` and `analysis.nc` files, files of the form `Lag_NNNNN_Diag.nc` will be generated. Each of these has N fewer timesteps than the lag=0 run, starting at the same time but ending N timesteps sooner. The `obs_seq.final` file and the `preassim.nc` and `analysis.nc` files will be the same as the non-lagged version; the new output will be in each of the `Lag_NNNNN_Diag.nc` files.

6.209.2 Example

If you have a `true_state.nc` file and want to use the `plot_total_err` matlab function to plot the error, you must do the following steps to generate analogs of lagged `true_state.nc` files to use as a comparison. (The logic is not currently implemented in the matlab scripts to be able to compare netCDF files with unequal time coordinates.)

Make N separate versions of the true_state.nc with the last N timesteps removed. Using the netCDF NCO operator program 'ncks' is one way. If the true_state.nc file has 1000 time steps, then this command removes the last one:

ncks -d time,0,998 true_state.nc True_Lag01.nc

Note that the first time is at index 0, so the last timestep is index 999 in the full file, and 998 in the truncated file. Repeat this step for all N lags. Here are NCO commands to generate 10 truth files for num_lags = 10, 1000 time steps in true_state.nc:

ncks -d time,0,998 true_state.nc True_Lag01.nc ncks -d time,0,997 true_state.nc True_Lag02.nc ncks -d time,0,996 true_state.nc True_Lag03.nc ncks -d time,0,995 true_state.nc True_Lag04.nc ncks -d time,0,994 true_state.nc True_Lag05.nc ncks -d time,0,993 true_state.nc True_Lag06.nc ncks -d time,0,992 true_state.nc True_Lag07.nc ncks -d time,0,991 true_state.nc True_Lag08.nc ncks -d time,0,990 true_state.nc True_Lag09.nc ncks -d time,0,989 true_state.nc True_Lag10.nc

Here is an example matlab session which plots the lag=0 results and then odd numbered lags from 1 to 9. It uses the `plot_total_err` function from the $DART/matlab directory:

```
datadir    = '.';
truth_file = fullfile(datadir,'true_state.nc');
diagn_file = fullfile(datadir,'preassim.nc');
plot_total_err
reply = input('original data.  hit enter to continue ');
```

(continues on next page)

(continued from previous page)

```
truth_file = fullfile(datadir,'True_Lag01.nc');
diagn_file = fullfile(datadir,'Lag_00001_Diag.nc');
plot_total_err
reply = input('Lag 01.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag03.nc');
diagn_file = fullfile(datadir,'Lag_00003_Diag.nc');
plot_total_err
reply = input('Lag 03.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag05.nc');
diagn_file = fullfile(datadir,'Lag_00005_Diag.nc');
plot_total_err
reply = input('Lag 05.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag07.nc');
diagn_file = fullfile(datadir,'Lag_00007_Diag.nc');
plot_total_err
reply = input('Lag 07.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag09.nc');
diagn_file = fullfile(datadir,'Lag_00009_Diag.nc');
plot_total_err
reply = input('Lag 09.  hit enter to continue ');
```

6.209.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&smoother_nml
   num_lags              = 0,
   start_from_restart    = .false.,
   output_restart        = .false.,
   restart_in_file_name  = 'ics',
   restart_out_file_name = 'restart'
/
```

Item	Type	Description
num_lags	integer	Number of smoother lags; < 1 means no smoother.
start_from_restart	logical	True if smoother states are to come from restart file(s). False if they are to be spun up from scratch.
output_restart	logical	True if restart file(s) are to be written, else false.
restart_in_file_name	character(len=129)	String used to construct the file name from which to read restart data. `Lag_NNNNN_` will be prepended to the specified value to create the actual filename. If each ensemble is to be read from a separate file, the .NNNN ensemble number will also be appended. e.g. specifying 'ics' here results in 'Lag_00001_ics' if all ensemble members are read from a single file, 'Lag_00001_ics.0001', 'Lag_00001_ics.0002', etc for multiples.
restart_out_file_name	character(len=129)	String used to construct the file name to which to write restart data. `Lag_NNNNN_` will be prepended to the specified value to create the actual filename. If each ensemble is to be written to a separate file, the .NNNN ensemble number will also be appended. e.g. specifying 'restart' here results in 'Lag_00001_restart' if all ensemble members are written to a single file, 'Lag_00001_restart.0001', 'Lag_00001_restart.0002', etc for multiples.

6.209.4 Other modules used

```
types_mod
mpi_utilities_mod
utilities_mod
ensemble_manager_mod
time_manager_mod
assim_model_mod
assim_tools_mod
obs_sequence_mod
adaptive_inflate_mod
```

6.209.5 Public interfaces

use smoother_mod, only :	smoother_read_restart
	advance_smoother
	smoother_gen_copy_meta_data
	smoother_write_restart
	init_smoother
	do_smoothing
	smoother_mean_spread
	smoother_assim
	filter_state_space_diagnostics
	smoother_ss_diagnostics
	smoother_end

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call smoother_read_restart(ens_handle, ens_size, model_size, time1, init_time_days)

```
type(ensemble_type), intent(inout) :: ens_handle
integer, intent(in)                :: ens_size
integer, intent(in)                :: model_size
type(time_type), intent(inout)     :: time1
integer, intent(in)                :: init_time_days
```

Reads in ensemble of states for all lag estimates from a restart file.

ens_handle	Handle of ensemble manager structure of single state; copied into all lags for startup.
ens_size	Size of the ensemble.
model_size	Size of the model state vector.
time1	Overwrite the time in the restart file with this value if init_time_days is non-negative.
init_time_days	If non-negative, use time1 instead of time in restart file.

call advance_smoother(ens_handle)

```
type(ensemble_type), intent(in) :: ens_handle
```

Advances smoother state estimates at all lags forward in time. This entails copying the most recent smoother state, contained in ens_handle, into the lag 1 smoother state and pushing back all other lags by 1 (i.e. lag 1 becomes lag 2, etc.).

ens_handle	Ensemble handle with most recent filtered state.

call smoother_gen_copy_meta_data(num_output_state_members, output_inflation)

```
integer, intent(in) :: num_output_state_members
logical, intent(in) :: output_inflation
```

Initializes the metadata required for the smoother state space diagnostic files.

num_output_state_members	Number of copies of smoother state vector that should be in state space diagnostic output.
output_inflation	True if smoother state space output should include inflation values.

call smoother_write_restart(start_copy, end_copy)

```
integer, intent(in) :: start_copy
integer, intent(in) :: end_copy
```

Outputs restart files for all lags of smoother state. Integer arguments specify the start and end global indices of a continguous set of copies that contain the ensemble members.

start_copy	Global index of ensemble copy that starts the actual ensemble members for smoother.
end_copy	Global index of ensemble copy that ends the actual ensemble members for smoother.

call init_smoother(ens_handle, POST_INF_COPY, POST_INF_SD_COPY)

```
type(ensemble_type), intent(inout) :: ens_handle
integer, intent(in)                :: POST_INF_COPY
integer, intent(in)                :: POST_INF_SD_COPY
```

Initializes the storage needed for a smoother. Also initializes an adaptive inflation type that does NO inflation (not currently supported for smoothers).

ens_handle	An ensemble handle for the filter that contains information about ensemble and model size.
POST_INF_COPY	Global index of ensemble copy that holds posterior state space inflation values.
POST_INF_SD_COPY	Global index of ensemble copy that holds posterior inflation standard deviation values.

var = do_smoothing()

```
logical, intent(out) :: do_smoothing
```

Returns true if smoothing is to be done, else false.

do_smoothing	Returns true if smoothing is to be done.

call smoother_mean_spread(ens_size,ENS_MEAN_COPY,ENS_SD_COPY, output_state_ens_mean,output_state_ens_spread)

```
integer, intent(in) :: ens_size
integer, intent(in) :: ENS_MEAN_COPY
integer, intent(in) :: ENS_SD_COPY
logical, intent(in) :: output_state_ens_mean
logical, intent(in) :: output_state_ens_spread
```

Computes the ensemble mean (and spread if required) of all state variables for all lagged ensembles. Spread is only computed if it is required for output.

ens_size	Size of ensemble.
ENS_MEAN_COPY	Global index of copy that stores ensemble mean.
ENS_SD_COPY	Global index of copy that stores ensemble spread.
output_state_ens_mean	True if the ensemble mean is to be output to state diagnostic file.
output_state_ens_spread	True if ensemble spread is to be output to state diagnostic file.

call smoother_assim(obs_ens_handle, seq, keys, ens_size, num_groups, obs_val_index, ENS_MEAN_COPY, ENS_SD_COPY, PRIOR_INF_COPY, PRIOR_INF_SD_COPY, OBS_KEY_COPY, OBS_GLOBAL_QC_COPY, OBS_PRIOR_MEAN_START, OBS_PRIOR_MEAN_END, OBS_PRIOR_VAR_START, OBS_PRIOR_VAR_END)

```
type(ensemble_type), intent(inout)  :: obs_ens_handle
type(obs_sequence_type), intent(in) :: seq
integer, dimension(:), intent(in)   :: keys
integer, intent(in)                 :: ens_size
integer, intent(in)                 :: num_groups
integer, intent(in)                 :: obs_val_index
```

(continues on next page)

(continued from previous page)

```
integer, intent(in)                 :: ENS_MEAN_COPY
integer, intent(in)                 :: ENS_SD_COPY
integer, intent(in)                 :: PRIOR_INF_COPY
integer, intent(in)                 :: PRIOR_INF_SD_COPY
integer, intent(in)                 :: OBS_KEY_COPY
integer, intent(in)                 :: OBS_GLOBAL_QC_COPY
integer, intent(in)                 :: OBS_PRIOR_MEAN_START
integer, intent(in)                 :: OBS_PRIOR_MEAN_END
integer, intent(in)                 :: OBS_PRIOR_VAR_START
integer, intent(in)                 :: OBS_PRIOR_VAR_END
```

Does assimilation of a set of observations for each smoother lag.

obs_ens_handle	Handle for ensemble manager holding prior estimates of observations.
seq	Observation sequence being assimilated.
keys	A one dimensional array containing indices in seq of observations to as similate at current time.
ens_size	Ensemble size.
num_groups	Number of groups in filter.
obs_val_index	Integer index of copy of data in seq that contains the observed value from instruments.
ENS_MEAN_COPY	Global index in smoother's state ensemble that holds ensemble mean.
ENS_SD_COPY	Global index in smoother's state ensemble that holds ensemble standard deviation.
PRIOR_INF_COPY	Global index in obs_ens_handle that holds inflation values (not used for smoother).
PRIOR_INF_SD_COPY	Global index in obs_ens_handle that holds inflation sd values (not used for smoother).
OBS_KEY_COPY	Global index in obs_ens_handle that holds the key for the observation.
OBS_GLOBAL_QC_COPY	Global index in obs_ens_handle that holds the quality control value.
OBS_PRIOR_MEAN_START	Global index in obs_ens_handle that holds the first group's prior mean.
OBS_PRIOR_MEAN_END	Global index in obs_ens_handle that holds the last group's prior mean.
OBS_PRIOR_VAR_START	Global index in obs_ens_handle that holds the first group's prior variance.
OBS_PRIOR_VAR_END	Global index in obs_ens_handle that holds the last group's prior variance.

call filter_state_space_diagnostics(out_unit, ens_handle, model_size, num_output_state_members, output_state_mean_index, output_state_spread_index, output_inflation, temp_ens, ENS_MEAN_COPY, ENS_SD_COPY, inflate, INF_COPY, INF_SD_COPY)

```
type(netcdf_file_type), intent(inout)   :: out_unit
type(ensemble_type), intent(inout)      :: ens_handle
integer, intent(in)                     :: model_size
integer, intent(in)                     :: num_output_state_members
integer, intent(in)                     :: output_state_mean_index
integer, intent(in)                     :: output_state_spread_index
logical, intent(in)                     :: output_inflation
real(r8), intent(out)                   :: temp_ens(model_size)
integer, intent(in)                     :: ENS_MEAN_COPY
integer, intent(in)                     :: ENS_SD_COPY
type(adaptive_inflate_type), intent(in) :: inflate
integer, intent(in)                     :: INF_COPY
```

(continues on next page)

(continued from previous page)

```
integer, intent(in)                    :: INF_SD_COPY
```

Writes state space diagnostic values including ensemble members, mean and spread, and inflation mean and spread to a netcdf file.

out_unit	Descriptor for the netcdf file being written.
ens_handle	Ensemble handle whose state space values are to be written.
model_size	Size of the model state vector.
num_output_state_members	Number of individual state members to be output.
output_state_mean_index	Index in netcdf file for ensemble mean.
output_state_spread_index	Index in netcdf file for ensemble spread.
output_inflation	True if the inflation values are to be output. Default is .TRUE.
temp_ens	Storage passed in to avoid having to allocate extra space.
ENS_MEAN_COPY	Global index in ens_handle for ensemble mean.
ENS_SD_COPY	Global index in ens_handle for ensemble spread.
inflate	Contains description and values of state space inflation.
INF_COPY	Global index in ens_handle of inflation values.
INF_SD_COPY	Global index in ens_handle of inflation standard deviation values.

call smoother_ss_diagnostics(model_size, num_output_state_members, output_inflation, temp_ens, ENS_MEAN_COPY, ENS_SD_COPY, POST_INF_COPY, POST_INF_SD_COPY)

```
integer, intent(in)   :: model_size
integer, intent(in)   :: num_output_state_members
logical, intent(in)   :: output_inflation
real(r8), intent(out) :: temp_ens(model_size)
integer, intent(in)   :: ENS_MEAN_COPY
integer, intent(in)   :: ENS_SD_COPY
integer, intent(in)   :: POST_INF_COPY
integer, intent(in)   :: POST_INF_SD_COPY
```

Outputs state space diagnostics files for all smoother lags.

model_size	Size of the model state vector.
num_output_state_members	Number of state copies to be output in the state space diagnostics file.
output_inflation	True if the inflation values are to be output. Default is .TRUE.
temp_ens	Storage passed in to avoid having to allocate extra space.
ENS_MEAN_COPY	Global index of the ensemble mean in the lag smoother ensemble handles.
ENS_SD_COPY	Global index of the ensemble spread in the lag smoother ensemble handles.
POST_INF_COPY	Global index of the inflation value in the lag smoother ensemble handles (not currently used).
POST_INF_SD_COPY	Global index of the inflation spread in the lag smoother ensemble handles (not currently used).

call smoother_end()

Releases storage allocated for smoother.

call smoother_inc_lags()

Increments the number of lags that are in use for smoother. Used when a smoother is being started up and there have not been enough times to propagate the state to all requested lags.

6.209.6 Files

- input.nml
- smoother initial condition files
- smoother restart files

6.209.7 References

1. none

6.209.8 Private components

N/A

6.210 MODULE assim_model_mod

6.210.1 Overview

This module acts as an intermediary between DART compliant models and the filter. At one time the assim_model_type, which combines a state vector and a time_type, was envisioned as being fundamental to how DART views model states. This paradigm is gradually being abandoned so that model state vectors and times are handled as separate data types. It is important to call static_init_assim_model before using routines in assim_model_mod. Interfaces to work with model time stepping, restart files, and computations about the locations of model state variables and the distance between observations and state variables. Many of the interfaces are passed through nearly directly to the model_mod.

Notes

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

6.210.2 Namelist

This module does not have a namelist.

6.210.3 Other modules used

```
types_mod
location_mod (model dependent choice)
time_manager_mod
utilities_mod
model_mod
netcdf
typeSizes (part of netcdf)
```

6.210.4 Public interfaces

use assim_model_mod, only :	
	adv_1step
	aoutput_diagnostics
	aread_state_restart
	assim_model_type
	awrite_state_restart
	close_restart
	copy_assim_model
	end_assim_model
	ens_mean_for_model
	finalize_diag_output
	get_close_maxdist_init
	get_close_obs
	get_close_obs_init
	get_closest_state_time_to
	get_diag_input_copy_meta_data
	get_initial_condition
	get_model_size
	get_model_state_vector
	get_model_time
	get_model_time_step
	get_state_meta_data
	init_assim_model
	init_diag_input
	init_diag_output
	input_diagnostics
	interpolate
	nc_append_time

continues on next page

Table 9 – continued from previous page

	nc_get_tindex
	nc_write_calendar_atts
	netcdf_file_type
	open_restart_read
	open_restart_write
	output_diagnostics
	pert_model_state
	read_state_restart
	set_model_state_vector
	set_model_time
	static_init_assim_model
	write_state_restart

```
type assim_model_type
   private
   real(r8), pointer   :: state_vector(:)
   type(time_type)     :: time
   integer             :: model_size
   integer             :: copyID
end type assim_model_type
```

This type is used to represent both the state and time of a state from a model.

Component	Description
state_vector	A one dimensional representation of the model state vector.
time	The time of the model state.
model_s	Size of the model state vector.
copyID	Not used in present implementation.

```
type netcdf_file_type
   integer             :: ncid
   integer             :: Ntimes
   integer             :: NtimesMAX
   real(r8), pointer   :: rtimes(:)
   type(time_type), pointer :: times(:)
   character(len = 80)      :: fname
end type netcdf_file_type
```

Basically, we want to keep a local mirror of the unlimited dimension coordinate variable (i.e. time) because dynamically querying it causes unacceptable performance degradation over "long" integrations.

Component	Description
ncid	The netcdf file unit id.
Ntimes	The current working length.
NtimesMAX	Allocated length.
rtimes	Times as real (r8).
times	Times as time_types.
fname	Netcdf file name.

call static_init_assim_model()

Initializes the assim_model class. Must be called before any other assim_model_mod interfaces are used. Also calls the static initialization for the underlying model. There are no arguments.

ncFileID = init_diag_output(FileName, global_meta_data, copies_of_field_per_time, meta_data_per_copy [, lagID])

```
type(netcdf_file_type)          :: init_diag_output
character (len = *), intent(in) :: FileName
character (len = *), intent(in) :: global_meta_data
integer, intent(in)             :: copies_of_field_per_time
character (len = *), intent(in) :: meta_data_per_copy(copies_of_field_per_time)
integer, optional, intent(in)   :: lagID
```

Initializes a netCDF file for output of state space diagnostics. A handle to the channel on which the file is opened is returned.

ncFileID	Identifier for the netcdf file is returned. This is not an integer unit number, but a derived type containing additional information about the opened file.
FileName	Name of file to open.
global_meta_data	Global metadata that describes the contents of this file.
copies_of_field_per_time	Number of copies of data to be written at each time. For instance, these could be the prior ensemble members, prior ensemble mean, prior ensemble spread, posterior ensemble members, posterior spread and mean, etc..
meta_data_per_copy	Metadata describing each of the copies.
lagID	If using the smoother, which lag number this output is for.

var = get_model_size()

```
integer :: get_model_size
```

Returns the size of the model state vector. This is a direct pass through to the model_mod.

var = get_closest_state_time_to(model_time, time)

```
type(time_type)              ::  get_closest_state_time_to
type(time_type), intent(in)  ::  model_time
type(time_type), intent(in)  ::  time
```

Returns the closest time that a model is capable of advancing a given state to a specified time. For instance, what is the closest time to 12GMT 01 January, 2004 that a model state at 00GMT 01 January, 2004 can be advanced? If the model time is past the time, the model time is returned (new feature in releases after Hawaii).

var	The closest time to which the model can be advanced is returned.
model_time	The time of a model state vector.
time	A time that one would like to get close to with the model.

call get_state_meta_data()

Pass through to model_mod. See model_mod documentation for arguments and description.

var = get_model_time(assim_model)

```
type(time_type)                    :: get_model_time
type(assim_model_type), intent(in) :: assim_model
```

Returns time from an assim_model type.

var	Returned time from assim_model
assim_model	Assim_model type from which to extract time

var = get_model_state_vector(assim_model)

```
real(r8)                           :: get_model_state_vector(model_size)
type(assim_model_type), intent(in) :: assim_model
```

Returns the state vector component from an assim_model_type.

var	Returned state vector
assim_model	Input assim_model_type

call copy_assim_model(model_out, model_in)

```
type(assim_model_type), intent(out) :: model_out
type(assim_model_type), intent(in)  :: model_in
```

Copies one assim_model_type to another.

model_out	Copy.
model_in	Data to be copied.

call interpolate(x, location, loctype, obs_vals, istatus)

```
real(r8),            intent(in)  :: x(:)
type(location_type), intent(in)  :: location
integer,             intent(in)  :: loctype
real(r8),            intent(out) :: obs_vals
integer,             intent(out) :: istatus
```

Interpolates a given model state variable type to a location given the model state vector. Nearly direct call to model_interpolate in model_mod. See model_mod for the error return values in istatus.

x	Model state vector.
location	Location to which to interpolate.
loctype	Type of variable to interpolate.
obs_vals	Returned interpolated value.
istatus	Returned as 0 if all is well, else various errors.

call set_model_time(assim_model, time)

```
type(assim_model_type), intent(inout) :: assim_model
type(time_type), intent(in)           :: time
```

Sets the time in an assim_model_type.

assim_model	Set the time in this assim_model_type.
time	Set to this time

call set_model_state_vector(assim_model, state)

```
type(assim_model_type), intent(inout) :: assim_model
real(r8), intent(in)                  :: state(:)
```

Set the state in an assim_model_type.

assim_model	Set the state vector in this assim_model_type.
state	The state vector to be inserted.

call write_state_restart(assim_model, funit [, target_time])

```
type(assim_model_type),    intent(in) :: assim_model
integer,                   intent(in) :: funit
type(time_type), optional, intent(in) :: target_time
```

Writes a restart from an assim_model_type with an optional target_time.

assim_model	Write a restart from this assim_model_type.
funit	Integer file unit id open for output of restart files.
target_time	If present, put this target time at the front of the restart file.

call read_state_restart(assim_model, funit [, target_time])

```
type(assim_model_type),    intent(out) :: assim_model
integer,                   intent(in)  :: funit
type(time_type), optional, intent(out) :: target_time
```

Read a state restart file into assim_model_type. Optionally read a prepended target time.

assim_model	Read the time and state vector from restart into this.
funit	File id that has been opened for reading restart files.
target_time	If present, read a target time from the front of the file into this.

call output_diagnostics(ndFileID, state [, copy_index])

```
type(netcdf_file_type), intent(inout) :: ndFileID
type(assim_model_type), intent(in)    :: state
integer, optional,      intent(in)    :: copy_index
```

Writes one copy of the state time and vector to a netCDF file.

ndFileID	An identifier for a netCDF file
state	State vector and time
copy_index	Which copy of state is to be output

call end_assim_model()

Called to clean-up at end of assim_model use. For now just passes through to model_mod.

call input_diagnostics(file_id, state, copy_index)

```
integer,                intent(in)    :: file_id
type(assim_model_type), intent(inout) :: state
integer,                intent(out)   :: copy_index
```

Used to read in a particular copy of the state vector from an open state diagnostics file.

file_id	Integer descriptor (channel number) for a diagnostics file being read.
state	Assim_model_type to read in data.
copy_index	Which copy of state to be read.

var = init_diag_input(file_name, global_meta_data, model_size, copies_of_field_per_time)

```
integer                       :: init_diag_input
character(len=*), intent(in)  :: file_name
character(len=*), intent(out) :: global_meta_data
integer,          intent(out) :: model_size
integer,          intent(out) :: copies_of_field_per_time
```

Opens a state diagnostic file and reads the global meta data, model size, and number of data copies.

var	Returns the unit number on which the file is open.
file_name	File name of state diagnostic file.
global_meta_data	Global metadata string from file.
model_size	Size of model.
copies_of_field_per_time	Number of copies of the state vector at each time.

call init_assim_model(state)

```
type(assim_model_type), intent(inout) :: state
```

Creates storage for an assim_model_type.

state	An assim_model_type that needs storage created.

call get_diag_input_copy_meta_data(file_id, model_size_out, num_copies, location, meta_data_per_copy)

```
integer,             intent(in)  :: file_id
integer,             intent(in)  :: model_size_out
integer,             intent(in)  :: num_copies
type(location_type), intent(out) :: location(model_size_out)
character(len = *)               :: meta_data_per_copy(num_copies)
```

Reads meta-data describing state vectors in a state diagnostics file. Given the file, the model_size, and the number of copies, returns the locations of each state variable and the text description of each copy.

file_id	Integer channel open to state diagostic file being read
Model_size_out	model size
num_copies	Number of copies of state in file
location	Returned locations for state vector
meta_data_per_copy	Meta data describing what is in each copy of state vector

var = finalize_diag_output(ncFileID)

```
integer                                :: finalize_diag_output
type(netcdf_file_type), intent(inout) :: ncFileID
```

Used to complete writing on and open netcdf file. An error return is provided for passing to the netcdf error handling routines.

var	Returns an error value.
ncFileID	Netcdf file id of an open file.

call aread_state_restart(model_time, model_state, funit [, target_time])

```
type(time_type),           intent(out) :: model_time
real(r8),                  intent(out) :: model_state(:)
integer,                   intent(in)  :: funit
type(time_type), optional, intent(out) :: target_time
```

Reads a model time and state, and optionally a prepended target time, from a state restart file.

`model_time`	Returned time of model state
`model_state`	Returned model state.
`funit`	Channel open for reading a state restart file.
target_time	If present, this time is read from the front of the restart file.

call aoutput_diagnostics(ncFileID, model_time, model_state [, copy_index])

```
type(netcdf_file_type), intent(inout) :: ncFileID
type(time_type),        intent(in)    :: model_time
real(r8),               intent(in)    :: model_state(:)
integer, optional,      intent(in)    :: copy_index
```

Write a state vector to a state diagnostics netcdf file.

`ncFileID`	Unit for a state vector netcdf file open for output.
`model_time`	The time of the state to be output
`model_state`	A model state vector to be output.
copy_index	Which copy of state vector is to be written, default is copy 1

call awrite_state_restart(model_time, model_state, funit [, target_time])

```
type(time_type),           intent(in) :: model_time
real(r8),                  intent(in) :: model_state(:)
integer,                   intent(in) :: funit
type(time_type), optional, intent(in) :: target_time
```

Writes a model time and state vector to a restart file and optionally prepends a target time.

`model_time`	Time of model state.
`model_state`	Model state vector.
`funit`	Channel of file open for restart output.
target_time	If present, time to be prepended to state time / vector.

call pert_model_state()

Passes through to pert_model_state in model_mod. See model_mod documentation for arguments and details.

var = nc_append_time(ncFileID, time)

```
integer                                 :: nc_append_time
type(netcdf_file_type), intent(inout) :: ncFileID
type(time_type),        intent(in)    :: time
```

Appends the time to the time coordinate variable of the netcdf file. The new length of the time variable is returned. Requires that time is a coordinate variable AND it is the unlimited dimension.

var	Returns new length of time variable.
ncFileID	Points to open netcdf file.
time	The next time to be added to the file.

var = nc_write_calendar_atts(ncFileID, TimeVarID)

```
integer                              :: nc_write_calendar_atts
type(netcdf_file_type), intent(in) :: ncFileID
integer,                intent(in) :: TimeVarID
```

Sets up the metadata for the appropriate calendar being used in the time manager an writes it to a netcdf file.

var	Returns a netcdf error code.
ncFileID	Netcdf file id pointing to a file open for writing.
TimeVarID	The index of the time variable in the netcdf file.

var = nc_get_tindex(ncFileID, statetime)

```
integer                                 :: nc_get_tindex
type(netcdf_file_type), intent(inout) :: ncFileID
type(time_type),        intent(in)    :: statetime
```

Returns the index of a time from the time variable in a netcdf file. This function has been replaced with more efficient approaches and may be deleted from future releases.

var	The index of the time in the netcdf file.
ncFileID	File id for an open netcdf file.
statetime	The time to be found in the netcdf file.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

This passes through to model_mod. See model_mod documentation for arguments and details.

var	Returns time step of model.

var = open_restart_read(file_name)

```
integer                       :: open_restart_read
character(len=*), intent(in) :: file_name
```

Opens a restart file for readig.

var	Returns a file descriptor (channel number).
file_name	Name of restart file to be open for reading.

var = open_restart_write(file_name)

```
integer                       :: open_restart_write
character(len=*), intent(in) :: file_name
```

Open a restart file for writing.

var	Returns a file descriptor (channel) for a restart file.
file_name	File name of restart file to be opened.

call close_restart(file_unit)

```
integer, intent(in) :: file_unit
```

Closes a restart file.

file_unit	File descriptor (channel number) of open restart file.

call adv_1step()

Advances a model by one step. Pass through to model_mod. See model_mod documentation for arguments and details.

call get_initial_condition(time, x)

```
type(time_type), intent(out) :: time
real(r8),        intent(out) :: x
```

Obtains an initial condition from models that support this option.

time	the valid time of the model state
x	the initial model state

call ens_mean_for_model(ens_mean)

```
type(r8), intent(in) :: ens_mean(:)
```

An array of length model_size containing the ensemble means. This is a direct pass through to the model_mod.

ens_mean	Array of length model_size containing the mean for each entry in the state vector.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
type(r8), intent(in)                :: maxdist
```

Sets the threshold distance. Anything closer than this is deemed to be close. This is a direct pass through to the model_mod, which in turn can pass through to the location_mod.

gc	Data for efficiently finding close locations.
maxdist	Anything closer than this distance is a close location.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
```

(continues on next page)

(continued from previous page)

```
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8),  optional,  intent(out) :: dist(:)
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The observation kinds are passed in to allow more sophisticated distance computations to be done if needed. This is a direct pass through to the model_mod, which in turn can pass through to the location_mod.

gc	Data for efficiently finding close locations.
base_obs_loc	Single given location.
base_obs_kind	Kind of the single location.
obs	List of observations from which close ones are to be found.
obs_kind	Kind associated with observations in obs list.
num_close	Number of observations close to the given location.
close_ind	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(:)
```

Initialize storage for efficient identification of locations close to a given location. Allocates storage for keeping track of which 'box' each observation in the list is in. This is a direct pass through to the model_mod, which in turn can pass through to the location_mod.

gc	Data for efficiently finding close locations.
num	The number of locations in the list.
obs	The location of each element in the list, not used in 1D implementation.

6.210.5 Files

filename	purpose/comment
filter_restart	specified in &filter_nml:restart_in_filename
filter_restart	specified in &filter_nml:restart_out_filename
input.nml	to read namelists

6.210.6 References

- none

6.210.7 Private components

N/A

6.211 MODULE assim_tools_mod

6.211.1 Overview

This module provides subroutines that implement the parallel versions of the sequential scalar filter algorithms. These include the standard sequential filter as described in Anderson 2001, 2003 along with systematic correction algorithms for both mean and spread. In addition, algorithms to do a variety of flavors of filters including the EAKF, ENKF, particle filter, and kernel filters are included. The parallel implementation that allows each observation to update all state variables that are close to it at the same time is described in Anderson and Collins, 2007.

6.211.2 Filter types

Available observation space filter types include:

- 1 = EAKF (Ensemble Adjustment Kalman Filter, see Anderson 2001)
- 2 = ENKF (Ensemble Kalman Filter)
- 3 = Kernel filter
- 4 = Observation Space Particle filter
- 5 = Random draw from posterior (contact dart@ucar.edu before using)
- 6 = Deterministic draw from posterior with fixed kurtosis (ditto)
- 7 = Boxcar kernel filter
- 8 = Rank Histogram filter (see Anderson 2010)
- 9 = Particle filter (see Poterjoy 2016)

We recommend using type=1, the EAKF. Note that although the algorithm is expressed in a slightly different form, the EAKF is identical to the EnSRF (Ensemble Square Root Filter) described by Whitaker and Hamill in 2002. Highly non-gaussian distributions may get better results from type=8, Rank Histogram filter.

6.211.3 Localization

Localization controls how far the impact of an observation extends. The namelist items related to localization are spread over several different individual namelists, so we have made a single collected description of them here along with some guidance on setting the values.

This discussion centers on the mechanics of how you control localization in DART with the namelist items, and a little bit about pragmatic approaches to picking the values. There is no discussion about the theory behind localization - contact Jeff Anderson for more details. Additionally, the discussion here applies specifically to models using the 3d-sphere location module. The same process takes place in 1d models but the details of the location module namelist is different.

The following namelist items related to 3d-sphere localization are all found in the `input.nml` file:

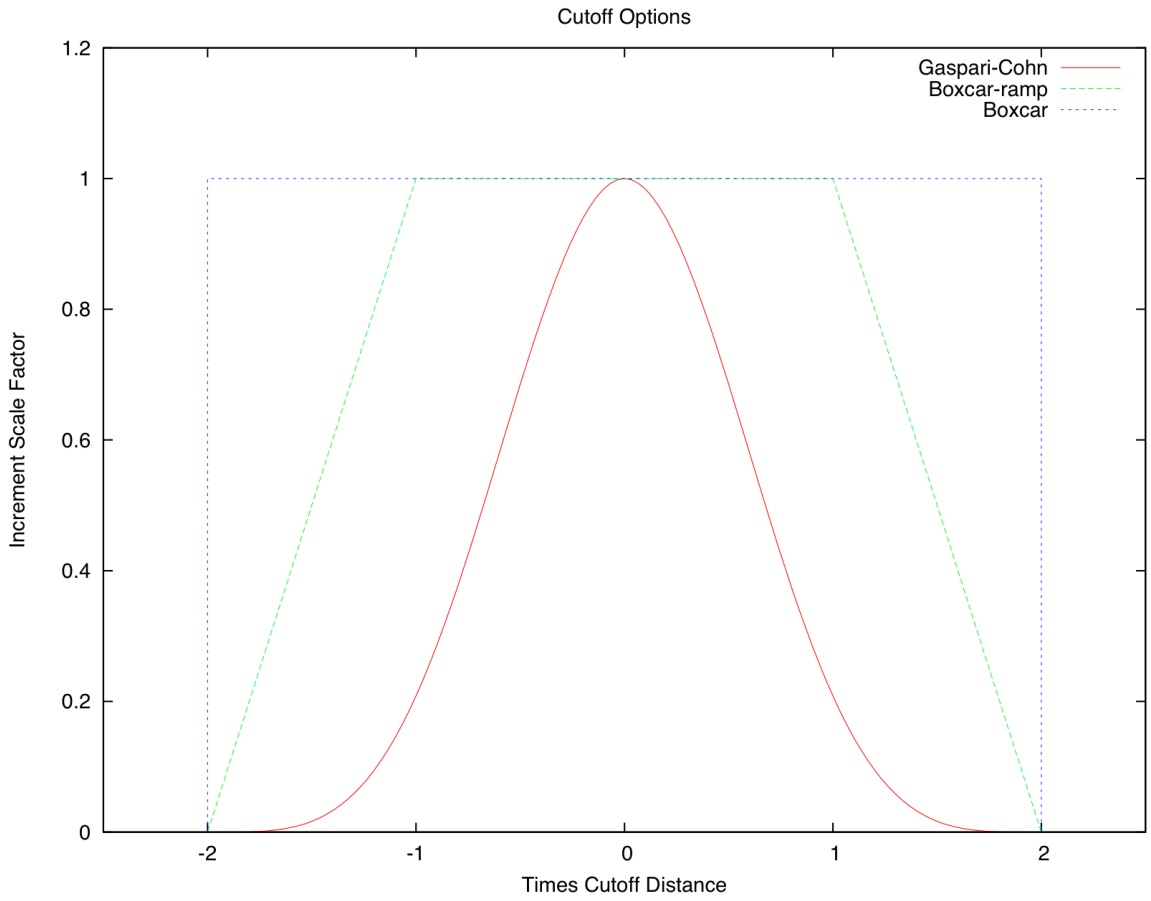
&assim_tools_nml :: cutoff
valid values: 0.0 to infinity

This is the value, in radians, of the half-width of the localization radius (this follows the terminology of an early paper on localization). For each observation, a state vector item increment is computed based on the covariance values. Then a multiplier, based on the 'select_localization' setting (see below) decreases the increment as the distance between the obs and the state vector item increases. In all cases if the distance exceeds 2*cutoff, the increment is 0.

&cov_cutoff_nml :: select_localization
valid values: 1=Gaspari-Cohn; 2=Boxcar; 3=Ramped Boxcar

Controls the shape of the multiplier function applied to the computed increment as the distance increases between the obs and the state vector item. Most users use type 1 localization.

- Type 1 (Gaspari-Cohn) has a value of 1 at 0 distance, 0 at 2*cutoff, and decreases in an approximation of a gaussian in between.
- Type 2 (Boxcar) is 1 from 0 to 2*cutoff, and then 0 beyond.
- Type 3 (Ramped Boxcar) is 1 to cutoff and then ramps linearly down to 0 at 2*cutoff.



&location_nml :: horiz_dist_only

valid values: .true., .false.

If set to .true., then the vertical location of all items, observations and state vector both, are ignored when computing distances between pairs of locations. This has the effect that all items within a vertical-cylindrical area are considered the same distance away.

If set to .false., then the full 3d separation is computed. Since the localization is computed in radians, the 2d distance is easy to compute but a scaling factor must be given for the vertical since vertical coordinates can be in meters, pressure, or model levels. See below for the 'vert_normalization_xxx' namelist items.

&location_nml :: vert_normalization_{pressure,height,level,scale_height}

valid values: real numbers, in pascals, meters, index, and value respectively

If 'horiz_dist_only' is set to .true., these are ignored. If set to .false., these are required. They are the amount of that quantity that is equivalent to 1 radian in the horizontal. If the model is an earth-based one, then one radian is roughly 6366 kilometers, so if vert_normalization_height is set to 6366000 meters, then the localization cutoff will be a perfect sphere. If you want to localize over a larger distance in the vertical than horizontal, use a larger value. If you want to localize more sharply in the vertical, use a smaller number. The type of localization used is set by which type of vertical coordinate the observations and state vector items have.

If you have observations with different vertical coordinates (e.g. pressure and height), or if your observations have a different vertical coordinate than your state vector items, or if you want to localize in a different type of unit than your normal vertical coordinate (e.g. your model uses pressure in the vertical but you wish to localize in meters), then you will need to modify or add a `get_close()` routine in your `model_mod.f90` file. See the discussion in the *MODULE location_mod (threed_sphere)* documentation for how to transform vertical coordinates before localization.

&assim_tools_nml ::adaptive_localization_threshold
valid values: integer counts, or -1 to disable

Used to dynamically shrink the localization cutoff in areas of dense observations. If set to something larger than 0, first the number of other observations within 2*cutoff is computed. If it is larger than this given threshold, the cutoff is decreased proportionally so if the observations were evenly distributed in space, the number of observations within 2*revised_cutoff would now be the threshold value. The cutoff value is computed for each observation as it is assimilated, so can be different for each one.

&assim_tools_nml :: adaptive_cutoff_floor
valid values: 0.0 to infinity, or -1 to disable

If using adaptive localization (adaptive_localization_threshold set to a value greater than 0), then this value can be used to set a minimum cutoff distance below which the adaptive code will not shrink. Set to -1 to disable. Ignored if not using adaptive localization.

&assim_tools_nml :: output_localization_diagnostics
valid values: .true., .false.

If .true. and if adaptive localization is on, a single text line is printed to a file giving the original cutoff and number of observations, and the revised cutoff and new number of counts within this smaller cutoff for any observation which has nearby observations which exceed the adaptive threshold count.

&assim_tools_nml :: localization_diagnostics_file
valid values: text string

Name of the file where the adaptive localization diagnostic information is written.

&assim_tools_nml :: special_localization_obs_types
valid values: list of 1 or more text strings

The cutoff localization setting is less critical in DART than it might be in other situations since during the assimilation DART computes the covariances between observations and nearby state vector locations and that is the major factor in controlling the impact an observation has. For conventional observations fine-tuning the cutoff based on observation type is not recommended (it is possible to do more harm than good with it). But in certain special cases there may be valid reasons to want to change the localization cutoff distances drastically for certain kinds of observations. This and the following namelist items allow this.

Optional list of observation types (e.g. "RADAR_REFLECTIVITY", "AIRS_TEMPERATURE") which will use a different cutoff distance. Any observation types not listed here will use the standard cutoff distance (set by the 'cutoff' namelist value). This is only implemented for the threed_sphere location module (the one used by most geophysical models.)

&assim_tools_nml :: special_localization_cutoffs
valid values: list of 1 or more real values, 0.0 to infinity

A list of real values, the same length as the list of observation types, to be used as the cutoff value for each of the given observation types. This is only implemented for the threed_sphere location module (the one used by most geophysical models.)

Guidance regarding localization

There are a large set of options for localization. Individual cases may differ but in general the following guidelines might help. Most users use the Gaspari-Cohn covariance cutoff type. The value of the cutoff itself is the item most often changed in a sensitivity run to pick a good general value, and then left as-is for subsequent runs. Most localize in the vertical, but tend to use large values so as to not disturb vertical structures. Users do not generally use adaptive localization, unless their observations are very dense in some areas and sparse in others.

The advice for setting good values for the cutoff value is to err on the larger side - to estimate for all types of observations under all conditions what the farthest feasible impact or correlated structure size would be. The downsides of guessing too large are 1) run time is slower, and 2) there can be spurious correlations between state vector items and observations which aren't physically related and noise can creep into the assimilation results this way. The downside of guessing too small is that state vector items that should get an impact from an observation won't. This might disrupt organized features in a field and the model may take more time to recover/reconstruct the feature.

6.211.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&assim_tools_nml
   filter_kind                     = 1
   cutoff                          = 0.2
   distribute_mean                 = .false.
   sort_obs_inc                    = .false.
   spread_restoration              = .false.
   sampling_error_correction       = .false.
   adaptive_localization_threshold = -1
   adaptive_cutoff_floor           = 0.0
   output_localization_diagnostics = .false.
   localization_diagnostics_file   = "localization_diagnostics"
   print_every_nth_obs             = 0
   rectangular_quadrature          = .true.
   gaussian_likelihood_tails       = .false.
   close_obs_caching               = .true.
   adjust_obs_impact               = .false.
   obs_impact_filename             = ""
   allow_any_impact_values         = .false.
   convert_all_obs_verticals_first   = .true.
   convert_all_state_verticals_first = .false.
   special_localization_obs_types  = 'null'
   special_localization_cutoffs    = -888888.0
  /
```

Description of each namelist entry

filter_kind
type: integer

Selects the variant of filter to be used.

- 1 = EAKF (Ensemble Adjustment Kalman Filter, see Anderson 2001)
- 2 = ENKF (Ensemble Kalman Filter)
- 3 = Kernel filter
- 4 = Observation Space Particle filter
- 5 = Random draw from posterior (contact dart@ucar.edu before using)
- 6 = Deterministic draw from posterior with fixed kurtosis (ditto)
- 7 = Boxcar kernel filter
- 8 = Rank Histogram filter (see Anderson 2010)
- 9 = Particle filter (see Poterjoy 2016)

The EAKF is the most commonly used filter. Note that although the algorithm is expressed in a slightly different form, the EAKF is identical to the EnSRF (Ensemble Square Root Filter) described by Whitaker and Hamill in 2002.

The Rank Histgram filter can be more successful for highly nongaussian distributions.

Jon Poterjoy's Particle filter is included with this code release. To use, it, overwrite `assim_tools_mod.f90` with `assim_tools_mod.pf.f90` and rebuild filter.

```
$ mv assimilation_code/modules/assimilation/assim_tools_mod.pf.f90 assimilation_
↪code/modules/assimilation/assim_tools_mod.f90
```

There are additional namelist items in this version specific to the particle filter. Read the code for more details.

cutoff
type: real(r8)

Cutoff controls a distance dependent weight that modulates the impact of an observation on a state variable. The units depend both on the location module being used and on the covariance cutoff module options selected. As defined in the original paper, this is the half-width; the localization goes to 0 at 2 times this value.

distribute_mean
type: logical

If your model uses coordinates that have no options for different vertical coordinates then this setting has no effect on speed and should be .true. to use less memory. If your model has code to convert between different coordinate systems, for example Pressure, Height, Model Levels, etc, then setting this .false. will generally run much faster at assimilation time but will require more memory per MPI task. If you run out of memory, setting this to .true. may allow you to run but take longer.

sort_obs_inc
type: logical

If true, the final increments from obs_increment are sorted so that the mean increment value is as small as possible. This minimizes regression errors when non-deterministic filters or error correction algorithms are applied. HOWEVER, when using deterministic filters (filter_kind == 1 or 8) with no inflation or a combination of a determinstic filter and deterministic inflation (filter_nml:inf_deterministic = .TRUE.) sorting the increments is both unnecessary and expensive. A warning is printed to stdout and the log and the sorting is skipped.

spread_restoration
type: logical

True turns on algorithm to restore amount of spread that would be expected to be lost if underlying obs/state variable correlation were really 0.

sampling_error_correction
type: logical

If true, apply sampling error corrections to the correlation values based on the ensemble size. See Anderson 2012. This option uses special input files generated by the gen_sampling_err_table tool in the assimilation_code/programs directory. The values are generated for a specific ensemble size and most common ensemble sizes have precomputed entries in the table. There is no dependence on which model is being used, only on the number of ensemble members. The input file must exist in the directory where the filter program is executing.

adaptive_localization_threshold
type: integer

Used to reduce the impact of observations in densely observed regions. If the number of observations close to a given observation is greater than the threshold number, the cutoff radius for localization is adjusted to try to make the number of observations close to the given observation be the threshold number. This should be dependent on the location module and is tuned for a three_dimensional spherical implementation for numerical weather prediction models at present.

adaptive_cutoff_floor
type: real

If adaptive localization is enabled and if this value is greater than 0, then the adaptive cutoff distance will be set to a value no smaller than the distance specified here. This guarentees a minimum cutoff value even in regions of very dense observations.

output_localization_diagnostics
type: logical

Setting this to `.true.` will output an additional text file that contains the obs key, the obs time, the obs location, the cutoff distance and the number of other obs which are within that radius. If adaptive localization is enabled, the output also contains the updated cutoff distance and the number of other obs within that new radius. Without adaptive localization there will be a text line for each observation, so this file could get very large. With adaptive localization enabled, there will only be one line per observation where the radius is changed, so the size of the file will depend on the number of changed cutoffs.

localization_diagnostics_file
type: character(len=129)

Filename for the localization diagnostics information. This file will be opened in append mode, so new information will be written at the end of any existing data.

print_every_nth_obs
type: integer

If set to a value `N` greater than 0, the observation assimilation loop prints out a progress message every `N`th observations. This can be useful to estimate the expected run time for a large observation file, or to verify progress is being made in cases with suspected problems.

rectangular_quadrature
type: logical

Only relevant for filter type 8 and recommended to leave `.true.`.

gaussian_likelihood_tails
type: logical

Only relevant for filter type 8 and recommended to leave `.false.`.

close_obs_caching
type: logical

Should remain .TRUE. unless you are using specialized_localization_cutoffs. In that case to get accurate results, set it to .FALSE.. This also needs to be .FALSE. if you have a get_close_obs() routine in your model_mod file that uses the types/kinds of the obs to adjust the distances.

adjust_obs_impact
type: logical

If true, reads a table of observation quantities and types which should be artifically adjusted regardless of the actual correlation computed during assimilation. Setting the impact value to 0 prevents items from being adjusted by that class of observations. The input file can be constructed by the 'obs_impact_tool' program, included in this release. See the documentation for more details.

obs_impact_filename
type: character(len=256)

If adjust_obs_impact is true, the name of the file with the observation types and quantities and state quantities that should have have an additional factor applied to the correlations during assimilation.

allow_any_impact_values
type: logical

If .false., then the impact values can only be zero or one (0.0 or 1.0) - any other value will throw an error. .false. is the recommended setting.

convert_all_obs_verticals_first
type: logical

Should generally always be left .True.. For models without vertical conversion choices the setting of this item has no impact.

convert_all_state_verticals_first
type: logical

If the model has multiple choices for the vertical coordinate system during localization (e.g. pressure, height, etc) then this should be .true. if previous versions of get_state_meta_data() did a vertical conversion or if most of the state is going to be impacted by at least one observation. If only part of the state is going to be updated or if get_state_meta_data() never used to do vertical conversions, leave it .false.. The results should be the same but the run time may be impacted by doing unneeded conversions up front. For models without vertical conversion choices the setting of this item has no impact.

special_localization_obs_types
type: character(len=32), dimension(:)

Optional list of observation types (e.g. "RADAR_REFLECTIVITY", "RADIOSONDE_TEMPERATURE") which will use a different cutoff value other than the default specified by the 'cutoff' namelist. This is only implemented for the 'threed_sphere' locations module.

special_localization_cutoffs
type: real(r8), dimension(:)

Optional list of real values which must be the same length and in the same order as the observation types list given for the 'special_localization_obs_types' item. These values will set a different cutoff distance for localization based on the type of the observation currently being assimilated. Any observation type not in the list will use the default cutoff value. This is only implemented for the 'threed_sphere' locations module.

6.211.5 Other modules used

```
types_mod
utilities_mod
sort_mod
random_seq_mod
obs_sequence_mod
obs_def_mod
cov_cutoff_mod
reg_factor_mod
location_mod (model dependent choice)
ensemble_manager_mod
mpi_utilities_mod
adaptive_inflate_mod
time_manager_mod
assim_model_mod
```

6.211.6 Public interfaces

use assim_tools_mod, only :	filter_assim

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call filter_assim(ens_handle, obs_ens_handle, obs_seq, keys, ens_size, num_groups, obs_val_index, inflate, ens_mean_copy, ens_sd_copy, ens_inf_copy, ens_inf_sd_copy, obs_key_copy, obs_global_qc_copy, obs_prior_mean_start, obs_prior_mean_end, obs_prior_var_start, obs_prior_var_end, inflate_only)

```
type(ensemble_type), intent(inout)         :: ens_handle
type(ensemble_type), intent(inout)         :: obs_ens_handle
type(obs_sequence_type), intent(in)        :: obs_seq
integer, intent(in)                        :: keys(:)
integer, intent(in)                        :: ens_size
integer, intent(in)                        :: num_groups
integer, intent(in)                        :: obs_val_index
type(adaptive_inflate_type), intent(inout) :: inflate
integer, intent(in)                        :: ens_mean_copy
integer, intent(in)                        :: ens_sd_copy
integer, intent(in)                        :: ens_inf_copy
integer, intent(in)                        :: ens_inf_sd_copy
integer, intent(in)                        :: obs_key_copy
integer, intent(in)                        :: obs_global_qc_copy
integer, intent(in)                        :: obs_prior_mean_start
integer, intent(in)                        :: obs_prior_mean_end
integer, intent(in)                        :: obs_prior_var_start
integer, intent(in)                        :: obs_prior_var_end
logical, intent(in)                        :: inflate_only
```

Does assimilation and inflation for a set of observations that is identified by having integer indices listed in keys. Only the inflation is updated if inflation_only is true, otherwise the state is also updated.

ens_handle	Contains state variable ensemble data and description.
obs_ens_handle	Contains observation prior variable ensemble and description.
obs_seq	Contains the observation sequence including observed values and error variances.
keys	A list of integer indices of observations in obs_seq that are to be used at this time.
ens_size	Number of ensemble members in state and observation prior ensembles.
num_groups	Number of groups being used in assimilation.
obs_val_index	Integer index of copy in obs_seq that contains the observed value from instrument.
inflate	Contains inflation values and all information about inflation to be used.
ens_mean_copy	Index of copy containing ensemble mean in ens_handle.
ens_sd_copy	Index of copy containing ensemble standard deviation in ens_handle.
ens_inf_copy	Index of copy containing state space inflation in ens_handle.
ens_inf_sd_copy	Index of copy containing state space inflation standard deviation in ens_handle.
obs_key_copy	Index of copy containing unique key for observation in obs_ens_handle.
obs_global_qc_copy	Index of copy containing global quality control value in obs_ens_handle.
obs_prior_mean_start	Index of copy containing first group's prior mean in obs_ens_handle.
obs_prior_mean_end	Index of copy containing last group's prior mean in obs_ens_handle.
obs_prior_var_start	Index of copy containing first group's ensemble variance in obs_ens_handle.
obs_prior_var_end	Index of copy containing last group's ensemble variance in obs_ens_handle.
inflate_only	True if only inflation is to be updated, and not state.

6.211.7 Files

filename	purpose
input.nml	to read assim_tools_nml

6.211.8 References

- Anderson, J. L., 2001: An Ensemble Adjustment Kalman Filter for Data Assimilation. Mon. Wea. Rev., 129, 2884-2903. doi: 10.1175/1520-0493(2001)129<2884:AEAKFF>2.0.CO;2
- Anderson, J. L., 2003: A Local Least Squares Framework for Ensemble Filtering. Mon. Wea. Rev., 131, 634-642. doi: 10.1175/1520-0493(2003)131<0634:ALLSFF>2.0.CO;2
- Anderson, J., Collins, N., 2007: Scalable Implementations of Ensemble Filter Algorithms for Data Assimilation. Journal of Atmospheric and Oceanic Technology, 24, 1452-1463. doi: 10.1175/JTECH2049.1
- Anderson, J. L., 2010: A Non-Gaussian Ensemble Filter Update for Data Assimilation. Mon. Wea. Rev., 139, 4186-4198. doi: 10.1175/2010MWR3253.1
- Anderson, J. L., 2012:, Localization and Sampling Error Correction in Ensemble Kalman Filter Data Assimilation. Mon. Wea. Rev., 140, 2359-2371. doi: 10.1175/MWR-D-11-00013.1
- Poterjoy, J., 2016:, A localized particle filter for high-dimensional nonlinear systems. Mon. Wea. Rev. 144 59-76. doi:10.1175/MWR-D-15-0163.1

6.211.9 Private components

N/A

6.212 MODULE cov_cutoff_mod

6.212.1 Overview

Computes the weight with which an observation should impact a state variable that is separated by a given distance. The distance is in units determined by the location module being used.

6.212.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&cov_cutoff_nml
   select_localization = 1
/
```

Item	Type	Description
select_localization	integer	Selects the localization function. • 1 = Gaspari-Cohn 5th order polynomial with halfwidth c. • 2 = Boxcar with halfwidth c (goes to 0 for z_in < 2c). • 3 = Ramped Boxcar. Has value 1 for z_in < c and then reduces linearly to 0 at z_in = 2c.

6.212.3 Other modules used

```
types_mod
utilities_mod
location_mod
```

6.212.4 Public interfaces

use cov_factor_mod, only :	comp_cov_factor

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

var = comp_cov_factor(z_in, c [, obs_loc] [, obs_type] [, target_loc] [, target_kind] [, localization_override])

```
real(r8)                                    :: comp_cov_factor
real(r8), intent(in)                        :: z_in
real(r8), intent(in)                        :: c
type(location_type), optional, intent(in) :: obs_loc
integer, optional, intent(in)               :: obs_type
type(location_type), optional, intent(in) :: target_loc
integer, optional, intent(in)               :: target_kind
integer, optional, intent(in)               :: localization_override
```

Returns a weighting factor for observation and a target variable (state or observation) separated by distance z_in and with a half-width distance, c. Three options are provided and controlled by a namelist parameter. The optional argument localization_override controls the type of localization function if present. The optional arguments obs_loc, obs_type and target_loc, target_kind are not used in the default code. They are made available for users who may want to design more sophisticated localization functions.

var	Weighting factor.
z_in	The distance between observation and target.
c	Factor that describes localization function. Describes half-width of functions used here.
obs_loc	Location of the observation.
obs_type	Observation specific type.
target_loc	Location of target.
target_kind	Generic kind of target.
localization_override	Controls localization type if present. Same values as for namelist control.

6.212.5 Files

filename	purpose
input.nml	to read `cov_cutoff_nml`

6.212.6 References

1. Gaspari and Cohn, 1999, QJRMS, **125**, 723-757. (eqn. 4.10)

6.212.7 Error codes and conditions

Routine	Message	Comment
comp_cov_factor	Illegal value of "select_localization" in cov_cutoff_mod namelist	Only values 1 through 3 select a localization function.

6.212.8 Private components

N/A

6.213 MODULE obs_model_mod

6.213.1 Overview

The code in this module computes the assimilation windows, and decides if the model needs to run in order for the data to be at the appropriate time to assimilate the next available observations. It also has the code to write out the current states, advance the model (in a variety of ways) and then read back in the updated states.

6.213.2 Other modules used

```
types_mod
utilities_mod
assim_model_mod
obs_sequence_mod
obs_def_mod
time_manager_mod
ensemble_manager_mod
mpi_utilities_mod
```

6.213.3 Public interfaces

use obs_model_mod, only :	advance_state
	move_ahead

call move_ahead(ens_handle, ens_size, seq, last_key_used, window_time, key_bounds, num_obs_in_set, curr_ens_time, next_ens_time, trace_messages)

```
type(ensemble_type),     intent(in)  :: ens_handle
integer,                 intent(in)  :: ens_size
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: last_key_used
type(time_type),         intent(in)  :: window_time
integer, dimension(2),   intent(out) :: key_bounds
integer,                 intent(out) :: num_obs_in_set
type(time_type),         intent(out) :: curr_ens_time
type(time_type),         intent(out) :: next_ens_time
logical, optional,       intent(in)  :: trace_messages
```

Given an observation sequence and an ensemble, determines how to advance the model so that the next set of observations can be assimilated. Also returns the first and last keys and the number of observations to be assimilated at this time. The algorithm implemented here (one might want to have other variants) first finds the time of the next observation that has not been assimilated at a previous time. It also determines the time of the ensemble state vectors. It then uses information about the model's time stepping capabilities to determine the time to which the model can be advanced that is CLOSEST to the time of the next observation. For now, this algorithm assumes that the model's timestep is a constant. A window of width equal to the model timestep is centered around the closest model time to the next observation and all observations in this window are added to the set to be assimilated.

Previous versions of this routine also made the call which actually advanced the model before returning. This is no longer true. The routine only determines the time stepping and number of observations. The calling code must then call advance_state() if indeed the next observation to be assimilated is not within the current window. This is determined by comparing the current ensemble time with the next ensemble time. If equal no advance is needed. Otherwise, next ensemble time is the target time for advance_state().

ens_handle	Identifies the model state ensemble
ens_size	Number of ensemble members
seq	An observation sequence
last_key_used	Identifies the last observation from the sequence that has been used
window_time	Reserved for future use.
key_bounds	Returned lower and upper bound on observations to be used at this time
num_obs_in_set	Number of observations to be used at this time
curr_ens_time	Time of the ensemble data passed into this routine.
next_ens_time	Time the ensemble data should be advanced to. If equal to curr_ens_time, the model does not need to advance to assimilate the next observation.
trace_messages	Optional argument. By default, detailed time trace messages are disabled but can be turned on by passing this in as .True. . The messages will print the current window times, data time, next observation time, next window time, next data time, etc.

call advance_state(ens_handle, ens_size, target_time, async, adv_ens_command, tasks_per_model_advance)

```
type(ensemble_type), intent(inout) :: ens_handle
integer, intent(in)                :: ens_size
type(time_type), intent(in)        :: target_time
integer, intent(in)                :: async
character(len=*), intent(in)       :: adv_ens_command
integer, intent(in)                :: tasks_per_model_advance
```

Advances all ensemble size copies of an ensemble stored in ens_handle to the target_time. If async=0 this is done by repeated calls to the `adv_1step()` subroutine. If async=2, a call to the shell with the command `adv_ens_command` is used. If async=4, the filter program synchronizes with the MPI job shell script using the `block_task()` and `restart_task()` routines to suspend execution until all model advances have completed. The script can start the model advances using MPI and have it execute in parallel in this mode.

ens_handle	Structure for holding ensemble information and data
ens_size	Ensemble size.
target_time	Time to which model is to be advanced.
async	How to advance model: 0 = subroutine adv_1step 2 = shell executes adv_ens_command 4 = MPI job script advances models and syncs with filter task
adv_ens_command	Command to be issued to shell to advance model if async=2.
tasks_per_model_advance	Reserved for future use.

6.213.4 Namelist

This module does not have a namelist.

6.213.5 Files

filename	purpose
assim_model_state_ic####	a binary representation of the state vector prepended by a small header consisting of the 'advance-to' time and the 'valid-time' of the state vector. The #### represents the ensemble member number if &ensemble_manager_nml: single_restart_file_out = .true..
assim_model_state_ud####	a binary representation of the state vector prepended by a small header consisting of the 'valid-time' of the state vector. This is the 'updated' model state (after the model has advanced the state to the desired 'advance-to' time).
filter_control####	a text file containing information needed to advance the ensemble members; i.e., the ensemble member number, the input state vector file, the output state vector file - that sort of thing.

6.213.6 References

- none

6.213.7 Private components

N/A

6.214 MODULE reg_factor

6.214.1 Overview

Computes a weighting factor to reduce the impact of observations on state variables using information from groups of ensembles. Can be run using groups or using archived summary information available from previous group filter experiments.

6.214.2 Other modules used

```
types_mod
utilities_mod
time_manager_mod
```

6.214.3 Public interfaces

use reg_factor_mod, only :	comp_reg_factor

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

var = comp_reg_factor(num_groups, regress, obs_index, state_index [, obs_state_ind] [, obs_state_max])

```
real(r8)                                    ::  comp_reg_factor
integer,                          intent(in) ::  num_groups
real(r8), dimension(num_groups), intent(in) ::  regress
integer,                          intent(in) ::  obs_index
integer,                          intent(in) ::  state_index
integer, optional,                intent(in) ::  obs_state_ind
integer, optional,                intent(in) ::  obs_state_max
```

Returns a weighting factor given regression factors from each group of a group filter or retrieves a factor generated by previous group filter runs from a file.

num_groups	Number of groups. Set to 1 when using information from previously run group filter from file.
regress	Regression factor from each group for a given state variable and observation variable pair.
obs_index	Integer index of the observation being processed. Not used in current implementation .
state_index	Integer index of state variable being processed. Not used in current implementation.
obs_state_ind	Index into file generated for Bgrid model which could be duplicated in other large models.
obs_state_max	Maximum number of observation state variable pairs with non-zero impacts for a given model and observation sequence. Used for generating Bgrid statistic files.

6.214.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"
/
```

Item	Type	Description
select_regression	integer	Selects the method for computing regression factor. • 1 = compute using sampling theory for any ensemble size. • 2 = low order model format. Works from archived time mean or time median regression files generated by low-order models like Lorenz-96. • 3 = selects bgrid archived file. This is not currently supported in released versions.
input_reg_file	character(len=129)	File name from which statistics are to be read for select_regression = 3.
save_reg_diagnostics	logical	True if regression diagnostics should be computed.
reg_diagnostics_file	character(len=129)	File name to which to write diagnostics.

6.214.5 Files

- (optional) input regression file from namelist variable input_reg_file.
- reg_factor_mod.nml in input.nml

filename	purpose
from `input.nml`&reg_factor_mod:input_reg_file	file of regression coefficients

6.214.6 References

- none

6.214.7 Private components

N/A

6.215 MODULE adaptive_inflate_mod

6.215.1 Overview

This module implements a variety of hierarchical Bayesian adaptive inflation algorithms for use with ensemble filters. It can provide constant valued inflation in state or observation space, consistent with previous DART releases. It can provide spatially-constant, time-varying adaptive inflation. It can provide spatially-varying, time-varying adaptive inflation and it can provide temporally-varying observation space inflation. And finally, it can provide adaptive damped inflation, which decreases inflation through time when observation density varies. Diagnostic output and restart files are available. Several papers on the NCAR DART website document the algorithms in detail. The `DART/tutorial/section12` chapter has more information.

Details on controlling the inflation options are contained in the documentation for the filter. The filter_nml controls what inflation options are used.

Inflation flavor 3 (spatially-constant state space) reads and writes a restart file that is the full size of the state vector, however it takes the first value in the array and replicates that throughout the array. This allows one to switch between flavors 2 and 3. Going from inflation flavor 3 to 2 the initial value for all items in the state vector will be a constant value and will then start to adapt. Going from inflation flavor 2 to 3 whatever value is in the array at index 1 will be replicated and used for the entire rest of the state vector items.

6.215.2 Other modules used

```
types_mod
utilities_mod
random_seq_mod
time_manager_mod
ensemble_manager_mod
```

6.215.3 Public interfaces

use adaptive_inflate_mod, only :	update_inflation
	adaptive_inflate_end
	inflate_ens
	output_inflate_diagnostics
	do_obs_inflate
	do_single_ss_inflate
	do_varying_ss_inflate
	adaptive_inflate_init
	adaptive_inflate_type
	get_inflate
	set_inflate
	set_sd
	set_sd
	deterministic_inflate

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call update_inflation(inflate_handle, inflate, inflate_sd, prior_mean, prior_var, obs, obs_var, gamma)

```
type(adaptive_inflate_type), intent(in)    :: inflate_handle
real(r8),                    intent(inout) :: inflate
real(r8),                    intent(inout) :: inflate_sd
real(r8),                    intent(in)    :: prior_mean
real(r8),                    intent(in)    :: prior_var
real(r8),                    intent(in)    :: obs
real(r8),                    intent(in)    :: obs_var
real(r8),                    intent(in)    :: gamma
```

Updates the mean and standard deviation of an inflation distribution given the prior values, the prior observation ensemble mean and variance, and the observation and its error variance. The factor gamma is the expected impact (0 to 1) of the state variable corresponding to the inflation on the observation and is the product of the ensemble correlation plus an additional localization factor or group regression factors.

inflate_handle	Handle to object that describes the inflation type and values.
inflate	Prior mean value of the inflation distribution.
inflate_sd	Prior standard deviation of the inflation distribution.
prior_mean	Mean of the prior observation ensemble.
prior_var	Variance of the prior observation ensemble.
obs	The observed value.
obs_var	Observational error variance.
gamma	Expected impact factor, product of correlation, localization, regression factor.

call adaptive_inflate_end(inflate_handle, ens_handle, ss_inflate_index, ss_inflate_sd_index)

```
type(adaptive_inflate_type), intent(in)    :: inflate_handle
type(ensemble_type),         intent(inout) :: ens_handle
integer,                     intent(in)    :: ss_inflate_index
integer,                     intent(in)    :: ss_inflate_sd_index
```

Outputs the values of inflation to restart files using the ensemble_manager for state space inflation and file output for observation space inflation. Releases allocated storage in inflate_handle.

inflate_handle	Handle for the details of the inflation being performed.
ens_handle	Handle for ensemble storage that holds values of state space inflation.
ss_inflate_index	Index in ensemble storage copies for state space inflation.
ss_inflate_sd_index	Index in ensemble storage copies for state space inflation standard deviation.

call inflate_ens(inflate_handle, ens,mean, inflate [,var_in])

```
type(adaptive_inflate_type),               intent(in)  :: inflate_handle
real(r8),                    dimension(:), intent(out) :: ens
real(r8),                                  intent(in)  :: mean
real(r8),                                  intent(in)  :: inflate
real(r8),                    optional,     intent(in)  :: var_in
```

Given an ensemble, its mean and the covarance inflation factor, inflates the ensemble.

inflate_handle	Handle for the details of the inflation being performed.
ens	Values for the ensemble to be inflated
mean	The mean of the ensemble.
inflate	The covariance inflation factor.
var_in	The variance of the ensemble.

call output_inflate_diagnostics(inflate_handle, time)

```
type(adaptive_inflate_type), intent(in) :: inflate_handle
type(time_type),             intent(in) :: time
```

Outputs diagnostic record of inflation for the observation space of spatially constant state space inflation. Spatially varying state space diagnostics are in the Posterior and Prior Diagnostic netcdf files and are written with calls from filter.f90.

inflate_handle	Handle for the details of the inflation being performed.
time	Time of this diagnostic info.

var = do_obs_inflate(inflate_handle)

```
logical,               intent(out) :: do_obs_inflate
adaptive_inflate_type, intent(in)  :: inflate_handle
```

Returns true if observation space inflation is being done by this handle.

do_obs_inflate	True if obs space inflation is being done by this handle.
inflate_handle	Handle to inflation details.

var = do_varying_ss_inflate(inflate_handle)

```
logical,               intent(out) :: do_varying_ss_inflate
adaptive_inflate_type, intent(in)  :: inflate_handle
```

Returns true if spatially varying state space inflation is being done by this handle.

do_varying_ss_inflate	True if spatially varying state space inflation is being done by this handle.
inflate_handle	Handle to inflation details.

var = do_single_ss_inflate(inflate_handle)

```
logical,               intent(out) :: do_single_ss_inflate
adaptive_inflate_type, intent(in)  :: inflate_handle
```

Returns true if spatially fixed state space inflation is being done by this handle.

do_single_ss_inflate	True if spatially fixed state space inflation is being done by this handle.
inflate_handle	Handle to inflation details.

call adaptive_inflate_init(inflate_handle, inf_flavor, mean_from_restart, sd_from_restart, output_restart, deterministic, in_file_name, out_file_name, diag_file_name, inf_initial, sd_initial, inf_lower_bound, inf_upper_bound, sd_lower_bound, ens_handle, ss_inflate_index, ss_inflate_sd_index, label)

```
type(adaptive_inflate_type), intent(inout) :: inflate_handle
integer, intent(in)                        :: inf_flavor
logical, intent(in)                        :: mean_from_restart
logical, intent(in)                        :: sd_from_restart
logical, intent(in)                        :: output_restart
logical, intent(in)                        :: deterministic
character(len=*), intent(in)               :: in_file_name
character(len=*), intent(in)               :: out_file_name
character(len=*), intent(in)               :: diag_file_name
real(r8), intent(in)                       :: inf_initial
real(r8), intent(in)                       :: sd_initial
real(r8), intent(in)                       :: inf_lower_bound
real(r8), intent(in)                       :: inf_upper_bound
real(r8), intent(in)                       :: sd_lower_bound
type(ensemble_type), intent(inout)         :: ens_handle
integer, intent(in)                        :: ss_inflate_index
integer, intent(in)                        :: ss_inflate_sd_index
character(len=*), intent(in)               :: label
```

Initializes a descriptor of an inflation object.

inflate_handle	Handle for the inflation descriptor being initialized.
inf_flavor	Type of inflation, 1=obs_inflate, 2=varying_ss_inflate, 3=single_ss_inflate.
mean_from_restart	True if inflation mean values to be read from restart file.
sd_from_restart	True if inflation standard deviation values to be read from restart file.
output_restart	True if an inflation restart file is to be output.
deterministic	True if deterministic inflation is to be done.
in_file_name	File name from which to read restart.
out_file_name	File name to which to write restart.
diag_file_name	File name to which to write diagnostic output; obs space inflation only .
inf_initial	Initial value of inflation for start_from_restart=.false.
sd_initial	Initial value of inflation standard deviation for start_from_restart=.false.
inf_lower_bound	Lower bound on inflation value.
inf_upper_bound	Upper bound on inflation value.
sd_lower_bound	Lower bound on inflation standard deviation.
ens_handle	Ensemble handle with storage for state space inflation.
ss_inflate_index	Index op copy in ensemble storage for inflation value.
ss_inflate_sd_index	Index of copy in ensemble storage for inflation standard deviation.
label	Character label to be used in diagnostic output (e.g. 'Prior', 'Posterior').

var = get_sd(inflate_handle)

```
real(r8), intent(out)                   :: get_sd
type(adaptive_inflate_type), intent(in) :: inflate_handle
```

Returns value of observation space inflation standard deviation.

get_sd	Returns the value of observation space inflation.
inflate_handle	Handle for inflation descriptor.

var = get_inflate(inflate_handle)

```
real(r8), intent(out)                   :: get_inflate
type(adaptive_inflate_type), intent(in) :: inflate_handle
```

Returns value of observation space inflation.

get_inflate	Returns the value of observation space inflation.
inflate_handle	Handle for inflation descriptor.

call set_inflate(inflate_handle, inflate)

```
type(adaptive_inflate_type), intent(inout) :: inflate_handle
real(r8), intent(in)                       :: inflate
```

Set the value of observation space inflation.

inflate_handle	Handle for inflation descriptor.
inflate	Set observation space inflation to this value.

call set_sd(inflate_handle, sd)

```
type(adaptive_inflate_type), intent(inout) :: inflate_handle
real(r8), intent(in)                       :: sd
```

Set the value of observation space inflation standard deviation.

inflate_handle	Handle for inflation descriptor.
sd	Set observation space inflation standard deviation to this value.

var = deterministic_inflate(inflate_handle)

```
logical, intent(out)                    :: deterministic_inflate
type(adaptive_inflate_type), intent(in) :: inflate_handle
```

Returns true if deterministic inflation is being done.

deterministic_inflate	Returns true if deterministic inflation is being done.
inflate_handle	Handle for inflation descriptor.

```
type adaptive_inflate_type
   private
   integer :: inflation_flavor
   integer :: obs_diag_unit
   logical :: start_from_restart
   logical :: output_restart
   logical :: deterministic
   character(len = 129) :: in_file_name
   character(len = 129) :: out_file_name
   character(len = 129) :: diag_file_name
   real(r8) :: inflate
   real(r8) :: sd
   real(r8) :: sd_lower_bound
   real(r8) :: inf_lower_bound
   real(r8) :: inf_upper_bound
   type(random_seq_type) :: ran_seq
end type adaptive_inflate_type
```

Provides a handle for a descriptor of inflation. Includes type of inflation, values controlling it, input and output file names, an output file descriptor for observation space inflation diagnotics, and a random sequence for doing reproducible non-determinstic inflation. There are 2 instances of this type, one for Prior and one for Posterior inflation.

Component	Description
inflation_flavor	Type of inflation; 0=none, 1=obs. space, 2=spatially varying, 3=spatially-fixed.
obs_diag_unit	Unit descriptor for output diagnostic file.
start_from_restart	True if initial inflation to be read from file.
output_restart	True if final inflation values to be written to file.
deterministic	True if inflation is to be done be deterministic algorithm.
in_file_name	File name containing restart.
out_file_name	File to contain output restart.
diag_file_name	File to hold observation space diagnostics.
inflate	Initial value of inflation for all types; current value for obs. space.
sd	Initial value of sd for all types; current value for obs. space.
sd_lower_bound	Don't allow standard deviation to get smaller than this.
inf_lower_bound	Don't let inflation get smaller than this.
inf_upper_bound	Don't let inflation get larger than this.
ran_seq	Handle to random number sequence to allow reproducing non-deterministic inflate.

6.215.4 Namelist

The adaptive_inflate module no longer has a namelist. Control has been moved to &filter_nml in filter.

6.215.5 Files

Three files are opened from this module, but all names are passed in from the filter_nml now, and there are 2 values for each name: one for the prior and one for the posterior inflation.

- inf_in_file_name Mean and standard deviation values read in restart file format.
- inf_out_file_name Mean and standard deviation values written in restart file format.
- inf_diag_file_name Contains diagnostic history of inflation values for obs space and spatially-fixed state space inflation. Diagnostics for spatially-varying state space inflation are extra fields on the Posterior and Prior diagnostic netcdf files created in filter.f90.

6.215.6 References

- Anderson, J. L., 2007: An adaptive covariance inflation error correction algorithm for ensemble filters. Tellus A, 59, 210-224. doi: 10.1111/j.1600-0870.2006.00216.x
- Anderson, J. L., 2009: Spatially and temporally varying adaptive covariance inflation for ensemble filters. Tellus A, 61, 72-83. doi: 10.1111/j.1600-0870.2008.00361.x

6.215.7 Private components

no discussion

6.216 MODULE quality_control_mod

6.216.1 Overview

Routines in this module deal with two different types of quality control (QC) related functions. The first is to support interpretation of the *incoming* data quality, to reject observations at assimilation time which are marked as poor quality. The second is to document how DART disposed of each observation; whether it was successfully assimilated or rejected, and if rejected, for which reason.

6.216.2 Usage

Incoming data quality control

DART currently supports a single incoming quality control scheme compatible with NCEP usage. Lower values are considered better and higher values are considered poorer. A single namelist item, `input_qc_threshold` sets the boundary between accepted and rejected observations. Values *larger* than this value are rejected; values equal to or lower are accepted. Note that observations could be subsequently rejected for other reasons, including failing the outlier threshold test or all observations of this type being excluded by namelist control. See the obs_kind_mod namelist documentation for more details on how to enable or disable assimilation by observation type at runtime.

The incoming quality control value is set when an observation sequence file is created. If the data provider user a different scheme the values must be translated into NCEP-consistent values. Generally we use the value 3 for most runs.

Observations can also be rejected by the assimilation if the observation value is too far from the mean of the ensemble of expected values (the forward operator results). This is controlled by the `outlier_threshold` namelist item.

Specifically, the outlier test computes the difference between the observation value and the prior ensemble mean. It then computes a standard deviation by taking the square root of the sum of the observation error variance and the prior ensemble variance for the observation. If the difference between the ensemble mean and the observation value is more than the specified number of standard deviations then the observation is not used. This can be an effective way to discard clearly erroneous observation values. A commonly used value is 3. To assimilate all possible observations, a value of -1 can be used, but may result in 'chasing bad observations' and _prevents_ the calculation of the number of observations that are grossly inconsistent with the ensemble; a useful indicator of *filter divergence*.

There is an option to add code to this module to specialize the outlier threshold routine. For example, it is possible to allow all observations of one type to be assimilated regardless of the outlier value, and enforce the outlier threshold only on other types of observations. To enable this capability requires two actions: setting the `enable_special_outlier_code` namelist to `.TRUE.`, and adding your custom code to the `failed_outlier()` subroutine in this module.

DART outgoing quality control

As DART assimilates each observation it adds a *DART Quality Control* value to the output observation sequence (frequently written to a file named `obs_seq.final`). This flag indicates how the observation was used during the assimilation. The flag is a numeric value with the following meanings:

0:	Observation was assimilated successfully
1:	Observation was evaluated only so not used in the assimilation
2:	The observation was used but one or more of the posterior forward observation operators failed
3:	The observation was evaluated only so not used AND one or more of the posterior forward observation operators failed
4:	One or more prior forward observation operators failed so the observation was not used
5:	The observation was not used because it was not selected in the namelist to be assimilated or evaluated
6:	The incoming quality control value was larger than the threshold so the observation was not used
7:	Outlier threshold test failed (as described above)
8:	The location conversion to the vertical localization unit failed so the observation was not used

6.216.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&quality_control_nml
   input_qc_threshold           = 3
   outlier_threshold            = -1
   enable_special_outlier_code = .false.
  /
```

Items in this namelist control whether an observation is assimilated or not.

Item	Type	Description
input_qc_threshold	real(r8)	Numeric value indicating whether this observation is considered "good quality" and should be assimilated, or whether it is suspect because of previous quality control processes. This value would have been set when the observation was created and added to the observation sequence file. Observations with an incoming QC value larger than this threshold are rejected and not assimilated.
outlier threshold	real(r8)	This numeric value defines the maximum number of standard deviations an observation value can be away from the ensemble forward operator mean and still be assimilated. Setting it to the value -1 disables this check.
enable_special_outlier_code	logical	Setting this value to .TRUE. will call a subroutine `failed_outlier()` instead of using the default action. The user can then customize the tests in this subroutine, for example to accept all observations of a particular type, or use different numerical thresholds for different observation types or locations.

6.216.4 Discussion

Small ensemble spread

If an ensemble is spun up from a single state the ensemble spread may be very small to begin and many observations may be rejected by the `outlier_threshold`. But as the ensemble spread increases the assimilation should be able to assimilate more and more observations as the model trajectory becomes consistent with those observations.

6.216.5 Other modules used

```
types_mod
utilities_mod
random_seq_mod
```

6.216.6 Public interfaces

use quality_control_mod, only :	initialize_qc
	input_qc_ok
	get_dart_qc
	check_outlier_threshold
	good_dart_qc
	set_input_qc
	dart_flags

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call check_outlier_threshold(obs_prior_mean, obs_prior_var, obs_val, obs_err_var, & obs_seq, this_obs_key, dart_qc)

```
real(r8),                intent(in)    :: obs_prior_mean !>  prior observation mean
real(r8),                intent(in)    :: obs_prior_var  !>  prior observation variance
real(r8),                intent(in)    :: obs_val        !>  observation value
real(r8),                intent(in)    :: obs_err_var    !>  observation error variance
type(obs_sequence_type), intent(in)    :: obs_seq        !>  observation sequence
integer,                 intent(in)    :: this_obs_key   !>  index for this observation
integer,                 intent(inout) :: dart_qc        !>  possibly modified DART QC
```

Computes whether this observation failed the outlier threshold test and if so, updates the DART QC.

var = input_qc_ok(input_qc, qc_to_use)

```
real(r8), intent(in)  :: input_qc    !> incoming QC data value
integer,  intent(out) :: qc_to_use   !> resulting DART QC
logical               :: input_qc_ok !> true if input_qc is good
```

Returns true if the input qc indicates this observation is good to use.

```
! Dart quality control variables
integer, parameter :: DARTQC_ASSIM_GOOD_FOP        = 0
integer, parameter :: DARTQC_EVAL_GOOD_FOP         = 1
integer, parameter :: DARTQC_ASSIM_FAILED_POST_FOP = 2
integer, parameter :: DARTQC_EVAL_FAILED_POST_FOP  = 3
integer, parameter :: DARTQC_FAILED_FOP            = 4
integer, parameter :: DARTQC_NOT_IN_NAMELIST       = 5
integer, parameter :: DARTQC_BAD_INCOMING_QC       = 6
integer, parameter :: DARTQC_FAILED_OUTLIER_TEST   = 7
integer, parameter :: DARTQC_FAILED_VERT_CONVERT   = 8
!!integer, parameter :: DARTQC_OUTSIDE_DOMAIN        = 9  ! we have no way (yet) for the␣
↪model_mod to signal this
```

These are public constants for use in other parts of the DART code.

6.216.7 Files

filename	purpose
input.nml	to read the quality_control_mod namelist

6.216.8 References

1. none

6.216.9 Error codes and conditions

Routine	Message	Comment
routine name	output string	description or comment

6.216.10 Future plans

Should support different incoming data QC schemes.

It would be nice to have a different DART QC flag for observations which fail the forward operator because they are simply outside the model domain. The diagnosic routines may indicate a large number of failed forward operators which make it confusing to identify observations where the forward operator should have been computed and can skew the statistics. Unfortunately, this requires adding an additional requirement on the model-dependent *model_mod.f90* code in the `model_interpolate()` routine. The current interface defines a return status code of 0 as success, any

positive value as failure, and negative numbers are reserved for other uses. To identify obs outside the domain would require reserving another value that the interpolate routine could return.

At this time the best suggestion is to cull out-of-domain obs from the input observation sequence file by a preprocessing program before assimilation.

6.216.11 Private components

N/A

6.217 MODULE filter_mod

6.217.1 Overview

Main module for driving ensemble filter assimilations. Used by filter.f90, perfect_model_obs.f90, model_mod_check.f90, and a variety of test programs. See the *PROGRAM filter* for a general description of filter capabilities and controls.

`filter_mod` is a Fortran 90 module, and provides a large number of options for controlling execution behavior and parameter configuration that are driven from its namelist. See the namelist section below for more details. The number of assimilation steps to be done is controlled by the input observation sequence and by the time-stepping capabilities of the model being used in the assimilation.

See *Welcome to the Data Assimilation Research Testbed* for more documentation, including a discussion of the capabilities of the assimilation system, a diagram of the entire execution cycle, the options and features.

6.217.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&filter_nml
   single_file_in               = .false.,
   input_state_files            = '',
   input_state_file_list        = '',
   init_time_days               = 0,
   init_time_seconds            = 0,
   perturb_from_single_instance = .false.,
   perturbation_amplitude       = 0.2,

   stages_to_write              = 'output'

   single_file_out              = .false.,
   output_state_files           = '',
   output_state_file_list       = '',
   output_interval              = 1,
   output_members               = .true.,
   num_output_state_members     = 0,
   output_mean                  = .true.,
   output_sd                    = .true.,
```

(continues on next page)

(continued from previous page)

```
   write_all_stages_at_end      = .false.,
   compute_posterior            = .true.

   ens_size                     = 20,
   num_groups                   = 1,
   distributed_state            = .true.,

   async                        = 0,
   adv_ens_command              = "./advance_model.csh",
   tasks_per_model_advance      = 1,

   obs_sequence_in_name         = "obs_seq.out",
   obs_sequence_out_name        = "obs_seq.final",
   num_output_obs_members       = 0,
   first_obs_days               = -1,
   first_obs_seconds            = -1,
   last_obs_days                = -1,
   last_obs_seconds             = -1,
   obs_window_days              = -1,
   obs_window_seconds           = -1,

   inf_flavor                   = 0,                       0,
   inf_initial_from_restart     = .false.,                 .false.,
   inf_sd_initial_from_restart  = .false.,                 .false.,
   inf_deterministic            = .true.,                  .true.,
   inf_initial                  = 1.0,                     1.0,
   inf_lower_bound              = 1.0,                     1.0,
   inf_upper_bound              = 1000000.0,               1000000.0,
   inf_damping                  = 1.0,                     1.0,
   inf_sd_initial               = 0.0,                     0.0,
   inf_sd_lower_bound           = 0.0,                     0.0,
   inf_sd_max_change            = 1.05,                    1.05,

   trace_execution              = .false.,
   output_timestamps            = .false.,
   output_forward_op_errors     = .false.,
   write_obs_every_cycle        = .false.,
   allow_missing_clm            = .false.,
   silence                      = .false.,
/
```

Particular options to be aware of are: ens_size, cutoff (localization radius), inflation flavor, outlier_threshold, input and output state filenames, obs_sequence_in_name, horiz_dist_only, and the binary or ascii controls for observation sequence file formats. Some of these important items are located in other namelists, but all are in the same *input.nml* file.

The inflation control variables are all dimensioned 2, the first value controls the prior inflation and the second controls the posterior inflation.

Item	Type	Description
single_file_in	logical	`.true.` means all ensemble members are read from a single netCDF file (which can only be used with subroutine-callable models). `.false.` means each member is in a separate file.
input_state_files	character(len=256), dimension(MAXFILES)	A list of the NetCDF files to open to read the state vectors. Models using multiple domains must put the domain and ensemble numbers in the file names. The order and format of those is to be determined. NOT SUPPORTED as of March, 2017.
input_state_file_list	character(len=256), dimension(MAXFILES)	A list of files, one per domain. Each file must be a text file containing the names of the NetCDF files to open, one per ensemble member, one per line.
init_time_days	integer	If negative, use the initial days read from the state data restart file. If positive, override the initial days read from state data restart files. Days since 1 Jan 1601.
init_time_seconds	integer	If negative use the initial seconds read from the state data restart file. If positive, override the initial seconds read from state data restart files. Seconds since midnight.
perturb_from_single_instance	logical	`.true.` means perturb a single state vector from one restart file to create an ensemble. This may be done by model_mod, if model_mod provides subroutine pert_model_copies. `.false.` means an an ensemble-sized set of restart files is provided.
perturbation_amplitude	real(r8)	Standard deviation for the noise model used when generating ensemble members. This value is available to the model_mod for use in the required interface `pert_model_copies`. For more, see *pert_model_copies* below. Ignored if `perturb_from_single_instance = .false.`
stages_to_write	character(len=10), dimension(6)	Controls diagnostic and restart output. Valid values are: 'input', 'forecast', 'preassim', 'postassim', 'analysis', 'output', and 'null'. Input is case-insensitive.

continues on next page

Table 10 – continued from previous page

Item	Type	Description
single_file_out	logical	`.true.` means all ensemble members are written to a single netCDF file. `.false.` means each member is output in a separate file. Only subroutine-callable models may write a single output file.
output_state_files	character(len=256), dimension(MAXFILES)	A list of the netCDF files to open for writing updated state vectors. Not supported when using multiple domains.
output_state_file_list	character(len=256),	A list of files, one per domain. Each file must be a text file containing the names of the netCDF files to open, one per ensemble member, one per line.
output_interval	integer	Output state and observation diagnostics every 'N'th assimilation time, N is output_interval.
output_members	logical	`.true.` means output the ensemble members in any stage that is enabled.
num_output_state_members	integer	Number of ensemble members to be included in the state diagnostic output for stages 'forecast', 'preassim', 'postassim' and 'analysis'. `output_members` must be `.true.`
output_mean	logical	`.true.` means output the ensemble mean in any stage that is enabled.
output_sd	logical	`.true.` means output the ensemble standard deviation (spread) in any stage that is enabled.
write_all_stages_at_end	logical	For most cases this should be `.false.`; data will be output as it is generated for the 'forecast', 'preassim', 'postassim', and 'analysis' diagnostics, and then restart data will be output at the end. However, if I/O time dominates the runtime, setting this to `.true.` will store the data and it can all be written in parallel at the end of the execution. This will require slightly more memory at runtime, but can lower the job cost significantly in some cases.

continues on next page

Table 10 – continued from previous page

Item	Type	Description
compute_posterior	logical	If `.false.`, skip computing posterior forward operators and do not write posterior values in the obs_seq.final file. Those are rarely worth examining. Saves time and memory. Posterior inflation is not possible. For backwards compatibility the default `.true.`
ens_size	integer	Size of ensemble.
num_groups	integer	Number of groups for hierarchical filter. It should evenly divide ens_size.
distributed_state	logical	`.true.` means the ensemble data is distributed across all tasks as it is read in, so a single task never has to have enough memory to store the data for an ensemble member. Large models should always set this to `.true.`, while for small models it may be faster to set this to `.false.` This is different from `&assim_tools_mod :: distributed_mean`.
async	integer	Controls method for advancing model: • 0 is subroutine call • 2 is shell command • 4 is mpi-job script Ignored if filter is not controlling the model advance, e.g. in CESM, WRF, etc
adv_ens_command	character(len=256)	Command sent to shell if async is 2.
tasks_per_model_advance	integer	Number of tasks to assign to each ensemble member advance.
obs_sequence_in_name	character(len=256)	File name from which to read an observation sequence.
obs_sequence_out_name	character(len=256)	File name to which to write output observation sequence.
num_output_obs_members	integer	Number of ensemble members to be included in the output observation sequence file.
first_obs_days	integer	If negative, don't use. If non-negative, ignore all observations before this time.
first_obs_seconds	integer	If negative, don't use. If non-negative, ignore all observations before this time.

continues on next page

Table 10 – continued from previous page

Item	Type	Description
last_obs_days	integer	If negative, don't use. If non-negative, ignore all observations after this time.
last_obs_seconds	integer	If negative, don't use. If non-negative, ignore all observations after this time.
obs_window_days	integer	Assimilation window days; defaults to model timestep size.
obs_window_seconds	integer	Assimilation window seconds; defaults to model timestep size.
All variables named inf_* are arrays of length 2. The first element controls the prior, the second element controls the posterior inflation. See *PROGRAM filter* for a discussion of inflation and effective strategies.		
inf_flavor	character(len=32), dimension(2)	Inflation flavor [prior, posterior] see *Inflation Options* below.
inf_initial_from_restart	logical, dimension(2)	If `.true.`, get initial mean values for inflation from inflation file. If `.false.` , use the corresponding namelist value `inf_initial`.
inf_sd_initial_from_restart	logical, dimension(2)	If `.true.`, get initial standard deviation values for inflation from file. If `.false.` , use the corresponding namelist value `inf_sd_initial`.
inf_deterministic	logical, dimension(2)	`.true.` means deterministic inflation, `.false.` means stochastic.
inf_initial	real(r8), dimension(2)	Initial value of inflation if not read from restart file.
inf_lower_bound	real(r8), dimension(2)	Lower bound for inflation value.
inf_upper_bound	real(r8), dimension(2)	Upper bound for inflation value.
inf_damping	real(r8), dimension(2)	Damping factor for inflation mean values. The difference between the current inflation value and 1.0 is multiplied by this factor and added to 1.0 to provide the next inflation mean. The value should be between 0.0 and 1.0. Setting a value of 0.0 is full damping, which in fact turns off all inflation by fixing the inflation value at 1.0. A value of 1.0 turns inflation *damping* off leaving the original inflation value unchanged.
inf_sd_initial	real(r8) dimension(2)	Initial value of inflation standard deviation if not read from restart file. If 0, do not update the inflation values, so they are time-constant. If positive, the inflation values will adapt through time.
inf_sd_lower_bound	real(r8), dimension(2)	Lower bound for inflation standard deviation. If using a negative value for inf_sd_initial this should also be negative to preserve the setting.

continues on next page

Table 10 – continued from previous page

Item	Type	Description
inf_sd_max_change	real(r8), dimension(2)	For `inf_flavor` 5 (enhanced inflation), controls the maximum change of the inflation standard deviation when adapting for the next assimilation cycle. The value should be between 1.0 and 2.0. 1.0 prevents any changes, while 2.0 allows 100% change. For the enhanced inflation option, if the standard deviation initial value is equal to the standard deviation lower bound, the standard deviation will not adapt in time. See *PROGRAM filter* for a discussion of how the standard deviation adapts based on different types of inflation.
trace_execution	logical	`.true.` means output very detailed messages about what routines are being called in the main filter loop. Useful if a job hangs or otherwise doesn't execute as expected.
output_timestamps	logical	`.true.` means write timing information to the log before and after the model advance and the observation assimilation phases.
output_forward_op_errors	logical	`.true.` means output errors from forward observation operators. This is the 'istatus' error return code from the model_interpolate routine. An ascii text file prior_forward_op_errors and/or post_forward_op_errors will be created in the current directory. For each ensemble member which returns a non-zero return code, a line will be written to this file. Each line will have three values listed: the observation number, the ensemble member number, and the istatus return code. Be cautious when turning this option on. The number of lines in this file can be up to the number of observations times the number of ensemble members times the number of assimilation cycles performed. This option is generally most useful when run with a small observation sequence file and a small number of ensemble members to diagnose forward operator problems.

continues on next page

Table 10 – continued from previous page

Item	Type	Description
write_obs_every_cycle	logical	For debug use; this option can significantly slow the execution of filter. True means to write the entire output observation sequence diagnostic file each time through the main filter loop even though only observations with times up to and including the current model time will have been assimilated. Unassimilated observations have the value -888888.0 (the DART "missing value"). If filter crashes before finishing it may help to see the forward operator values of observations that have been assimilated so far.
allow_missing_clm	logical	Some models are allowed to have MISSING_R8 values in the DART state. If `.true.` extra caution is taken (at considerable computational cost) to allow missing values in the DART state. So far, only CLM requires this to be `.true.`
silence	logical	`.true.` means output almost no runtime messages. Not recommended for general use, but can speed long runs of the lower order models if the execution time becomes dominated by the volume of output.

6.217.3 Inflation Options

The value for the `inf_flavor` is a character string. For backwards compatiblity (it was an integer code), the specification of the integer is still supported. Inflation values (for flavors other than 0) will be time-varying only if `inf_sd_initial` > 0.

inflation option	description
0 ‘0’ ‘NO_INFLATION’	no inflation
2 ‘2’ ‘VARYING_SS_INFLATION’	spatially-varying state-space (gaussian)
3 ‘3’ ‘SINGLE_SS_INFLATION’	spatially-fixed state-space (gaussian)
4 ‘4’ ‘RELAXATION_TO_PRIOR_SPREAD’ ‘RTPS	Relaxation To Prior Spread (Posterior inflation only)
5 ‘5’ ‘ENHANCED_SS_INFLATION’	Enhanced spatially-varying state-space (inverse gamma). Refer to `inf_sd_initial` for how to set the time evolution options.

Create an initial ensemble from a single file

If the default `pert_model_copies` routine is used, random noise values drawn from a gaussian distribution with the standard deviation specified by `perturbation_amplitude` will be added to the data in a single initial ensemble member to generate the rest of the members. This option is more frequently used in the low order models and less frequently used in large models. This is in part due to the different scales of real geophysical variable values, and the resulting inconsistencies between related field values. A more successful initial condition generation strategy is to generate climatological distributions from long model runs which have internally consistent structures and values and then use observations with a ‘spin-up’ period of assimilation to shape the initial states into a set of members with enough spread and which match the current set of observations. Each *model_mod* is required to provide a *pert_model_copies* routine which can be used to either pass-through to the default routine or can be customized for that specific model.

6.217.4 Modules used

```
types_mod
obs_sequence_mod
obs_def_mod
obs_def_utilities_mod
time_manager_mod
utilities_mod
assim_model_mod
assim_tools_mod
obs_model_mod
ensemble_manager_mod
adaptive_inflate_mod
mpi_utilities_mod
smoother_mod
random_seq_mod
state_vector_io_mod
io_filenames_mod
forward_operator_mod
quality_control_mod
```

6.217.5 Files

See the filter overview for the list of files.

6.217.6 Error codes and conditions

Routine	Message	Comment
filter_main	ens_size in namelist is ###: Must be > 1	Ensemble size must be at least 2.
filter_main	inf_flavor= ### Must be 0, 2, 3.	Observation Inflation is no longer supported (i.e flavor 1).
filter_main	Posterior observation space inflation (type 1) not supported.	Posterior observation space inflation doesn't work.
filter_main	Number of processes > model size.	Number of processes can't exceed model size for now.
filter_generate_copy_metadata	output metadata in filter needs state ensemble size < 10000, not ###.	Only up to 10000 ensemble members with state output for now.
filter_generate_copy_metadata	output metadata in filter needs obs ensemble size < 10000, not ###.	Only up to 10000 ensemble members with obs space output for now.
filter_setup_obs_sequence	input obs_seq file has ### qc fields; must be < 2.	Only 0 or 1 qc fields in input obs sequence for now.
get_obs_copy_index	Did not find observation copy with metadata observation.	Only 0 or 1 qc fields in input obs sequence for now.

6.218 MODULE location_mod

6.218.1 Overview

DART provides a selection of options for the coordinate system in which all observations and all model state vector locations are described. All executables are built with a single choice from the available location modules. The names of these modules are all `location_mod`.

6.218.2 Introduction

The core algorithms of DART work with many different models which have a variety of coordinate systems. This directory provides code for creating, setting/getting, copying location information (coordinates) independently of the actual specific coordinate information. It also contains distance routines needed by the DART algorithms.

Each of the different location_mod.f90 files provides the same set of interfaces and defines a 'module location_mod', so by selecting the proper version in your path_names_xxx file you can compile your model code with the main DART routines.

- *MODULE location_mod (threed_sphere)*: The most frequently used version for real-world 3d models. It uses latitude and longitude for horizontal coordinates, plus a vertical coordinate which can be meters, pressure, model level, surface, or no specific vertical location.
- *MODULE (1D) location_mod*: The most frequently used for small models (e.g. the Lorenz family). It has a cyclic domain from 0 to 1.
- ***MODULE location_mod (threed_cartesian)*:**
  A full 3D X,Y,Z coordinate system.
- *MODULE location_mod (channel)*: a 3d domain periodic in x, limited in y, and unlimited z.
- column: no x,y but 1d height, pressure, or model level for vertical.
- annulus: a hollow 3d cylinder with azimuth, radius, and depth.
- twod: a periodic 2d domain with x,y coordinates between 0 and 1.
- twod_sphere: a 2d shell with latitude, longitude pairs.
- threed: a periodic 3d domain with x,y,z coordinates between 0 and 1.

Other schemes can be added, as needed by the models. Possible ideas are a non-periodic version of the 1d, 2d cartesian versions. Email dart at ucar.edu if you have a different coordinate scheme which we might want to support.

6.218.3 Namelist

Each location module option has a different namelist. See the specific documentation for the location option of choice.

6.218.4 Files

- none

6.218.5 References

- none

6.218.6 Private components

N/A

6.219 forward operator test README

6.219.1 Contents

1. *Overview*
2. *rttov_test.f90*
3. *rttov_unit_tests.f90*
4. *make_COS_input*
5. *make_assim_list*
6. *Terms of Use*

6.219.2 Overview

The `developer_tests/forward_operators` directory contains the testing framework for three kinds of tests:

- *rttov_test.f90* which tests basic functionality of the RTTOV radiance forward operators for AMSUA and AIRS observations (MW and IR, respectively)
  - create a dummy rttov_sensor_db_file to read
  - check the IR instrument ID
  - check the MW instrument ID
  - exercise the IR (direct) forward operator
  - exercise the MW (scatt) forward operator
- *rttov_unit_tests.f90* performs a host of unit tests.
  - If run with a TERMLEVEL of 3, all tests will be completed even if previous tests are not successful.
- *make_COS_input* and *make_assim_list* creates input for create_obs_sequence (COS) and the appropriate namelist settings to test the forward operator code.

Different sets of observations are grouped into separate files based on certain criteria - are they atmospheric observations, oceanic … do they require special metadata, etc. The following files are intended to be supplied as input to *make_COS_input* and will result in a text file that will generate an observation sequence file when used as input to *create_obs_sequence*.

- all_atm_obs_types
- all_commoncode_atm_obs_types
- all_f90s
- all_fwdop_atm_obs_types
- all_obs_types
- forward_op_code
- no_special_forward_op_code

See the *make_COS_input* section for more detail.

6.219.3 rttov_test.f90

This test requires several coefficient files that are not part of the default set provided by the RTTOV 12.3 distribution. Specifically:

- *rtcoef_eos_2_amsua.dat*
- *rtcoef_eos_2_airs.H5*
- *mietable_eos_amsua.dat* (same file as *mietable_noaa_amsua.dat*)

These coefficient files may be downloaded by using the *rtcoef_rttov12/rttov_coef_download.sh* script provided in the RTTOV distribution.

6.219.4 rttov_unit_tests.f90

These unit tests are best run with a TERMLEVEL of 3, which allows DART to continue past errors that would otherwise be fatal. If any of the unit tests are unable to start, the error code from *rttov_unit_tests* is 102. This is to give an error for *test_dart.csh* to detect.

Test	Pass	Fail
metadata growth	metadata arrays grow correctly as observations are added	incorrect metadata length
metadata content	metadata arrays contain correct data	incorrect data

6.219.5 make_COS_input

make_COS_input takes one filename as an argument and creates a text file that can be used as input for *create_obs_sequence*. The output text file has a name based on the input filename. For example:

```
<prompt> ./make_COS_input forward_op_code
 ready to run create_obs_sequence < forward_op_code_COS.in
```

create_obs_sequence must be created with the *preprocess_nml* settings to support the observation definitions required by the input file.

6.219.6 make_assim_list

make_assim_list is a follow-on step to *make_COS_input* and simply creates the text for the *input.nml:filter_nml:assimilate_these_obs* variable.

```
<prompt> forward_operators > ./make_assim_list forward_op_code
 created forward_op_code_obskind.nml
 add this section to your &obs_kind_nml in input.nml
 <prompt> head -n 10 forward_op_code_obskind.nml
 assimilate_these_obs_types =
 'ACARS_DEWPOINT,',
 'ACARS_RELATIVE_HUMIDITY,',
 'AIRCRAFT_DEWPOINT,',
 'AIRCRAFT_RELATIVE_HUMIDITY,',
 'AIREP_DEWPOINT,',
 'AIRS_DEWPOINT,',
 'AIRS_RELATIVE_HUMIDITY,',
 'AMDAR_DEWPOINT,',
 'AMSR_TOTAL_PRECIPITABLE_WATER,',
```

6.219.7 Terms of Use

© University Corporation for Atmospheric Research

Licensed under the Apache License, Version 2.0. Unless required by applicable law or agreed to in writing, software distributed under this license is distributed on an "as is" basis, without warranties or conditions of any kind, either express or implied.

6.220 PROGRAM `PrecisionCheck`

6.220.1 Overview

This is a self-contained program to explore the interaction between the compiler options to 'autopromote' variables from one precision to another and the intrinsic F90 mechanism for getting consistent behavior without relying on autopromotion - namely, the `SELECT_INT_KIND()` and `SELECT_REAL_KIND()` functions. The most portable code explicity types the variables to avoid relying on compiler flags. The core DART code abides by these rules; some pieces that are derived from dynamical models may have original code fragments.

All that is required is to compile the single file and run the resulting executable. There are no required libraries - any F90 compiler should have no trouble with this program. There is no input of any kind.

You are encouraged to view the source code. It's pretty obvious what is being tested.

6.220.2 Examples

The following examples have differences from the default configuration highlighted in boldface. You are strongly encouraged to test your compiler and its autopromotion options. The Absoft compiler actually does what I consider to be reasonable and logical (as long as you know that "-dp" means **d**emote **p**recision). Many other compilers are surprising.

PowerPC chipset : Absoft Pro Fortran 9.0

```
[~/DART/utilities] % f90 PrecisionCheck.f90
[~/DART/utilities] % ./a.out

 This explores the use of the intrinsic SELECTED_[REAL,INT]_KIND() functions
 and the interplay with the compiler options. You are encouraged to use the
 "autopromotion" flags on your compiler and compare the results.

-----------------------------------------------
 "integer"
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
-----------------------------------------------
 "integer(i4)" i4 = SELECTED_INT_KIND(8)
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
-----------------------------------------------
 "integer(i8)" i8 = SELECTED_INT_KIND(13)
 DIGITS    =   63
 HUGE      =   9223372036854775807
 KIND      =   8
-----------------------------------------------
 "real"
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
-----------------------------------------------
 "real(r4)" r4 = SELECTED_REAL_KIND(6,30)
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
-----------------------------------------------
 "real(r8)" r8 = SELECTED_REAL_KIND(13)
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
```

(continues on next page)

(continued from previous page)

```
-------------------------------------------
 "double precision"
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
```

PowerPC chipset : Absoft Pro Fortran 9.0 : "-dp"

```
[~/DART/utilities] % f90 -dp PrecisionCheck.f90
[~/DART/utilities] % ./a.out

 This explores the use of the intrinsic SELECTED_[REAL,INT]_KIND() functions
 and the interplay with the compiler options. You are encouraged to use the
 "autopromotion" flags on your compiler and compare the results.

-------------------------------------------
 "integer"
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
-------------------------------------------
 "integer(i4)" i4 = SELECTED_INT_KIND(8)
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
-------------------------------------------
 "integer(i8)" i8 = SELECTED_INT_KIND(13)
 DIGITS    =   63
 HUGE      =   9223372036854775807
 KIND      =   8
-------------------------------------------
 "real"
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
-------------------------------------------
 "real(r4)" r4 = SELECTED_REAL_KIND(6,30)
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
-------------------------------------------
 "real(r8)" r8 = SELECTED_REAL_KIND(13)
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
```

(continues on next page)

(continued from previous page)

```
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
-----------------------------------------------
 "double precision"
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
```

PowerPC chipset : Absoft Pro Fortran 9.0 : "-n113"

```
[~/DART/utilities] % f90 -N113 PrecisionCheck.f90
[~/DART/utilities] % ./a.out

 This explores the use of the intrinsic SELECTED_[REAL,INT]_KIND() functions
 and the interplay with the compiler options. You are encouraged to use the
 "autopromotion" flags on your compiler and compare the results.

-----------------------------------------------
 "integer"
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
-----------------------------------------------
 "integer(i4)" i4 = SELECTED_INT_KIND(8)
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
-----------------------------------------------
 "integer(i8)" i8 = SELECTED_INT_KIND(13)
 DIGITS    =   63
 HUGE      =   9223372036854775807
 KIND      =   8
-----------------------------------------------
 "real"
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
-----------------------------------------------
 "real(r4)" r4 = SELECTED_REAL_KIND(6,30)
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
-----------------------------------------------
```

(continues on next page)

(continued from previous page)

```
 "real(r8)" r8 = SELECTED_REAL_KIND(13)
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
------------------------------------------------
 "double precision"
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
```

6.221 MODULE obs_def_gps_mod

6.221.1 Overview

DART GPS Radio Occultation observation module, including the observation operators for both local and non-local refractivity computations.

Author information:

- Dr. Hui Liu

6.221.2 Namelist

This namelist is now enabled by default. The maximum number of GPS observations is settable at runtime by changing the value in the namelist. If you get an error about a missing namelist add `&obs_def_gps_nml` using the example below to your `input.nml` namelist file and rerun. No recompiling is needed.

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_def_gps_nml
  max_gpsro_obs = 100000,
/
```

Item	Type	Description
max_gpsro_obs	integer	The maximum number of GPS refractivity observations supported for a single execution. Generally the default will be sufficient for a single run of filter, but not enough for a long diagnostics run to produce a time series.

6.221.3 Other modules used

```
types_mod
utilities_mod
location_mod (threed_sphere)
assim_model_mod
obs_kind_mod
```

6.221.4 Public interfaces

use obs_def_gps_mod, only :	read_gpsro_ref
	write_gpsro_ref
	get_expected_gpsro_ref
	interactive_gpsro_ref
	set_gpsro_ref
	get_gpsro_ref

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call read_gpsro_ref(gpskey, ifile, [, fform])

```
integer,          intent(out)          :: gpskey
integer,          intent(in)           :: ifile
character(len=*), intent(in), optional :: fform
```

Refractivity observations have several items of auxiliary data to read or write. This routine reads in the data for the next observation and returns the private GPS key index number that identifies the auxiliary data for this observation.

`gpskey`	GPS key number returned to the caller.
`ifile`	Open file unit number to read from.
fform	If specified, indicate whether the file was opened formatted or unformatted. Default is 'formatted'.

call write_gpsro_ref(gpskey, ifile, [, fform])

```
integer,          intent(in)           :: gpskey
integer,          intent(in)           :: ifile
character(len=*), intent(in), optional :: fform
```

Refractivity observations have several items of auxiliary data to read or write. This routine writes out the auxiliary data for the specified observation to the file unit given.

gpskey	GPS key number identifying which observation to write aux data for.
ifile	Open file unit number to write to.
fform	If specified, indicate whether the file was opened formatted or unformatted. Default is 'formatted'.

call get_expected_gpsro_ref(state_vector, location, gpskey, ro_ref, istatus)

```
real(r8),            intent(in)  :: state_vector(:)
type(location_type), intent(in)  :: location
integer,             intent(in)  :: gpskey
real(r8),            intent(out) :: ro_ref
integer,             intent(out) :: istatus
```

Given a location and the state vector from one of the ensemble members, compute the model-predicted GPS refractivity that would be observed at that location. There are two types of operators: modeled *local* refractivity (N-1)*1.0e6 or *non_local* refractivity (excess phase, m) The type is indicated in the auxiliary information for each observation.

state_vector	Some dimensional representation of the model state vector
location	Location of this observation
gpskey	Integer key identifying which GPS observation this is, so the correct corresponding auxiliary information can be accessed.
ro_ref	The returned GPS refractivity value
istatus	Returned integer status code describing problems with applying forward operator. 0 is a good value; any positive value indicates an error; negative values are reserved for internal DART use only.

call interactive_gpsro_ref(gpskey)

```
integer, intent(out) :: gpskey
```

Prompts the user for the auxiliary information needed for a GPS refractivity observation, and returns the new key associated with this data.

gpskey	Unique identifier associated with this GPS refractivity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine returns the incremented value associated with this data.

call set_gpsro_ref(gpskey, nx, ny, nz, rfict0, ds, htop, subset0)

```
integer,          intent(out) :: gpskey
real(r8),         intent(in)  :: nx
real(r8),         intent(in)  :: ny
real(r8),         intent(in)  :: nz
real(r8),         intent(in)  :: rfict0
real(r8),         intent(in)  :: ds
real(r8),         intent(in)  :: htop
character(len=6), intent(in)  :: subset0
```

Sets the auxiliary information associated with a GPS refractivity observation. This routine increments and returns the new key associated with these values.

gpskey	Unique identifier associated with this GPS refractivity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine returns the incremented value associated with this data.
nx	X component of direction of ray between the LEO (detector) satellite and the GPS transmitter satellite at the tangent point.
ny	Y component of tangent ray.
nz	Z component of tangent ray.
rfict0	Local curvature radius (meters).
ds	Delta S, increment to move along the ray in each direction when integrating the non-local operator (meters).
htop	Elevation (in meters) where integration stops along the ray.
subset0	The string 'GPSREF' for the local operator (refractivity computed only at the tangent point), or 'GPSEXC' for the non-local operator which computes excess phase along the ray.

call get_gpsro_ref(gpskey, nx, ny, nz, rfict0, ds, htop, subset0)

```
integer,          intent(in)  :: gpskey
real(r8),         intent(out) :: nx
real(r8),         intent(out) :: ny
real(r8),         intent(out) :: nz
real(r8),         intent(out) :: rfict0
real(r8),         intent(out) :: ds
real(r8),         intent(out) :: htop
character(len=6), intent(out) :: subset0
```

Gets the auxiliary information associated with a GPS refractivity observation, based on the GPS key number specified.

gpskey	Unique identifier associated with this GPS refractivity observation. In this code it is an integer index into module local arrays which hold the additional data. The value specified selects which observation to return data for.
nx	X component of direction of ray between the LEO (detector) satellite and the GPS transmitter satellite at the tangent point.
ny	Y component of tangent ray.
nz	Z component of tangent ray.
rfict0	Local curvature radius (meters).
ds	Delta S, increment to move along the ray in each direction when integrating the non-local operator (meters).
htop	Elevation (in meters) where integration stops along the ray.
subset0	The string 'GPSREF' for the local operator (refractivity computed only at the tangent point), or 'GPSEXC' for the non-local operator which computes excess phase along the ray.

6.221.5 Files

- A DART observation sequence file containing GPS obs.

6.221.6 References

- Assimilation of GPS Radio Occultation Data for Numerical Weather Prediction, Kuo,Y.H., Sokolovskiy,S.V., Anthes,R.A., Vendenberghe,F., Terrestrial Atm and Ocn Sciences, Vol 11, pp157-186, 2000.

6.221.7 Error codes and conditions

Routine	Message	Comment
initialize_module	initial allocation failed for gps observation data, itemcount = (max_gpsro_obs)	Need to increase max_gpsro_obs count in namelist
gpskey_out_of_range	gpskey (key#) exceeds max_radial_gps_obs (maxval)	The number of GPS observations exceeds the array size allocated in the module. Need to increase max_gpsro_obs count in namelist.
read_gpsro_ref	Expected header 'gpsroref' in input file	The format of the input obs_seq file is not consistent.
get_expected_gpsro_ref	vertical location must be height; gps obs key #	GPS observations must have vertical coordinates of height

6.221.8 Future Plans

- The current code first bins the very densely-sampled vertical profile into 200 bins, and then interpolates the requested vertical location from that. The original profiles have been plotted and are smooth; there appears to be no need to pre-bin the ata.
- The local operator needs no additional auxiliary data. The observation files would be much smaller if the local operator observation was a separate type without aux data, and only the non-local operator observation types would need the ray direction, the curvature, etc.

6.222 MODULE obs_def_dew_point_mod

6.222.1 Overview

Provides a subroutine to calculate the dew point temperature from model temperature, specific humidity, and pressure.

Revision 2801 (April 2007) implements a more robust method (based on Bolton's Approximation) for calculating dew point. This has been further revised to avoid a numerical instability that could lead to failed forward operators for dewpoints almost exactly 0 C.

6.222.2 Other modules used

```
types_mod
utilities_mod
location_mod (most likely threed_sphere)
assim_model_mod
obs_kind_mod
```

6.222.3 Public interfaces

use obs_def_dew_point_mod, only :	get_expected_dew_point

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call get_expected_dew_point(state_vector, location, key, td, istatus)

```
real(r8),            intent(in)  :: state_vector
type(location_type), intent(in)  :: location
integer,             intent(in)  :: key
real(r8),            intent(out) :: td
integer,             intent(out) :: istatus
```

Calculates the dew point temperature (Kelvin).

state_vector	A one dimensional representation of the model state vector
location	Location for this obs
key	Controls whether upper levels (key = 1) or 2-meter (key = 2) is required.
td	The returned dew point temperature value
istatus	Returned integer describing problems with applying forward operator

6.222.4 Files

- NONE

6.222.5 References

1. Bolton, David, 1980: The Computation of Equivalent Potential Temperature. Monthly Weather Review, 108, 1046-1053.

6.222.6 Error codes and conditions

Routine	Message	Comment
get_expected_dew_point	'key has to be 1 (upper levels) or 2 (2-meter), got ',key	The input value of key is not allowed.

6.223 MODULE obs_def_ocean_mod

6.223.1 Overview

DART includes a flexible, powerful, and slightly complicated mechanism for incorporating new types of observations. The obs_def_ocean_mod module being described here is used by the program preprocess to insert appropriate definitions of ocean observations into the DEFAULT_obs_def_mod.f90 template and generate the source files obs_def_mod.f90 and obs_kind_mod.f90 that are used by filter and other DART programs.

Only HFRADAR_RADIAL_VELOCITY observations require a forward operator, as evidenced by the fact there is no COMMON_CODE in the third column of the type definitions table. All other observations types map to quantities that must be available in the model state; the observations types flagged with COMMON_CODE will use the model_interpolate() routine as the forward operator.

The mandatory header line is followed by lines that have the observation type name (an all caps Fortran 90 identifier) and their associated generic quantity identifier from the obs_kind module. If there is no special processing needed for an observation type, and no additional data needed beyond the standard contents of an observation, then a third word on the line, the COMMON_CODE will instruct the preprocess program to automatically generate all stubs and code needed for this type. For observation types needing any special code or additional data, this word should not be specified and the user must supply the code manually. One of the future extensions of this module will be to support acoustic tomographic observations, which will necessitate specific support routines.

Ocean variable types and their corresponding quantities

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
! SALINITY,                       QTY_SALINITY,              COMMON_CODE
! TEMPERATURE,                    QTY_TEMPERATURE,           COMMON_CODE
! U_CURRENT_COMPONENT,            QTY_U_CURRENT_COMPONENT,   COMMON_CODE
! V_CURRENT_COMPONENT,            QTY_V_CURRENT_COMPONENT,   COMMON_CODE
! SEA_SURFACE_HEIGHT,             QTY_SEA_SURFACE_HEIGHT,    COMMON_CODE
! SEA_SURFACE_PRESSURE,           QTY_SEA_SURFACE_PRESSURE,  COMMON_CODE
! ARGO_U_CURRENT_COMPONENT,       QTY_U_CURRENT_COMPONENT,   COMMON_CODE
! ARGO_V_CURRENT_COMPONENT,       QTY_V_CURRENT_COMPONENT,   COMMON_CODE
! ARGO_SALINITY,                  QTY_SALINITY,              COMMON_CODE
! ARGO_TEMPERATURE,               QTY_TEMPERATURE,           COMMON_CODE
! ADCP_U_CURRENT_COMPONENT,       QTY_U_CURRENT_COMPONENT,   COMMON_CODE
! ADCP_V_CURRENT_COMPONENT,       QTY_V_CURRENT_COMPONENT,   COMMON_CODE
! ADCP_SALINITY,                  QTY_SALINITY,              COMMON_CODE
! ADCP_TEMPERATURE,               QTY_TEMPERATURE,           COMMON_CODE
! FLOAT_SALINITY,                 QTY_SALINITY,              COMMON_CODE
! FLOAT_TEMPERATURE,              QTY_TEMPERATURE,           COMMON_CODE
! DRIFTER_U_CURRENT_COMPONENT,    QTY_U_CURRENT_COMPONENT,   COMMON_CODE
! DRIFTER_V_CURRENT_COMPONENT,    QTY_V_CURRENT_COMPONENT,   COMMON_CODE
! DRIFTER_SALINITY,               QTY_SALINITY,              COMMON_CODE
! DRIFTER_TEMPERATURE,            QTY_TEMPERATURE,           COMMON_CODE
! GLIDER_U_CURRENT_COMPONENT,     QTY_U_CURRENT_COMPONENT,   COMMON_CODE
! GLIDER_V_CURRENT_COMPONENT,     QTY_V_CURRENT_COMPONENT,   COMMON_CODE
! GLIDER_SALINITY,                QTY_SALINITY,              COMMON_CODE
! GLIDER_TEMPERATURE,             QTY_TEMPERATURE,           COMMON_CODE
! MOORING_U_CURRENT_COMPONENT,    QTY_U_CURRENT_COMPONENT,   COMMON_CODE
! MOORING_V_CURRENT_COMPONENT,    QTY_V_CURRENT_COMPONENT,   COMMON_CODE
! MOORING_SALINITY,               QTY_SALINITY,              COMMON_CODE
! MOORING_TEMPERATURE,            QTY_TEMPERATURE,           COMMON_CODE
! MOORING_PRESSURE,               QTY_PRESSURE,              COMMON_CODE
! BOTTLE_SALINITY,                QTY_SALINITY,              COMMON_CODE
! BOTTLE_TEMPERATURE,             QTY_TEMPERATURE,           COMMON_CODE
! CTD_SALINITY,                   QTY_SALINITY,              COMMON_CODE
! CTD_TEMPERATURE,                QTY_TEMPERATURE,           COMMON_CODE
! TCTD_SALINITY,                  QTY_SALINITY,              COMMON_CODE
! TCTD_TEMPERATURE,               QTY_TEMPERATURE,           COMMON_CODE
! STD_SALINITY,                   QTY_SALINITY,              COMMON_CODE
! STD_TEMPERATURE,                QTY_TEMPERATURE,           COMMON_CODE
! XCTD_SALINITY,                  QTY_SALINITY,              COMMON_CODE
! XCTD_TEMPERATURE,               QTY_TEMPERATURE,           COMMON_CODE
! MBT_SALINITY,                   QTY_SALINITY,              COMMON_CODE
! MBT_TEMPERATURE,                QTY_TEMPERATURE,           COMMON_CODE
! XBT_SALINITY,                   QTY_SALINITY,              COMMON_CODE
! XBT_TEMPERATURE,                QTY_TEMPERATURE,           COMMON_CODE
! DBT_SALINITY,                   QTY_SALINITY,              COMMON_CODE
! DBT_TEMPERATURE,                QTY_TEMPERATURE,           COMMON_CODE
! APB_SALINITY,                   QTY_SALINITY,              COMMON_CODE
! APB_TEMPERATURE,                QTY_TEMPERATURE,           COMMON_CODE
! DOPPLER_U_CURRENT_COMPONENT,    QTY_U_CURRENT_COMPONENT,   COMMON_CODE
! DOPPLER_V_CURRENT_COMPONENT,    QTY_V_CURRENT_COMPONENT,   COMMON_CODE
! DOPPLER_W_CURRENT_COMPONENT,    QTY_W_CURRENT_COMPONENT,   COMMON_CODE
```

(continues on next page)

(continued from previous page)

```
! SATELLITE_MICROWAVE_SST,       QTY_TEMPERATURE,           COMMON_CODE
! SATELLITE_INFRARED_SST,        QTY_TEMPERATURE,           COMMON_CODE
! SATELLITE_BLENDED_SST,         QTY_TEMPERATURE,           COMMON_CODE
! SATELLITE_SSH,                 QTY_SEA_SURFACE_HEIGHT,    COMMON_CODE
! SATELLITE_SSS,                 QTY_SALINITY,              COMMON_CODE
! J1_SEA_SURFACE_ANOMALY,        QTY_SEA_SURFACE_ANOMALY,   COMMON_CODE
! EN_SEA_SURFACE_ANOMALY,        QTY_SEA_SURFACE_ANOMALY,   COMMON_CODE
! GFO_SEA_SURFACE_ANOMALY,       QTY_SEA_SURFACE_ANOMALY,   COMMON_CODE
! DRY_LAND,                      QTY_DRY_LAND,              COMMON_CODE
! OI_SEA_SURFACE_TEMPERATURE,    QTY_TEMPERATURE,           COMMON_CODE
! HFRADAR_U_CURRENT_COMPONENT,   QTY_U_CURRENT_COMPONENT,   COMMON_CODE
! HFRADAR_V_CURRENT_COMPONENT,   QTY_V_CURRENT_COMPONENT,   COMMON_CODE
! HFRADAR_RADIAL_VELOCITY,       QTY_VELOCITY
! FERRYBOX_SALINITY,             QTY_SALINITY,              COMMON_CODE
! FERRYBOX_TEMPERATURE,          QTY_TEMPERATURE,           COMMON_CODE
! END DART PREPROCESS TYPE DEFINITIONS
```

New observation types may be added to this list with no loss of generality. Supporting the observations and actually **assimilating** them are somewhat different and is controlled by the `input.nml&obs_kind_nml` assimilate_these_obs_types variable. This provides the flexibility to have an observation sequence file containing many different observation types and being able to selectively choose what types will be assimilated.

6.223.2 Other modules used

```
types_mod
utilities_mod
location_mod (threed_sphere)
assim_model_mod
obs_kind_mod
ensemble_manager_mod
obs_def_utilities_mod
```

6.223.3 Public interfaces

use obs_def_ocean_mod, only :	read_hf_radial_vel
	write_hf_radial_vel
	interactive_hf_radial_vel
	get_expected_hf_radial_vel
	get_obs_def_hf_radial_vel
	set_hf_radial_vel

6.223.4 Namelist

Namelist interface `obs_def_ocean_nml` is read from file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_def_ocean_nml
   max_radial_vel_obs = 1000000
   debug = .false.
   /
```

Item	Type	Description
max_radial_vel_obs	integer	The maximum number of radial velocity observations to be read at one time. An error is thrown if more observations are encountered. Increase value and rerun.
debug	logical	Switch to control how much run-time output is created. `.false.` indicates less output, `.true.` indicates more output.

6.223.5 Public components

none

6.223.6 Files

none

6.223.7 References

none

6.223.8 Private components

N/A

6.224 MODULE obs_def_1d_state_mod

6.224.1 Overview

The list of observation types to be supported by the DART executables is defined at compile time. The observations DART supports can be changed at any time by adding or removing items from the preprocess namelist and rerunning *quickbuild.sh*.

`Preprocess` takes observation specific code sections from special obs_def files to generate `obs_def_mod.f90` and `obs_kind_mod.f90` which are then compiled into `filter` and other DART programs. One of the motivations behind creating `obs_def_1d_state_mod.f90` was to provide a prototype for people developing more complicated specialized observation definition modules.
`Obs_def_1d_state_mod.f90` is an extended format Fortran 90 module that provides the definition for observation types designed for use with idealized low-order models that use the 1D location module and can be thought of as having a state vector that is equally spaced on a 1D cyclic domain. Observation types include:

- RAW_STATE_VARIABLE - A straight linear interpolation to a point on a [0,1] domain.
- RAW_STATE_VAR_POWER - The interpolated RAW_STATE_VARIABLE raised to a real-valued power.
- RAW_STATE_1D_INTEGRAL - An area-weighted 'integral' of the state variable over some part of the cyclic 1D domain.

RAW_STATE_VAR_POWER is convenient for studying non-gaussian, non-linear assimilation problems. RAW_STATE_VAR_POWER can be used to do idealized studies related to remote sensing observations that are best thought of as weighted integrals of some quantity over a finite volume.
The RAW_STATE_1D_INTEGRAL has an associated half_width and localization type (see the *MODULE cov_cutoff_mod* documentation) and a number of points at which to compute the associated integral by quadrature. The location of the observation defines the center of mass of the integral. The integral is centered around the location and extends outward on each side to 2*half_width. The weight associated with the integral is defined by the weight of the localization function (for instance Gaspari Cohn) using the same localization options as defined by the cov_cutoff module. The number of points are used to equally divide the range for computing the integral by quadrature.
Special observation modules like `obs_def_1d_state_mod.f90` contain Fortran 90 code *and* additional specially formatted commented code that is used to guide the preprocess program in constructing obs_def_mod.f90 and obs_kind_mod.f90. The specially formatted comments are most conveniently placed at the beginning of the module and comprise seven sections, each beginning and ending with a special F90 comment line that must be included *verbatim*.
The seven sections and their specific instances for the 1d_raw_state_mod are:

1. A list of all observation types defined by this module and their associated generic quantities (see *PROGRAM preprocess* for details on quantity files). The header line is followed by lines that have the observation type name (an all caps Fortran 90 identifier) and their associated generic quantity identifier. If there is no special processing needed for an observation type, and no additional data needed beyond the standard contents of an observation then a third word on the line, `COMMON_CODE`, will instruct the preprocess program to automatically generate all stubs and code needed for this type. For observation types needing special code or additional data, this word should not be specified and the user must supply the code manually.

```
! BEGIN DART PREPROCESS KIND LIST
! RAW_STATE_VARIABLE,    QTY_STATE_VARIABLE,   COMMON_CODE
```

(continues on next page)

(continued from previous page)

```
! RAW_STATE_1D_INTEGRAL, QTY_1D_INTEGRAL
! END DART PREPROCESS KIND LIST
```

2. A list of all the use statements that the completed obs_def_mod.f90 must have in order to use the public interfaces provided by this special obs_def module. This section is optional if there are no external interfaces.

```
! BEGIN DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
!   use obs_def_1d_state_mod, only : write_1d_integral, read_1d_integral,  &
!                                    interactive_1d_integral, get_expected_1d_
↪integral, &
!                                    set_1d_integral
! END DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
```

3. Case statement entries for each observation type defined by this special obs_def module stating how to compute the forward observation operator. There must be a case statement entry for each type of observation, *except* for observation types defined with COMMON_CODE.

```
! BEGIN DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
!         case(RAW_STATE_1D_INTEGRAL)
!            call get_expected_1d_integral(state, location, obs_def%key, obs_val,␣
↪istatus)
! END DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
```

4. Case statement entries for each observation type defined by this special obs_def module stating how to read any extra required information from an obs sequence file. There must be a case statement entry for each type of observation, *except* for observation types defined with COMMON_CODE. If no special action is required put a `continue` statement as the body of the case instead of a subroutine call.

```
! BEGIN DART PREPROCESS READ_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call read_1d_integral(obs_def%key, ifile, fform)
! END DART PREPROCESS READ_OBS_DEF
```

5. Case statement entries for each observation type defined by this special obs_def module stating how to write any extra required information to an obs sequence file. There must be a case statement entry for each type of

observation, *except* for observation types defined with COMMON_CODE. If no special action is required put a `continue` statement as the body of the case instead of a subroutine call.

```
! BEGIN DART PREPROCESS WRITE_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call write_1d_integral(obs_def%key, ifile, fform)
! END DART PREPROCESS WRITE_OBS_DEF
```

6. Case statement entries for each observation type defined by this special obs_def module stating how to interactively create any extra required information. There must be a case statement entry for each type of observation, *except* for observation types defined with COMMON_CODE. If no special action is required put a `continue` statement as the body of the case instead of a subroutine call.

```
! BEGIN DART PREPROCESS INTERACTIVE_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call interactive_1d_integral(obs_def%key)
! END DART PREPROCESS INTERACTIVE_OBS_DEF
```

7. Any executable F90 module code must be tagged with the following comments. All lines between these markers will be copied, verbatim, to obs_def_mod.f90. This section is not required if there are no observation-specific subroutines.

```
! BEGIN DART PREPROCESS MODULE CODE
module obs_def_1d_state_mod

... (module executable code)

end module obs_def_1d_state_mod
! END DART PREPROCESS MODULE CODE
```

6.224.2 Other modules used

```
types_mod
utilities_mod
location_mod (1d_location_mod_only)
time_manager_mod
assim_model_mod
cov_cutoff_mod
```

6.224.3 Public interfaces

use obs_def_mod, only :	write_1d_integral
	read_1d_integral
	interactive_1d_integral
	get_expected_1d_integral
	set_1d_integral
	write_power
	read_power
	interactive_power
	get_expected_power
	set_power

call write_1d_integral(igrkey, ifile, fform)

```
integer,          intent(in) :: igrkey
integer,          intent(in) :: ifile
character(len=*), intent(in) :: fform
```

Writes out the extra information for observation with unique identifier key for a 1d_integral observation type. This includes the half-width, localization type and number of quadrature points for this observation.

igrkey	Unique integer key associated with the 1d integral observation being processed. This is not the same as the key that all types of observations have and uniquely distinguishes all observations from each other; this is a key that is only set and retrieved by this code for 1d integral observations. It is stored in the obs_def derived type, not in the main obs_type definition.
ifile	Unit number on which observation sequence file is open
fform	String noting whether file is opened for 'formatted' or 'unformatted' IO.

call read_1d_integral(igrkey, ifile, fform)

```
integer,          intent(out) :: igrkey
integer,          intent(in)  :: ifile
character(len=*), intent(in)  :: fform
```

Reads the extra information for observation with unique identifier key for a 1d_integral observation type. This information includes the half-width, localization type and number of quadrature points for this observation. The key that is returned is uniquely associated with the definition that has been created and is used by this module to keep track of the associated parameters for this observation.

igrkey	Unique integer key associated with the observation being processed.
ifile	Unit number on which observation sequence file is open
fform	String noting whether file is opened for 'formatted' or 'unformatted' IO.

call interactive_1d_integral(igrkey)

```
integer, intent(out) :: igrkey
```

Uses input from standard in to define the characteristics of a 1D integral observation. The key that is returned is uniquely associated with the definition that has been created and can be used by this module to keep track of the associated parameters (half_width, localization option, number of quadrature points) for this key.

igrkey	Unique identifier associated with the created observation definition in the obs sequence.

call get_expected_1d_integral(state, location, igrkey, val, istatus)

```
real(r8), intent(in)            :: state
type(location_type), intent(in) :: location
integer, intent(in)             :: igrkey
real(r8), intent(out)           :: val
integer, intent(out)            :: istatus
```

Computes the forward observation operator for a 1d integral observation. Calls the `interpolate()` routine multiple times to invoke the forward operator code in whatever model this has been compiled with.

state	Model state vector (or extended state vector).
location	Location of this observation.
igrkey	Unique integer key associated with this observation.
val	Returned value of forward observation operator.
istatus	Returns 0 if forward operator was successfully computed, else returns a positive value. (Negative values are reserved for system use.)

call set_1d_integral(integral_half_width, num_eval_pts, localize_type, igrkey, istatus)

```
real(r8), intent(in)  :: integral_half_width
integer,  intent(in)  :: num_eval_pts
integer,  intent(in)  :: localize_type
integer,  intent(out) :: igrkey
integer,  intent(out) :: istatus
```

Available for use by programs that create observations to set the additional metadata for these observation types. This information includes the integral half-width, localization type and number of quadrature points for this observation. The key that is returned is uniquely associated with the definition that has been created and should be set in the obs_def structure by calling `set_obs_def_key()`. This key is different from the main observation key which all observation types have. This key is unique to this observation type and is used when reading in the observation sequence to match the corresponding metadata with each observation of this type.

`integral_half_width`	Real value setting the half-width of the integral.
`num_eval_pts`	Integer, number of evaluation points. 5-20 recommended.
`localize_type`	Integer localization type: 1=Gaspari-Cohn; 2=Boxcar; 3=Ramped Boxcar
`igrkey`	Unique integer key associated with the observation being processed.
`istatus`	Return code. 0 means success, any other value is an error

call write_power(powkey, ifile, fform)

```
integer,          intent(in) :: powkey
integer,          intent(in) :: ifile
character(len=*), intent(in) :: fform
```

Writes out the extra information, the power, for observation with unique identifier key for a power observation type.

`powkey`	Unique integer key associated with the power observation being processed. This is not the same as the key that all types of observations have and uniquely distinguishes all observations from each other; this is a key that is only set and retrieved by this code for power observations. It is stored in the obs_def derived type, not in the main obs_type definition.
`ifile`	Unit number on which observation sequence file is open
`fform`	String noting whether file is opened for 'formatted' or 'unformatted' IO.

call read_power(powkey, ifile, fform)

```
integer,          intent(out) :: powkey
integer,          intent(in)  :: ifile
character(len=*), intent(in)  :: fform
```

Reads the extra information, the power, for observation with unique identifier key for a power observation type. The key that is returned is uniquely associated with the definition that has been created and is used by this module to keep track of the associated parameters for this observation.

powkey	Unique integer key associated with the observation being processed.
ifile	Unit number on which observation sequence file is open
fform	String noting whether file is opened for 'formatted' or 'unformatted' IO.

call interactive_power(powkey)

```
integer, intent(out) :: powkey
```

Uses input from standard in to define the characteristics of a power observation. The key that is returned is uniquely associated with the definition that has been created and can be used by this module to keep track of the associated parameter, the power, for this key.

powkey	Unique identifier associated with the created observation definition in the obs sequence.

call get_expected_power(state, location, powkey, val, istatus)

```
real(r8), intent(in)            :: state
type(location_type), intent(in) :: location
integer, intent(in)             :: powkey
real(r8), intent(out)           :: val
integer, intent(out)            :: istatus
```

Computes the forward observation operator for a power observation. Calls the `interpolate()` routine to invoke the forward operator code in whatever model this has been compiled with, then raises the result to the specified power associated with this powkey.

state	Model state vector (or extended state vector).
location	Location of this observation.
powkey	Unique integer key associated with this observation.
val	Returned value of forward observation operator.
istatus	Returns 0 if forward operator was successfully computed, else returns a positive value. (Negative values are reserved for system use.)

call set_power(power_in, powkey, istatus)

```
real(r8), intent(in)  :: power_in
integer,  intent(out) :: powkey
integer,  intent(out) :: istatus
```

Available for use by programs that create observations to set the additional metadata for these observation types. This information includes the power to which to raise the state variable. The key that is returned is uniquely associated with the definition that has been created and should be set in the obs_def structure by calling `set_obs_def_key()`. This key is different from the main observation key which all observation types have. This key is unique to this observation type and is used when reading in the observation sequence to match the corresponding metadata with each observation of this type.

`power_in`	Real value setting the power.
`powkey`	Unique integer key associated with the observation being processed.
`istatus`	Return code. 0 means success, any other value is an error

6.224.4 Namelist

This module has no namelist.

6.224.5 Files

- NONE

6.224.6 References

1. none

6.224.7 Error codes and conditions

Routine	Message	Comment
interactive_1d_integral	Out of space, max_1d_integral_obs limit NNNN (currently 1000).	There is only room for a fixed number of 1d integral observations. The max number is defined by max_1d_integral_obs. Set this to a larger value if more are needed.

6.225 MODULE obs_def_radar_mod

6.225.1 Overview

DART radar observation module, including the observation operators for the two primary radar-observation types – Doppler velocity and reflectivity – plus other utility subroutines and functions. A number of simplifications are employed for the observation operators. Most notably, the model state is mapped to a "point" observation, whereas a real radar observation is a volumetric sample. The implications of this approximation have not been investigated fully, so in the future it might be worth developing and testing more sophisticated observation operators that produce volumetric power- weighted samples.

This module is able to compute reflectivity and precipitation fall speed (needed for computing Doppler radial velocity) from the prognostic model fields only for simple single-moment microphysics schemes such as the Kessler and Lin schemes. If a more complicated microphysics scheme is used, then reflectivity and fall speed must be accessible instead as diagnostic fields in the model state.

Author and Contact information:

- Radar Science: David Dowell, david.dowell at noaa.gov, Glen Romine, romine at ucar.edu
- DART Code: Nancy Collins, nancy at ucar.edu
- Original DART/Radar work: Alain Caya

Backward compatibility note

For users of previous versions of the radar obs_def code, here are a list of changes beginning with subversion revision 3616 which are not backward compatible:

- The namelist has changed quite a bit; some items were removed, some added, and some renamed. See the namelist documention in this file for the current item names and default values.
- Some constants which depend on the microphysics scheme have been added to the namelist to make it easier to change the values for different schemes, but the defaults have also changed. Verify they are appropriate for the scheme being used.
- The interactive create routine prompts for the beam direction differently now. It takes azimuth and elevation, and then does the trigonometry to compute the three internal values which are stored in the file. The previous version prompted for the internal values directly.
- The get_expected routines try to call the model interpolate routine for `QTY_POWER_WEIGHTED_FALL_SPEED` and `QTY_RADAR_REFLECTIVITY` values. If they are not available then the code calls the model interpolate routines for several other quantities and computes these quantities. However, this requires that the model_mod interpolate code returns gracefully if the quantity is unknown or unsupported. The previous version of the WRF model_mod code used to print an error message and stop if the quantity was unknown. The updated version in the repository which went in with this radar code has been changed to return an error status code but continue if the quantity is unknown.
- The value for gravity is currently hardcoded in this module. Previous versions of this code used the gravity constant in the DART types_mod.f90 code, but in reality the code should be using whatever value of gravity is being used in the model code. For now, the value is at least separated so users can change the value in this code if necessary.

6.225.2 Other modules used

```
types_mod
utilities_mod
location_mod (threed_sphere)
assim_model_mod
obs_kind_mod
```

6.225.3 Public interfaces

use obs_def_radar_mod, only :	read_radar_ref
	get_expected_radar_ref
	read_radial_vel
	write_radial_vel
	interactive_radial_vel
	get_expected_radial_vel
	get_obs_def_radial_vel
	set_radial_vel

Namelist interface `&obs_def_radar_mod_nml` is read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call read_radar_ref(obsvalue, refkey)

```
real(r8),                 intent(inout) :: obsvalue
integer,                  intent(out)   :: refkey
```

Reflectivity observations have no auxiliary data to read or write, but there are namelist options that can alter the observation value at runtime. This routine tests the observation value and alters it if required.

`obsvalue`	Observation value.
`refkey`	Set to 0 to avoid uninitialized values, but otherwise unused.

call get_expected_radar_ref(state_vector, location, ref, istatus)

```
real(r8),            intent(in)  :: state_vector(:)
type(location_type), intent(in)  :: location
real(r8),            intent(out) :: ref
integer,             intent(out) :: istatus
```

Given a location and the state vector from one of the ensemble members, compute the model-predicted radar reflectivity that would be observed at that location. The returned value is in dBZ.
If `apply_ref_limit_to_fwd_op` is .TRUE. in the namelist, reflectivity values less than `reflectivity_limit_fwd_op` will be set to `lowest_reflectivity_fwd_op`.

`state_vector`	A one dimensional representation of the model state vector
`location`	Location of this observation
`ref`	The returned radar reflectivity value
`istatus`	Returned integer status code describing problems with applying forward operator. 0 is a good value; any positive value indicates an error; negative values are reserved for internal DART use only.

call read_radial_vel(velkey, ifile [, fform])

```
integer,                    intent(out) :: velkey
integer,                    intent(in)  :: ifile
character(len=*), optional, intent(in)  :: fform
```

Reads the additional auxiliary information associated with a radial velocity observation. This includes the location of the radar source, the beam direction, and the nyquist velocity.

`velkey`	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine increments it and returns the new value.
`ifile`	File unit descriptor for input file
fform	File format specifier: FORMATTED or UNFORMATTED; default FORMATTED

call write_radial_vel(velkey, ifile [, fform])

```
integer,                    intent(in) :: velkey
integer,                    intent(in) :: ifile
character(len=*), optional, intent(in) :: fform
```

Writes the additional auxiliary information associated with a radial velocity observation. This includes the location of the radar source, the beam direction, and the nyquist velocity.

velkey	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine uses the value to select the appropriate data to write for this observation.
ifile	File unit descriptor for output file
fform	File format specifier: FORMATTED or UNFORMATTED; default FORMATTED

call get_obs_def_radial_vel(velkey, radar_location, beam_direction, nyquist_velocity)

```
integer,             intent(in)  :: velkey
type(location_type), intent(out) :: radar_location
real(r8),            intent(out) :: beam_direction(3)
real(r8),            intent(out) :: nyquist_velocity
```

Returns the auxiliary information associated with a given radial velocity observation.

velkey	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine uses the value to select the appropriate data to return.
radar_location	Location of the radar.
beam_orientation	Orientation of the radar beam at the observation location. The three values are: sin(azimuth)*cos(elevation), cos(azimuth)*cos(elevation), and sin(elevation).
nyquist_velocity	Nyquist velocity at the observation point in meters/second.

call set_radial_vel(velkey, radar_location, beam_direction, nyquist_velocity)

```
integer,             intent(out) :: velkey
type(location_type), intent(in)  :: radar_location
real(r8),            intent(in)  :: beam_direction(3)
real(r8),            intent(in)  :: nyquist_velocity
```

Sets the auxiliary information associated with a radial velocity observation. This routine increments and returns the new key associated with these values.

velkey	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine returns the incremented value associated with this data.
radar_location	Location of the radar.
beam_orientation	Orientation of the radar beam at the observation location. The three values are: sin(azimuth)*cos(elevation), cos(azimuth)*cos(elevation), and sin(elevation).
nyquist_velocity	Nyquist velocity at the observation point in meters/second.

call interactive_radial_vel(velkey)

```
integer, intent(out) :: velkey
```

Prompts the user for the auxiliary information needed for a radial velocity observation, and returns the new key associated with this data.

velkey	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine returns the incremented value associated with this data.

call get_expected_radial_vel(state_vector, location, velkey, radial_vel, istatus)

```
real(r8),            intent(in)  :: state_vector(:)
type(location_type), intent(in)  :: location
integer,             intent(in)  :: velkey
real(r8),            intent(out) :: radial_vel
integer,             intent(out) :: istatus
```

Given a location and the state vector from one of the ensemble members, compute the model-predicted radial velocity in meters/second that would be observed at that location. `velkey` is the unique index for this particular radial velocity observation. The value is returned in `radial_vel`, `istatus` is the return code.

The along-beam component of the 3-d air velocity is computed from the u, v, and w fields plus the beam_direction. The along-beam component of power-weighted precipitation fall velocity is added to the result.

state_vector	Some dimensional representation of the model state vector
location	Location of this observation
velkey	Unique identifier associated with this radial velocity observation
radial_vel	The returned radial velocity value in meters/second
istatus	Returned integer status code describing problems with applying forward operator. 0 is a good value; any positive value indicates an error; negative values are reserved for internal DART use only.

6.225.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_def_radar_mod_nml
   apply_ref_limit_to_obs      =   .false.,
   reflectivity_limit_obs      =     -10.0,
```

(continues on next page)

(continued from previous page)

```
   lowest_reflectivity_obs     =     -10.0,
   apply_ref_limit_to_fwd_op   =   .false.,
   reflectivity_limit_fwd_op   =     -10.0,
   lowest_reflectivity_fwd_op  =     -10.0,
   max_radial_vel_obs          =   1000000,
   allow_wet_graupel           =   .false.,
   microphysics_type           =       2  ,
   allow_dbztowt_conv          =   .false.,
   dielectric_factor           =     0.224,
   n0_rain                     =     8.0e6,
   n0_graupel                  =     4.0e6,
   n0_snow                     =     3.0e6,
   rho_rain                    =    1000.0,
   rho_graupel                 =     400.0,
   rho_snow                    =     100.0
   /
```

Item	Type	Description
apply_ref_limit_to_obs	logical	If .TRUE. replace all reflectivity values less than "reflectivity_limit_obs" with "lowest_reflectivity_obs" value. If .FALSE. leave all values as-is.
reflectivity_limit_obs	real(r8)	The threshold value. Observed reflectivity values less than this threshold will be set to the "lowest_reflectivity_obs" value. Units are dBZ.
lowest_reflectivity_obs	real(r8)	The 'set-to' value. Observed reflectivity values less than the threshold will be set to this value. Units are dBZ.
apply_ref_limit_to_fwd_op	logical	Same as "apply_ref_limit_to_obs", but for the forward operator.
reflectivity_limit_fwd_op	real(r8)	Same as "reflectivity_limit_obs", but for the forward operator values.
lowest_reflectivity_fwd_op	real(r8)	Same as "lowest_reflectivity_obs", but for the forward operator values.
max_radial_vel_obs	integer	Maximum number of observations of this type to support at run time. This is combined total of all obs_seq files, for example the observation diagnostic program potentially opens multiple obs_seq.final files, or the obs merge program can also open multiple obs files.
allow_wet_graupel	logical	It is difficult to predict/diagnose whether graupel/hail has a wet or dry surface. Even when the temperature is above freezing, evaporation and/or absorption can still result in a dry surface. This issue is important because the reflectivity from graupel with a wet surface is significantly greater than that from graupel with a dry surface. Currently, the user has two options for how to compute graupel reflectivity. If allow_wet_graupel is .false. (the default), then graupel is always assumed to be dry. If allow_wet_graupel is .true., then graupel is assumed to be wet (dry) when the temperature is above (below) freezing. A consequence is that a sharp gradient in reflectivity will be produced at the freezing level. In the future, it might be better to provide the option of having a transition layer.
microphysics_type	integer	If the state vector contains the reflectivity or the power weighted fall speed, interpolate directly from those regardless of the setting of this item. If the state vector does not contain the fields, this value should be

6.225.5 Files

- A DART observation sequence file containing Radar obs.

6.225.6 References

- Battan, L. J., 1973: *Radar Observation of the Atmosphere.* Univ. of Chicago Press, 324 pp.
- Caya, A. *Radar Observations in Dart.* DART Subversion repository.
- Doviak, R. J., and D. S. Zrnic, 1993: *Doppler Radar and Weather Observations.* Academic Press, 562 pp.
- Ferrier, B. S., 1994: A double-moment multiple-phase four-class bulk ice scheme. Part I: Description. *J. Atmos. Sci.*, **51**, 249-280.
- Lin, Y.-L., Farley R. D., and H. D. Orville, 1983: Bulk parameterization of the snow field in a cloud model. *J. Climate Appl. Meteor.*, **22**, 1065-1092.
- Smith, P. L. Jr., 1984: Equivalent radar reflectivity factors for snow and ice particles. *J. Climate Appl. Meteor.*, 23, 1258-1260.
- Smith, P. L. Jr., Myers C. G., and H. D. Orville, 1975: Radar reflectivity factor calculations in numerical cloud models using bulk parameterization of precipitation. *J. Appl. Meteor.*, **14**, 1156-1165.

6.225.7 Error codes and conditions

Routine	Message	Comment
initialize_module	initial allocation failed for radial vel obs data, itemcount = (max_radial_vel_obs)	Need to increase max_radial_vel_obs count in namelist
read_radial_vel	Expected location header "platform" in input file	The format of the input file is not consistent.
velkey_out_of_range	key (val) exceeds max_radial_vel_obs (maxval)	The number of radial velocity observations exceeds the array size allocated in the module. Need to increase max_radial_vel_obs count in namelist.
read_nyquist_velocity	bad value for nyquist velocity	The format of the input obs_seq file is not consistent.
read_beam_direction	beam direction value must be between -1 and 1, got ()	The format of the input obs_seq file is not consistent.
read_beam_direction	Expected orientation header "dir3d" in input file	The format of the input obs_seq file is not consistent.

6.225.8 Private components

use obs_def_radar_mod, only :	initialize_module
	read_beam_direction
	read_nyquist_velocity
	write_beam_direction
	write_nyquist_velocity
	interactive_beam_direction
	interactive_nyquist_velocity
	get_reflectivity
	get_precip_fall_speed
	initialize_constants
	print_constants
	pr_con
	velkey_out_of_range
	check_namelist_limits
	ascii_file_format

call initialize_module()

Reads the namelist, allocates space for the auxiliary data associated wtih radial velocity observations, initializes the constants used in subsequent computations (possibly altered by values in the namelist), and prints out the list of constants and the values in use. These may need to change depending on which microphysics scheme is being used.

beam_direction = read_beam_direction(ifile, is_asciiformat)

```
real(r8), dimension(3)           :: read_beam_direction
integer,               intent(in) :: ifile
logical,               intent(in) :: is_asciiformat
```

Reads the beam direction at the observation location. Auxiliary data for doppler radial velocity observations.

read_beam_direction	Returns three real values for the radar beam orientation
ifile	File unit descriptor for input file
is_asciiformat	File format specifier: .TRUE. if file is formatted/ascii, or .FALSE. if unformatted/binary. Default .TRUE.

nyquist_velocity = read_nyquist_velocity(ifile, is_asciiformat)

```
real(r8),            :: read_nyquist_velocity
integer,  intent(in) :: ifile
logical,  intent(in) :: is_asciiformat
```

Reads nyquist velocity for a doppler radial velocity observation.

read_nyquist_velocity	Returns a real value for the nyquist velocity value
ifile	File unit descriptor for input file
is_asciiformat	File format specifier: .TRUE. if file is formatted/ascii, or .FALSE. if unformatted/binary. Default .TRUE.

call write_beam_direction(ifile, beam_direction, is_asciiformat)

```
integer,                intent(in) :: ifile
real(r8), dimension(3), intent(in) :: beam_direction
logical,                intent(in) :: is_asciiformat
```

Writes the beam direction at the observation location. Auxiliary data for doppler radial velocity observations.

ifile	File unit descriptor for output file
beam_direction	Three components of the radar beam orientation
is_asciiformat	File format specifier: .TRUE. if file is formatted/ascii, or .FALSE. if unformatted/binary. Default .TRUE.

call write_nyquist_velocity(ifile, nyquist_velocity, is_asciiformat)

```
integer,  intent(in) :: ifile
real(r8), intent(in) :: nyquist_velocity
logical,  intent(in) :: is_asciiformat
```

Writes nyquist velocity for a doppler radial velocity observation.

ifile	File unit descriptor for output file
nyquist_velocity	The nyquist velocity value for this observation
is_asciiformat	File format specifier: .TRUE. if file is formatted/ascii, or .FALSE. if unformatted/binary. Default .TRUE.

call interactive_beam_direction(beam_direction)

```
real(r8), dimension(3), intent(out) :: beam_direction
```

Prompts the user for input for the azimuth and elevation of the radar beam at the observation location. Will be converted to the three values actually stored in the observation sequence file.

beam_direction	Three components of the radar beam orientation

call interactive_nyquist_velocity(nyquist_velocity)

```
real(r8), intent(out) :: nyquist_velocity
```

Prompts the user for input for the nyquist velocity value associated with a doppler radial velocity observation.

nyquist_velocity	Nyquist velocity value for the observation.

call get_reflectivity(qr, qg, qs, rho, temp, ref)

```
real(r8), intent(in)  :: qr
real(r8), intent(in)  :: qg
real(r8), intent(in)  :: qs
real(r8), intent(in)  :: rho
real(r8), intent(in)  :: temp
real(r8), intent(out) :: ref
```

Computes the equivalent radar reflectivity factor in mm^6 m^{-3} for simple single-moment microphysics schemes such as Kessler and Lin, et al. See the references for more details.

qr	Rain water content (kg kg^{-1})
qg	Graupel/hail content (kg kg^{-1})
qs	Snow content (kg kg^{-1})
rho	Air density (kg m^{-3})
temp	Air temperature (K)
ref	The returned radar reflectivity value

call get_precip_fall_speed(qr, qg, qs, rho, temp, precip_fall_speed)

```
real(r8), intent(in)  :: qr
real(r8), intent(in)  :: qg
real(r8), intent(in)  :: qs
real(r8), intent(in)  :: rho
real(r8), intent(in)  :: temp
real(r8), intent(out) :: precip_fall_speed
```

Computes power-weighted precipitation fall speed in m s^{-1} for simple single-moment microphysics schemes such as Kessler and Lin, et al. See the references for more details.

qr	Rain water content (kg kg^{-1})
qg	Graupel/hail content (kg kg^{-1})
qs	Snow content (kg kg^{-1})
rho	Air density (kg m^{-3})
temp	Air temperature (K)
precip_fall_speed	The returned precipitation vall speed

call initialize_constants()

Set values for a collection of constants used throughout the module during the various calculations. These are set once in this routine and are unchanged throughout the rest of the execution. They cannot be true Fortran `parameters` because some of the values can be overwritten by namelist entries, but once they are set they are treated as read-only parameters.

call print_constants()

Print out the names and values of all constant parameters used by this module. The error handler message facility is used to print the message, which by default goes to both the DART log file and to the standard output of the program.

call pr_con(c_val, c_str)

```
real(r8),         intent(in)  :: c_val
character(len=*), intent(in)  :: c_str
```

Calls the DART error handler routine to print out a string label and a real value to both the log file and to the standard output.

Value of constant	A real value.
Name of constant	A character string.

call velkey_out_of_range(velkey)

```
integer, intent(in)  :: velkey
```

Range check key and trigger a fatal error if larger than the allocated array for observation auxiliary data.

velkey	Integer key into a local array of auxiliary observation data.

call check_namelist_limits(apply_ref_limit_to_obs, reflectivity_limit_obs, lowest_reflectivity_obs, apply_ref_limit_to_fwd_op, reflectivity_limit_fwd_op, lowest_reflectivity_fwd_op)

```
logical,  intent(in) :: apply_ref_limit_to_obs
real(r8), intent(in) :: reflectivity_limit_obs
real(r8), intent(in) :: lowest_reflectivity_obs
logical,  intent(in) :: apply_ref_limit_to_fwd_op
real(r8), intent(in) :: reflectivity_limit_fwd_op
real(r8), intent(in) :: lowest_reflectivity_fwd_op
```

Check the values set in the namelist for consistency. Print out a message if the limits and set-to values are different; this may be intentional but is not generally expected to be the case. In all cases below, see the namelist documentation for a fuller explanation of each value.

apply_ref_limit_to_obs	Logical. See namelist.
reflectivity_limit_obs	Real value. See namelist.
lowest_reflectivity_obs	Real value. See namelist.
apply_ref_limit_to_fwd_op	Logical. See namelist.
reflectivity_limit_fwd_op	Real value. See namelist.
lowest_reflectivity_fwd_op	Real value. See namelist.

is_asciifile = ascii_file_format(fform)

```
logical                                 :: ascii_file_format
character(len=*), intent(in), optional :: fform
```

Should be moved to DART utility module at some point. Returns .TRUE. if the optional argument is missing or if it is not one of the following values: "unformatted", "UNFORMATTED", "unf", "UNF".

ascii_file_format	Return value. Logical. Default is .TRUE.
fform	Character string file format.

6.226 MODULE DEFAULT_obs_def_mod

6.226.1 Overview

DEFAULT_obs_def.F90 is a template used by the program `preprocess` to create `obs_def_mod.f90`.

To read more detailed instructions on how to add new observation types, see the documentation for *MODULE obs_def_mod*. `obs_def_*_mod.f90` files are specified as input to the `preprocess` program by namelist, and a new `obs_def_mod.f90` file is generated which contains all the selected observation types.

Information from zero or more special obs_def modules, such as `obs_def_1d_state_mod.f90` or `obs_def_reanalyis_bufr_mod.f90`, (also documented in this directory) are incorporated into the DEFAULT_obs_def_mod.F90 template by `preprocess`. If no special obs_def files are included in the preprocess namelist, a minimal `obs_def_mod.f90` is created which can only support identity forward observation operators. Any identity observations on the obs_seq.out file will be assimilated, regardless of the obs types specified in assimilate_these_obs_types.

The documentation below describes the special formatting that is included in the `DEFAULT_obs_def_mod.F90` in order to guide the `preprocess` program.

Up to seven sections of code are inserted into `DEFAULT_obs_def_mod.F90` from each of the special `obs_def_*_mod.f90` files. The insertion point for each section is denoted by a special comment line that must be included *verbatim* in `DEFAULT_obs_def_mod.F90`. These special comment lines and their significance are:

1. `! DART PREPROCESS MODULE CODE INSERTED HERE`

   Some special observation definition modules (see for instance `obs_def_1d_state_mod.f90`) contain code for evaluating forward observation operators, reading or writing special information about an observation definition to an obs sequence file, or for interactive definition of an observation. The entire module code section is inserted here, so the resulting output file will be completely self-contained. Fortran 90 allows multiple modules to be defined in a single source file, and subsequent module code can use previously defined modules, so this statement must preceed the rest of the other comment lines.

1. `! DART PREPROCESS USE FOR OBS_QTY_MOD INSERTED HERE`

   The quantities available to DART are defined by passing quantity files from `DART/assimilation_code/modules/observations` to `preprocess`. Unique integer values for each quantity are assigned by `preprocess` and the use statements for these entries are inserted here.

2. `! DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE INSERTED HERE`

   Some special observation definition modules (see for instance `obs_def_1d_state_mod.f90`) contain code for evaluating forward observation operators, reading or writing special information about an observation definition to an obs sequence file, or for interactive definition of an observation. The use statements for these routines from the special observation definition modules are inserted here.

3. `! DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF INSERTED HERE`

   Special observation definition modules must contain case statement code saying what to do to evaluate a forward observation operator for each observation type that they define. This code is inserted here.

4. `! DART PREPROCESS READ_OBS_DEF INSERTED HERE`

   Special observation definition modules must contain case statement code saying what to do to read any additional information required for each observation type that they define from an observation sequence file. This code is inserted here.

5. `! DART PREPROCESS WRITE_OBS_DEF INSERTED HERE`

   Special observation definition modules must contain case statement code saying what to do to write any additional information required for each observation type that they define to an observation sequence file. This code is inserted here.

6. `! DART PREPROCESS INTERACTIVE_OBS_DEF INSERTED HERE`

   Special observation definition modules must contain case statement code saying what to do to interactively create any additional information required for each observation type that they define. This code is inserted here.

6.227 MODULE obs_def_mod

6.227.1 Overview

The DART Fortran90 derived type `obs_def` provide an abstraction of the definition of an observation. An observation sequence `obs_seq` at a higher level is composed of observation definitions associated with observed values. For now, the basic operations required to implement an observation definition are an ability to compute a forward operator given the model state vector, the ability to read/write the observation definition from/to a file, and a capability to do a standard input driven interactive definition of the observation definition.

DART makes a distinction between specific `observation types` and generic `observation quantities`. The role of the various obs_def input files is to define the mapping between the types and quantities, and optionally to provide type-specific processing routines.

A single obs_def output module is created by the program `preprocess` from two kinds of input files. First, a DEFAULT obs_def module (normally called `DEFAULT_obs_def_mod.F90` and documented in this directory) is used as a template into which the preprocessor incorporates information from zero or more special obs_def modules (such as `obs_def_1d_state_mod.f90` or `obs_def_reanalysis_bufr_mod.f90`, also documented in this directory). If no special obs_def files are included in the preprocessor namelist, a minimal `obs_def_mod.f90` is created which can only support identity forward observation operators.

New Observation Types

To add a new observation type which does not fit into any of the already-defined obs_def files, a new file should be created in the `obs_def` directory. These files are usually named according the the pattern `obs_def_X_mod.f90`, where the X is either an instrument name, a data source, or a class of observations. See the existing filenames in that directory for ideas. Then this new filename must be listed in the `input.nml` namelist for the model, in the `&preprocess_nml` section, in the `obs_type_files` variable. This variable is a string list type which can contain multiple filenames. Running the `preprocess` program will then use the contents of the new file to generate the needed output files for use in linking to the rest of the DART system.

Simple observations

If the new observation type can be directly interpolated by a model_mod interpolation routine, and has no additional observation-specific code for reading, writing, or initializing the observation, then the entire contents of the new file is:

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
! type, quantity, COMMON_CODE
! (repeat lines for each type)
! END DART PREPROCESS TYPE DEFINITIONS
```

DART will automatically generate all interface code needed for these new observation types. For example, here is a real list:

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
!VELOCITY,                    QTY_VELOCITY,              COMMON_CODE
!TRACER_CONCENTRATION,        QTY_TRACER_CONCENTRATION,  COMMON_CODE
!TRACER_SOURCE,               QTY_TRACER_SOURCE,         COMMON_CODE
!MEAN_SOURCE,                 QTY_MEAN_SOURCE,           COMMON_CODE
!SOURCE_PHASE,                QTY_SOURCE_PHASE,          COMMON_CODE
! END DART PREPROCESS TYPE DEFINITIONS
```

The first column is the specific observation `type` and should be unique. The second column is the generic observation `quantity`. The quantities available to DART are defined at compile time by *preprocess* via the option 'quantity_files' in the *preprocess_nml* namelist. The third column must be the keyword `COMMON_CODE` which tells the `preprocess` program to automatically generate all necessary interface code for this type.

Observations needing special handling

For observation types which have observation-specific routines, must interpolate using a combination of other generic quantities, or require additional observation-specific data to be stored, the following format is used:

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
! type, quantity
! (repeat lines for each type/quantity pair)
! END DART PREPROCESS TYPE DEFINITIONS
```

DART will need user-supplied interface code for each of the listed types. For example, here is a real list:

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
! DOPPLER_RADIAL_VELOCITY, QTY_VELOCITY
! RADAR_REFLECTIVITY,      QTY_RADAR_REFLECTIVITY
! END DART PREPROCESS TYPE DEFINITIONS
```

In this case, DART needs additional information for how to process these types. They include code sections delimited by precisely formatted comments, and possibly module code sections:

1. ```
   ! BEGIN DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
   ! END DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
   ```

   Any fortran use statements for public subroutines or variables from other modules should be placed between these lines, with comment characters in the first column.

   For example, if the forward operator code includes a module with public routines then a "use" statement like:
   ```

```
use obs_def_1d_state_mod, only : write_1d_integral, read_1d_integral, &
                                 interactive_1d_integral, get_expected_1d_integral
```

needs to be added to the obs_def_mod so the listed subroutines are available to be called. This would look like:

```
! BEGIN DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
! use obs_def_1d_state_mod, only : write_1d_integral, read_1d_integral, &
!                                  interactive_1d_integral, get_expected_1d_integral
! END DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
```

2.
```
! BEGIN DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
! END DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
```

These comments must enclose a case statement for each defined type that returns the expected observation value based on the current values of the state vector. The code must be in comments, with the comment character in the first column.

The variables available to be passed to subroutines or used in this section of code are:

state	the entire model state vector
state_time	the time of the state data
ens_index	the ensemble member number
location	the observation location
obs_kind_ind	the index of the specific observation type
obs_time	the time of the observation
error_val	the observation error variance

The routine must fill in the values of these variables:

obs_val	the computed forward operator value
istatus	return code: 0=ok, >0 is error, <0 reserved for system use

To call a model_mod interpolate routine directly, the argument list must match exactly:

```
interpolate(state, location, QTY_xxx, obs_val, istatus)
```

This can be useful if the forward operator needs to retrieve values for fields which are typically found in a model and then compute a derived value from them.

3.
```
! BEGIN DART PREPROCESS READ_OBS_DEF
! END DART PREPROCESS READ_OBS_DEF
```

These comments must enclose a case statement for each defined type that reads any additional data associated with a single observation. If there is no information beyond that for the basic obs_def type, the case statement must still be provided, but the code can simply be `continue`. The code must be in comments, with the comment character in the first column.

The variables available to be passed to subroutines or used in this section of code are:

`ifile`	the open unit number positioned ready to read, read-only
`obs_def`	the rest of the obs_def derived type for this obs, read-write
`key`	the index observation number in this sequence, read-only
`obs_val`	the observation value, if needed. in general should not be changed
`is_ascii`	logical to indicate how the file was opened, formatted or unformatted

The usual use of this routine is to read in additional metadata per observation and to set the private key in the `obs_def` to indicate which index to use for this observation to look up the corresponding metadata in arrays or derived types. Do not confuse the key in the obs_def with the key argument to this routine; the latter is the global observation sequence number for this observation.

4.
```
! BEGIN DART PREPROCESS WRITE_OBS_DEF
! END DART PREPROCESS WRITE_OBS_DEF
```

These comments must enclose a case statement for each defined type that writes any additional data associated with a single observation. If there is no information beyond that for the basic obs_def type, the case statement must still be provided, but the code can simply be `continue`. The code must be in comments, with the comment character in the first column.

The variables available to be passed to subroutines or used in this section of code are:

`ifile`	the open unit number positioned ready to write, read-only
`obs_def`	the rest of the obs_def derived type for this obs, read-only
`key`	the index observation number in this sequence, read-only
`is_ascii`	logical to indicate how the file was opened, formatted or unformatted

The usual use of this routine is to write the additional metadata for this observation based on the private key in the `obs_def`. Do not confuse this with the key in the subroutine call which is the observation number relative to the entire observation sequence file.

5.
```
! BEGIN DART PREPROCESS INTERACTIVE_OBS_DEF
! END DART PREPROCESS INTERACTIVE_OBS_DEF
```

These comments must enclose a case statement for each defined type that prompts the user for any additional data associated with a single observation. If there is no information beyond that for the basic obs_def type, the

case statement must still be provided, but the code can simply be `continue`. The code must be in comments, with the comment character in the first column.

The variables available to be passed to subroutines or used in this section of code are:

`obs_def`	the rest of the obs_def derived type for this obs, read-write
`key`	the index observation number in this sequence, read-only

The DART code will prompt for the rest of the obs_def values (location, type, value, error) but any additional metadata needed by this observation type should be prompted to, and read from, the console (e.g. `write(*,*)`, and `read(*, *)`). The code will generally set the `obs_def%key` value as part of setting the metadata.

6.
```
! BEGIN DART PREPROCESS MODULE CODE
! END DART PREPROCESS MODULE CODE
```

If the code to process this observation requires module data and/or subroutines, then these comments must surround the module definitions. Unlike all the other sections, this comment pair is optional, and if used, the code must not be in comments; it will be copied verbatim over to the output file.

Generally the code for a forward operator should be defined inside a module, to keep module variables and other private subroutines from colliding with unrelated routines and variables in other forward operator files.

It is possible to mix automatic code types and user-supplied code types in the same list. Simply add the COMMON_CODE keyword on the lines which need no special data or interfaces. For example, here is an extract from the 1d state obs_def module, where the raw state variable needs only autogenerated code, but the 1d integral has user-supplied processing code:

```
! BEGIN DART PREPROCESS TYPE LIST
! RAW_STATE_VARIABLE,    QTY_STATE_VARIABLE, COMMON_CODE
! RAW_STATE_1D_INTEGRAL, QTY_1D_INTEGRAL
! END DART PREPROCESS TYPE LIST


! BEGIN DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
!   use obs_def_1d_state_mod, only : write_1d_integral, read_1d_integral, &
!                                    interactive_1d_integral, get_expected_1d_integral
! END DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE

! BEGIN DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
!         case(RAW_STATE_1D_INTEGRAL)
!            call get_expected_1d_integral(state, location, obs_def%key, obs_val,␣
↪istatus)
! END DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF

! BEGIN DART PREPROCESS READ_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call read_1d_integral(obs_def%key, ifile, fileformat)
```

(continues on next page)

(continued from previous page)

```
! END DART PREPROCESS READ_OBS_DEF

! BEGIN DART PREPROCESS WRITE_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call write_1d_integral(obs_def%key, ifile, fileformat)
! END DART PREPROCESS WRITE_OBS_DEF

! BEGIN DART PREPROCESS INTERACTIVE_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call interactive_1d_integral(obs_def%key)
! END DART PREPROCESS INTERACTIVE_OBS_DEF

! BEGIN DART PREPROCESS MODULE CODE
module obs_def_1d_state_mod

use        types_mod, only : r8
use    utilities_mod, only : register_module, error_handler, E_ERR, E_MSG
use     location_mod, only : location_type, set_location, get_location
use  assim_model_mod, only : interpolate
use   cov_cutoff_mod, only : comp_cov_factor

implicit none

public :: write_1d_integral, read_1d_integral, interactive_1d_integral, &
          get_expected_1d_integral

...  (module code here)

end module obs_def_1d_state_mod
! END DART PREPROCESS MODULE CODE
```

See the *MODULE obs_def_1d_state_mod* documentation for more details and examples of each section. Also see `obs_def_wind_speed_mod.f90` for an example of a 3D geophysical forward operator.

In addition to collecting and managing any additional observation type-specific code, this module provides the definition of the obs_def_type derived type, and a collection of subroutines for creating, accessing, and updating this type. The remainder of this document describes the subroutines provided by this module.

6.227.2 Other modules used

```
types_mod
utilities_mod
location_mod (depends on model choice)
time_manager_mod
assim_model_mod
obs_kind_mod
Other special obs_def_kind modules as required
```

6.227.3 Public interfaces

use obs_def_mod, only :	obs_def_type
	init_obs_def
	get_obs_def_location
	get_obs_def_type_of_obs
	get_obs_def_time
	get_obs_def_error_variance
	get_obs_def_key
	set_obs_def_location
	set_obs_def_type_of_obs
	set_obs_def_time
	set_obs_def_error_variance
	set_obs_def_key
	interactive_obs_def
	write_obs_def
	read_obs_def
	get_expected_obs_from_def
	destroy_obs_def
	copy_obs_def
	assignment(=)
	get_name_for_type_of_obs

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

```
type obs_def_type
   private
   type(location_type)  :: location
```

(continues on next page)

(continued from previous page)

```
   integer              :: kind
   type(time_type)      :: time
   real(r8)             :: error_variance
   integer              :: key
end type obs_def_type
```

Models all that is known about an observation except for actual values. Includes a location, type, time and error variance.

Component	Description
location	Location of the observation.
kind	Despite the name, the specific type of the observation.
time	Time of the observation.
error_variance	Error variance of the observation.
key	Unique identifier for observations of a particular type.

call init_obs_def(obs_def, location, kind, time, error_variance)

```
type(obs_def_type),  intent(out) :: obs_def
type(location_type), intent(in)  :: location
integer,             intent(in)  :: kind
type(time_type),     intent(in)  :: time
real(r8),            intent(in)  :: error_variance
```

Creates an obs_def type with location, type, time and error_variance specified.

obs_def	The obs_def that is created
location	Location for this obs_def
kind	Observation type for obs_def
time	Time for obs_def
error_variance	Error variance of this observation

call copy_obs_def(obs_def1, obs_def2)

```
type(obs_def_type), intent(out) :: obs_def1
type(obs_def_type), intent(in)  :: obs_def2
```

Copies obs_def2 to obs_def1, overloaded as assignment (=).

obs_def1	obs_def to be copied into
obs_def2	obs_def to be copied from

var = get_obs_def_key(obs_def)

```
integer                          :: get_obs_def_key
type(obs_def_type), intent(in) :: obs_def
```

Returns key from an observation definition.

var	Returns key from an obs_def
obs_def	An obs_def

var = get_obs_def_error_variance(obs_def)

```
real(r8)                         :: get_obs_def_error_variance
type(obs_def_type), intent(in) :: obs_def
```

Returns error variance from an observation definition.

var	Error variance from an obs_def
obs_def	An obs_def

var = get_obs_def_location(obs_def)

```
type(location_type)              :: get_obs_def_location
type(obs_def_type), intent(in)   :: obs_def
```

Returns the location from an observation definition.

var	Returns location from an obs_def
obs_def	An obs_def

var = get_obs_def_type_of_obs(obs_def)

```
integer                          :: get_obs_def_type_of_obs
type(obs_def_type),  intent(in) :: obs_def
```

Returns an observation type from an observation definition.

var	Returns the observation type from an obs_def
obs_def	An obs_def

var = get_obs_def_time(obs_def)

```
type(time_type)                 :: get_obs_def_time
type(obs_def_type), intent(in) :: obs_def
```

Returns time from an observation definition.

var	Returns time from an obs_def
obs_def	An obs_def

obs_name = get_name_for_type_of_obs(obs_kind_ind)

```
character(len = 32)             :: get_name_for_type_of_obs
integer, intent(in)             :: obs_kind_ind
```

Returns an observation name from an observation type.

var	Returns name from an observation type
obs_kind_ind	An observation type

call set_obs_def_location(obs_def, location)

```
type(obs_def_type),  intent(inout) :: obs_def
type(location_type), intent(in)    :: location
```

Set the location in an observation definition.

obs_def	An obs_def
location	A location

call set_obs_def_error_variance(obs_def, error_variance)

```
type(obs_def_type), intent(inout) :: obs_def
real(r8), intent(in)              :: error_variance
```

Set error variance for an observation definition.

obs_def	An obs_def
error_variance	Error variance

call set_obs_def_key(obs_def, key)

```
type(obs_def_type), intent(inout) :: obs_def
integer,            intent(in)    :: key
```

Set the key for an observation definition.

obs_def	An obs_def
key	Unique identifier for this observation

call set_obs_def_type_of_obs(obs_def, kind)

```
type(obs_def_type), intent(inout) :: obs_def
integer,            intent(in)    :: kind
```

Set the type of observation in an observation definition.

obs_def	An obs_def
kind	An integer observation type

call set_obs_def_time(obs_def, time)

```
type(obs_def_type), intent(inout) :: obs_def
type(time_type), intent(in)       :: time
```

Sets time for an observation definition.

obs_def	An obs_def
time	Time to set

call get_expected_obs_from_def(key, obs_def, obs_kind_ind, ens_index, state, state_time, obs_val, istatus, assimilate_this_ob, evaluate_this_ob)

```
integer,            intent(in)  :: key
type(obs_def_type), intent(in)  :: obs_def
integer,            intent(in)  :: obs_kind_ind
integer,            intent(in)  :: ens_index
real(r8),           intent(in)  :: state(:)
type(time_type),    intent(in)  :: state_time
real(r8),           intent(out) :: obs_val
integer,            intent(out) :: istatus
logical,            intent(out) :: assimilate_this_ob
logical,            intent(out) :: evaluate_this_ob
```

Compute the observation (forward) operator for a particular obs definition.

key	descriptor for observation type
obs_def	The input obs_def
obs_kind_ind	The obs type
ens_index	The ensemble member number of this state vector
state	Model state vector
state_time	Time of the data in the model state vector
istatus	Returned integer describing problems with applying forward operator (0 == OK, >0 == error, <0 reserved for sys use).
assimilate_this_ob	Indicates whether to assimilate this obs or not
evaluate_this_ob	Indicates whether to evaluate this obs or not

call read_obs_def(ifile, obs_def, key, obs_val [,fform])

```
integer,                    intent(in)    :: ifile
type(obs_def_type),         intent(inout) :: obs_def
integer,                    intent(in)    :: key
real(r8),                   intent(inout) :: obs_val
character(len=*), optional, intent(in)    :: fform
```

Reads an obs_def from file open on channel ifile. Uses format specified in fform or FORMATTED if fform is not present.

ifile	File unit open to output file
obs_def	Observation definition to be read
key	Present if unique identifier key is needed by some obs type. Unused by default code.
obs_val	Present if needed to perform operations based on value. Unused by default code.
fform	File format specifier: FORMATTED or UNFORMATTED; default FORMATTED (FORMATTED in this case is the human readable/text option as opposed to UNFORMATTED which is binary.)

call interactive_obs_def(obs_def, key)

```
type(obs_def_type), intent(inout) :: obs_def
integer,            intent(in)    :: key
```

Creates an obs_def via input from standard in.

obs_def	An obs_def to be created
key	Present if unique identifier key is needed by some obs type. Unused by default code.

call write_obs_def(ifile, obs_def, key [,fform])

```
integer,                    intent(in) :: ifile
type(obs_def_type),         intent(in) :: obs_def
integer,                    intent(in) :: key
character(len=*), optional, intent(in) :: fform
```

Writes an obs_def to file open on channel ifile. Uses format specified in fform or FORMATTED if fform is not present.

ifile	File unit open to output file
obs_def	Observation definition to be written
key	Present if unique identifier key is needed by some obs type. Unused by default code.
fform	File format specifier: FORMATTED or UNFORMATTED; default FORMATTED

call destroy_obs_def(obs_def)

```
type(obs_def_type), intent(inout) :: obs_def
```

Releases all storage associated with an obs_def and its subcomponents.

obs_def	An obs_def to be released.

6.227.4 Files

- The read_obs_def() and write_obs_def() routines are passed an already-opened file channel/descriptor and read to or write from it.

6.227.5 References

- none

6.227.6 Error codes and conditions

Routine	Message	Comment
get_expected_obs_from_def	Attempt to evaluate undefined observation type	An observation type for which no forward operator has been defined is an error.
read_obs_def	Expected header "obdef" in input file	The format of the input file is not consistent.
read_obs_def	Expected kind header "kind " in input file	The format of the input file is not consistent.
read_obs_def	Attempt to read for undefined obs_kind index	Reading for an observation type for which no forward operator has been defined is an error.
write_obs_def	Attempt to write for undefined obs_kind index	Writing for an observation type for which no forward operator has been defined is an error.
interactive_obs_def	Attempt to interactively create undefined obs_kind index	Creating an observation type for which no forward operator has been defined is an error.

6.227.7 Private components

N/A

6.228 MODULE obs_def_rttov_mod

6.228.1 Overview

DART RTTOV observation module, including the observation operators for the two primary RTTOV-observation types – visible/infrared radiances and microwave radiances/brightness temperatures.

The obs_def_rttov_mod.f90 module acts as a pass-through for RTTOV version 12.3. For more information, see the RTTOV site.

For RTTOV v13 use the obs_def_rttov13_mod.f90 module contributed by Lukas Kugler of the University of Vienna.

DART supports both RTTOV-direct for visible/infrared/microwave as well as RTTOV-scatt for microwave computations. The code, in principle, supports all features of version 12.3 as a pass-through from the model to RTTOV, includes aerosols, trace gases, clouds, and atmospheric variables. The code also includes directly specifying scattering properties.

However, a model may not have all of the variables necessary for these functions depending on your model's setup. For example, DART can use any of the RTTOV clw or ice schemes, but the WRF model is not directly compatible with the IR default cloud classification of marine/continental stratus/cumulus clean/dirty. We also offer a simple classification based on maximum vertical velocity in the column and land type, but due to lack of aerosol information, WRF/DART

cannot differentiate between clean and dirty cumulus. This may have some impact on the forward calculations - but in experience the difference in cloud phase (ice versus water) makes a much larger difference. Trace gases and aerosols may be important for actual observation system experiments using visible/infrared; this may depend on the precise frequencies you wish to use.

Although a model may not have the necessary inputs by itself, the defaults in RTTOV based on climatology can be used. The impact on the quality of the results should be investigated.

Known issues:

- DART does not yet provide any type of bias correction
- Cross-channel error correlations are not yet supported. A principal component approach has been discussed. For now, the best bet is to use a subset of channels that are nearly independent of one another.
- Vertical localization will need to be tuned. Turning off vertical localization may work well if you have a large number of ensemble members. Using the maximum peak of the weighting function or the cloud-top may be appropriate. There are also other potential approaches being investigated.

Author and Contact information:

- DART Code: Jeff Steward
- Original DART/RTTOV work: Nancy Collins, Johnny Hendricks

Backward compatibility note

6.228.2 Other modules used

```
types_mod
utilities_mod
location_mod (threed_sphere)
assim_model_mod
obs_def_utilitie_mod
ensemble_manager_mod
utilities_mod
parkind1 (from RTTOV)
rttov_types (from RTTOV)
obs_kind_mod
```

6.228.3 Public interfaces

use obs_def_rttov_mod, only :	set_visir_metadata
	set_mw_metadata
	get_expected_radiance
	get_rttov_option_logical

Namelist interface `&obs_def_rttov_mod_nml` is read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call set_visir_metadata(key, sat_az, sat_ze, sun_az, sun_ze, & platform_id, sat_id, sensor_id, channel, specularity)

```
integer,  intent(out) :: key
real(r8), intent(in)  :: sat_az
real(r8), intent(in)  :: sat_ze
real(r8), intent(in)  :: sun_az
real(r8), intent(in)  :: sun_ze
integer,  intent(in)  :: platform_id, sat_id, sensor_id, channel
real(r8), intent(in)  :: specularity
```

Visible / infrared observations have several auxillary metadata variables. Other than the key, which is standard DART fare, the RTTOV satellite azimuth and satellite zenith angle must be specified. See the RTTOV user guide for more information (in particular, see figure 4). If the `addsolar` namelist value is set to true, then the solar azimuth and solar zenith angles must be specified - again see the RTTOV user guide. In addition to the platform/satellite/ sensor ID numbers, which are the RTTOV unique identifiers, the channel specifies the chanenl number in the RTTOV coefficient file. Finally, if `do_lambertian` is true, specularity must be specified here. Again, see the RTTOV user guide for more information.

`key`	The DART observation key.
`sat_az`	The satellite azimuth angle.
`sat_ze`	The satellite zenith angle.
`sun_az`	The solar azimuth angle. Only relevant if addsolar is true.
`sun_ze`	The solar zenith angle. Only relevant if addsolar is true.
`platform_id`	The RTTOV platform ID.
`sat_id`	The RTTOV satellite ID.
`sensor_id`	The RTTOV sensor ID.
`channel`	The RTTOV channel number.
`specularity`	The surface specularity. Only relevant if do_lambertian is true.

call set_mw_metadata(key, sat_az, sat_ze, platform_id, sat_id, sensor_id, channel, mag_field, cosbk, fastem_p1, fastem_p2, fastem_p3, fastem_p4, fastem_p5)

```
integer,  intent(out) :: key
real(r8), intent(in)  :: sat_az
real(r8), intent(in)  :: sat_ze
integer,  intent(in)  :: platform_id, sat_id, sensor_id, channel
real(r8), intent(in)  :: mag_field
real(r8), intent(in)  :: cosbk
real(r8), intent(in)  :: fastem_p[1-5]
```

Microwave observations have several auxillary metadata variables. Other than the key, which is standard DART fare, the RTTOV satellite azimuth and satellite zenith angle must be specified. See the RTTOV user guide for more information (in particular, see figure 4). In addition to the platform/satellite/ sensor ID numbers, which are the RTTOV unique

identifiers, the channel specifies the chanenl number in the RTTOV coefficient file. In addition, if use_zeeman is true, the magnetic field and cosine of the angle between the magnetic field and angle of propagation must be specified. See the RTTOV user guide for more information. Finally, the fastem parameters for land must be specified here. This may be difficult for observations to set, so default values (see table 21 in the RTTOV user guide) can be used until a better solution is devised.

key	The DART observation key.
sat_az	The satellite azimuth angle.
sat_ze	The satellite zenith angle.
platform_id	The RTTOV platform ID.
sat_id	The RTTOV satellite ID.
sensor_id	The RTTOV sensor ID.
channel	The RTTOV channel number.
mag_field	The strength of the magnetic field. Only relevant if add_zeeman is true.
cosbk	The cosine of the angle between the magnetic field and direction of EM propagation. Only relevant if add_zeeman is true.
fastem_p[1-5]	The five parameters used for fastem land/sea ice emissivities. For ocean emissivities, an internal model is used based on the value of fastem_version.

call get_expected_radiance(obs_kind_ind, state_handle, ens_size, location, key, val, istatus)

```
integer,             intent(in)  :: obs_kind_ind
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: ens_size
type(location_type), intent(in)  :: location
integer,             intent(in)  :: key
real(r8),            intent(out) :: val(ens_size)
integer,             intent(out) :: istatus(ens_size)
```

Given a location and the state vector from one of the ensemble members, compute the model-predicted satellite observation. This can be either in units of radiance (mW/cm-1/sr/sq.m) or a brightness temperature (in K), depending on if this is a visible/infrared observation or a microwave observation.

obs_kind_ind	The index of the observation kind; since many observation kinds are handled by this module, this can be used to determine precisely which observation kind is being used.
state_handle	The ensemble of model states to be used for the observation operator calculations.
location	Location of this observation
key	Unique identifier associated with this satellite observation
val	The returned observation in units of either radiance or brightness temperature.
istatus	Returned integer status code describing problems with applying forward operator. 0 is a good value; any positive value indicates an error; negative values are reserved for internal DART use only.

p = get_rttov_option_logical(field_name)

```
character(len=*),           intent(in)  :: field_name
logical,                    result      :: p
```

Return the logical value of the RTTOV parameter associated with the field_name.

field_name	The name of the RTTOV parameter from the namelist.
p	The logical return value associated with the parameter.

6.228.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_def_rttov_nml
   rttov_sensor_db_file   = 'rttov_sensor_db.csv'
   first_lvl_is_sfc       = .true.
   mw_clear_sky_only      = .false.
   interp_mode            = 1
   do_checkinput          = .true.
   apply_reg_limits       = .true.
   verbose                = .true.
   fix_hgpl               = .false.
   do_lambertian          = .false.
   lambertian_fixed_angle = .true.
   rad_down_lin_tau       = .true.
   use_q2m                = .true.
   use_uv10m              = .true.
   use_wfetch             = .false.
   use_water_type         = .false.
   addrefrac              = .false.
   plane_parallel         = .false.
   use_salinity           = .false.
   apply_band_correction  = .true.
   cfrac_data             = .true.
   clw_data               = .true.
   rain_data              = .true.
   ciw_data               = .true.
   snow_data              = .true.
   graupel_data           = .true.
   hail_data              = .false.
   w_data                 = .true.
   clw_scheme             = 1
   clw_cloud_top          = 322.
   fastem_version         = 6
   supply_foam_fraction   = .false.
   use_totalice           = .true.
   use_zeeman             = .false.
```

(continues on next page)

(continued from previous page)

```
   cc_threshold           = 0.05
   ozone_data             = .false.
   co2_data               = .false.
   n2o_data               = .false.
   co_data                = .false.
   ch4_data               = .false.
   so2_data               = .false.
   addsolar               = .false.
   rayleigh_single_scatt  = .true.
   do_nlte_correction     = .false.
   solar_sea_brdf_model   = 2
   ir_sea_emis_model      = 2
   use_sfc_snow_frac      = .false.
   add_aerosl             = .false.
   aerosl_type            = 1
   add_clouds             = .true.
   ice_scheme             = 1
   use_icede              = .false.
   idg_scheme             = 2
   user_aer_opt_param     = .false.
   user_cld_opt_param     = .false.
   grid_box_avg_cloud     = .true.
   cldstr_threshold       = -1.0
   cldstr_simple          = .false.
   cldstr_low_cloud_top   = 750.0
   ir_scatt_model         = 2
   vis_scatt_model        = 1
   dom_nstreams           = 8
   dom_accuracy           = 0.0
   dom_opdep_threshold    = 0.0
   addpc                  = .false.
   npcscores              = -1
   addradrec              = .false.
   ipcreg                 = 1
   use_htfrtc             = .false.
   htfrtc_n_pc            = -1
   htfrtc_simple_cloud    = .false.
   htfrtc_overcast        = .false.
/
```

Item	Type	Description
rttov_sensor_db_file	character(len=512)	The location of the RTTOV sensor database. The format for the database is a comma-
first_lvl_is_sfc	logical	Whether the first level of the model represents the surface (true) or the top of the atm
mw_clear_sky_only	logical	If microwave calculations should be “clear-sky” only (although cloud-liquid water ab
interp_mode	integer	The interpolation mode (see the RTTOV user guide).
do_checkinput	logical	Whether to check the input for reasonableness (see the RTTOV user guide).

Item	Type	Description
apply_reg_limits	logical	Whether to clamp the atmospheric values to the RTTOV bounds (see the RTTOV use
verbose	logical	Whether to output lots of additional output (see the RTTOV user guide).
fix_hgpl	logical	Whether the surface pressure represents the surface or the 2 meter value (see the RTT
do_lambertian	logical	Whether to include the effects of surface specularity (see the RTTOV user guide).
lambertian_fixed_angle	logical	Whether to include a fixed angle for the lambertian effect (see the RTTOV user guide
rad_down_lin_tau	logical	Whether to use the linear-in-tau approximation (see the RTTOV user guide).
use_q2m	logical	Whether to use 2m humidity information (see the RTTOV user guide). If true, the QT
use_q2m	logical	Whether to use 2m humidity information (see the RTTOV user guide). If true, the QT
use_uv10m	logical	Whether to use 10m wind speed information (see the RTTOV user guide). If true, the
use_wfetch	logical	Whether to use wind fetch information (see the RTTOV user guide). If true, the QTY
use_water_type	logical	Whether to use water-type information (0 = fresh, 1 = ocean; see the RTTOV user gui
addrefrac	logical	Whether to enable atmospheric refraction (see the RTTOV user guide).
plane_parallel	logical	Whether to treat the atmosphere as plane parallel (see the RTTOV user guide).
use_salinity	logical	Whether to use salinity (see the RTTOV user guide). If true, the QTY_SALINITY wi
apply_band_correction	logical	Whether to apply band correction from the coefficient field for microwave data (see th
cfrac_data	logical	Whether to use the cloud fraction from 0 to 1 (see the RTTOV user guide). If true, th
clw_data	logical	Whether to use cloud-liquid water data (see the RTTOV user guide). If true, the QTY
rain_data	logical	Whether to use precipitating water data (see the RTTOV user guide). If true, the QTY
ciw_data	logical	Whether to use non-precipiting ice information (see the RTTOV user guide). If true, t
snow_data	logical	Whether to use precipitating fluffy ice (see the RTTOV user guide). If true, the QTY_
graupel_data	logical	Whether to use precipting small, hard ice (see the RTTOV user guide). If true, the QT
hail_data	logical	Whether to use precipitating large, hard ice (see the RTTOV user guide). If true, the
w_data	logical	Whether to use vertical velocity information. This will be used to crudely classify if a
clw_scheme	integer	The clw_scheme to use (see the RTTOV user guide).
clw_cloud_top	real(r8)	Lower hPa limit for clw calculations (see the RTTOV user guide).
fastem_version	integer	Which FASTEM version to use (see the RTTOV user guide).
supply_foam_fraction	logical	Whether to use sea-surface foam fraction (see the RTTOV user guide). If true, the QT
use_totalice	logical	Whether to use totalice instead of precip/non-precip ice for microwave (see the RTTO
use_zeeman	logical	Whether to use the Zeeman effect (see the RTTOV user guide). If true, the magnetic
cc_threshold	real(r8)	Cloud-fraction value to treat as clear-sky (see the RTTOV user guide).
ozone_data	logical	Whether to use ozone (O3) profiles (see the RTTOV user guide). If true, the QTY_O
co2_data	logical	Whether to use carbon dioxide (CO2) profiles (see the RTTOV user guide). If true, th
n2o_data	logical	Whether to use nitrous oxide (N2O) profiles (see the RTTOV user guide). If true, the
co_data	logical	Whether to use carbon monoxide (CO) profiles (see the RTTOV user guide). If true, t
ch4_data	logical	Whether to use methane (CH4) profiles (see the RTTOV user guide). If true, the QTY
so2_data	logical	Whether to use sulfur dioxide (SO2) (see the RTTOV user guide). If true, the QTY_S
addsolar	logical	Whether to use solar angles (see the RTTOV user guide). If true, the sun_ze and sun_
rayleigh_single_scatt	logical	Whether to use only single scattering for Rayleigh scattering for visible calculations (
do_nlte_correction	logical	Whether to include non-LTE bias correction for HI-RES sounder (see the RTTOV use
solar_sea_brdf_model	integer	The solar sea BRDF model to use (see the RTTOV user guide).
ir_sea_emis_model	logical	The infrared sea emissivity model to use (see the RTTOV user guide).
use_sfc_snow_frac	logical	Whether to use the surface snow fraction (see the RTTOV user guide). If true, the QT
add_aerosl	logical	Whether to use aerosols (see the RTTOV user guide).
aerosl_type	integer	Whether to use OPAC or CAMS aerosols (see the RTTOV user guide).
add_clouds	logical	Whether to enable cloud scattering for visible/infrared (see the RTTOV user guide).
ice_scheme	integer	The ice scheme to use (see the RTTOV user guide).
use_icede	logical	Whether to use the ice effective diameter for visible/infrared (see the RTTOV user gu
idg_scheme	integer	The ice water effective diameter scheme to use (see the RTTOV user guide).
user_aer_opt_param	logical	Whether to directly specify aerosol scattering properties (see the RTTOV user guide)

Item	Type	Description
user_cld_opt_param	logical	Whether to directly specify cloud scattering properties (see the RTTOV user guide).
grid_box_avg_cloud	logical	Whether to cloud concentrations are grid box averages (see the RTTOV user guide).
cldstr_threshold	real(r8)	Threshold for cloud stream weights for scattering (see the RTTOV user guide).
cldstr_simple	logical	Whether to use one clear and one cloudy column (see the RTTOV user guide).
cldstr_low_cloud_top	real(r8)	Cloud fraction maximum in layers from the top of the atmosphere down to the specifi
ir_scatt_model	integer	Which infrared scattering method to use (see the RTTOV user guide).
vis_scatt_model	integer	Which visible scattering method to use (see the RTTOV user guide).
dom_nstreams	integer	The number of streams to use with DOM (see the RTTOV user guide).
dom_accuracy	real(r8)	The convergence criteria for DOM (see the RTTOV user guide).
dom_opdep_threshold	real(r8)	Ignore layers below this optical depth (see the RTTOV user guide).
addpc	logical	Whether to do principal component calculations (see the RTTOV user guide).
npcscores	integer	Number of principal components to use for addpc (see the RTTOV user guide).
addradrec	logical	Reconstruct the radiances using addpc (see the RTTOV user guide).
ipcreg	integer	Number of predictors to use with addpc (see the RTTOV user guide).
use_htfrtc	logical	Whether to use HTFRTC (see the RTTOV user guide).
htfrtc_n_pc	integer	Number of PCs to use with HTFRTC (see the RTTOV user guide).
htfrtc_simple_cloud	logical	Whether to use simple cloud scattering with htfrtc (see the RTTOV user guide).
htfrtc_overcast	logical	Whether to calculate overcast radiances with HTFRTC (see the RTTOV user guide).

6.228.5 Files

- A DART observation sequence file containing Radar obs.

6.228.6 References

- RTTOV user guide

6.228.7 Private components

use obs_def_rttov_mod, only :	initialize_module
	initialize_rttov_sensor_runtime
	initialize_rttov_sensor_runtime

call initialize_module()

Reads the namelist, allocates space for the auxiliary data associated wtih satellite observations, initializes the constants used in subsequent computations (possibly altered by values in the namelist), and prints out the list of constants and the values in use.

call initialize_rttov_sensor_runtime(sensor,ens_size,nlevels)

```
type(rttov_sensor_type), pointer    :: sensor
integer,                 intent(in) :: ens_size
integer,                 intent(in) :: nlevels
```

Initialize a RTTOV sensor runtime. A rttov_sensor_type instance contains information such as options and coefficients that are initialized in a "lazy" fashion only when it will be used for the first time.

sensor	The sensor type to be initialized
ens_size	The size of the ensemble
nlevels	The number of vertical levels in the atmosphere

6.228.8 Error codes and conditions

Routine	Message	Comment
initialize_module	initial allocation failed for satellite observation data	Need to increase MAXrttovkey
initialize_rttov_sensor_runtime	Module or sensor is not initialized	Both the module and the sensor must be initialized before calling this routine.
get_visir_metadata	The key exceeds the size of the metadata arrays, or the key is not a VIS/IR type	The number of satellite observations exceeds the array size allocated in the module. Check the input and/or increase MAXrttovkey.
get_mw_metadata	The key exceeds the size of the metadata arrays, or the key is not a MW type	The number of satellite observations exceeds the array size allocated in the module. Check the input and/or increase MAXrttovkey.
read_rttov_metadata	bad value for RTTOV fields	The format of the input obs_seq file is not consistent.
get_expected_radiance	Could not find the platform/satellite/sensor id combination in the RTTOV sensor database file.	An unknown RTTOV instrument ID was encountered. Check the database and/or the observation metadata.

6.229 Multi-Component CESM+DART Setup

6.229.1 CESM+DART setup overview

If you found your way to this file without reading more basic DART help files, please read those first. *Getting Started* is a good place to find pointers to those files. Then see *CESM* for an overview of DART's interfaces to CESM. Finally, see the ../../{your_cesm_component(s)}/readme.html documentation about the code-level interfaces and namelist values for various CESM component models. This document gives specific help in setting up a CESM+DART assimilation for the first time.

Warning: The scripts for multi-component assimilation were developed in the context of DART's Lanai release (or earlier) and CESM1. They won't work in later versions. The instructions below should be considered a template for setting up and running a multi-component assimilation, to be modified as needed. Some of them reference code that may be found in $DART/models/cam or cam-old.

The overall strategy is to set up an environment where;

- CESM is set up as a ''B" component set configuration (''fully coupled" = active atmosphere, ocean, land, and possibly others)
- a separate assimilation can be run using each component model interface for which there are observations.

Each CESM hindcast advances all of the active components model states, which are then used by the several `filter` programs. So you will need to build separate filters in the models/{your_CESM_component_models}/work directories. You will also need to assemble an initial ensemble of CESM files, which consists of restart and initial files for all of the active components. Each filter will read a separate observation sequence file.

6.229.2 Assimilation set-up procedure

Here is a list of steps to set up an assimilation. It assumes you have downloaded DART and learned how to use it with low order models. Some of the steps can be skipped if you have a suitable replacement, as noted.

1. Decide which component(s) you want to use as the assimilating model(s). (The rest of this list assumes that you're building a cam-fv assimilation, as an example. Steps will need to be repeated for your other models.)
2. Look in models/{your_models}/shell_scripts to see which CESM versions are supported.
3. CESM: locate that version on your system, or check it out from http://www.cesm.ucar.edu/models/current.html
4. Choose the options in $dart/mkmf/mkmf.template that are best for your assimilation. These will not affect the CESM build, only filter.
5. In models/cam-fv/work/input.nml, be sure to include all of your required obs_def_${platform}_mod.f90 file names in preprocess_nml:input_files. It's also convenient to modify the rest of input.nml to make it do what you want for the first assimilation cycle. That may include creating spread in the initial ensemble by perturbing it. Input.nml will be copied to the $CASEROOT directory and used by assimilate.csh. That copy can be modified for whichever cycles will be run next.
6. Build the DART executables using quickbuild.sh.
7. Follow the directions in CESM/shell_scripts/*setup* to set up the CESM case and integrate DART into it. The DART team recommends a tiny ensemble to start with, to more quickly test whether everything is in order.
8. Choose a start date for your assimilation. Choosing/creating the initial ensemble is a complicated issue.
   - It's simpler for CAM assimilations. If you don't have an initial state and/or ensemble for this date, build a single instance of CESM (Fxxxx compset for cam-fv) and run it from the default Jan 1 start date until 2 weeks before your start date. Be sure to set the cam namelist variable inithist = 'ENDOFRUN' during the last stage, so that CAM will write an "initial" file, which DART needs.
   - For ocean and land assimilations, which cannot spin up as quickly as the atmosphere, creating usable initial ensemble is a more complicated process. See those models' readme files.

9. In the CESM run directory, create a cam-fv ensemble (virtual in the case of a single instance) by linking files with instance numbers in them to the restart file set (which may have no instance number) using CESM/shell_scripts/link_ens_to_single.csh.
10. Link the other model's restart file sets into the run directory (also possibly using link_ens_to_single.csh).
11. After convincing yourself that the CESM+DART framework is working with no_assimilate.csh, activate the assimilation by changing CESM's env_run.xml:DATA_ASSIMILATION_SCRIPT to use assimilate.csh.
12. After the first hindcast+assimilation cycle finishes correctly, change the input.nml, env_run.xml and env_batch.xml to do additional cycle(s) without the perturbation of the initial state, and with using the restart files just created by the first cycle. You may also want to turn on the st_archive program. Instructions are in setup_hybrid and cam-fv/work/input.nml.
13. Finally, build a new case with the full ensemble, activate the assimilate.csh script and repeat the previous item.

6.229.3 Output directory

CESM's short term archiver (case.st_archive) is controlled by its `env_archive.xml`. DART's setup scripts modify that file to archive DART output along with CESM's. (See the ../../../guide/controlling-files-output.html for a description of DART's output). DART's output is archived in `$arch_dir/esp/{hist,rest,logs,...}`, where arch_dir is defined in `setup_{hybrid,advanced}`, `hist` contains all of the state space and observation space output, and `rest` contains the inflation restart files.

The cam-XX assimilate.csh script may make a copy of its obs_seq.final files in a scratch space ($scratch/$case/Obs_seqs) which won't be removed by assimilate.csh.

6.229.4 Shell_scripts for building and running multi-component assimilations

These scripts are outdated relative to Manhattan (path names, batch submission, long-term archiver, . . .), but can serve as a template for multi-component assimilations.

CESM1_1_1_setup_pmo

- set up, stage, and build a single-instance, B compset configuration of CESM.
- The initial state can come from any single member of a reference case.
- Synthetic observations are harvested from the CESM model states.

CESM1_1_1_setup_hybrid

- Set up, stage, and build an ensemble assimilation
- using a B compset configuration of CESM.
- The initial states come from a single, multi-instance, reference case

CESM1_1_1_setup_special

- Same as CESM1_1_1_setup_hybrid, but the initial states for the 5 active models
- come from up to 5 sources:
- The ICs source directories need to be updated.

CESM1_1_1_setup_initial

- Same as CESM1_1_1_setup_hybrid, but fewer comments and error checks.

CESM1_2_1_setup_pmo

- Same as CESM1_2_1_setup_hybrid, but for _pmo.

CESM1_2_1_setup_hybrid

- Same as CESM1_1_1_setup_hybrid, but updated to accommodate CESM's wave and land ice models.
- (DART has no interfaces for those components). Somewhat different handling of SourceMods.

CESM_DART_config

- Integrates DART into a pre-existing CESM case, either single- or multi-instance.
- Typically run by or after one of the _setup_ scripts.

perfect_model.csh

- Run by the CESM $CASE.run batch job, which was created by . . . setup_**pmo** .
- Can call the [component]_perfect_model.csh script for each component which will be used for assimilation.

{cam,pop,clm}_perfect_model.csh

- Runs perfect_model_obs_{cam,pop,clm}

assimilate.csh

- Run by the CESM $CASE.run batch job, which was created by . . . setup_{**hybrid,initial,special**}.
- Can call the assimilate.csh script for each component which will be used for assimilation.
- See [component]_assimilate.csh below (which were derived from $DART/models/[component]/shell_scripts/. . . /assimilate.csh

cam_assimilate.csh

- Sets up and runs filter for CAM and related observations.
- Uses cam_to_dart and dart_to_cam, which are not used in the Manhattan release and later.

clm_assimilate.csh

- similar to cam_assimilate.csh

pop_assimilate.csh

- similar to cam_assimilate.csh

no_assimilate.csh

- The script used as a placeholder in the CESM run scripts when a case is set up.

cam_no_assimilate.csh

- The CAM no_assimilate script needs to make an initial file available for the next CAM hindcast.

run_perfect_model_obs.csh

- Batch script to run perfect_model_obs for POP (only!)

CLM_convert_restarts.csh

- Converts 'old' CLM restart files to whatever resolution you like.

link_ens_to_single.csh

- Helper script to generate a virtual ensemble from a single instance (member).

st_archive.sh

- A CESM archiving script, modified to handle DART output files.

6.229.5 Helpful hints

You will probably want to use your computer resources efficiently. In addition to the Tips and Warnings in ../readme.html, The DART team recommends:

- Experiment with a single instance CASE to learn the smallest number of nodes on which it will run reliably. Strange andor nonreproducible errors often are the result of giving insufficient memory to the job. (node = several to dozens of central processing units which share memory in ways that allow very fast communication). Build the multi-instance case using that number of nodes per instance. This has 2 benefits; it minimizes queue wait times, and it minimizes internode communication, which can increase exponentially with the number of nodes used.
- Carefully select the output to be saved and the archiving frequency. Output from large ensemble, large model assimilations can quickly fill the available disk space, resulting in an ugly ending to your job, from which it is time consuming to recover; discarding the partial files and keeping the output needed for evaluation and restarting the assimilation.
- Evaluate the output frequently to determine whether it is worthwhile to continue. Looking at the model output in its gridded form can be useful, but the DART team has learned that you can do a much more thorough and efficient evaluation in "observation space", using obs_diag and scripts in "$DART/diagnostics/matlab" described in the Observation Space

There are, no doubt, things missing from these lists, so don't struggle too long before contacting dart'at'ucar.edu.

6.230 PROGRAM `create_ocean_obs`

`create_ocean_obs` is responsible for converting an interim ASCII file of ocean observations into a DART observation sequence file. The interim ASCII file is a simple 'whitespace separated' table where each row is an observation and each column is specific information about the observation.

column number	quantity	description
1	longitude (in degrees)	longitude of the observation
2	latitude (in degrees)	latitude of the observation
3	depth (in meters)	depth of the observation
4	observation value	such as it is …
5	vertical coordinate flag	There is a pathological difference between a surface observation and an observation with a depth of zero. See location_mod:location_type for a full explanation. The short explanation is that *surface == -1*, and *depth == 3*
6	observation variance	good luck here …
7	Quality Control flag	integer value passed through to DART. There is a namelist parameter for `filter` to ignore any observation with a QC value <= input_qc_threshold
8	obs_kind_name	a character string that must match a string in *MODULE obs_def_ocean_mod*
9	startDate_1	the year-month-date of the observation (YYYYMMDD format)
10	startDate_2	the hour-minute-second of the observation (HHMMSS format)

For example:

```
273.7500 21.3500 -2.5018 28.0441  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.4500 -2.5018 28.1524  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.5500 -2.5018 28.0808  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.6500 -2.5018 28.0143  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.7500 -2.5018 28.0242  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.8500 -2.5018 28.0160  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.9500 -2.5018 28.0077  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 22.0500 -2.5018 28.3399  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 22.1500 -2.5018 27.8852  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 22.2500 -2.5018 27.8145  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
...
```

It is always possible to combine observation sequence files with the program *program obs_sequence_tool*, so it was simply convenient to generate a separate file for each observation platform and type ('GLIDER' and 'TEMPERATURE'), however it is by no means required.

6.230.1 Modules used

Some of these modules use modules ... **those** modules and namelists are not discussed here. probably should be ...

```
types_mod
utilities_mod
dart_MITocean_mod
obs_sequence_mod
```

6.230.2 Namelist

This program has a namelist of its own, and some of the underlying modules require namelists. To avoid duplication and, possibly, some inconsistency in the documentation; only a list of the required namelists is provided - with a hyperlink to the full documentation for each namelist.

Namelist	Primary Purpose
utilities_nml	set the termination level and file name for the run-time log
obs_sequence_nml	write binary or ASCII observation sequence files

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/'. Consider yourself forewarned that filenames that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
namelist /create_ocean_obs_nml/  year, month, day, &
         tot_days, max_num, fname, output_name, lon1, lon2, lat1, lat2
```

This namelist is read in a file called `input.nml`

Contents	Type	Description
year	integer *[default: 1996]*	The first year of interest.
month	integer *[default: 1]*	The first month of interest.
day	integer *[default: 1]*	The first day of interest.
tot_days	integer *[default: 31]*	Stop processing after this many days.
max_num	integer *[default: 800000]*	The maximum number of observations to read/write.
fname	character(len=129) *[default: 'raw_ocean_obs.txt']*	The name of the interim ASCII file of observations.
output_name	character(len=129) *[default: 'raw_ocean_obs_seq.out']*	The output file name.
lon1	real *[default: 0.0]*	The leftmost longitude of interest.
lon2	real *[default: 360.0]*	The rightmost longitude of interest.
lat1	real *[default: -90.0]*	The most southern latitude of interest.
lat2	real *[default: 90.0]*	The most northern latitude of interest.

6.230.3 Files

- input namelist file: `input.nml`
- input data file: as listed by `input.nml&create_ocean_obs_nml:fname`
- output data file: as listed by `input.nml&create_ocean_obs_nml:output_name`

6.230.4 References

- none

6.231 PROGRAM `dart_to_ncommas`

Attention: `NCOMMAS` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `NCOMMAS` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

`dart_to_ncommas` is the program that **updates** a ncommas netCDF-format restart file (usually `ncommas_restart.nc`) with the state information contained in a DART output/restart file (e.g. `perfect_ics, filter_ics, ...`). Only the CURRENT values in the ncommas restart file will be updated. The DART model time is compared to the time in the ncommas restart file. If the last time in the restart file does not match the DART model time, the program issues an error message and aborts.

From the user perspective, most of the time `dart_to_ncommas` will be used on DART files that have a header containing one time stamp followed by the model state.

The dart_to_ncommas_nml namelist allows `dart_to_ncommas` to read the `assim_model_state_ic` files that have *two* timestamps in the header. These files are temporarily generated when DART is used to advance the model. One timestamp is the 'advance_to' time, the other is the 'valid_time' of the model state. In this case, a namelist for ncommas (called `ncommas_in.DART`) is written that contains the `&time_manager_nml` settings appropriate to advance ncommas to the time requested by DART. The repository version of the `advance_model.csh` script has a section to ensure the proper DART namelist settings for this case.

Conditions required for successful execution of `dart_to_ncommas`:

- a valid `input.nml` namelist file for DART
- a valid `ncommas_vars.nml` namelist file for ncommas - the same one used to create the DART state vector, naturally,
- a DART file (typically `filter_restart.xxxx` or `filter_ics.xxxx`)
- a ncommas restart file (typically `ncommas_restart.nc`).

Since this program is called repeatedly for every ensemble member, we have found it convenient to link the DART input file to the default input filename (`dart_restart`). The same thing goes true for the ncommas output filename `ncommas_restart.nc`.

6.231.1 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   ncommas_restart_filename     = 'ncommas_restart.nc';
   assimilation_period_days     = 1,
   assimilation_period_seconds  = 0,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .true.,
   calendar                     = 'Gregorian',
   debug                        = 0
/
```

```
&dart_to_ncommas_nml
   dart_to_ncommas_input_file = 'dart_restart',
   advance_time_present    = .false.
/
```

`dart_to_ncommas_nml` and `model_nml` are always read from a file called `input.nml`. The full description of the `model_nml` namelist is documented in the NCOMMAS model_mod.

Item	Type	Description
dart_to_ncommas_input_file	character(len=128)	The name of the DART file containing the model state to insert into the ncommas restart file.
advance_time_present	logical	If you are converting a DART initial conditions or restart file this should be `.false.`; these files have a single timestamp describing the valid time of the model state. If `.true.` TWO timestamps are expected to be the DART file header. In this case, a namelist for ncommas (called `ncommas_in.DART`) is created that contains the `&time_manager_nml` settings appropriate to advance ncommas to the time requested by DART.

`ncommas_vars_nml` is always read from a file called `ncommas_vars.nml`.

Item	Type	Description
ncommas_state_variables	character(len=NF90_MAX_NAME) :: dimension(160)	The list of variable names in the NCOMMAS restart file to use to create the DART state vector and their corresponding DART kind.

```
&ncommas_vars_nml
   ncommas_state_variables = 'U',    'QTY_U_WIND_COMPONENT',
                             'V',    'QTY_V_WIND_COMPONENT',
                             'W',    'QTY_VERTICAL_VELOCITY',
                             'TH',   'QTY_POTENTIAL_TEMPERATURE',
                             'DBZ',  'QTY_RADAR_REFLECTIVITY',
                             'WZ',   'QTY_VERTICAL_VORTICITY',
                             'PI',   'QTY_EXNER_FUNCTION',
                             'QV',   'QTY_VAPOR_MIXING_RATIO',
                             'QC',   'QTY_CLOUDWATER_MIXING_RATIO',
                             'QR',   'QTY_RAINWATER_MIXING_RATIO',
                             'QI',   'QTY_ICE_MIXING_RATIO',
                             'QS',   'QTY_SNOW_MIXING_RATIO',
                             'QH',   'QTY_GRAUPEL_MIXING_RATIO'
  /
```

6.231.2 Modules used

```
assim_model_mod
location_mod
model_mod
null_mpi_utilities_mod
obs_kind_mod
random_seq_mod
time_manager_mod
types_mod
utilities_mod
```

6.231.3 Files read

- DART initial conditions/restart file; e.g. `filter_ic`
- DART namelist file; `input.nml`
- ncommas namelist file; `ncommas_vars.nml`
- ncommas restart file `ncommas_restart.nc`

6.231.4 Files written

- ncommas restart file; `ncommas_restart.nc`
- ncommas namelist file; `ncommas_in.DART`

6.231.5 References

none

6.232 PROGRAM `ncommas_to_dart`

Attention: `NCOMMAS` works with versions of DART *before* Manhattan (9.x.x) and has yet to be updated. If you are interested in using `NCOMMAS` with more recent versions of DART, contact DAReS staff to assess the feasibility of an update. Until that time, you should consider this documentation as out-of-date.

`ncommas_to_dart` is the program that reads a ncommas restart file (usually `ncommas_restart.nc`) and creates a DART state vector file (e.g. `perfect_ics, filter_ics, ...`).
The list of variables used to create the DART state vector are specified in the `ncommas_vars.nml` file.
Conditions required for successful execution of `ncommas_to_dart`:

- a valid `input.nml` namelist file for DART
- a valid `ncommas_vars.nml` namelist file for ncommas
- the ncommas restart file mentioned in the `input.nml&model_nml:ncommas_restart_filename` variable.

Since this program is called repeatedly for every ensemble member, we have found it convenient to link the ncommas restart files to the default input filename (`ncommas_restart.nc`). The default DART state vector filename is `dart_ics` - this may be moved or linked as necessary.

6.232.1 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   ncommas_restart_filename     = 'ncommas_restart.nc';
   assimilation_period_days     = 1,
   assimilation_period_seconds  = 0,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .true.,
   calendar                     = 'Gregorian',
   debug                        = 0
/
```

```
&ncommas_to_dart_nml
   ncommas_to_dart_output_file = 'dart_ics'
/
```

ncommas_to_dart_nml and model_nml are always read from a file called input.nml. The full description of the model_nml namelist is documented in the NCOMMAS model_mod.

Item	Type	Description
ncommas_to_dart_output_file	character(len=128)	The name of the DART file which contains the updated model state info that should be written into the NCOMMAS file.

ncommas_vars_nml is always read from a file called ncommas_vars.nml.

Item	Type	Description
ncommas_state_variables	character(len=NF90_MAX_NAME) :: dimension(160)	The list of variable names in the NCOMMAS restart file to use to create the DART state vector and their corresponding DART kind.

```
&ncommas_vars_nml
   ncommas_state_variables = 'U',    'QTY_U_WIND_COMPONENT',
                             'V',    'QTY_V_WIND_COMPONENT',
                             'W',    'QTY_VERTICAL_VELOCITY',
                             'TH',   'QTY_POTENTIAL_TEMPERATURE',
                             'DBZ',  'QTY_RADAR_REFLECTIVITY',
                             'WZ',   'QTY_VERTICAL_VORTICITY',
                             'PI',   'QTY_EXNER_FUNCTION',
                             'QV',   'QTY_VAPOR_MIXING_RATIO',
                             'QC',   'QTY_CLOUDWATER_MIXING_RATIO',
                             'QR',   'QTY_RAINWATER_MIXING_RATIO',
                             'QI',   'QTY_ICE_MIXING_RATIO',
                             'QS',   'QTY_SNOW_MIXING_RATIO',
                             'QH',   'QTY_GRAUPEL_MIXING_RATIO'
  /
```

6.232.2 Modules used

```
assim_model_mod
location_mod
model_mod
null_mpi_utilities_mod
obs_kind_mod
random_seq_mod
time_manager_mod
types_mod
utilities_mod
```

6.232.3 Files read

- ncommas restart file; `ncommas_restart.nc`
- DART namelist files; `input.nml` and `ncommas_vars.nml`

6.232.4 Files written

- DART state vector file; e.g. `dart_ics`

6.232.5 References

none

6.233 Copyright

Copyright 2021 University Corporation for Atmospheric Research

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

6.234 Changelog

This file documents the most user-visible changes to the DART code. It is not intended to document every change, but instead is intended to inform people what features are now available or have been removed. Detailed changes are always available through the version control framework.

DART now uses Git for version control but has preserved the revision history from when subversion (and CVS before that) was used. The previous revision numbers can be related to git hashes by searching the output of `git log`

```
0[1011] machine:DART % git log > full_git_log.txt
```

A reminder that since many files were moved or renamed, the best way to get the complete log is to use `git log --follow` for information on individual files.

The changes are now listed with the most recent at the top.

December 12 2022 :: Automated testing of pull requests

- GitHub actions for pull requests which checkout, compile and run a given model. Current workflow: lorenz_96 (mpi) and lorenz_63 (no mpi)

contributed by Anderson Chauphan

December 2 2022 :: Bug-fix cam-fv. Tag v10.5.6

- Fix for assimilate.csh purge of restart files when the interval for restart saves is given as a string rather than an integer.
- Fix for setting ptype when no_normalization_of_scale_heights = .false.

November 8 2022 :: Improved clean_nml and CLM quickbuild.sh. Tag v10.5.5

- clean_nml tool for comparing input.nmls given optional arguments to keep the original order of nmls and/or entries, and optionally keep namelist comments.
- fill_inflation_restart now a default build for CLM.

November 3 2022 :: Bug-fix release. Tag v10.5.4

- Perfect_model_obs (pmo) fixed for running with MPI and advancing the model inside pmo.
- MPAS_ATM xtime string padded with blanks for easier bitwise comparison.
- lorenz_96_tracer_advection quickbuild.sh fixed.

October 13 2022 :: Bug-fix for read variables. Tag v10.5.3

- Per-file check for unlimited dimension before variable read. Netcdf dimension counts adjusted accordingly. Fixes problems when reading from DART created netcdf files, for example, from fill_inflation_restart
- Bug-fix for verbose printing of state_structure info

October 10 2022 :: Bug-fix for obs_converter builds. Tag v10.5.2

- Bug fix for converter builds using the template model_mod.f90
- Performance fix for MPAS_ATM

September 23 2022 :: Bug-fix for pertub_single_instance. Tag v10.5.1

- Program perturb_single_instance was running without perturbing when interf_provided = .false. (silent fail). Model specific pert_model_copies is now required to run perturb_single_instance

September 22 2022 :: CAM-SE. Tag: v10.5.0

- CAM-SE interface for Manhattan
- Shared code for CAM-SE and CAM-FV model_mods in cam-common-code

September 21 2022 :: ROMS model_mod perturbation routine Tag: v10.4.0

- Adds a pert_model_copies subroutine to the ROMS model_mod to enable proper functioning of perturb_single_instance for ROMS.
- Updates ROMS documentation with an explanation of how to generate an initial ensemble of history files.

September 16 2022 :: Bug-fix for CLM shell scripts. Tag: v10.3.2

- Fixes dtlimit bug in cesm2.2 CLM shell scripts. Loads all CAM reanalysis files within datm data stream file regardless if it is single year or multi-year run.

- Fixes single instance bug in cems2.2 CLM shell scripts. Allows for freerun scripts to generate single instance simulations.

September 14 2022 :: Bug-fix for POP shell scripts. Tag: v10.3.1

- Fixes bug in POP CESM2.1 shell scripts in which inflation files were not being propagated properly due to link destination already existing.

August 19 2022 :: Automated setup of new model interfaces. Tag: v10.3.0

- Automated initial setup of new model interfaces to aid users developing model_mod code and documentation.

contributed by Benjamin Gunn

August 18 2022 :: Bug-fixes for obs_utilities build and mpas_atm. Tag: v10.2.1

- obs_utilities_mod no longer included by default for model/work builds because these utilities are for threed_sphere and threed_cartesian location_mods only.
- mpas_atm model_mod check for required quantities changed to handle multiple variables of the same quantity.

August 3 2022 :: TIEGCM. Tag v10.2.0

- TIEGCM model_mod updated to Manhattan
- Added hyperslice subroutine to state_structure_mod to read only part of a netcdf variable into the state.
- Replaced stub get_expected_vtec with COMMON_CODE.

August 2 2022 :: RTTOV v13. Tag v10.1.0

- Support for RTTOV v13

Contributed by Lukas Kugler

July 27 2022 :: Bug-fix for WRF model_mod nc_write_model_atts. Tag: v10.0.5

- Removes obsolete domain dimension from nc_write_model_atts for WRF model_mod. This code was left over from when multiple domains could be written to a single NetCDF file in DART.

July 21 2022 :: Bug-fixes for DART_LAB rank histograms and get_close caching. Tag: v10.0.4

- Fixes DART_LAB prior and posterior rank histogram calculation.
- Fix for intent(inout) for get_close_X_caching routines.
- Obsolete GitHub workflow removed.

July 14 2022 :: Performance improvement - removal of redundant caching. Tag: v10.0.3

- Reduces the runtime by removing redundant caching in the get_close_obs_cached and get_close_state_cached subroutines in assim_tools_mod.f90

June 24 2022 :: Bug-fixes for MITgcm_ocean and Var obs converter. Tag: v10.0.2

- MITgcm_ocean pert_model_copies routine fixed to use the correct variable clamping value and indices for each element of the copies array.
- Var obs converter quicklbuild.sh fixed to correctly locate the required 3DVAR_OBSPROC code.
- Documentation for Var obs converter updated with information for where to get the latest WRF 3DVAR_OBSPROC code.

June 2 2022 :: Bug-fixes for ps_rand_local in the Bgrid Model. Tag: v10.0.1

- performs the missing call for initialize_utilities()
- fixes improper formatting when writing into ps_rand.out

May 24 2022 :: New build tools for DART. Tag: v10.0.0

- mkmf_ and path_names_ files replaced with buildfunctions to collect source code.
- quickbuild.csh replaced with quickbuild.sh
- developer_tests/build_everything for simultaneous runs of every quickbuild.sh
- Several build/compilation related bug fixes

May 23 2022 :: Bug-fix for RTPS inflation flavor. Tag: v9.16.4

- Order of operations changed to avoid inadvertent changes to ens when using RTPS.

May 16 2022 :: Installation documentation update. Tag: v9.16.3

- Improved installation documentation.

April 5 2022 :: Bug-fix for NetCDF variables with NaN attributes. Tag: v9.16.2

- Fix for checking attributes of NetCDF variables that have a NaN as the missing or _FillValue.

April 1 2022 :: Per-obs-type localization for 3D Cartesian location_mod. Tag: v9.16.1

- Optional per-obs-type localization for 3D Cartesian location

Contributed by Jon Labriola for use with CM1

March 31 2022 :: MiTgcm-ocean NBLING. Tag: v9.16.0

- MITgcm-ocean interface updated to Manhattan.
- Support for NBLING (ocean biogeochemistry) for MITgcm-ocean.
- New observation converter for ocean color.

March 22 2022 :: CLM5-DART Tutorial. Tag: v9.15.0

- New CLM5-DART tutorial providing new users with 13 steps for download, setup, execution, and diagnosis of a simple global assimilation run.
- Stage_cesm_files script corrected to re-stage prior inflation files correctly and provide reproducibility

March 10 2022 :: Tracer advection model. Tag v9.14.0

- New model for tracer advection based on Lorenz_96 using a Semi-Lagrangian scheme.
- Forward operator documentation updated to Manhattan

lorenz_96_tracer_advection contributed by Fairuz Ishraque (SIParCS)

February 11 2022 :: Bug-fix for nag compiler. Tag v9.13.2

- Bug-fix for fixsystem preprocessing for NAG compiler

February 7 2022 :: CM1 and 3D Cartesian location_mod updates. Tag v9.13.1

Contributed by Jon Labriola

- Updated CM1 model_mod to use mixed-case boundary conditions, for example periodic in the x-direction but non-periodic in the y-direction.
- Added capability to CM1 model_mod to interpolate 3D fields such as reflectivity.
- Added capability to use multiple localization radii to threed_cartisian location_mod.
- Bug-fix for threed_cartesian location_mod for periodic boundaries.

February 3 2022 :: CLM with SWE repartitioning. Tag: v9.13.0

- Updated Community Land Model (CLM) model_mod, scripting, and diagnostics.

- New capability to repartition snow layer mass and dimension variables. The repartitioning ensures that the adjustments applied to the snow layers are consistent (mass and dimensions are conserved) with the adjustment of the total snow water equivalent.
- New observation converters for NASA_Earthdata and NSIDC
- Support for netcdf _FillValue
- Bug-fix for clm nc_write_model_atts lon vs nlon

January 11 2022 :: Bug fix for inflation namelist options. Tag: v9.12.1

- reverted inf_flavor namelist option to be integer only to conform to Fortran standards.

December 7 2021 :: Refactored filter_assim. Tag: v9.12.0

- Filter_assim refactored so each process calcuates increments
- Code readability changes

November 22 2021 :: Bug fix for groups with posterior spatially-varying adaptive inflation. Tag: v9.11.13

- Removed the additional outlier threshold test for each group when using posterior spatially-varying adaptive inflation. The outlier test is done for the entire ensemble when the posterior forward operators are computed.

October 27 2021 :: Observation converter documentation update. Tag: v9.11.12

- Improved documentation for radar observation converters

September 30 2021 :: Bug fix for very large models. Tag v9.11.11

- mpi_utilties_mod using correct check for message length > SNDRCV_MAXSIZE
- new developers test for large message sizes
- State vector IO updated to use i8 for state indexing
- WRF model_mod now using i8

September 21 2021 :: Bug fix for perfect_model_obs. Tag v9.11.10

- perfect_model_obs now exits cleanly when no filenames are given for the input_state_files or output_state_files namelist options.

August 30 2021 :: Repository clean-up and DART registration form. Tag: v9.11.9

removed obsolete code:

- cam-old
- restart_file_tool
- html boilerplate from pre-readthedocs documentation
- null versions of clamp_mod and vert_convert_mod
- io test harnesses
- jekyll website

removed experimental code:

- pnetcdf (for restart files)
- filter.separate_seq (split obs_seq across cores)

August 26 2021 :: NAG compiler fixes and updates to developer tests Tag: v9.11.8

- bug fix for fixsytem for the NAG compiler

- new developer test for mpi one-sided communication
- removed obsolete async 4 developer tests

August 19 2021 :: WRF-Hydro diagnostics Tag: v9.11.7

- Improved DART diagnostic routines for WRF-Hydro

August 10 2021 :: Documentation and GitHub template update Tag: v9.11.6

- External forward operator documentation
- Typo fixes for GitHub templates

August 5 2021 :: bug fix for obs_seq_to_netcdf and grabbufr.x Tag: v9.11.5

- obs_seq_to_netcdf now works correctly with mulitple obs_seq per epoch.
- grabbufr.x STAT function returns correctly for long filenames when using PGI

July 23 2021 :: bug fix for wrf non-initialized unique levels. Tag: v9.11.4

- The array uniquek is now initialized to an invalid level to prevent random reasonable level values in the array.

June 24 2021 :: bug fix for cam-fv model_interpolate. Tag: v.9.11.3

- cam-fv model_interpolate now passes the correct array slice of quad_vals to quad_lon_lat_evaluate

June 24 2021 :: latest version of local particle filter. Tag: v9.11.2

- latest version of particle filter from Jon Potterjoy
- new mpi routine get_global_max

June 18 2021 :: build fixes for PGI compiler and intel compiler osx. Tag: v9.11.1

- mkmf.templates fixes for intel.osx and pgi
- input.nml fix for obs_total_error
- path_names fix for test_quad_*_interp

June 8 2021 :: New observation converter for Solar Induced Fluorescence (SIF). Tag: v9.11.0

- Converter for harmonized SIF retrievals

Jun 7 2021 :: fix typos in POP documentation Tag: v9.10.6

- fix some spelling mistakes, does not change meaning.

May 18 2021 :: updated process to generate obs on a sphere. Tag: v9.10.5

- Matlab scripts and new scripts for cam-fv make it simpler to generate synthetic observations evenly spaced around the sphere.
- Moved create_sphere_obs into the even_sphere directory.

May 10 2021 :: obs_info support for identity obs Tag: v9.10.4

- programs/obs_utils/obs_info.f90 now supports reporting identity obs

May 6 2021 :: fix AMSUA converter bug. Tag: v9.10.3

- AIRS/convert_amsu_L1.f90 correctly handles multiple input files
- separated AIRS/README, convert_amsu_L1, and convert_airs_L2 documentation

May 4 2021 :: issue and pull request templates. Tag: v9.10.2

Github changes

- Templates for pull requests, bug reports and feature requests

Documentation updates

- Removed outdated instructions for checking out a tag

April 29 2021 :: change default GitHub branch. Tag: v9.10.1

- Replaced the default branch ("Manhattan") with "main". "main" is now the latest and stable version. The HEAD of "main" will be the source of releases using the vX.Y.Z format.

April 27 2021 :: preprocess, inflation options, external FO output, wrf-hydro, AMSU-A, DART_LAB. Tag: v9.10.0

New features

- Updated `preprocess`:
  - Integers for quantities (kinds) are created and managed by preprocess instead of through a list of integers in `DEFAULT_obs_kind_mod.F90`.
  - Quantities are defined by name in files: `xxx_quantities_mod.f90`.
  - `preprocess` is backwards compatible with existing (v9.9.0) `DEFAULT_obs_kind_mod.F90` files and corresponding `&preprocess_nml` options.
- Inflation algorithm options in `&filter_nml` can be given as strings.
- External forward operators can be selectively written out by observation type in `obs_sequence_tool`.
- Updated wrf-hydro interface from **James McCreight**.
- Added `AIRS/convert_amsu_L1.f90` and `amsua_bt_mod.f90` to support converting AMSUA brightness temperatures to obs_seq.
- `AIRS/airs_JPL_mod.f90` strictly supports HDF-EOS2 (not HDF-EOS5) and is only used for Level 2 (i.e. retrievals of) temperature and humidity observations.
- POP CESM2.1 scripts use the unzipped CAM reanalysis files available on the Research Data Archive (RDA).
- Enhanced adaptive inflation added to DART_LAB.
- Improved support for RTPS: output posterior inflation files now contain posterior inflation values when using RTPS.
- Improved support for RTTOV in MPAS:
  - `loc_sea` variable used to create sfc, 2m, 10m locations relative to model surface elevation.
  - new error code for pressure *not* monotonically decreasing with level.
  - QTY_CLOUD_FRACTION added.
- `E_CONTINUE` added to allow programs continue after throwing an error. Used in developer tests.
- Support for more Flux Tower observations (`obs_def_tower_mod.f90`)
- Expanded support for netcdf in `netcdf_utilities_mod`.
- Documentation converted to reStructuredText and available online. Reorganization of directories to support this: docs -> guide, docs/tutorial -> theory.

Bug fixes

- Check for monotonically decreasing pressure from TOA down to surface in `obs_def_rttov_mod.f90` now checks for greater than or *equal* to previous level.
- External forward operators now use the correct ensemble members when `distributed_state=false`.

- The `obs_sequence_tool` now writes out external forward operator values. Thanks to **Chris Riedel** for reporting this and providing the original bug-fix.
- `obs_def_radar_mod.f90` now correctly applies `apply_ref_limit_to_fwd_op` when QTY_RADAR_REFLECTIVITY is in the state. Thanks to **Craig Schwartz** for providing the bug-fix.
- `quality_control_mod.f90` now correctly handles `enable_special_outlier = .true.` Thanks to **Craig Schwartz** for providing the bug-fix.

Removed

- Doxygen directory.
- Removed svn logging variables and `register_module` for cleaner log messages. The svn info has not been used since DART moved to Git.

Oct 29 2020 :: radiance support, MPAS, obs converters Tag: v9.9.0

- Use RTTOV (Radiative Transfer for TOVS) routines to support radiance assimilation.
  - *Introduction to DART support for RTTOV*
  - WRF, MPAS, and CAM-FV model interfaces now support radiance assimilation.
  - Added GOES 16-19 ABI converter
- *NOTE*: The `build_templates/mkmf.template` file has been removed from version control. You must now explicitly copy the best example `mkmf.template` into place before compiling. If there is no `mkmf.template` when you try to build, an error message is displayed.
- MPAS regional configurations now supported.
- Converted CHANGELOG to a markdown document, put newest content at top.
- Converted many HTML documents to markdown
  - renamed `observations/obs_converters/observations.html` to `observations/obs_converters/README.md` for example.
- Updated Publications
- declare hexadecimal constants according to the Fortran standard.
- GSI2DART converter updated - Thanks to **Craig Schwartz** & **Jamie Bresch**.
- The WRF-DART tutorial has been rewritten as `models/wrf/tutorial/README.md`
- Hydro-DART (AKA wrf-hydro/DART) has been updated to be Manhattan-compliant.
  - also support masked bucket
  - added perturbed forcing capability
- The support for POP and CESM2 has been implemented and documented.
- `obs_diag` now correctly handles the special case when the observation is properly assimilated or evaluated but the posterior forward operator fails. The posterior DART QC in the `obs_diag_output.nc` should be a '2', not a '4'. The prior DART QC value in obs_diag_output.nc can still be a 7 if need be.
- `obs_def_tower_mod.f90` was refactored into `obs_def_tower_mod.f90` and `obs_def_land_mod.f90`.
- WRF-Chem/DART documentation and datasets have been updated for Manhattan. Dr. Arthur Mizzi is the father of the WRF-Chem/DART project. If you'd like to use WRF-Chem/DART, please email Dr. Mizzi.
- Fixed bug in `obs_seq_to_netcdf` to correctly append to existing netCDF files.
- Support absolute humidity observations - Thanks to **Michael Ying**.

- DEFAULT_obs_kind_mod.F90 has many added quantities.
- new observation converters including (but not limited to):
  - absolute humidity
  - streamflow observations from the Mexican water agency
  - streamflow observations from the USGS
  - total water storage observations from GRACE
  - radiance observations from GOES
- the following forward operator modules are either new or modified:
  - (M) observations/forward_operators/DEFAULT_obs_def_mod.F90
  - (M) observations/forward_operators/obs_def_GRACE_mod.f90
  - (A) observations/forward_operators/obs_def_abs_humidity_mod.f90
  - (M) observations/forward_operators/obs_def_altimeter_mod.f90
  - (A) observations/forward_operators/obs_def_land_mod.f90
  - (A) observations/forward_operators/obs_def_mesonet_mod.f90
  - (M) observations/forward_operators/obs_def_oxygen_ion_density_mod.f90
  - (M) observations/forward_operators/obs_def_reanalysis_bufr_mod.f90
  - (M) observations/forward_operators/obs_def_rel_humidity_mod.f90
  - (A) observations/forward_operators/obs_def_rttov_mod.f90
  - (A) observations/forward_operators/obs_def_streamflow_mod.f90
  - (M) observations/forward_operators/obs_def_tower_mod.f90
  - (M) observations/forward_operators/obs_def_upper_atm_mod.f90
  - (A) observations/forward_operators/rttov_sensor_db.csv
- fill_inflation_restart now correctly creates inflation values for all variables in the DART state, regardless of the setting of the no update metadata.
- GITM is now fully Manhattan compliant.
- fix bug in madis rawin converter
- avoid computing posterior inflation if using the 'relaxation to prior spread' inflation option – Thanks to **Craig Schwartz**.
- add additional reporting options to the obs_assim_count utility

Nov 20 2019 :: FESOM,NOAH-MP model support, better testing Tag: v9.8.0

- first release entirely from GIT
- fixed bug in fill_inflation_restart tool which used the prior inflation mean and sd for both prior and posterior inflation files. now correctly uses the posterior mean/sd if requested.
- fixed a typo in the location test script that prevented it from running
- additional functionality in the quad interpolation code, now supports grids which start at 90 (north) and end at -90 (south).
- if possible, send shorter MPI messages. improves performance on some platforms and MPI implementations.

- add explicit call to `initalize_utilities()` where it was missing in a couple of the WRF utility routines.
- added an example of how to use a namelist to the `text_to_obs.f90` observation converter program.
- Removing the clamping messages in `clamp_variable()` of clamped values
- changed argument names using reserved keywords.
  - `state_vector_io_mod:read_state() 'time' to 'model_time'`
  - `random_seq_mod:random_gamma() 'shape' to 'rshape', 'scale' to 'rscale'`
  - `random_seq_mod:random_inverse_gamma() 'shape' to 'rshape', 'scale' to 'rscale'`
  - `obs_def_mod:init_obs_def() 'kind' to 'obkind', 'time' to 'obtime'`
  - `obs_utilities_mod:  'start' to 'varstart', 'count' to 'varcount'`
- The **FESOM** model is now Manhattan-ready. Thanks to **Ali Aydogdu**
- The **noah** model is now Manhattan-ready and may be used with NOAH-MP.
- bugfixed references to the `documentation` directory that was renamed `docs` to comply with GitHub Pages.
- improved `test_dart.csh` functionality.

Apr 30 2019 :: cam-fv refactor, posteriors optional, QC 8 Revision: 13138

- The CAM Finite Volume (**cam-fv**) `model_mod.f90` has undergone substantial refactoring to improve simplicity and remove code for unsupported CAM variants while also supporting WACCM and WACCM-X. Namelist changes will be required.
- **cam-fv** setup and scripting support added for CESM 2.1, including advanced archiving and compression
- fix for WRF's wind direction vectors when using the Polar Stereographic map projection. Thanks to **Kevin Manning** for the fix.
- Add filter namelist option to avoid calling the posterior forward operators and to not create those copies in the `obs_seq.final` file.
- Use less memory if writing ensemble member values into the `obs_seq.final` file.
- added a DART QC of 8 for failed vertical conversions
- updated Matlab scripts support QC=8 and no posterior in obs sequence files.
- sampling error correction table now has all ensemble sizes between 3 and 200
- `closest_member_tool` can be compiled with other MPI targets
- `COSMIC_ELECTRON_DENSITY` has been moved from `obs_def_gps_mod.f90` to `obs_def_upper_atm_mod.f90`, which has new quantities for `ION_O_MIXING_RATIO` and `ATOMIC_H_MIXING_RATIO`
- `obs_converters/gps/convert_cosmic_ionosphere.f90` has a test dataset
- support for NAG compiler
- fixed Intel compiler bug in `lorenz_96` comparing long integers to integer loop indices
- `get_maxdist()` now a required routine all location modules
- Default routines now create a time variable as `time(time)` to allow multiple files to be concatenated along the unlimited dimension more easily. Also conforms to the netCDF convention for coordinate dimensions.
- `obs_impact_tool` handles a continuum of values, not just discrete 0 or 1.
- `fill_inflation_restart` now produces files with names consistent with filter defaults.
- expanded functionality in `xyz_location_mod.f90`

- Removed 'slow' sorting routines from `sort_mod.f90`
- replacing some repeated native netCDF library calls with routines from the `netcdf_utilities_mod.f90`
- Updated dewpoint equation to avoid dividing by zero given a very unlikely scenario (r12832)
- More efficient implementation of adaptive inflation
- *Yongfei Zhang* and *Cecilia Bitz* added improvements to the CICE model and observation converters and forward operators. These changes also use the locations of the 'new' glade filesystem. They used CESM tag: cesm2_0_alpha06n
- Worked with Yongfei Zhang to remove prototype codes and more completely document observation converters and data sources for cice assimilation.
- removed `allow_missing_in_clm` flag from the `&assim_tools_nml` namelist in the CICE work directory. The flag moved to a different namelist and the CICE model doesn't care about it.
- increased the maximum number of input files to `obs_diag` from 100 to 10000.
- Updated the `developer_tests` to include more cases.
- Updated `oned/obs_diag.f90` to support `obs_seq.out` files.
- Better error and informational messages in various routines.

Aug 03 2018 :: performance fix for distributed mean Revision: 12758

- Important performance fix if model does vertical conversion for localization. Results were not wrong but performance was poor if `distribute_mean = .true.` was selected in the `&assim_tools_nml` namelist.

  Now distributing the mean runs in close to the non-distributed time and uses much less memory for large models. This only impacts models which do a vertical conversion of either the observation or state vertical coordinate for localization AND which set `&assim_tools_nml :: distribute_mean = .true.` to use less memory.

  When using a distributed mean `convert_all_obs_verticals_first = .true.` should be set. If your observations will impact most of the model state, then `convert_all_state_verticals_first = .true.` can also be set.

Jun 18 2018 :: CAM/CESM 2.0, DART QC 8, closest_member_tool Revision: 12682

- Support for **cam-fv** assimilations in the CESM 2.0 release. See documentation in `models/cam-fv/doc/README_cam-fv` for details.
- `obs_diag` and matlab scripts updated to report statistics on DART QC 8, observation failed vertical conversion
- Updates to fix minor problems with the new WRF scripts
- Added the `inf_sd_max_change` namelist item to all `input.nml` files for the enhanced inflation option
- Revival of the `closest_member_tool`, which now runs in parallel on all ensemble members at one time. This tool can be used as a template for any other tools which need to process something for all ensemble members in parallel.
- Revival of the `fill_inflation_restart` tool as a Fortran 90 program. Using `ncap2` is still possible, but if the correct version is not installed or available this tool can be used.
- Added more functions to the `netcdf_utilities_mod.f90`

May 21 2018 :: enhanced inflation option, scripting Revision: 12591

- Enhanced inflation algorithm added. See the `filter_mod.html` for new documentation on this option.
- Updated WRF scripts for the Manhattan release.
- `obs_diag` reports statistics on DART QC 8, observation failed vertical conversion. Matlab scripts also updated to support QC 8.

- New parallel conversion scripts for GPS Radio Occultation observations and NCEP prepbufr conversions.
- Further updates to documentation files to change KIND to QTY or Quantity.
- Documented required changes when moving from the Lanai/Classic release to Manhattan in `documentation/html/Manhattan_diffs_from_Lanai.html`
- Expanded the routines in the `netcdf_utilities_mod.f90`
- Add an ensemble handle parameter to the 6 ensemble manager routines where it was missing.
- The `advance_time` program can read/generate CESM format time strings (YYYY-MM-DD-SSSSS).
- Fixed a bug in the netcdf read routines that under certain circumstances could report an array was using the unlimited dimension incorrectly.
- Removed the option to try to bitwise reproduce Lanai results; due to the number of changes this is no longer possible.
- Minor bug fixes to the (seldom used) perturb routines in the **WRF** and **mpas_atm** `model_mod.f90` files. (used to add gaussian noise to a single model state to generate an ensemble; this is never the recommended method of starting a new experiment but the code remains for testing purposes.)
- Several remaining model-specific `model_mod_check` programs were removed in favor of a single common program source file.
- Keep `filter_mod.dopplerfold.f90` in sync with `filter_mod.f90`, and `assim_tools_mod.pf.f90` in sync with `assim_tools_mod.f90`.
- Removed makefiles for the obsolete `trans_time` program.

Mar 01 2018 :: ROMS, MMC, PMO, mpas_atm debug, etc Revision: 12419

- Fix a debug message in the **mpas_atm** model which might have caused a buffer overflow crash when formatting a message for a larger ensemble size.
- Update the **ROMS** shell scripts to support PBS, SLURM, as well as LSF. Update the ROMS model_mod html documentation.
- Update the default **cam-fv** `input.nml` to have more realistic values for the highest observation assimilated, and for where the ramp starts that decreases the increments at the model top. If running with a higher model top than the default check these items carefully.
- Fixed variable type for `time` variables we create in diagnostic files
- Miscellaneous minor Bug fixes:
  - Print format wider for fractional levels in `threed_sphere` locations
  - Fixed a deallocate call at program shutdown time
  - Fixed an indexing problem computing **cam-fv** U_WIND observations if the observation used HEIGHT as the vertical coordinate (very unusual).
  - Fixed grid creation bug in a test program used with `model_mod_check`. Now uses correct spacing for grids in the x,y coordinates.
  - Fixed an allocate problem in a test interpolate routine.
- Add surface pressure to the default state list in the **wrf** `work/input.nml`
- `developer_tests/test_dart.csh` can run PMO for more models. required updates to the `work/input.nml` in several directories (wrf, cm1, POP, mpas_atm) to match the current namelist.

- several `model_mod_check` programs were combined into a single version that allows for selection of individual tests. many of the input.nml `models/xxx/work/input.nml` files have either had a `&model_mod_check_nml` section added or updated to match the updated interface.
- the DART QTYs are now available via the state structure in the **wrf** and **clm** `model_mod`s.
- support the NAG compiler better. (contact dart@ucar.edu for more help if you want to use this compiler. some hand work is still needed.)
- streamlined the debug output from the `state_structure_info()` call to avoid replicating information that was the same for all variables.
- minor formatting change to the dart log file output for the list of observation types being assimilated, evaluated, and using precomputed forward operators.
- fixed an uninitialized variable in the BGRID model code in a routine that isn't normally used.
- Updated the `threed_sphere` location module documentation with some usage notes about issues commonly encountered.
- Fixed an incorrect test when printing out a log message describing if the inflation would be variance-adaptive or not.
- Change the location of the POP MDT reference file to be relative to the current run directory and not an absolute file location on cheyenne.
- Make the ROMS, CM1, and POP model_mod log namelist information to the namelist log file and not the main DART log file.
- Updated several html documentation files, including the `template/model_mod.html` which describes the current model_mod required interfaces.
- Updated the instructions for the GSI to DART obs converter to suggest some needed compiler flags in certain cases.
- Updated the location module test programs.

Dec 01 2017 :: ROMS scripting, debugging aids Revision: 12166

- Added an option to the ROMS model scripting to advance the model ensemble members in parallel using a job array.
- Updated the DART_LAB Matlab GUIs to log a history of the settings and results.
- Added a debug option to the filter namelist, `write_obs_every_cycle`, to output the full `obs_seq.final` during each cycle of filter.
  (Very slow - use only when debugging a filter crash.)
- Allow the test grid in `model_mod_check` to cross the prime meridian for testing longitude interpolation in grids that cross the 360/0 line.

Nov 22 2017 :: minor updates for DA challenge files Revision: 12144

- added `obs_seq.in.power` to the Lorenz 96 directory
- added new obs types to the workshop version of the `input.nml` assimilation list

Nov 21 2017 :: 1D obs_diag fix, 1D power forward operator Revision: 12138

- fixed a bad URL reference in tutorial section 18
- fixed a crash with the 1D version of the observation diagnostics program when including identity observations.
- all models with a `workshop_setup.csh` now build the same set of programs. (some/most did not build obs_diag - which is used in the tutorial)

- added a 1D obs-to-a-power forward operator.
- updates to the matlab plotting routines for NetCDF observation formats
- World Ocean Database (WOD) converter supports partial year conversions and 2013 file formats.

Oct 17 2017 :: mpas_atm bug fix, various other updates. Revision: 12002

- Fixed a bug in the **mpas_atm** `model_mod` that affected surface observations, in particular altimeter obs. also fixed a bug in the vertical conversion if using 'scale height' as the vertical localization type.
- Fixed a bug in the **cam-fv** `model_mod` which might have excluded observations with a vertical coordinate of height (meters) which were in fact below the equivalent highest_obs_pressure_Pa namelist setting. also fixed a possible memory leak.
- Added two new modules: `options_mod.f90` and `obs_def_utilities_mod.f90` this was required so we didn't have circular dependencies in our modules as we reused common code in more places. We have updated all the `path_names*` files which are in the repository. if you have your own path_names files you may need to add these new modules to your path lists.
  - `assimilation_code/modules/utilities/options_mod.f90`
  - `observations/forward_operators/obs_def_utilities_mod.f90`
- Removed `QTY_SURFACE_TEMPERATURE` from the default obs quantities list and added `QTY_2M_SPECIFIC_HUMIDITY`. `QTY_2M_TEMPERATURE` exists for atmospheric models, and `QTY_SKIN_TEMPERATURE` and `QTY_SOIL_TEMPERATURE` exist for other models. if you were using `QTY_SURFACE_TEMPERATURE` please replace it with the corresponding other temperature quantity.
- Updated and improved the observation converter for ionospheric observations from the COSMIC GPS satellite.
- Updated the **cam-fv** scripts for cesm2_0_beta05.
- Updated the Matlab diagnostics documentation. 'help DART' or 'doc DART' will give an overview of the available Matlab diagnostics shipped with the dart distribution.
- Added the observation type `COSMIC_ELECTRON_DENSITY` to the `obs_def_upper_atm_mod`
- `dart_to_clm` and `clm_to_dart` were resurrected to correctly handle conversions for the SWE (snow water equivalent) field.
- Updated the channel and column location modules to be compatible with the current required interfaces.
- Updated the `model_mod_check.f90` program (most often used when porting DART to a new model). there is now more control over exactly which tests are being run. updated the nml and html documentation files to match the current code and describe the tests in more detail.
- Fixed a misleading status message in the `obs_sequence_tool` when all obs are excluded by the min/max lon/lat box namelist items. the incorrect message blamed it on observation height instead of the bounding box.
- Added some additional debugging options to the mpi utilities module. if you have problems that appear to be MPI related, contact us for more help in enabling them.
- Improved some error messages in `location_io_mod` and `state_structure_mod`

Aug 2 2017 :: single filenames, random distributions, bug fixes. Revision: 11864

- added code to support listing input and output filenames directly in the namelist instead of having to go through an indirect text file. most useful for programs that take a single input and output file, but works for all cases.
- bug fix in `location_io_mod.f90` that affected `obs_seq_to_netcdf` (error in adding vertical location types to output file).
- fix to `convert_gpsro_bufr.f90` converter (GPS obs from BUFR files) that failed if r8 defined to be r4.
- added draws from gamma, inverse gamma, and exponential distributions to the random sequence module.

- various updates to the **cam** scripts to work more smoothly with the most recent CIME changes and DART Manhattan updates.
- added `QTY_CWP_PATH` and `QTY_CWP_PATH_ZERO` to the default quantities list for the `obs_def_cwp_mod.f90` forward operator.
- improved some error messages in the diagnostic matlab scripts

July 18 2017 :: bug fixes, documentation updates. Revision: 11830

- fixed bug in `obs_impact_tool` when generating the run-time table. specifying a generic quantity resulted in selecting the wrong specific obs types.
- fixed a bug that would not allow filter to start from a single ensemble member if `single_file_in = .true.`
- updates to HTML documentation especially for types/quantities (replacing kinds)
- updates to `input.nml` namelists, code comments, and shell scripts where names changed from `restart` to `state` for input and output files.

July 7th 2017 :: cam-fv, mpas_atm scripts, single file i/o. Revision: 11807

- **mpas_atm**: scripts completely revised for the Manhattan release. Many thanks to **Soyoung Ha** and **Ryan Torn** for the contributed code.
- **cam-fv**: scripts and `model_mod.f90` updated for cesm2_0_beta05.

Single File I/O:

- Now we are able to run `single_file_in` and `single_file_out` with MPI.
- `single_file_io_mod.f90` has been removed and its functionality has been moved to `direct_netcdf_mod.f90`.
- `single_file_io_mod.f90` has been removed from all of the `path_names_*` files in the repository. (Remove it from any private `path_names_*` files.)

June 27rd 2017 :: CICE 5, model_mod_check, tutorial. Revision: 11770

- Updated support for CICE5.
- Updated support for `model_mod_check` - now compatible with netCDF input files, input is through [input,output]_state_files namelist variable (variables renamed).
- Ensured consistency between low-order namelists and the updated DART tutorial. Updated documentation of many namelists. More to come.
- `location_mod`: namelist variable `maintain_original_vert` was deprecated, it is now removed. You must remove it from your existing namelists or DART will error out immediately.
- `obs_diag`: namelist variables `rat_cri` and `input_qc_threshold` have been deprecated for years, they have been removed. You must remove them from your existing namelists or obs_diag will error out immediately.

Jun 2nd 2017 :: tutorial, DART_LAB, and various updates. Revision: 11696

- bring the DART tutorial pdf slides up to date with the current release.
- include new GUIs with adaptive inflation options in DART_LAB:
  - `oned_model_inf.m`
  - `run_lorenz_96_inf.m`
- added the **lorenz_96_2scale** model - additional kinds of `QTY_SMALL_SCALE_STATE` and `QTY_LARGE_SCALE_STATE` added as required.
- add useful attributes to the variables in the diagnostic files

- updates and minor bug fixes to the matlab diagnostic scripts
- updates to the default input.nmls for models
- updates to the **cam-fv** shell scripts to work with the CESM2.0 framework
- updates to the **cam-fv** `model_mod` for support of `cam-chem` variables Added more QUANTITIES/KINDS for chemistry species. Removed support for 'stand-alone' **cam** and **cam-se** (**cam-se** will be a separate 'model').
- major bug fix in the **simple_advection** `model_mod`: Fixed an error with the layout of the state vector.
- `obs_def_radar_mod`: Fixed a serious bug in the fall velocity forward operator. If the fall speed field is not in the state the test for a bad istatus from the interpolate() call was looking at the wrong variable and returning ok even if interpolate() had set bad values.
- bug fix in the **wrf** model_mod for fields which have a vertical stagger
- fix to the makefiles for the GSI2DART observation converter
- added additional netcdf and location utility routines
- various fixes to documentation and test code
- renamed `QTY_RAW_STATE_VARIABLE` to `QTY_STATE_VARIABLE` (RAW is redundant)
- `direct_netcdf_mod`: Renamed `limit_mem` to `buffer_state_io`. `buffer_state_io` is now a logical that states if a variable that tells DART it it should read and write variables all at once or variable-by-variable.

May 5th 2017 :: major changes to model_mod interfaces. Revision: 11615

A long-awaited overhaul of the model_mod interfaces. All models which are in our subversion repository and are supported in the Manhattan release have been updated to match the new interfaces. If you have model_mods with extensive changes, our recommendation is to diff your changes with the version you checked out and insert those changes into the new version. The changes for this update are unfortunately extensive.

The detailed list of changes:

`model_mod::get_state_meta_data()` is no longer passed an ensemble_handle as the first argument. it should not do vertical coordinate conversion. that will be done as a separate step by `convert_vertical_state()`

`model_mod::vert_convert` is replaced by `convert_vertical_state()` and `convert_vertical_obs()` Any vertical conversion code that was in `get_state_meta_data` should be moved to `convert_vertical_state()` which has access to the state vector index, so the code should move easily.

`model_mod::query_vert_localization_coord` is no longer a required interface `model_mod::get_close_maxdist_init` is not longer a required interface `model_mod::get_close_obs_init` is not longer a required interface

`model_mod::get_close_obs` has a different calling convention and is split into `get_close_obs()` and `get_close_state()`. the close obs routine is passed both the obs types and quantities, and the close state routine is passed both the state quantities and the state index, for ease in vertical conversion if needed.

`model_mod::nc_write_model_vars()` is deprecated for now; it may return in a slightly different form in the future.

`model_mod::nc_write_model_atts()` is now a subroutine with different arguments. it should now only write any global attributes wanted, and possibly some grid information. it should NOT write any of the state variables; those will be written by DART routines.

`model_mod::get_model_size()` needs to return an `i8` (a long integer) for the size.

A new module `default_model_mod` supplies default routines for any required interfaces that don't need to be specialized for this model.

A new module `netcdf_utilities_mod` can do some simple netcdf functions for you and we plan to add many more over the next couple months.

model_mod::get_model_time_step has been replaced by shortest_time_between_assimilations() since in fact it has always controlled the minimum time filter would request a model advance and never had anything to do with the internal time step of the dynamics of the model.

We have removed output_state_vector from the namelist of all model_mods since we no longer output a single 1d vector. all i/o is now in netcdf format.

Models now have more control over when vertical conversion happens - on demand as needed, or all up front before assimilation.

Models that were doing vertical conversion in get_state_meta_data should set:

```
&assim_tools_nml
   convert_all_state_verticals_first = .true.
   convert_all_obs_verticals_first = .true.

Models which were not should set:
   convert_all_state_verticals_first = .false.
   convert_all_obs_verticals_first = .true.
```

The location_mod::vert_is_xxx() routines have become a single is_vertical(loc, "string") where string is one of: "PRESSURE", "HEIGHT", "SURFACE", "LEVEL", "UNDEFINED", "SCALE_HEIGHT"

Models doing vertical localization should add a call to set_vertical_localization_coord() in their static_init_model() routine to tell dart what vertical coordinate system they are expecting to convert to for vert localization

Most path_names_xxx files have been updated to add additional modules. compare against what is checked out to see the differences.

Some of the internal changes include pulling common code from the locations modules into a location_io_mod which contains common functions for creating and writing 'location' variables for any location type.

QTY_RAW_STATE_VARIABLE is redundant and was shortened to QTY_STATE_VARIABLE

Many utility programs use the template/model_mod.f90 because they do not depend on any model-specific functions. this file was also updated to match the new interfaces.

The obs_impact facility is enabled in the assim_tools namelist. you can use the obs_impact_tool to construct a table which prevents one class of observations from impacting another class of state.

Sampling Error Correction now reads the values it needs from a single netcdf file found in assimilation_code/programs/gen_sampling_err_table/work. Copy it to the same directory as where filter is running. All ensemble sizes which were previously in final_full.XX files are included, and there is a tool to generate and append to the file any other ensemble size required.

April 27th 2017 :: diagnostic file changes. Revision: 11545

Two additional Diagnostic Files (forecast and analysis) in Filter which can be set with the namelist option (stages_to_write)

- **input** writes out mean and sd if requested.
  - For low order models, mean and sd are only inserted into restart files with a single time step.
- **forecast**
  - contains the forecast and potentially the mean and sd for the, this is mostly important for lower order models which cycle
- **preassim** before assimilation
  - No Inflation: same as forecast

  - Prior Inf: the inflated ensemble and damped prior inf
  - Post Inf: same as forecast
  - Prior and Post Inf: the inflated ensemble and damped prior inf
- **postassim** after assimilation (before posterior infation)
  - No Inflation: same as analysis
  - Prior Inf: same as analysis
  - Post Inf: assimilated ensemble and damped posterior inflation
  - Prior and Post Inf: assimilated ensemble and damped posterior inflation
- **analysis** after assimilation and before potentially update posterior inflation ensemble and updated prior inf
  - No Inflation: assimilated ensemble
  - Prior Inf: assimilated ensemble and updated prior inf
  - Post Inf: post inflated ensemble and updated posterior inflation
  - Prior and Post Inf: post inflated ensemble and updated prior inf and posterior inflation
- **output**
  - a single time step of the output ensemble and potentially updated prior inf and posterior inflation

Feb 15th 2017 :: filter updates. Revision: 11160

The postassim diagnostics file was being incorrectly written after posterior inflation was applied. It is now written immediately after the assimilation update, and then posterior inflation, if enabled, is applied.

Sampling Error Correction now reads data from a single netcdf file for any ensemble size. To add other sizes, a program can generate any ensemble size and append it to this file. The default file is currently in `system_simulation`:

`system_simulation/work/sampling_error_correction_table.nc`

Filter and PMO no longer need the "has_cycling" flag.

Changes to the filter_nml are :

- `has_cycling` REMOVED for low order models

Changes to the perfect_model_obs_nml are :

- `has_cycling` REMOVED for low order models

Feb 15th 2017 :: rma_single_file merge changes. Revision: 11136

Filter and PMO can now run with multiple cycles for low order models. The output for this is only supported with single file output (members, inflation, mean, sd are all in the same file).

Added matlab support for diagnostics format in lower order models.

Changes to the filter_nml are :

- `output_restart` RENAMED to `output_members`
- `restart_in_file_name` RENAMED to `input_state_file_list`
- `restart_out_file_name` RENAMED to `output_state_file_list`
- `single_restart_file_in` RENAMED to `single_file_in`
- `single_restart_file_out` RENAMED to `single_file_out`
- `input_state_files` ADDED - not currently working

- output_state_files ADDED - not currently working
- has_cycling ADDED for low order models

Changes to the perfect_model_obs_nml are :

- start_from_restart RENAMED read_input_state_from_file
- output_restart RENAMED write_output_state_to_file
- restart_in_file_name RENAMED input_state_files
- restart_out_file_name RENAMED output_state_files
- single_file_in ADDED for low order models
- single_file_out ADDED for low order models
- has_cycling ADDED for low order models

Jan 13th 2017 :: rma_fixed_filenames merge changes. Revision: 10902

Specific namelist changes include:

1. Earlier versions of the RMA branch code supported both direct NetCDF reads/writes and the original binary/ascii DART format restart files.

   As of the next update DART format files are no longer supported. All I/O is NetCDF only. If your model does not use NetCDF you will still need a model_to_dart and dart_to_model converter; otherwise all DART programs read the model's NetCDF files directly. The namelist options related to selecting direct netcdf I/O have been removed.

2. Diagnostic and state space data (such as inflation, mean and sd information) that were previously stored in {Prior,Posterior}_Diag.nc are now broken up into multiple files and have fixed filenames. This decreases the IO time for diagnostic output and reduces the number of namelist options.

3. There is no longer support for observation space inflation (i.e. inf_flavor = 1). Contact us at dart@ucar.edu if you have an interest in using this option.

Changes to the filter_nml are :

- restart_in_file_name has been replaced with input_restart_file_list. The namelist must contain one or more file names, each of which is a textfile containing a list of N NetCDF restart files, one per line for each ensemble member. For models with multiple domains (e.g. nested WRF or CLM) you must specify a listfile for each domain.
- restart_out_file_name has been replaced with output_restart_file_list. Same format as input_restart_file_list.
- inf_in_file_name REMOVED, now have fixed names of the form input_{prior,posterior}inf_{mean,sd}.nc
- inf_out_file_name REMOVED, now have fixed names of the form output_{prior,posterior}inf_{mean,sd}.nc.
- inf_diag_filename REMOVED
- inf_output_restart REMOVED, inflation restarts will be written out if inflation is turned on
- output_inflation REMOVED, inflation diagnostic files will be written if inflation is turned on
- stages_to_write There is more control over what state data to write. Options are at stages : 'input', 'preassim', postassim', 'output'.

  Stages preassim and postassim will output state data originally contained within the copies of Prior_Diag.nc and Posterior_Diag.nc. See rma_doc/rma.html for details on the filename conventions. For example, running filter with prior inflation enabled with stage 'preassim' enabled will produce files with names:

  – preassim_member_####.nc

  - preassim_{mean,sd}.nc
  - preassim_priorinf_{mean,sd}.nc

- `write_all_stages_at_end` important for large models - all output file I/O is deferred until the end of filter, but will use more memory to store the data. More detailed info is in rma_doc/rma.html
- `output_restart_mean` renamed output_mean
- `output_restart` renamed output_restarts
- `direct_netcdf_{read,write}` REMOVED, always true
- `restart_list_file` renamed input_restart_file_list
- `single_restart_file_in` renamed single_file_in
- `single_restart_file_out` renamed single_file_out
- `add_domain_extension` REMOVED
- `use_restart_list` REMOVED
- `overwrite_state_input` REMOVED, equivalent functionality can be set with `single_restart_file_in = single_restart_file_out`

Changes to the perfect_model_obs_nml are :

- `restart_in_filename` renamed `restart_in_file_names` takes a NetCDF file. For multiple domains you can specify a list.
- `direct_netcdf_{read,write}` REMOVED, always true

Changes to the state_space_diag_nml are :

- `single_file` REMOVED, diagnostic files are now controlled in `filter_nml` with `stages_to_write`
- `make_diagnostic_files` REMOVED, no longer produce original `Prior_Diag.nc` and `Posterior_Diag.nc`
- `netCDF_large_file_support` REMOVED, always true

Changes to the state_vector_io_nml are :

- `write_binary_restart_files` REMOVED

Changes to the ensemble_manager_nml are :

- `flag_unneeded_transposes` – REMOVED

Changes to the integrate_model_nml are :

- `advance_restart_format` – REMOVED, only supporting NetCDF format.

Scripting with CESM :

See `models/cam-fv/scripts_cesm1_5/assimilate.csh` for an example of how to handle the new filename conventions.

```
(To help find things:  input_priorinf_mean output_priorinf_mean )
{in,out}put_{prior,post}inf_{mean,sd}.nc   ARE in use;
    Search for stage_metadata%filenames turned up
    interface set_file_metadata
       module procedure set_explicit_file_metadata
       module procedure set_stage_file_metadata
```

(continues on next page)

(continued from previous page)

```
      ! stage_name is {input,preassim,postassim,output}
      ! base_name  is {mean,sd,{prior,post}inf_{mean,sd}} from filter/filter_mod.f90.
      write(string1,'(A,''.nc'')') trim(stage_name)//'_'//trim(base_name)
      file_info%stage_metadata%filenames(my_copy,1) = trim(string1)

   This shows where inflation file names are defined.
     > grep -I set_file_metadata */*.f90 | grep inf
   filter/filter_mod.f90:
      call set_file_metadata(file_info, PRIOR_INF_MEAN, stage, 'priorinf_mean', 'prior␣
↪inflation mean')
      call set_file_metadata(file_info, PRIOR_INF_SD,   stage, 'priorinf_sd',   'prior␣
↪inflation sd')
      call set_file_metadata(file_info, POST_INF_MEAN,  stage, 'postinf_mean',
↪'posterior inflation mean')
      call set_file_metadata(file_info, POST_INF_SD,    stage, 'postinf_sd',
↪'posterior inflation sd')

   subroutine set_member_file_metadata(file_info, ens_size, my_copy_start)
      call set_file_metadata(file_info, icopy, stage_name, base_name, desc, offset)

   subroutine set_stage_file_metadata(file_info, copy_number, stage, base_name, desc,␣
↪offset)
      write(string1,'(A,''.nc'')') trim(stage_name)//'_'//trim(base_name)

   subroutine set_explicit_file_metadata(file_info, cnum, fnames, desc)
      file_info%stage_metadata%filenames(cnum,idom)        = trim(fnames(idom))
      file_info%stage_metadata%file_description(cnum,idom) = trim(string1)

   function construct_file_names(file_info, ens_size, copy, domain)
      write(construct_file_names, '(A, ''_member_'', I4.4, A, ''.nc'')') &
                           trim(file_info%root_name), copy, trim(dom_str)

Also see
  harnesses/filename_harness/files:  ENS_MEAN_COPY       PriorDiag_mean.nc
```

ADDITIONAL NOTES :

1. currently the closest_member_tool is broken but plans on being fixed soon.
2. restart_file_tool and most model_to_dart/dart_to_model programs have been deprecated, since DART formatted restarts are no longer supported.
3. some programs such as model_mod_check have not been fully tested and need to be exercised with the new naming conventions.

ancient history

To see previous history, it is probably best to use

- `git log --follow`
- `git diff --name-status XXXX YYYY` where XXXX and YYYY are commits, branches, ...

or something along those lines.

6.235 404 Error

The requested page could not be found.