MDTF Developer's Walkthrough

Release 3.0 beta 1

Yi-Hung Kuo[a] Dani Coleman[b] Thomas Jackson[c]
Chih-Chieh (Jack) Chen[b] Andrew Gettelman[b] J. David Neelin[a]
Eric Maloney[d] John Krasting[c]
(a: UCLA; b: NCAR; c: GFDL; d:CSU)

Jun 19, 2020

CHAPTER

ONE

DEVELOPER INFORMATION

1.1 Migration from framework version 2.0

In this page we summarize issues to be aware of for developers familiar with the organization of version 2.0 of the framework. New developers can skip this section, as the developer documentation is self-contained.

1.1.1 Getting Started and Developer's Walkthrough

A main source of documentation for version 2.0 of the framework were the "Getting Started" and "Developer's Walkthrough" documents. Updated versions of these documents are:

- Getting Started v3.0 (PDF)[1]
- Developer's Walkthrough v3.0 (PDF)[2]

Note: these documents contain a subset of information available on this website, rather than new material: the text is reorganized to be placed in the same order as the version 2.0 documents, for ease of comparison.

1.1.2 Checklist for migrating a POD from version 2.0

Here we list the broad set of tasks needed to update a diagnostic written for version 2.0 of the framework to version 3.0.

- Update settings and varlist files: In version 3.0 these have been combined into a single `settings.jsonc` file. See the settings file format guide (page 9), example POD, or reference documentation (page 19) for a description of the new format.
- Update references to framework environment variables: See the table below for an overview, and the reference documentation (page 29) for complete information on what environment variables the framework sets. Note that your diagnostic should not use any hard-coded paths or variable names, but should read this information in from the framework's environment variables.
- Resubmit digested observational data: To minimize the size of supporting data users need to download, we ask that you only supply observational data specifically needed for plotting, as well as any code used to perform that data reduction from raw sources.

[1] https://mdtf-diagnostics.readthedocs.io/en/latest/_static/MDTF_getting_started.pdf
[2] https://mdtf-diagnostics.readthedocs.io/en/latest/_static/MDTF_walkthrough.pdf

- Remove HTML templating code: Version 2.0 of the framework required that your POD's top-level driver script take particular steps to assemble its HTML file. In version 3.0 these tasks are done by the framework: all that your diagnostic needs to do is generate .eps files of the appropriate names in the `model/PS` and `obs/PS` folders, and the framework will convert and link them appropriately.

1.1.3 Conversion from v2.0 environment variables

In version 3.0, the paths referred to by the framework's environment variables have been changed to be specific to your POD. The variables themselves have been renamed to avoid possible confusion. Here's a table of the appropriate substitutions to make:

Table 1: Environment variable name conversion

Path Description	**v2.0 environment variable expression**	**Equivalent v3.0 variable**
Top-level code repository	`$DIAG_HOME`	No variable set: PODs should not access files outside of their own source code directory within `$POD_HOME`
POD's source code	`$VARCODE`/<pod name>	`$POD_HOME`
POD's observational/supporting data	`$VARDATA`/<pod name>	`$OBS_DATA`
POD's working directory	`$variab_dir`/<pod name>	`$WK_DIR`
Path to requested netcdf data file for <variable name> at date frequency <freq>	Currently unchanged: `$DATADIR`/<freq>/`$CASENAME`.<variable name>.<freq>.nc	
Other v2.0 paths	`$DATA_IN`, `$DIAG_ROOT`, `$WKDIR`	No equivalent variable set. PODs shouldn't access files outside of their own directories; instead use one of the quantities above.

1.2 Developing for MDTF Diagnostics with git

There are many git tutorials online, such as:

- The official git tutorial[3].
- A more verbose introduction[4] to the ideas behind git and version control.
- A still more detailed walkthrough[5], assuming no prior knowledge.

In the interests of making things self-contained we give some step-by-step instructions here:

[3] https://git-scm.com/docs/gittutorial
[4] https://www.atlassian.com/git/tutorials/what-is-version-control
[5] http://swcarpentry.github.io/git-novice/

1.2.1 Using SSH with Github

- It's recommended you generate an SSH key[6] and add it[7] to your github account. This will save you from having to re-enter your github username and password every time you interact with their servers.
- The following instructions assume you're doing this. If you're using manual authentication instead, replace the "`git@github.com:`" addresses in what follows with "`https://github.com/`".

1.2.2 Getting started

- Create a fork of the project by clicking the button in the upper-right corner of the main Github page[8]. This will create a copy in your own Github account which you have full control over.
- Clone your fork onto your computer: `git clone git@github.com:<your_github_account>/MDTF-diagnostics.git`. This not only downloads the files, but due to the magic of git also gives you the full commit history of all branches.
- Enter the project directory: `cd MDTF-diagnostics`.
- Clone additional dependencies of the code: `git submodule update --recursive --init`.
- Git knows about your fork, but you need to tell it about NOAA's repo if you wish to contribute changes back to the code base. To do this, type `git remote add upstream git@github.com:NOAA-GFDL/MDTF-diagnostics.git`. Now you have two remote repos: `origin`, your Github fork which you can read and write to, and `upstream`, NOAA's code base which you can only read from.

1.2.3 Coding a feature

- Start from the `develop` branch: `git checkout develop`.
- If it's been a while since you created your fork, other people may have updated NOAA's `develop` branch. To make sure you're up-to-date, get these changes with `git pull upstream develop` and `git submodule update --recursive --remote`.
- That command updates the working copy on your computer, but you also need to tell your fork on github about the changes: `git push origin develop`.
- Now you're up-to-date and ready to start working on a new feature. `git checkout -b feature/<my_feature_name>` will create a new branch (`-b` flag) off of `develop` and switch you to working on that branch.
- Write your code! Useful commands are `git status` to remind you what branch you're on and what uncommitted changes there are, and `git branch -a` to list all branches.
- Commit changes with `git commit -m <your commit message>`. This means you enter a snapshot of the code base into the history in your local repo.

[6] https://help.github.com/en/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent
[7] https://help.github.com/en/articles/adding-a-new-ssh-key-to-your-github-account
[8] https://github.com/NOAA-GFDL/MDTF-diagnostics

  - Don't commit code that you know is buggy or non-functional!
  - Good commit messages are key to making the project's history useful. To make this easier, instead of using the `-m` flag, you can configure git to launch your text editor of choice with `git config --global core.editor "<command string to launch your editor>"`.
  - Write in the present tense. Commit messages should describe what the commit, when applied, does to the code – not what you did to the code.
  - Messages should start with a brief, one-line summary, less than 80 characters. If this is too short, you may want to consider entering your changes as multiple commits.
  - To provide further information, add a blank line after the summary and wrap text to 72 columns if your editor supports it (this makes things display nicer on some tools). Here's an example[9].

- If you want to let others work on your feature, push its branch to your github fork with `git push -u origin feature/<my_feature_name>`. The `-u` flag is for creating a new branch remotely and only needs to be used the first time.
- When your feature is finished, merge it back into `develop`: first `git checkout develop` then `git merge --no-ff feature/<my_feature_name>`. The '–no-ff' flag is important: it tells git not to compress ("fast-forward") your commit history onto the `develop` branch.
- `git push origin`.
- When your feature is ready, submit a pull request by going to the Github page of your fork and clicking on that button. This is your proposal to the maintainers to incorporate your feature into NOAA's code base.
- When it's no longer needed, delete the branch locally with `git branch -d feature/<my_feature_name>`. If you pushed it to your fork, you can delete it remotely with `git push --delete origin feature/<my_feature_name>`. Remember that branches in git are just pointers to a particular commit, so by deleting a branch you don't lose any history.

1.3 Development instructions

We list several important points to be aware of when developing your POD. This may require you to modify existing code.

1.3.1 Scope of the analysis your POD conducts

See the BAMS article[10] describing version 2.0 of the framework for a description of the project's scientific goals and what we mean by a "process oriented diagnostic" (POD). We encourage PODs to have a specific, focused scope.

PODs should be relatively lightweight in terms of computation and memory requirements (eg, run time measured in minutes, not hours): this is to enable rapid feedback and iteration cycles to assist users in model development. Bear in mind that your POD may be run on model output of potentially any date range and

[9] https://github.com/NOAA-GFDL/MDTF-diagnostics/commit/225b29f30872b60621a5f1c55a9f75bbcf192e0b
[10] https://doi.org/10.1175/BAMS-D-18-0042.1

spatial resolution. Your POD should not require strong assumptions about these quantities, or other details of the model's operation.

1.3.2 Choice of language(s)

In order to accomplish our goal of portability, the MDTF cannot accept PODs written in closed-source languages (eg MATLAB[11]; depending on your use case, it may be feasible to port MATLAB code to Octave[12]).

We also do not accept PODs written in compiled languages (C or Fortran): installation would rapidly become impractical if the user had to check compilation options for each POD. See below for options if your POD requires the performance of a compiled language.

We require that PODs that are funded through the CPO grant be developed in Python, specifically Python >= 3.6 (official support for Python 2 was discontinued as of January 2020). While the framework is able to call PODs written in any scripting language, Python support will be "first among equals" in terms of priority for allocating developer resources, etc. At the same time, if your POD development is not being funded, we recommend against unnecessarily re writing existing analysis scripts in Python. Doing so is likely to introduce new bugs into stable code, especially if you're unfamiliar with Python.

1.3.3 Managing language and library dependencies

We recommend developing your POD using miniconda/Anaconda to manage your POD's dependencies during development for the same reasons we recommend it to end users: it allows you to keep your development environment separate from the rest of your system. Note that conda is not python-specific, but allows coexisting versioned environments of most scripting languages (R[13], NCL[14], pyFerret[15], etc.)

To prevent the proliferation of dependencies, we suggest that new python development use libraries in the python_base[16] conda environment, if possible.

If your POD requires libraries that aren't available in an existing environment, we ask that you notify us (since this situation may be relevant to other developers) and submit a YAML file[17] that creates the environment needed for your POD.

- The environment filename should be `env_<your POD's name>.yml`, and committed in `src/conda`.
- The name of the environment should be `_MDTF_<your POD's name>`.
- We recommend listing conda-forge[18] as the first channel to search, as it's entirely open source and has the largest range of packages. Note that combining packages from different channels (in particular, conda-forge and anaconda's channel) may create incompatibilities.
- We recommend constructing the list of packages manually, and not exporting your development environment with `conda env export`. The latter command gives platform-specific version information

[11] https://www.mathworks.com/products/matlab.html
[12] https://www.gnu.org/software/octave/
[13] https://anaconda.org/conda-forge/r-base
[14] https://anaconda.org/conda-forge/ncl
[15] https://anaconda.org/conda-forge/pyferret
[16] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/develop/src/conda/env_python_base.yml
[17] https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-file-manually
[18] https://anaconda.org/conda-forge

and will not be fully portable in all cases; it also does so for every package in the environment, not just the "top-level" ones you directly requested. It's straightforward to search your POD's code for `import` statements referencing third-party libraries.

- We recommend specifying versions as little as possible, out of consideration for the end user: if each POD specifies exact versions of all its dependencies, conda will need to install multiple versions of the same libraries. In general, specifying a version should only be needed in cases where backward compatibility was broken (eg. python 2 vs. 3) or a bug affecting your POD was fixed (eg. postscript font rendering on MacOS with older NCL). Conda installs the latest version of each package that's consistent with all other dependencies.

After creating the environment file and placing it in `src/conda`, verify that your POD works:

- Create the environment using the `conda_env_setup.sh` script described in the installation instructions:

```
% cd $CODE_ROOT
% ./src/conda/conda_env_setup.sh --env <your POD's name> --conda_
↪root $CONDA_ROOT --env_dir $CONDA_ENV_DIR
```

- Have the framework run your POD on suitable test data, as described in start_config. No additional steps are needed to specify the environment: if your conda environment follows the naming conventions above, the framework will always use it to run your POD, and your POD only.

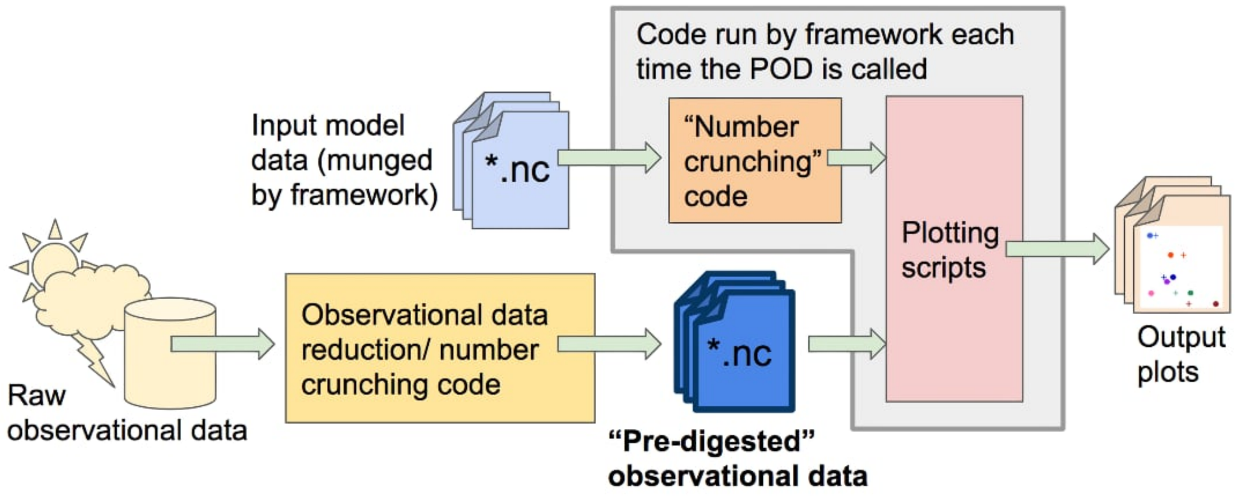
1.3.4 Scope of your POD's code

As described above, your POD should accept model data as input and express the results of its analysis in a series of figures, which are presented to the user in a web page. Input model data will be in the form of one netCDF file (with accompanying dimension information) per variable, as requested in your POD's settings file (page 9). Because your POD may be run on the output of any model, you should be careful about the assumptions your code makes about the layout of these files. Supporting data may be in any format and will not be modified by the framework.

The above data sources are your POD's only input: you may provide options in the settings file for the user to configure when the POD is installed, but these cannot be changed each time the POD is run. Furthermore, your POD should not access the internet or other networked resources.

The output of your POD should be a series of figures in vector format (.eps or .ps), written to a specific working directory (described below). Optionally, we encourage POD developers to also save relevant output data (eg, the output data being plotted) as netcdf files, to give users the ability to take the POD's output and perform further analysis on it.

1.3.5 Observational and supporting data; code organization.


Code run by framework each time the POD is called
Input model data (munged by framework)
*.nc
"Number crunching" code
Plotting scripts
Output plots
Raw observational data
Observational data reduction/ number crunching code
*.nc
"Pre-digested" observational data

In order to make your code run faster for the users, we request that you separate any calculations that don't depend on the model data (eg. pre-processing of observational data), and instead save the end result of these calculations in data files for your POD to read when it is run. We refer to this as "digested observational data," but it refers to any quantities that are independent of the model being analyzed. For purposes of data provenance, reproducibility, and code maintenance, we request that you include all the pre-processing/data reduction scripts used to create the digested data in your POD's code base, along with references to the sources of raw data these scripts take as input (yellow box in the figure).

Digested data should be in the form of numerical data, not figures, even if the only thing the POD does with the data is produce an unchanging reference plot. We encourage developers to separate their "number-crunching code" and plotting code in order to give end users the ability to customize output plots if needed. In order to keep the amount of supporting data needed by the framework manageable, we request that you limit the total amount of digested data you supply to no more than a few gigabytes.

In collaboration with PCMDI, a framework is being advanced that can help systematize the provenance of observational data used for POD development. Some frequently used datasets have been prepared with this framework, known as PCMDIobs. Please check to see if the data you require is available via PCMDIobs. If it is, we encourage you to use it, otherwise proceed as described above.

1.3.6 Other tips on implementation:

1. Structure of the code package: Implementing the constituent PODs in accordance with the structure described in sections 2 and 3 makes it easy to pass the package (or just part of it) to other groups.
2. Robustness to model file/variable names: Each POD should be robust to modest changes in the file/variable names of the model output; see section 5 regarding the model output filename structure, and section 6 regarding using the environment variables and robustness tests. Also, it would be easier to apply the code package to a broader range of model output.
3. Save intermediate output: Can be used, e.g. to save time when there is a substantial computation that can be re-used when re-running or re-plotting diagnostics. See section 3.I regarding where to save the

output.

4. Self-documenting: For maintenance and adaptation, to provide references on the scientific underpinnings, and for the code package to work out of the box without support. See step 5 in section 2.
5. Handle large model data: The spatial resolution and temporal frequency of climate model output have increased in recent years. As such, developers should take into account the size of model data compared with the available memory. For instance, the example POD precip_diurnal_cycle and Wheeler_Kiladis only analyze part of the available model output for a period specified by the environment variables `FIRSTYR` and `LASTYR`, and the convective_transition_diag module reads in data in segments.
6. Basic vs. advanced diagnostics (within a POD): Separate parts of diagnostics, e.g, those might need adjustment when model performance out of obs range.
7. Avoid special characters (`!@#$%^&*`) in file/script names.

1.4 Diagnostic settings file quickstart

This page gives a quick introduction to how to write the settings file for your diagnostic. See the full documentation (page 19) on this file format for a complete list of all the options you can specify.

1.4.1 Overview

The MDTF framework can be viewed as a "wrapper" for your code that handles data fetching and munging. Your code communicates with this wrapper in two ways:

- The settings file is where your code talks to the framework: when you write your code, you document what model data your code uses and what format it expects it in. When the framework is run, it will fulfill the requests you make here (or tell the user what went wrong).
- When your code is run, the framework talks to it by setting environment variables (page 29) containing paths to the data files and other information specific to the run (not covered on this page, follow the link for details).

In the settings file, you specify what model data your diagnostic uses in a vocabulary you're already familiar with:

- The CF conventions[19] for standardized variable names and units.
- The netCDF4 (classic) data model, in particular the notions of variables[20] and dimensions[21] as they're used in a netCDF file.

1.4.2 Example

[19] http://cfconventions.org/
[20] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcVars.html
[21] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html

```
// Any text to the right of a '//' is a comment
{
  "settings" : {
    "long_name": "My example diagnostic",
    "driver": "example_diagnostic.py",
    "realm": "atmos",
    "runtime_requirements": {
      "python": ["numpy", "matplotlib", "netCDF4"]
    }
  },
  "data" : {
    "frequency": "day"
  },
  "dimensions": {
    "lat": {
      "standard_name": "latitude"
    },
    "lon": {
      "standard_name": "longitude"
    },
    "plev": {
      "standard_name": "air_pressure",
      "units": "hPa",
      "positive": "down"
    },
    "time": {
      "standard_name": "time",
      "units": "day"
    }
  },
  "varlist" : {
    "my_precip_data": {
      "standard_name": "precipitation_flux",
      "path_variable": "PATH_TO_PR_FILE",
      "units": "kg m-2 s-1",
      "dimensions" : ["time", "lat", "lon"]
    },
    "my_3d_u_data": {
      "standard_name": "eastward_wind",
      "path_variable": "PATH_TO_UA_FILE",
      "units": "m s-1",
      "dimensions" : ["time", "plev", "lat", "lon"]
    }
  }
}
```

1.4.3 Settings section

This is where you describe your diagnostic and list the programs it needs to run.

`long_name`: Display name of your diagnostic, used to describe your diagnostic on the top-level index.html page. Can contain spaces.

`driver`: Filename of the driver script the framework should call to run your diagnostic.

`realm`: One or more of the eight CMIP6 modeling realms (aerosol, atmos, atmosChem, land, landIce, ocean, ocnBgchem, seaIce) describing what data your diagnostic uses. This is give the user an easy way to, eg, run only ocean diagnostics on data from an ocean model.

`runtime_requirements`: This is a list of key-value pairs describing the programs your diagnostic needs to run, and any third-party libraries used by those programs.

- The key is program's name, eg. languages such as "python[22]" or "ncl[23]" etc. but also any utilities such as "ncks[24]", "cdo[25]", etc.
- The value for each program is a list of third-party libraries in that language that your diagnostic needs. You do not need to list built-in libraries: eg, in python, you should to list numpy[26] but not math[27]. If no third-party libraries are needed, the value should be an empty list.

1.4.4 Data section

This section contains settings that apply to all the data your diagnostic uses. Most of them are optional.

`frequency`: The time frequency the model data should be provided at, eg. "1hr", "6hr", "day", "mon", …

1.4.5 Dimensions section

This section is where you list the dimensions (coordinate axes) your variables are provided on. Each entry should be a key-value pair, where the key is the name your diagnostic uses for that dimension internally, and the value is a list of settings describing that dimension. In order to be unambiguous, all dimensions must specify at least:

`standard_name`: The CF standard name[28] for that coordinate.

`units`: The units the diagnostic expects that coordinate to be in (using the syntax of the UDUnits library[29]). This is optional: if not given, the framework will assume you want CF convention canonical units[30].

In addition, any vertical (Z axis) dimension must specify:

[22] https://www.python.org/
[23] https://www.ncl.ucar.edu/
[24] http://nco.sourceforge.net/
[25] https://code.mpimet.mpg.de/projects/cdo
[26] https://numpy.org/
[27] https://docs.python.org/3/library/math.html
[28] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html
[29] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax
[30] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html

`positive`: Either `"up"` or `"down"`, according to the CF conventions[31]. A pressure axis is always `"down"` (increasing values are closer to the center of the earth).

1.4.6 Varlist section

This section is where you list the variables your diagnostic uses. Each entry should be a key-value pair, where the key is the name your diagnostic uses for that variable internally, and the value is a list of settings describing that variable. Most settings here are optional, but the main ones are:

`standard_name`: The CF standard name[32] for that variable.

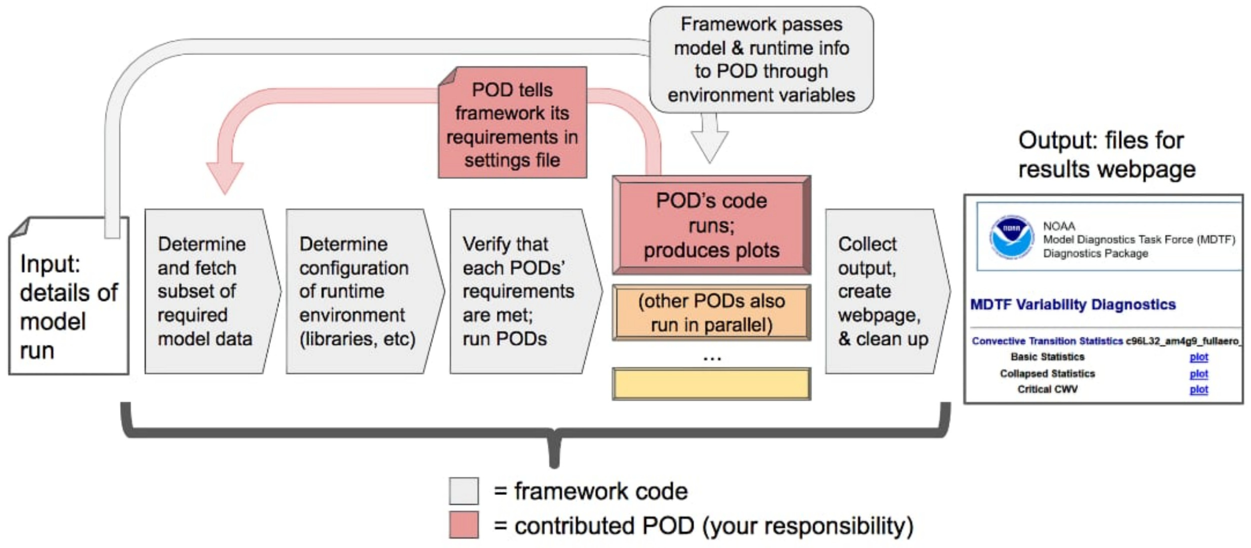
`path_variable`: Name of the shell environment variable the framework will use to pass the location of the file containing this variable to your diagnostic when it's run. See the environment variable documentation (page 29) for details.

`units`: The units the diagnostic expects the variable to be in (using the syntax of the UDUnits library[33]). This is optional: if not given, the framework will assume you want CF convention canonical units[34].

`dimensions`: List of names of dimensions specified in the "dimensions" section, to specify the coordinate dependence of each variable.

1.5 Walkthrough of framework operation

We now describe in greater detail the actions that are taken when the framework is run, focusing only on aspects that are relevant for the operation of individual PODs. For the rest of this walkthrough, the Example Diagnostic POD[35] is used as a concrete example to illustrate how a POD is implemented and integrated into the framework.


Framework passes model & runtime info to POD through environment variables
POD tells framework its requirements in settings file
Output: files for results webpage
Input: details of model run
Determine and fetch subset of required model data
Determine configuration of runtime environment (libraries, etc)
Verify that each PODs' requirements are met; run PODs
POD's code runs; produces plots
(other PODs also run in parallel)
...
Collect output, create webpage, & clean up
NOAA
Model Diagnostics Task Force (MDTF) Diagnostics Package
MDTF Variability Diagnostics
Convective Transition Statistics c96L32_am4g9_fullaero
Basic Statistics plot
Collapsed Statistics plot
Critical CWV plot
= framework code
= contributed POD (your responsibility)

[31] http://cfconventions.org/faq.html#vertical_coords_positive_attribute
[32] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html
[33] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax
[34] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html
[35] https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example

1.5.1 Framework invocation

The user runs the framework by executing the framework's driver script, rather than executing the PODs directly. This is where the user specifies the model run to be analyzed. The user can choose which PODs to run, with the default being all of them except for the example.

See section 3 of the Getting Started for more details on how the package is called. See the command line reference for documentation on command line options (or run `mdtf --help`).

1.5.2 Data request

Each POD describes the model data it requires as input in the `"varlist"` section of its settings file. The most important features of this file are described in the settings file (page 9) and documented in full detail on the reference page (page 19).) Each entry in the `varlist` section corresponds to one model data file used by the POD.

The framework goes through all the PODs to be run and assembles a top-level list of required model data from their `varlist` sections. It then queries the source of the model data for the presence of each requested variable with the requested characteristics.

Variables are specified in the settings file in a model-independent way, using CF convention[36] standard terminology wherever possible. If your POD takes derived quantities as input (column weighted averages, etc.) we recommend that you incorporate whatever preprocessing is necessary to compute these into your POD's code. Your POD may request variables outside of the CF conventions (by requiring an exact match on the variable name), but please be aware that this will severely limit the situations in which your POD will be run (see below).

It may be that some of the variables your POD requests are not available: they were not saved during the model run, or they weren't output at the requested frequency (or other characteristics). You have the option to specify a "backup plan" for this situation by designating sets of variables as "alternates," where this is scientifically feasible: if the framework is unable to obtain a variable that has the "alternates" attribute set in the `varlist`, it will then (and only then) query the model data source for the variables named as alternates.

If no alternates are defined or the alternate variables are not available, the framework concludes that it's unable to run the POD on the provided model data. Your POD's code will not be executed, and an error message listing the missing variables will be presented to the user in your POD's entry in the top-level results page.

Once the framework has determined which PODs are able to run given the model data, it downloads a local copy of the requested variables.

Example diagnostic

The example diagnostic uses only one model variable in its varlist[37]: surface air temperature, recorded at monthly frequency.

[36] http://cfconventions.org/

[37] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/d8d9f951d2c887b9a30fc496298815ab7ee68569/diagnostics/example/settings.jsonc#L46

1.5.3 Runtime environment configuration

In the first section of your POD's settings file, we request that you provide a list of programs your POD uses to run (names of interpreters for scripting languages, as well as any utility programs) and any third-party libraries they use.

The framework will check that all these programs and libraries are available on the system it's running on. The mechanism for doing so will differ, depending on whether the framework is making use of the conda package manager or not. If these dependencies are not found (for whatever reason), your POD will not be run and an error message will be presented to the user.

Example diagnostic

In its settings file, the example diagnostic lists its requirements[38] as the python language interpreter, and the matplotlib, xarray and netCDF4 third-party libraries for python. In this walkthrough, we assume the framework is managing its dependencies using the conda package manager, so the framework assigns the POD to run in the "python-base"[39] conda environment, which was created when the user installed the framework.

1.5.4 POD execution

At this point, your POD's requirements have been met, so the framework begins execution of your POD's code by calling the top-level script listed in your POD's settings file.

All information is passed from the framework to your POD in the form of unix shell environment variables; see the reference documentation for details on their names and values.

You should avoid making assumptions about the environment in which your POD will run beyond what's listed here; a development priority is to interface the framework with cluster and cloud job schedulers to enable individual PODs to run in a concurrent, distributed manner.

We encourage that your POD produce a log of its progress as it runs: this can be useful in debugging. All text your POD writes to stdout or stderr is captured in a log file and made available to the user.

If your POD experiences a fatal or unrecoverable error, it should signal that to the framework in the conventional unix way by exiting with a return code different from zero. This error will be presented to the user, who can then look over the log file to determine what went wrong.

POD execution: paths

Recall that installing the code will create a directory titled `MDTF-diagnostics` containing the files listed on the github page. Below we refer to this MDTF-diagnostics directory as `$CODE_ROOT`. It contains the following subdirectories: diagnostics/ : directories containing source code of individual PODs doc/ : directory containing documentation (a local mirror of the github wiki and documentation site) src/ : source code of the framework itself tests/ : unit tests for the framework Please refer to the Getting Started document, section 3 for background on the paths.

[38] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/d8d9f951d2c887b9a30fc496298815ab7ee68569/diagnostics/example/settings.jsonc#L38

[39] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/conda/env_python_base.yml

The most important environment variables set by the framework describe the location of resources your POD needs. To achieve the design goal of portability, you should ensure that no paths are hard-coded in your POD, for any reason. Instead, they should reference one of the following variable names (note `$POD_HOME` is used in linux shell and NCL; in Python `os.environ["POD_HOME"]` would be used):

- `$POD_HOME`: Path to the top-level directory containing your diagnostic's source code. This will be of the form …/MDTF-diagnostics/diagnostics/<your POD's name>. This can be used to call sub-scripts from your diagnostic's driver script. This directory should be treated as read-only.
- `$OBS_DATA`: Path to the top-level directory containing any digested observational or reference data you've provided as the author of your diagnostic. Files and sub-directories will be present within this directory with the names and layout in which you supplied them. The framework will ensure this is copied to a local filesystem when your diagnostic is run, but this directory should be treated as read-only. The path to each model data file is provided in an environment variable you name in that variable's entry in the varlist section of the settings file.
- `$WK_DIR`: path to your POD's working directory. This is the only location to which your POD should write files. Within this, the framework will create sub-directories which should be where your output is written:
- `$WK_DIR/obs/PS` and `$WK_DIR/model/PS`: All output plots produced by your diagnostic should be written to one of these two directories. Only files in these locations will be converted to bitmaps for HTML output.
- `$WK_DIR/obs/netCDF` and `$WK_DIR/model/netCDF`: Any output data files your diagnostic wants to make available to the user should be saved to one of these two directories.

Example diagnostic

The framework starts a unix subprocess, sets environment variables and the conda environment, and runs the example-diag.py[40] script in python. See comments in the code. The script reads the model surface air temperature data located at `$TAS_FILE`, and reference digested temperature data at `$OBS_DATA/example_tas_means.nc`.

The calculation performed by the example POD is chosen to be simple: it just does a time average of the model data. The observational data was supplied in time-averaged form, following the instructions for digested results above.

The model time averages are saved to `$WK_DIR/model/netCDF/temp_means.nc` for use by the user. Then both the observational and model means are plotted: the model plot is saved to `$WK_DIR/model/PS/example_model_plot.eps` and the observational data plot is saved to `$WK_DIR/obs/PS/example_obs_plot.eps`.

1.5.5 Output and cleanup

At this point, your POD has successfully finished running, and all remaining tasks are handled by the framework. The framework converts the postscript plots to bitmaps according to the following rule:

- `$WK_DIR/model/PS/<filename>.eps` → `$WK_DIR/model/filename.png`

[40] https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/diagnostics/example/example_diag.py

- `$WK_DIR/obs/PS/<filename>.eps` → `$WK_DIR/obs/filename.png`

The webpage template is copied to `$WK_DIR` by the framework, so in writing the template file all plots should be referenced as relative links to this location, eg. "`<A href=model/filename.png>`".

Values of all environment variables are substituted in the html template, allowing you to reference the run's `CASENAME` and date range. Beyond this, we don't offer a way to alter the text of your POD's output webpage at run time.

The framework links your POD's html page to the top-level `index.html` page, and copies all files to the specified output location.

1.6 Development Checklist

The following are the necessary steps for the module implementation and integration into the framework. The POD name tag used in the code should closely resemble the full POD name but should not contain any space bar or special characters. Note that the convective_transition_diag tag here is used repeatedly and consistently for the names of sub-directories, script, and html template. Please follow this convention so that mdtf.py can automatically process through the PODs.

All the modules currently included in the code package have the same structure, and hence the descriptions below apply:

1. Provide all the scripts for the convective_transition_diag POD in the sub-directory DIAG_HOME/var_code/convective_transition_diag. Among the provided scripts, there should be a template html file convective_transition_diag.html, and a main script convective_transition_diag.py that calls the other scripts in the same sub-directory for analyzing, plotting, and finalizing html.
2. Provide all the pre-digested observation data/figures in the sub-directory DATA_IN/obs_data/convective_transition_diag. One can create a new html template by simply copying and modifying the example templates in DIAG_HOME/var_code/html/html_template_examples. Note that scripts therein are exact replications of the html-related scripts in the example PODs, serving merely as a reference, and are not called by `mdtf.py`.
3. Provide documentation following the templates:
   A. Provide a comprehensive POD documentation, including a one-paragraph synopsis of the POD, developers' contact information, required programming language and libraries, and model output variables, a brief summary of the presented diagnostics as well as references in which more in-depth discussions can be found (see an example).
   B. All scripts should be self-documenting by including in-line comments. The main script convective_transition_diag.py should contain a comprehensive header providing information that contains the same items as in the POD documentation, except for the "More about this diagnostic" section.
   C. The one-paragraph POD synopsis (in the POD documentation) as well as a link to the Full Documentation should be placed at the top of the template convective_transition_diag.html (see example).
4. Test before distribution. It is important that developers test their POD before sending it to the MDTF contact. Please take the time to go through the following procedures:

A. Test how the POD fails. Does it stop with clear errors if it doesn't find the files it needs? How about if the dates requested are not presented in the model data? Can developers run it on data from another model? If it fails, does it stop the whole mdtf.py script? (It should contain an error-handling mechanism so the main script can continue). Have developers added any code to mdtf.py? (Do not change mdtf.py! — if you find some circumstance where it is essential, it should only be done in consultation with the MDTF contact).

B. Make a clean tar file. For distribution, a tar file with obs_data/, var_code/, namelist, and model data that developers have thoroughly tested is needed. These should not include any extraneous files (output NetCDF, output figures, backups, `pyc`, `*~`, or `#` files). The model data used to test (if different from what is provided by the MDTF page) will need to be in its own tar file. Use `tar -tf` to see what is in the tar file. Developers might find it helpful to consult the script used to make the overall distributions mdtf/make_tars.sh.

C. Final testing: Once a tar file is made, please test it in a clean location where developers haven't run it before. If it fails, repeat steps 1)-3) until it passes. Next, ask a colleague or assign a group member not involved in the development to test it as well — download to a new machine to install, run, and ask for comments on whether they can understand the documentation.

5. Post on an ftp site and/or email the MDTF contact.

1.7 General developer resources

The following links to third-party pages contain information that may be helpful.

1.7.1 Git tutorials/references

- The official git tutorial[41].
- A more verbose introduction[42] to the ideas behind git and version control.
- A still more detailed walkthrough[43] which assumes no prior knowledge.

1.7.2 Python coding style

- PEP8[44], the officially recognized Python style guide.
- Google's Python style guide[45].

[41] https://git-scm.com/docs/gittutorial
[42] https://www.atlassian.com/git/tutorials/what-is-version-control
[43] http://swcarpentry.github.io/git-novice/
[44] https://www.python.org/dev/peps/pep-0008/
[45] https://github.com/google/styleguide/blob/gh-pages/pyguide.md

1.7.3 Code documentation

Documentation for the framework's code is managed using sphinx[46], which works with files in reStructured text[47] (reST, `.rst`) format. The framework uses Google style conventions for python docstrings.

- Sphinx quickstart[48].
- reStructured text introduction[49], quick reference[50] and in-depth guide[51].
- reST syntax comparison[52] to other text formats you may be familiar with.
- Style guide for google-style python docstrings[53] and quick examples[54].

[46] https://www.sphinx-doc.org/en/master/index.html
[47] https://docutils.sourceforge.io/rst.html
[48] http://www.sphinx-doc.org/en/master/usage/quickstart.html
[49] http://docutils.sourceforge.net/docs/user/rst/quickstart.html
[50] http://docutils.sourceforge.net/docs/user/rst/quickref.html
[51] http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html
[52] http://hyperpolyglot.org/lightweight-markup
[53] https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings
[54] https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html

CHAPTER

TWO

FRAMEWORK REFERENCE

2.1 Diagnostic settings file format

The settings file is how your diagnostic tells the framework what it needs to run, in terms of software and model data.

Each diagnostic must contain a text file named `settings.jsonc` in the JSON[55] format, with the addition that any text to the right of `//` is treated as a comment and ignored (sometimes called the "JSONC" format).

2.1.1 Brief summary of JSON

We'll briefly summarize subset of JSON syntax used in this configuration file. The file's JSON expressions are built up out of items, which may be either

1. a boolean, taking one of the values `true` or `false` (lower-case, with no quotes).
2. a number (integer or floating-point).
3. a case-sensitive string, which must be delimited by double quotes.

In addition, for the purposes of the configuration file we define

4. a "unit-ful quantity": this is a string containing a number followed by a unit, eg. `"6hr"`. In addition, the string `"any"` may be used to signify that any value is acceptable.

Items are combined in compound expressions of two types:

5. arrays, which are one-dimensional ordered lists delimited with square brackets. Entries can be of any type, eg `[true, 1, "two"]`.
6. objects, which are un-ordered lists of key:value pairs separated by colons and delimited with curly brackets. Keys must be strings and must all be unique within the object, while values may be any expression, eg. `{"red": 0, "green": false, "blue": "bagels"}`.

Compound expressions may be nested within each other to an arbitrary depth.

[55] https://en.wikipedia.org/wiki/JSON#Data_types_and_syntax

2.1.2 File organization

```
{
  "settings" : {
    <...properties describing the diagnostic..>
  },
  "data" : {
    <...properties for all requested model data...>
  },
  "dimensions" : {
    "my_first_dimension": {
      <...properties describing this dimension...>
    },
    "my_second_dimension": {
      <...properties describing this dimension...>
    },
    ...
  },
  "varlist" : {
    "my_first_variable": {
      <...properties describing this variable...>
    },
    "my_second_variable": {
      <...properties describing this variable...>
    },
    ...
  }
}
```

At the top level, the settings file is an object (page 19) containing four required entries, described in detail below.

- settings (page 20): properties that label the diagnostic and describe its runtime requirements.
- data (page 22): properties that apply to all the data your diagnostic is requesting.
- dimensions (page 24): properties that apply to the dimensions (in netCDF[56] terminology) of the model data. Each distinct dimension (coordinate axis) of the data being requested should be listed as a separate entry here.
- varlist (page 26): properties that describe the individual variables your diagnostic operates on. Each variable should be listed as a separate entry here.

2.1.3 Settings section

This section is an object (page 19) containing properties that label the diagnostic and describe its runtime requirements.

[56] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html

Example

```
"settings" : {
  "long_name" : "Effect of X on Y diagnostic",
  "driver" : "my_script.py",
  "realm" : ["atmos", "ocean"],
  "runtime_requirements": {
    "python": ["numpy", "matplotlib", "netCDF4", "cartopy"],
    "ncl": ["contributed", "gsn_code", "gsn_csm"]
  },
  "pod_env_vars" : {
    // RES: Spatial Resolution (degree) for Obs Data (0.25, 0.50, 1.00).
    "RES": "1.00"
  }
}
```

Diagnostic description

`long_name`: String, required. Human-readable display name of your diagnostic. This is the text used to describe your diagnostic on the top-level index.html page. It should be in sentence case (capitalize first word and proper nouns only) and omit any punctuation at the end.

`driver`: String, required. Filename of the top-level driver script the framework should call to run your diagnostic's analysis.

`realm`: String or array (page 19) (list) of strings, required. One of the eight CMIP6 modeling realms (aerosol, atmos, atmosChem, land, landIce, ocean, ocnBgchem, seaIce) describing what data your diagnostic uses. If your diagnostic uses data from multiple realms, list them in an array (eg. `["atmos", "ocean"]`). This information doesn't affect how the framework fetches model data for your diagnostic: it's provided to give the user a shortcut to say, eg., "run all the atmos diagnostics on this output."

Diagnostic runtime

`runtime_requirements`: object (page 19), required. Programs your diagnostic needs to run (for example, scripting language interpreters) and any third-party libraries needed in those languages. Each executable should be listed in a separate key-value pair:

- The key is the name of the required executable, eg. languages such as "python[57]" or "ncl[58]" etc. but also any utilities such as "ncks[59]", "cdo[60]", etc.
- The value corresponding to each key is an array (page 19) (list) of strings, which are names of third-party libraries in that language that your diagnostic needs. You do not need to list standard libraries or scripts that are provided in a standard installation of your language: eg, in python,

[57] https://www.python.org/
[58] https://www.ncl.ucar.edu/
[59] http://nco.sourceforge.net/
[60] https://code.mpimet.mpg.de/projects/cdo

you need to list numpy[61] but not math[62]. If no third-party libraries are needed, the value should be an empty list.

In the future we plan to offer the capability to request specific versions[63]. For now, please communicate your diagnostic's version requirements to the MDTF organizers.

`pod_env_vars`: object (page 19), optional. Names and values of shell environment variables used by your diagnostic, in addition to those supplied by the framework. The user can't change these at runtime, but this can be used to set site-specific installation settings for your diagnostic (eg, switching between low- and high-resolution observational data depending on what the user has chosen to download). Note that environment variable values must be provided as strings.

2.1.4 Data section

This section is an object (page 19) containing properties that apply to all the data your diagnostic is requesting.

Example

```
"data": {
  "format": "netcdf4_classic",
  "rename_dimensions": false,
  "rename_variables": false,
  "multi_file_ok": true,
  "frequency": "3hr",
  "min_frequency": "1hr",
  "max_frequency": "6hr",
  "min_duration": "5yr",
  "max_duration": "any"
}
```

Example

`format`: String. Optional: assumed `"any_netcdf_classic"` if not specified. Specifies the format(s) of model data your diagnostic is able to read. As of this writing, the framework only supports retrieval of netCDF formats, so only the following values are allowed:

- `"any_netcdf"` includes all of:
  - `"any_netcdf3"` includes all of:
    * `"netcdf3_classic"` (CDF-1, files restricted to < 2 Gb)
    * `"netcdf3_64bit_offset"` (CDF-2)
    * `"netcdf3_64bit_data"` (CDF-5)

[61] https://numpy.org/
[62] https://docs.python.org/3/library/math.html
[63] https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/pkg-specs.html#package-match-specifications

  - `"any_netcdf4"` includes all of:
    - `"netcdf4_classic"`
    - `"netcdf4"`
- `"any_netcdf_classic"` includes all the above except `"netcdf4"` (classic data model only).

See the netCDF FAQ[64] (under "Formats, Data Models, and Software Releases") for information on the distinctions. Any recent version of a supported language for diagnostics with netCDF support will be able to read all of these. However, the extended features of the `"netcdf4"` data model are not commonly used in practice and currently only supported at a beta level in NCL, which is why we've chosen `"any_netcdf_classic"` as the default.

`rename_dimensions`: Boolean. Optional: assumed `false` if not specified. If set to `true`, the framework will change the name of all dimensions (page 24) in the model data from the model's native value to the string specified in the `name` property for that dimension. If set to `false`, the diagnostic is responsible for reading dimension names from the environment variable. See the environment variable documentation (page 29) for details on how these names are provided.

`rename_variables`: Boolean. Optional: assumed `false` if not specified. If set to `true`, the framework will change the name of all variables (page 26) in the model data from the model's native value to the string specified in the `name` property for that variable. If set to `false`, the diagnostic is responsible for reading dimension names from the environment variable. See the environment variable documentation (page 29) for details on how these names are provided.

`multi_file_ok`: Boolean. Optional: assumed `false` if not specified. If set to `true`, the diagnostic can handle datasets for a single variable spread across multiple files, eg xarray[65].

`min_duration`, **`max_duration`**: Unit-ful quantities (page 19). Optional: assumed `"any"` if not specified. Set minimum and maximum length of the analysis period for which the diagnostic should be run: this overrides any choices the user makes at runtime. Some example uses of this setting are:

- If your diagnostic uses low-frequency (eg seasonal) data, you may want to set `min_duration` to ensure the sample size will be large enough for your results to be statistically meaningful.
- On the other hand, if your diagnostic uses high-frequency (eg hourly) data, you may want to set `max_duration` to prevent the framework from attempting to download a large volume of data for your diagnostic if the framework is called with a multi-decadal analysis period.

The following properties can optionally be set individually for each variable in the varlist section (page 26). If so, they will override the global settings given here.

`dimensions_ordered`: Boolean. Optional: assumed `false` if not specified. If set to `true`, the framework will ensure that the dimensions of each variable's array are given in the same order as listed in `dimensions`. If set to false, your diagnostic is responsible for handling arbitrary dimension orders: eg. it should not assume that 3D data will be presented as (time, lat, lon).

`frequency`, **`min_frequency`**, **`max_frequency`**: Unit-ful quantities (page 19). Time frequency at which the data is provided. Either `frequency` or the min/max pair, or both, is required:

[64] https://www.unidata.ucar.edu/software/netcdf/docs/faq.html
[65] http://xarray.pydata.org/en/stable/generated/xarray.open_mfdataset.html

- If only `frequency` is provided, the framework will attempt to obtain data at that frequency. If that's not available from the data source, your diagnostic will not run.
- If the min/max pair is provided, the diagnostic must be capable of using data at any frequency within that range (inclusive). The diagnostic is responsible for determining the frequency if this option is used.
- If all three properties are set, the framework will first attempt to find data at `frequency`. If that's not available, it will try data within the min/max range, so your code must be able to handle this possibility.

2.1.5 Dimensions section

This section is an object (page 19) contains properties that apply to the dimensions of model data. "Dimensions" are meant in the sense of the netCDF data model[66]: informally, they are "coordinate axes" holding the values of independent variables that the dependent variable is sampled at.

All dimensions (page 27) and scalar coordinates (page 28) referenced by variables in the varlist section must have an entry in this section. If two variables reference the same dimension, they will be sampled on the same set of values.

Note that the framework only supports the (simplest and most common) "independent axes" case of the CF conventions[67]. In particular, the framework only deals with data on lat-lon grids.

Example

```
"dimensions": {
  "lat": {
      "standard_name": "latitude",
      "units": "degrees_N",
      "range": [-90, 90],
      "need_bounds": false
  },
  "lon": {
      "standard_name": "longitude",
      "units": "degrees_E",
      "range": [-180, 180],
      "need_bounds": false
  },
  "plev": {
      "standard_name": "air_pressure",
      "units": "hPa",
      "positive": "down",
      "need_bounds": false
```

(continues on next page)

[66] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html

[67] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#_independent_latitude_longitude_vertical_and_time_axes

(continued from previous page)

```
  },
  "time": {
      "standard_name": "time",
      "units": "days",
      "calendar": "noleap",
      "need_bounds": false
  }
}
```

Latitude and Longitude

`standard_name`: Required, string. Must be `"latitude"` and `"longitude"`, respectively.

`units`: Optional. String, following syntax of the UDUnits library[68]. Units the diagnostic expects the dimension to be in. Currently the framework only supports decimal `degrees_north` and `degrees_east`, respectively.

`range`: Array (page 19) (list) of two numbers. Optional. If given, specifies the range of values the diagnostic expects this dimension to take. For example, `"range": [-180, 180]` for longitude will have the first entry of the longitude variable in each data file be near -180 degrees (not exactly -180, because dimension values are cell midpoints), and the last entry near +180 degrees.

`need_bounds`: Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that bounds are supplied for this dimension, in addition to its midpoint values, following the CF conventions[69]: the `bounds` attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

Time

`standard_name`: Required. Must be `"time"`.

`units`: String. Optional, defaults to "day". Units the diagnostic expects the dimension to be in. Currently the diagnostic only supports time axes of the form "<units> since <reference data>", and the value given here is interpreted in this sense (eg. settings this to "day" would accommodate a dimension of the form "[decimal] days since 1850-01-01".)

`calendar`: String, Optional. One of the CF convention calendars[70] or the string `"any"`. Defaults to "any" if not given. Calendar convention used by your diagnostic. Only affects the number of days per month.

`need_bounds`: Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that bounds are supplied for this dimension, in addition to its midpoint values, following the CF conventions[71]: the `bounds` attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

[68] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax
[69] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries
[70] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#calendar
[71] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries

Z axis (height/depth, pressure, ...)

`standard_name`: Required, string. Standard name[72] of the variable as defined by the CF conventions[73], or a commonly used synonym as employed in the CMIP6 MIP tables.

`units`: Optional. String, following syntax of the UDUnits library[74]. Units the diagnostic expects the dimension to be in. If not provided, the framework will assume CF convention canonical units[75].

`positive`: String, required. Must be `"up"` or `"down"`, according to the CF conventions[76]. A pressure axis is always `"down"` (increasing values are closer to the center of the earth), but this is not set automatically.

`need_bounds`: Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that bounds are supplied for this dimension, in addition to its midpoint values, following the CF conventions[77]: the `bounds` attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

Other dimensions (wavelength, ...)

`standard_name`: Required, string. Standard name[78] of the variable as defined by the CF conventions[79], or a commonly used synonym as employed in the CMIP6 MIP tables.

`units`: Optional. String, following syntax of the UDUnits library[80]. Units the diagnostic expects the dimension to be in. If not provided, the framework will assume CF convention canonical units[81].

`need_bounds`: Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that bounds are supplied for this dimension, in addition to its midpoint values, following the CF conventions[82]: the `bounds` attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

2.1.6 Varlist section

This section is an object (page 19) contains properties that apply to the model variables your diagnostic needs for its analysis. "Dimensions" are meant in the sense of the netCDF data model[83]: informally, they are the "independent variables" whose values are being computed as a function of the values stored in the dimensions.

Each entry corresponds to a distinct data file (or set of files, if `multi_file_ok` is `true`) downloaded by the framework. If your framework needs the same physical quantity sampled with different properties (eg. slices of a variable at multiple pressure levels), specify them as multiple entries.

[72] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html
[73] http://cfconventions.org/
[74] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax
[75] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html
[76] http://cfconventions.org/faq.html#vertical_coords_positive_attribute
[77] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries
[78] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html
[79] http://cfconventions.org/
[80] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax
[81] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html
[82] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries
[83] https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcVars.html

Varlist entry example

```
"u500": {
    "standard_name": "eastward_wind",
    "path_variable": "U500_FILE",
    "units": "m s-1",
    "dimensions" : ["time", "lat", "lon"],
    "dimensions_ordered": true,
    "scalar_coordinates": {"pressure": 500},
    "requirement": "optional",
    "alternates": ["another_variable_name", "a_third_variable_name"]
}
```

Varlist entry properties

The key in a varlist key-value pair is the name your diagnostic uses to refer to this variable (and must be unique). The value of the key-value pair is an object (page 19) containing properties specific to that variable:

`standard_name`: String, required. Standard name[84] of the variable as defined by the CF conventions[85], or a commonly used synonym as employed in the CMIP6 MIP tables (eg. "ua" instead of "eastward_wind").

`path_variable`: String, required. Name of the shell environment variable the framework will set with the location of this data. See the environment variable documentation (page 29) for details.

- If `multi_file_ok` is `false`, `<path_variable>` will be set to the absolute path to the netcdf file containing this variable's data.
- If `multi_file_ok` is `true`, `<path_variable>` will be a single path or a colon-separated list of paths to the files containing this data. Files will be listed in chronological order.
- If the variable is listed as `"optional"` or `"alternate"` or has `alternate` variables listed, `<path_variable>` will be defined but set to the empty string if the framework couldn't obtain this data from the data source. Your diagnostic should test for this possibility.

`use_exact_name`: Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ignore the model's naming conventions and only look for a variable with a name matching the key of this entry, regardless of what model or data source the framework is using. The only use case for this setting is to give diagnostics the ability to request data that falls outside the CF conventions: in general, you should rely on the framework to translate CF standard names to the native field names of the model being analyzed.

`units`: Optional. String, following syntax of the UDUnits library[86]. Units the diagnostic expects the variable to be in. If not provided, the framework will assume CF convention canonical units[87].

[84] http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html
[85] http://cfconventions.org/
[86] https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax
[87] http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html

`dimensions`: Required. List of strings, which must be selected the keys of entries in the dimensions (page 24) section. Dimensions of the array containing the variable's data. Note that the framework will not reorder dimensions (transpose) unless `dimensions_ordered` is additionally set to `true`.

`dimensions_ordered`: Boolean. Optional: assumed `false` if not specified. If `true`, the framework will ensure that the dimensions of this variable's array are given in the same order as listed in `dimensions`. If set to false, your diagnostic is responsible for handling arbitrary dimension orders: eg. it should not assume that 3D data will be presented as (time, lat, lon). If given here, overrides the values set globally in the `data` section (see description (page 23) there).

`scalar_coordinates`: object (page 19), optional. This implements what the CF conventions refer to as "scalar coordinates[88]", with the use case here being the ability to request slices of higher-dimensional data. For example, the snippet at the beginning of this section shows how to request the u component of wind velocity on a 500 mb pressure level.

- keys are the key (name) of an entry in the dimensions (page 24) section.
- values are a single number (integer or floating-point) corresponding to the value of the slice to extract. Units of this number are taken to be the `units` property of the dimension named as the key.

In order to request multiple slices (eg. wind velocity on multiple pressure levels, with each level saved to a different file), create one varlist entry per slice.

`frequency`, `min_frequency`, `max_frequency`: Unit-ful quantities (page 19). Optional. Time frequency at which the variable's data is provided. If given here, overrides the values set globally in the `data` section (see description (page 23) there).

`requirement`: String. Optional: assumed `"required"` if not specified. One of three values:

- `"required"`: variable is necessary for the diagnostic's calculations. If the data source doesn't provide the variable (at the requested frequency, etc., for the user-specified analysis period) the framework will not run the diagnostic, but will instead log an error message explaining that the lack of this data was at fault.
- `"optional"`: variable will be supplied to the diagnostic if provided by the data source. If not available, the diagnostic will still run, and the `path_variable` for this variable will be set to the empty string. The diagnostic is responsible for testing the environment variable for the existence of all optional variables.
- `"alternate"`: variable is specified as an alternate source of data for some other variable (see next property). The framework will only query the data source for this variable if it's unable to obtain one of the other variables that list it as an alternate.

`alternates`: Array (page 19) (list) of strings, which must be keys (names) of other variables. Optional: if provided, specifies an alternative method for obtaining needed data if this variable isn't provided by the data source.

- If the data source provides this variable (at the requested frequency, etc., for the user-specified analysis period), this property is ignored.
- If this variable isn't available as requested, the framework will query the data source for all of the variables listed in this property. If all of the alternate variables are available, the diagnostic will

[88] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#scalar-coordinate-variables

be run; if any are missing it will be skipped. Note that, as currently implemented, only one set of alternates may be given (no "plan B", "plan C", etc.)

2.2 MDTF Environment variables

This page describes the environment variables that the framework will set for your diagnostic when it's run.

2.2.1 Overview

The MDTF framework can be viewed as a "wrapper" for your code that handles data fetching and munging. Your code communicates with this wrapper in two ways:

- The settings file (page 9) is where your code talks to the framework: when you write your code, you document what model data your code uses (not covered on this page, follow the link for details).
- When your code is run, the framework talks to it by setting shell environment variables[89] containing paths to the data files and other information specific to the run. The framework communicates all runtime information this way: this is in order to 1) pass information in a language-independent way, and 2) to make writing diagnostics easier (you don't need to parse command-line settings).

Note that environment variables are always strings. Your script will need to cast non-text data to the appropriate type (eg. the bounds of the analysis time period, `FIRSTYR`, `LASTYR`, will need to be converted to integers.)

Also note that names of environment variables are case-sensitive.

2.2.2 Paths

`OBS_DATA`: Path to the top-level directory containing any observational or reference data you've provided as the author of your diagnostic. Any data your diagnostic uses that doesn't come from the model being analyzed should go here (ie, you supply it to the framework maintainers, they host it, and the user downloads it when they install the framework). The framework will ensure this is copied to a local filesystem when your diagnostic is run, but this directory should be treated as read-only.

`POD_HOME`: Path to the top-level directory containing your diagnostic's source code. This will be of the form `.../MDTF-diagnostics/diagnostics/<your POD's name>`. This can be used to call sub-scripts from your diagnostic's driver script. This directory should be treated as read-only.

`WK_DIR`: Path to your diagnostic's working directory, which is where all output data should be written (as well as any temporary files).

The framework creates the following subdirectories within this directory:

- `$WK_DIR/obs/PS` and `$WK_DIR/model/PS`: All output plots produced by your diagnostic should be written to one of these two directories. Only files in these locations will be converted to bitmaps for HTML output.

[89] https://en.wikipedia.org/wiki/Environment_variable

- `$WK_DIR/obs/netCDF` and `$WK_DIR/model/netCDF`: Any output data files your diagnostic wants to make available to the user should be saved to one of these two directories.

2.2.3 Model run information

`CASENAME`: User-provided label describing the run of model data being analyzed.

`FIRSTYR`, `LASTYR`: Four-digit years describing the analysis period.

2.2.4 Locations of model data files

These are set depending on the data your diagnostic requests in its settings file (page 9). Refer to the examples below if you're unfamiliar with how that file is organized.

Each variable listed in the `varlist` section of the settings file must specify a `path_variable` property. The value you enter there will be used as the name of an environment variable, and the framework will set the value of that environment variable to the absolute path to the file containing data for that variable.

From a diagnostic writer's point of view, this means all you need to do here is replace paths to input data that are hard-coded or passed from the command line with calls to read the value of the corresponding environment variable.

- If the framework was not able to obtain the variable from the data source (at the requested frequency, etc., for the user-specified analysis period), this variable will be set equal to the empty string. Your diagnostic is responsible for testing for this possibility for all variables that are listed as `optional` or have alternates listed (if a required variable without alternates isn't found, your diagnostic won't be run.)
- If `multi_file_ok` is set to `true` in the settings file, this environment variable may be a list of paths to multiple files in chronological order, separated by colons. For example, `/dir/precip_1980_1989.nc:/dir/precip_1990_1999.nc:/dir/precip_2000_2009.nc` for an analysis period of 1980-2009.

2.2.5 Names of variables and dimensions

These are set depending on the data your diagnostic requests in its settings file (page 9). Refer to the examples below if you're unfamiliar with how that file is organized.

For each dimension: If <key> is the name of the key labeling the key:value entry for this dimension, the framework will set an environment variable named `<key>_dim` equal to the name that dimension has in the data files it's providing.

- If `rename_dimensions` is set to `true` in the settings file, this will always be equal to <key>. If If `rename_dimensions` is `false`, this will be whatever the model or data source's native name for this dimension is, and your diagnostic should read the name from this variable. Your diagnostic should only use hard-coded names for dimensions if `rename_dimensions` is set to `true` in its settings file (page 19).

For each variable: If <key> be the name of the key labeling the key:value entry for this variable in the varlist section, the framework will set an environment variable named `<key>_var` equal to the name that variable has in the data files it's providing.

- If `rename_variables` is set to `true` in the settings file, this will always be equal to <key>. If If `rename_variables` is `false`, this will be whatever the model or data source's native name for this variable is, and your diagnostic should read the name from this variable. Your diagnostic should only use hard-coded names for dimensions if `rename_dimensions` is set to `true` in its settings file (page 19).

2.2.6 Simple example

We only give the relevant parts of the settings file (page 19) below.

```
"data": {
  "rename_dimensions": false,
  "rename_variables": false,
  "multi_file_ok": false,
  ...
},
"dimensions": {
  "lat": {
    "standard_name": "latitude",
    ...
  },
  "lon": {
    "standard_name": "longitude",
    ...
  },
  "time": {
    "standard_name": "time",
    ...
  }
},
"varlist": {
  "pr": {
    "standard_name": "precipitation_flux",
    "path_variable": "PR_FILE"
  }
}
```

The framework will set the following environment variables:

1. `lat_dim`: Name of the latitude dimension in the model's native format (because `rename_dimensions` is false).
2. `lon_dim`: Name of the longitude dimension in the model's native format (because `rename_dimensions` is false).

3. `time_dim`: Name of the time dimension in the model's native format (because `rename_dimensions` is false).
4. `pr_var`: Name of the precipitation variable in the model's native format (because `rename_variables` is false).
5. `PR_FILE`: Absolute path to the file containing `pr` data, eg. `/dir/precip.nc`.

2.2.7 More complex example

Let's elaborate on the previous example, and assume that the diagnostic is being called on model that provides precipitation_flux but not convective_precipitation_flux.

```
"data": {
  "rename_dimensions": true,
  "rename_variables": false,
  "multi_file_ok": true,
  ...
},
"dimensions": {
  "lat": {
    "standard_name": "latitude",
    ...
  },
  "lon": {
    "standard_name": "longitude",
    ...
  },
  "time": {
    "standard_name": "time",
    ...
  }
},
"varlist": {
  "prc": {
    "standard_name": "convective_precipitation_flux",
    "path_variable": "PRC_FILE",
    "alternates": ["pr"]
  },
  "pr": {
    "standard_name": "precipitation_flux",
    "path_variable": "PR_FILE"
  }
}
```

Comparing this with the previous example:

- `lat_dim`, `lon_dim` and `time_dim` will be set to "lat", "lon" and "time", respectively, because `rename_dimensions` is true. The framework will have renamed these dimensions to have these names in all data files provided to the diagnostic.

- `prc_var` and `pr_var` will be set to the model's native names for these variables. Names for all variables are always set, regardless of which variables are available from the data source.
- In this example, `PRC_FILE` will be set to `''`, the empty string, because it wasn't found.
- `PR_FILE` will be set to `/dir/precip_1980_1989.nc:/dir/precip_1990_1999.nc:/dir/precip_2000_2009.nc`, because `multi_file_ok` was set to `true`.

CHAPTER THREE

ACKNOWLEDGEMENTS

Development of this code framework for process-oriented diagnostics was supported by the National Oceanic and Atmospheric Administration[90] (NOAA) Climate Program Office Modeling, Analysis, Predictions and Projections[91] (MAPP) Program (grant # NA18OAR4310280). Additional support was provided by University of California Los Angeles[92], the Geophysical Fluid Dynamics Laboratory[93], the National Center for Atmospheric Research[94], Colorado State University[95], Lawrence Livermore National Laboratory[96] and the US Department of Energy[97].

Many of the process-oriented diagnostics modules (PODs) were contributed by members of the NOAA Model Diagnostics Task Force[98] under MAPP support. Statements, findings or recommendations in these documents do not necessarily reflect the views of NOAA or the US Department of Commerce.

3.1 Disclaimer

[90] https://www.noaa.gov/
[91] https://cpo.noaa.gov/Meet-the-Divisions/Earth-System-Science-and-Modeling/MAPP
[92] https://www.ucla.edu/
[93] https://www.gfdl.noaa.gov/
[94] https://ncar.ucar.edu/
[95] https://www.colostate.edu/
[96] https://www.llnl.gov/
[97] https://www.energy.gov/
[98] https://cpo.noaa.gov/Meet-the-Divisions/Earth-System-Science-and-Modeling/MAPP/MAPP-Task-Forces/Model-Diagnostics-Task-Force