MDTF Diagnostics Documentation

Release 3.0 beta 2

Model Diagnostics Task Force

Oct 01, 2020
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Getting started</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Quickstart installation instructions</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Framework configuration for user model data</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Site-specific information</td>
<td>12</td>
</tr>
<tr>
<td>2.1</td>
<td>GFDL-specific information</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Developer information</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction for POD developers</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Migration from framework v2.0</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>POD development checklist</td>
<td>18</td>
</tr>
<tr>
<td>3.4</td>
<td>Developer quickstart guide</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>POD development guidelines</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>POD settings file summary</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Walkthrough of framework operation</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>POD coding best practices</td>
<td>37</td>
</tr>
<tr>
<td>3.9</td>
<td>Git-based development workflow</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>Diagnostics reference</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of MDTF process-oriented diagnostics</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Example Diagnostic Documentation</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Convective Transition Diagnostic Package</td>
<td>54</td>
</tr>
<tr>
<td>4.4</td>
<td>ENSO Moist Static Energy Diagnostic Package</td>
<td>59</td>
</tr>
<tr>
<td>4.5</td>
<td>EOF of Geopotential Height Diagnostic Module From NCAR</td>
<td>64</td>
</tr>
<tr>
<td>4.6</td>
<td>MJO Propagation and Amplitude Diagnostic Package</td>
<td>66</td>
</tr>
<tr>
<td>4.7</td>
<td>MJO Suite Diagnostic Module From NCAR</td>
<td>70</td>
</tr>
<tr>
<td>4.8</td>
<td>MJO Teleconnection Diagnostic Package</td>
<td>75</td>
</tr>
<tr>
<td>4.9</td>
<td>Phase and Amplitude of Precipitation Diurnal Cycle</td>
<td>80</td>
</tr>
<tr>
<td>4.10</td>
<td>Soil moisture-Evapotranspiration Coupling Diagnostic Package</td>
<td>83</td>
</tr>
<tr>
<td>4.11</td>
<td>Wavenumber Frequency Spectra Diagnostic Module From NCAR</td>
<td>86</td>
</tr>
<tr>
<td>5</td>
<td>Framework reference</td>
<td>88</td>
</tr>
<tr>
<td>5.1</td>
<td>Command-line option reference</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>Diagnostic settings file format</td>
<td>91</td>
</tr>
<tr>
<td>5.3</td>
<td>MDTF Environment variables</td>
<td>101</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6</td>
<td>Internal framework code</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>src.mdtf module</td>
<td>105</td>
</tr>
<tr>
<td>6.2</td>
<td>src.data_manager module</td>
<td>106</td>
</tr>
<tr>
<td>6.3</td>
<td>src.environment_manager module</td>
<td>109</td>
</tr>
<tr>
<td>6.4</td>
<td>src.shared_diagnostic module</td>
<td>111</td>
</tr>
<tr>
<td>6.5</td>
<td>src.util module</td>
<td>115</td>
</tr>
<tr>
<td>7</td>
<td>Acknowledgements</td>
<td>121</td>
</tr>
<tr>
<td>7.1</td>
<td>Disclaimer</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>Python Module Index</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>123</td>
</tr>
</tbody>
</table>
1.1 Overview

Welcome! In this section we’ll describe what the Model Diagnostics Task Force (MDTF) framework is, how it works, and how you can contribute your own diagnostic scripts.

1.1.1 Purpose

The scientific motivation and content behind the framework was described in E. D. Maloney et al. (2019): Process-Oriented Evaluation of Climate and Weather Forecasting Models. BAMS, 100 (9), 1665–1686, doi:10.1175/BAMS-D-18-0042.1.¹

Also see the section of this site devoted to documentation of individual diagnostics (page 46).

¹ https://doi.org/10.1175/BAMS-D-18-0042.1
1.1.2 Framework operation

The design goal of the MDTF framework is to provide a portable and adaptable means to run process-oriented diagnostic scripts, abbreviated as PODs below. By “portability,” we mean the ideal of “run once, run anywhere”: the purpose of the framework is to automate retrieval of model data from different local or remote sources, and transform that data into a layout (field names, variable units, etc.) your script expects. This will empower your analysis to be run by a wider range of researchers on a wider range of models.

As shown in the figure above, the MDTF framework itself performs common data management and support tasks (gray boxes) before and after the individual POD scripts are run. The PODs (colored boxes) are developed by different research groups and run independently of one another. Each POD takes as input

1. requested variables from the model run, along with
2. any required observational or supporting data, performs an analysis, and produces
3. a set of figures which are presented to the user in a series of .html files.

We do not include or require a mechanism for publishing these webpages on the internet; html is merely used as a convenient way to present a multimedia report to the user.

1.1.3 Getting started for users

The rest of the documentation in this section describes next steps for end users of the framework:

- We provide instructions on how to download and install (page 3) the framework and run it on sample model data.

- We describe the most common configuration options (page 8) for running the framework on your own model data. Also see the full list of command-line options (page 88).

- If you encounter a bug, check the GitHub issue tracker².

² https://github.com/NOAA-GFDL/MDTF-diagnostics/issues
1.1.4 Getting started for POD developers

Information for researchers wishing to contribute a POD to the framework is provided in the Developer section; consult the quickstart guide for an overview and the checklist of items needed for submitting your POD.

The framework is designed to require minimal changes to existing analysis scripts. We recommend that developers of new PODs start independently of the framework and adapt it for the framework’s use once it’s fully debugged. As summarized in the figure above, the changes needed to convert an existing analysis script for use in the framework are:

- Provide a settings file which tells the framework what it needs to do: what languages and libraries your code need to run, and what model data your code takes as input.
- Adapt your code to load data files from locations set in unix shell environment variables (we use this as a language-independent way for the framework to communicate information to the POD).
- Provide a template web page which links to, and briefly describes, the plots generated by the script.

1.2 Quickstart installation instructions

This section provides instructions for downloading, installing and running a test of the MDTF framework using sample model data. The MDTF framework has been tested on UNIX/LINUX, Mac OS, and Windows Subsystem for Linux.

Throughout this document, % indicates the command line prompt and is followed by commands to be executed in a terminal in fixed-width font. $ indicates strings to be substituted, e.g., the string $CODE_ROOT below should be replaced by the actual path to the MDTF-diagnostics directory.

Summary of steps for installing the framework

You will need to download the code, digested observational data, and sample model data (Section 1.2.1). Afterwards, we describe how to install software dependencies using the conda package manager (Section 1.2.2, Section 1.2.3) and run the framework on sample model data (Section 1.2.4 and Section 1.2.5).

1.2.1 Download the framework code and supporting data

Obtaining the code

The official repo for the MDTF code is hosted at the NOAA-GFDL GitHub account. We recommend that end users download and test the latest official release.

To install the MDTF framework, create a directory named mdtf and unzip the code downloaded from the release page there. This will create a directory titled MDTF-diagnostics-3.0-beta.2 containing the files

---

3 https://docs.conda.io/en/latest/
4 https://github.com/NOAA-GFDL/MDTF-diagnostics
5 https://github.com/NOAA-GFDL/MDTF-diagnostics/releases/tag/v3.0-beta.2
6 https://github.com/NOAA-GFDL/MDTF-diagnostics/releases/tag/v3.0-beta.2
listed on the GitHub page. Below we refer to this MDTF-diagnostics directory as $CODE_ROOT. It contains the following subdirectories:

- diagnostics/: directory containing source code and documentation of individual PODs.
- doc/: directory containing documentation (a local mirror of the documentation site).
- src/: source code of the framework itself.
- tests/: unit tests for the framework.

For advanced users interested in keeping more up-to-date on project development and contributing feedback, the main branch contains features that haven’t yet been incorporated into an official release, which are less stable or thoroughly tested.

For POD developers, the develop branch is the “beta test” version of the framework. POD developers should begin by locally cloning the repo and checking out this branch, as described in Git-based development workflow (page 42).

**Obtaining supporting data**

Supporting observational data and sample model data are available via anonymous FTP at ftp://ftp.cgd.ucar.edu/archive/mdtf. The observational data is required for the PODs’ operation, while the sample model data is provided for default test/demonstration purposes. The required files are:

- Digested observational data (159 Mb): MDTF_v2.1.a.obs_data.tar7.
- NCAR-CESM-CAM sample data (12.3 Gb): model.QBOi.EXP1.AMIP.001.tar8.
- NOAA-GFDL-CM4 sample data (4.8 Gb): model.GFDL.CM4.c96L32.am4g10r8.tar9.

Note that the above paths are symlinks to the most recent versions of the data and will be reported as zero bytes in an FTP client.

Download these three files and extract the contents in the following hierarchy under the mdtf directory:

```
mdtf
    MDTF-diagnostics
    inputdata
        model ( = $MODEL_DATA_ROOT)
            GFDL.CM4.c96L32.am4g10r8
                day
                    GFDL.CM4.c96L32.am4g10r8.precip.day.nc
                        (... other .nc files )
            QBOi.EXP1.AMIP.001
                1hr
                    QBOi.EXP1.AMIP.001.PRECT.1hr.nc
                        (... other .nc files )
                3hr
                    QBOi.EXP1.AMIP.001.PRECT.3hr.nc
```

(continues on next page)

---

7 ftp://ftp.cgd.ucar.edu/archive/mdtf/MDTF_v2.1.a.obs_data.tar
8 ftp://ftp.cgd.ucar.edu/archive/mdtf/model.QBOi.EXP1.AMIP.001.tar
9 ftp://ftp.cgd.ucar.edu/archive/mdtf/model.GFDL.CM4.c96L32.am4g10r8.tar
The default test case uses the QBOi.EXP1.AMIP.001 data. The GFDL.CM4.c96L32.am4g10r8 data is only for testing the MJO Propagation and Amplitude POD.

You can put the observational data and model output in different locations (e.g., for space reasons) by changing the values of OBS_DATA_ROOT and MODEL_DATA_ROOT as described in Section 1.2.4.

### 1.2.2 Install the conda package manager, if needed

The MDTF framework code is written in Python 3, but supports running PODs written in a variety of scripting languages and combinations of libraries. We use conda\(^\text{10}\), a free, open-source package manager, to install and manage these dependencies. Conda is one component of the Miniconda\(^\text{11}\) and Anaconda\(^\text{12}\) Python distributions, so having Miniconda or Anaconda is sufficient but not required.

For maximum portability and ease of installation, we recommend that all users manage dependencies through conda, even if they have a pre-existing installations of the required languages. A complete installation of all dependencies requires roughly 5 Gb, and the location of this installation can be set with the $CONDA_ENV_DIR setting described below. Note that conda does not create duplicates of dependencies that are already installed (instead using hard links by default).

If these space requirements are prohibitive, we provide an alternate method of operation which makes no use of conda and relies on the user to install external dependencies, at the expense of portability. This is documented in a separate section.

#### Conda installation

Users with an existing conda installation should skip this section and proceed to Section 1.2.3.

- To determine if conda is installed, run `%% conda --version` as the user who will be using the framework. The framework has been tested against versions of conda >= 4.7.5.

---

\(^{10}\) [https://docs.conda.io/en/latest/](https://docs.conda.io/en/latest/)

\(^{11}\) [https://docs.conda.io/en/latest/miniconda.html](https://docs.conda.io/en/latest/miniconda.html)

\(^{12}\) [https://www.anaconda.com/](https://www.anaconda.com/)
• If you do not have a pre-existing conda installation, we recommend installing Miniconda 3.x, available here\textsuperscript{13}. This version is not required: any version of Miniconda/Anaconda (2 or 3) released after June 2019 will work equally well.
  
  – Follow the installation instructions\textsuperscript{14} appropriate for your system. Toward the end of the installation process, enter “yes” at “Do you wish the installer to initialize Miniconda3 by running conda init?” (or similar) prompt. This will allow the installer to add the conda path to the user’s shell startup script (e.g., ~/.bashrc or ~/.cshrc).
  
  – Restart the terminal to reload the updated shell startup script.
  
  – Mac OS users may encounter a message directing them to install the Java JDK. This can be ignored.

\subsection{1.2.3 Install framework dependencies with conda}

As described above, all software dependencies for the framework and PODs are managed through conda environments.

Run \texttt{% conda info --base} as the user who will be using the framework to determine the location of your conda installation. This path will be referred to as $\text{CONDA\_ROOT}$ below. If you don’t have write access to this location (e.g., on a multi-user system), you’ll need to tell conda to install files in a non-default location $\text{CONDA\_ENV\_DIR}$, as described below.

Next, run

\begin{verbatim}
% cd $\text{CODE\_ROOT}
% ./src/conda/conda_env_setup.sh --all --conda_root $\text{CONDA\_ROOT} --env_dir $\text{CONDA\_ENV\_DIR}
\end{verbatim} 

to install all dependencies, which takes \textasciitilde{10} min (depending on machine and internet connection). The names of all framework-created environments begin with “_MDTF”, so as not to conflict with user-created environments in a preexisting conda installation.

• Substitute the actual paths for $\text{CODE\_ROOT}$, $\text{CONDA\_ROOT}$, and $\text{CONDA\_ENV\_DIR}$.

• The optional --env_dir flag directs conda to install framework dependencies in $\text{CONDA\_ENV\_DIR}$ (for space reasons, or if you don’t have write access). If this flag is omitted, the environments will be installed in $\text{CONDA\_ROOT}/\text{envs}/$ by default.

• The --all flag makes the script install all dependencies for all PODs. To selectively update individual conda environments after installation, use the --env flag instead. For instance, \texttt{% ./ src/conda/conda_env_setup.sh --env base --conda_root $\text{CONDA\_ROOT} --env_dir $\text{CONDA\_ENV\_DIR}} will update the environment named “_MDTF_base” defined in src/conda/env_base.yml, and so on.

\textbf{Note}: After installing the framework-specific conda environments, you shouldn’t manually alter them (e.g., never run \texttt{conda update} on them). To update the environments after updating the framework code, re-run

\textsuperscript{13} https:\/\slash\/docs.conda.io\slash\slashen\slash\slashlatest\slash\slashminiconda.html

\textsuperscript{14} https:\slash\slashdocs.conda.io\slash\slashprojects\slash\slashconda\slash\slashen\slash\slashlatest\slash\slashuser-guide\slash\slashinstall\slash\slashindex.html
the above commands. These environments can be uninstalled by simply deleting the “_MDTF” directories under $CONDA_ENV_DIR (or $CONDA_ROOT/envs/ by default).

1.2.4 Configure framework paths

The MDTF framework supports setting configuration options in a file as well as on the command line. An example of the configuration file format is provided at src/default_tests.jsonc. We recommend configuring the following settings by editing a copy of this file.

Relative paths in the configuration file will be interpreted relative to $CODE_ROOT. The following settings need to be configured before running the framework:

- If you’ve saved the supporting data in the directory structure described in Obtaining supporting data (page 4), the default values for OBS_DATA_ROOT and MODEL_DATA_ROOT given in src/default_tests.jsonc (.../inputdata/obs_data and .../inputdata/model, respectively) will be correct. If you put the data in a different location, these paths should be changed accordingly.

- OUTPUT_DIR should be set to the desired location for output files. The output of each run of the framework will be saved in a different subdirectory in this location.

- conda_root should be set to the value of $CONDA_ROOT used above in Install framework dependencies with conda (page 6).

- If you specified a non-default conda environment location with $CONDA_ENV_DIR, set conda_env_root to that value; otherwise, leave it blank.

1.2.5 Run the MDTF framework on sample data

Location of the MDTF executable

The MDTF framework is run via a wrapper script at $CODE_ROOT/mdtf.

This is created by the conda environment setup script used in Section 1.2.3. The wrapper script activates the framework’s conda environment before calling the framework’s code (and individual PODs). To verify that the framework and environments were installed successfully, run

```bash
% cd $CODE_ROOT
% ./mdtf --version
```

This should print the current version of the framework.

---

15 https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/default_tests.jsonc
Run the framework on sample data

If you’ve downloaded the NCAR-CESM-CAM sample data (described in Obtaining supporting data (page 4) above), you can now perform a trial run of the framework:

```
% cd $CODE_ROOT
% ./mdtf -f src/default_tests.jsonc
```

Run time may be 10-20 minutes, depending on your system.

• If you edited or renamed `src/default_tests.jsonc`, as recommended in the previous section, pass the path to that configuration file instead.

• The output files for this test case will be written to `$OUTPUT_DIR/MDTF_QBOi.EXP1.AMIP.001_1977_1981`. When the framework is finished, open `$OUTPUT_DIR/QBOi.EXP1.AMIP.001_1977_1981/index.html` in a web browser to view the output report.

• The framework defaults to running all available PODs, which is overridden by the `pod_list` option in the `src/default_tests.jsonc` configuration file. Individual PODs can be specified as a comma-delimited list of POD names.

• Currently the framework only analyzes data from one model run at a time. To run the MJO_prop_amp POD on the GFDL.CM4.c96L32.am4g10r8 sample data, delete or comment out the section for QBOi.EXP1.AMIP.001 in `caselist` section of the configuration file, and uncomment the section for GFDL.CM4.c96L32.am4g10r8.

1.3 Framework configuration for user model data

In this section we describe how to run the framework with your own model data, and more configuration options than the test case described in Quickstart installation instructions (page 3).

The complete set of configuration options is described in Command-line option reference (page 88), or by running `% ./mdtf --help`. All options can be specified as a command-line flag (e.g., `--OUTPUT_DIR`) or as a JSON configuration input file of the form provided in `src/default_tests.jsonc`\(^\text{16}\). We recommend using this file as a template, making copies and customizing it as needed.

Options given on the command line always take precedence over the input file. This is so you can store options that don’t frequently change in the file (e.g., the input/output data paths) and use command-line flags to set only those options you want to change from run to run (e.g., the analysis period start and end years). In all cases, the complete set of option values used in each run of the framework will be included in the log file as part of the output, for reproducibility and provenance.

Summary of steps for running the framework on user data

1. Save or link model data files following the framework’s filename convention.
2. Select the variable name convention used by the model.
3. Edit the configuration input file accordingly, then
4. Run the framework.

\(^\text{16}\) https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/develop/src/default_tests.jsonc
1.3.1 Adding your model data

Currently the framework is only able to run on model data in the form of NetCDF files on a locally mounted disk following a specific directory hierarchy and filename convention, with one variable per file. We hope to offer more flexibility in this area in the near future.

The directory/filename convention we use is

$\text{MODEL\_DATA\_ROOT}/\text{CASENAME}/\text{frequency}/\text{CASENAME.\variable.\frequency.nc}$,

where

- $\text{CASENAME}$ is any string used to identify the model run (experiment) that generated the data,
- $\text{frequency}$ is the frequency at which the data is sampled: one of 1hr, 3hr, 6hr, day, mon or year,
- $\text{variable}$ is the name of the variable in your model’s convention.

As an example, here’s how the sample model data is organized:

```
inputdata
  model ( = $\text{MODEL\_DATA\_ROOT}$)
    GFDL.CM4.c96L32.am4g1or8
      day
        GFDL.CM4.c96L32.am4g1or8.precip.day.nc
        (... other .nc files )
      QBOi.EXP1.AMIP.001
        1hr
          QBOi.EXP1.AMIP.001.PRECT.1hr.nc
          (... other .nc files )
        3hr
          QBOi.EXP1.AMIP.001.PRECT.3hr.nc
      day
        QBOi.EXP1.AMIP.001.FLUT.day.nc
        (... other .nc files )
    QBOi.EXP1.AMIP.001.PS.mon.nc
    (... other .nc files )
  obs_data ( = $\text{OBS\_DATA\_ROOT}$)
    (... supporting data for individual PODs)
```

If your model data is available on a locally mounted disk, we recommend creating symlinks\(^{17}\) that have the needed filenames and point to the data, rather than making copies of the files. For example,

```
% mkdir -p inputdata/model/my_new_experiment/day
% ln -s $\text{path\_to\_file/pr\_day\_GFDL\_ESM4\_historical\_r1i1p1f1\_gr1\_20100101\_20141231.nc}$
  → inputdata/model/my_experiment/day/my_new_experiment.pr.day.nc
```

will create a link to the file in the first argument that can be accessed normally:

```
inputdata
  model ( = $\text{MODEL\_DATA\_ROOT}$)
```

\(^{17}\) https://en.wikipedia.org/wiki/Symbolic_link
1.3.2 Select the model's variable name convention

The framework requires specifying a convention for variable names used in the model data. Currently recognized conventions are

- **CMIP**, for CF-compliant output produced as part of CMIP6;
- **CESM**, for the NCAR community earth system model\(^\text{18}\);
- **AM4**, for the NOAA-GFDL atmosphere model\(^\text{19}\);
- **SPEAR**, for the NOAA-GFDL seasonal model\(^\text{20}\).

We hope to offer support for the variable naming conventions of a wider range of models in the future. For the time being, please process output of models not on this list with CMOR\(^\text{21}\) to make them CF-compliant.

Alternatively, the framework will load any lookup tables of the form *src/fieldlist_${convention}.jsonc* and use them for variable name conversion. Users can add new files in this format to specify new conventions. For example, in *src/fieldlist_CESM.jsonc* the line "pr_var" : "PRECT" means that the CESM name for the precipitation rate is PRECT (case sensitive). In addition, "pr_conversion_factor" : 1000 specifies the conversion factor to CF standard units for this variable.

1.3.3 Running the code on your data

After adding your model data to the directory hierarchy as described above, you can run the framework on that data using the following options. These can either be set in the caselist section of the configuration input file (see *src/default_tests.jsonc*\(^\text{22}\) for an example/template), or individually as command-line flags (e.g., `--CASENAME my_new_experiment`). Required settings are:

- **CASENAME** should be the same string used to label your model run.
- **convention** describes the variable naming convention your model uses, determined in the previous section.
- **FIRSTYR** and **LASTYR** specify the analysis period.
- **model** and **experiment** are recorded if given, but not currently used.

---

\(^{18}\) [http://www.cesm.ucar.edu/](http://www.cesm.ucar.edu/)

\(^{19}\) [https://www.gfdl.noaa.gov/am4/](https://www.gfdl.noaa.gov/am4/)


\(^{21}\) [https://cmor.llnl.gov/](https://cmor.llnl.gov/)

\(^{22}\) [https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/develop/src/default_tests.jsonc](https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/develop/src/default_tests.jsonc)
When the framework is run, it determines whether the data each POD needs to run is present in the model data being provided. Specifically, the model must provide all variables needed by a POD at the required frequency. Consult the documentation (page 46) for a POD to determine the data it requires.

If the framework can’t find data requested by a POD, an error message will be logged in place of that POD’s output that should help you diagnose the problem. We hope to add the ability to transform data (e.g., to average daily data to monthly frequency) in order to simplify this process.

1.3.4 Other framework settings

The paths to input and output data (described in Configure framework paths (page 7)) only need to be modified if the corresponding data is moved, or if you’d like to send output to a new location. Note that the framework doesn’t retain default values for paths, so if you don’t specify a configuration file, all required paths will need to be given explicitly on the command line.

Other relevant flags controlling the framework’s output are:

- `save_ps`: set to `true` to retain the vector .eps figures generated by PODs, in addition to the bitmap images linked to from the webpage.
- `save_nc`: set to `true` to retain netcdf files of any data output at intermediate steps by PODs for further analysis.
- `make_variab_tar`: set to `true` to save the entire output directory as a .tar file, for archival or file transfer purposes.
- `overwrite`: set to `true` to overwrite previous framework output in `$OUTPUT_DIR`. By default, output with the same CASENAME and date range is assigned a unique name to ensure preexisting results are never overwritten.

These can be set as command-line flags each time the framework is run (e.g., `--save_ps`), or as `true/false` values in the input file ("save_ps": true). Note that `true` and `false` in JSON must be written all lower-case, with no quotes.

1.3.5 Modifying POD settings

Individual PODs may provide user-configurable options in the "pod_env_vars" section of their settings .jsonc file, which is located in each POD’s source code directory under /diagnostics. These only need to be changed in rare or specific cases. Consult the POD’s documentation (page 46) for details.
2.1 GFDL-specific information

This page contains information specific to the site installation at the Geophysical Fluid Dynamics Laboratory.\textsuperscript{23}

2.1.1 Site installation

The DET team maintains a site-wide installation of the framework and all supporting data at /home/mdteam/DET/analysis/mdtf/MDTF-diagnostics. This is kept up-to-date and is accessible from both workstations and PPAN. Please contact us if your use case can’t be accommodated by this installation.

2.1.2 FRE-centric modes of operation

In addition to the standard, interactive method of running MDTF diagnostics as described in the rest of the documentation, the site installation provides alternative ways to run the diagnostics within GFDL’s existing workflow.

1. Within FRE XMLs. This is done by calling the mdtf_gfdl.csh\textsuperscript{24} wrapper script from an \textless analysis\textgreater tag in the XML. Currently, FRE requires that each analysis script be associated with a single model \textless component\textgreater. This poses difficulties for diagnostics which use data generated by multiple components. We provide two ways to address this issue:

   A. If it’s known ahead of time that a given \textless component\textgreater will dominate the run time and finish last, one can call mdtf_gfdl.csh from an \textless analysis\textgreater tag in that component only. In this case, the framework will search all data present in the /pp/ output directory when it’s called. The \textless component\textgreater being used doesn’t need to generate data analyzed by the diagnostics; in this case it’s only used to schedule the diagnostics’ execution.

   B. If one doesn’t know which \textless component\textgreater will finish last, a more robust solution is to call mdtf_gfdl.csh --component_only from each \textless component\textgreater generating data to be analyzed. When the --component_only flag is set, every time the framework is called it will only run the diagnostics for which all the input data is available and which haven’t run already (which haven’t written their output to $OUTPUT_DIR$).

\textsuperscript{23} https://www.gfdl.noaa.gov/
\textsuperscript{24} https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/feature/gfdl-data/src/mdtf_gfdl.csh
2. As a batch job on PPAN, managed via slurm. This is handled via the `mdtf_gfdl_interactive.csh` wrapper script.

3. Called from an interactive shell on PPAN or workstations.

### 2.1.3 Data retrieval options

The framework is currently configured to search data from two types of directory hierarchies. The framework will determine what’s intended based on its input, but this choice can be overridden by passing the following options with the `--data_manager` flag:

- The /pp/ hierarchy used by FRE (`--data_manager Gfdl_PP`). In this case `CASE_ROOT_DIR` should be set to the root of the directory hierarchy (ie, ending in /pp).

- The CMIP6 DRS for published data on the Unified Data Archive (`--data_manager Gfdl_UDA_CMIP6`). In this case `CASE_ROOT_DIR` should not be set, but the `--model` and `--experiment` settings should be populated.

- The CMIP6 DRS for unpublished data on /data_cmip6. This option must be requested explicitly with `--data_manager Gfdl_data_cmip6`. In this case `CASE_ROOT_DIR` should not be set, but the `--model` and `--experiment` settings should be populated.

### 2.1.4 GFDL-specific options

In addition to the framework’s normal command-line options, the following site-specific options are recognized:

- `--GFDL-PPAN-TEMP, --GFDL_PPAN_TEMP <DIR>`: If running on the GFDL PPAN cluster, set the `$MDTF_GFDL_TMPDIR` environment variable to this location and create temp files here. Note: must be accessible via gcp. Defaults to `$TMPDIR`.

- `--GFDL-WS-TEMP, --GFDL_WS_TEMP <DIR>`: If running on a GFDL workstation, set the `$MDTF_GFDL_TMPDIR` environment variable to this location and create temp files here. The directory will be created if it doesn’t exist. Note: must be accessible via gcp. Defaults to `/net2/$USER/tmp`.

- `--frepp`: Normally this is set by the `mdtf_gfdl.csh` wrapper script, and not directly by the user. Set flag to run framework in “online” mode (1a. or 1b. above), processing data as part of the FRE pipeline.

- `--ignore-component, --ignore_component`: Normally this is set by the `mdtf_gfdl.csh` wrapper script, and not directly by the user. If set, this flag tells the framework to search the entire /pp/ directory for model data (1a. above); default is to restrict to model component passed by FRE. Ignored if `--frepp` is not set.

---

2.1.5 GFDL-specific defaults

The following paths are set to more useful default values:

- **--OBS-DATA-REMOTE, --OBS_DATA_REMOTE <DIR>:** Site-specific installation of observational data used by individual PODs at /home/Oar.Gfdl.Mdteam/DET/analysis/mdtf/obs_data. If running on PPAN, this data will be GCP’ed to the current node.

- **--OBS-DATA-ROOT, --OBS_DATA_ROOT <DIR>:** Local directory for observational data. Defaults to $MDTF_GFDL_TMPDIR/inputdata/obs_data, where the environment variable $MDTF_GFDL_TMPDIR is defined as described above.

- **--MODEL-DATA-ROOT, --MODEL_DATA_ROOT <DIR>:** Local directory for model data. Defaults to $MDTF_GFDL_TMPDIR/inputdata/model, where the environment variable $MDTF_GFDL_TMPDIR is defined as described above.

- **--WORKING-DIR, --WORKING_DIR <DIR>:** Working directory. Defaults to $MDTF_GFDL_TMPDIR/wkdir, where the environment variable $MDTF_GFDL_TMPDIR is defined as described above.

- **--OUTPUT-DIR, --OUTPUT_DIR, -o <DIR>:** Destination for output files. Defaults to $HOME/mdtf_out, which will be created if it doesn’t exist.
3.1 Introduction for POD developers

This walkthrough contains information for developers wanting to contribute a process-oriented diagnostic (POD) module to the MDTF framework. There are two tracks through the material: one for developers who have an existing analysis script they want to adapt for use in the framework, and one for developers who are writing a POD from scratch.

Section 3.4 provides instructions for setting up POD development, in particular managing language and library dependencies through conda. For developers already familiar with version 2.0 of the framework, Section 3.2 summarizes changes from v2.0 to facilitate migration to v3.0. New developers can skip this section, as the rest of this walkthrough is self-contained.

Section 3.3 provides a list of instructions for submitting a POD for inclusion in the framework. We require developers to submit PODs through GitHub. See Section 3.9 for how to manage code through the GitHub website.

Section 3.5 provides overall guidelines for POD development. Section 3.6 is a reference for the POD’s settings file format. In Section 3.7, we walk the developers through the workflow of the framework, focusing on aspects that are relevant for the operation of individual PODs, and using the Example Diagnostic POD as a concrete example to illustrate how a POD works under the framework. Section 3.8 provides coding best practices to address common issues encountered in submitted PODs.

3.1.1 Scope of a process-oriented diagnostic

The MDTF framework imposes requirements on the types of data your POD outputs and takes as input. In addition to the scientific scope of process-oriented diagnostics, the analysis that you intend to do needs to fit the following model:

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 28). 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 (e.g., the range of longitude or the positive convention for

---

28 https://github.com/NOAA-GFDL/MDTF-diagnostics
29 https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example
30 http://cfconventions.org/faq.html#vertical_coords_positive_attribute
vertical coordinates). Supporting data may be in any format and will not be modified by the framework (see next section).

The above data sources are your POD’s only input: your POD should not access the internet or other networked resources. 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.

To achieve portability, the MDTF cannot accept PODs written in closed-source languages (eg, MATLAB or IDL). We also cannot accept PODs written in compiled languages (eg, C or Fortran): installation would rapidly become impractical if users had to check compilation options for each POD.

The output of your POD should be a series of figures in vector format (.eps or .ps). Optionally, we encourage POD developers to also save relevant output data (e.g., 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.

### 3.1.2 POD code organization and supporting 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 (e.g., 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. This section will be updated when this data source is ready
for public use.

3.2 Migration from framework v2.0

In this section we describe the major changes made from v2.0 to v3.0 of the framework that are relevant for POD developers. The scope of the framework has expanded in version 3.0, which required changes in the way the PODs and framework interact. New developers can skip this section, as the rest of this documentation is self-contained.

3.2.1 Getting Started and Developer’s Walkthrough

A main source of documentation for v2.0 of the framework were the “Getting Started” and “Developer’s Walkthrough” documents. Updated versions of these documents are:

- Getting Started v3.0 (PDF)
- Developer’s Walkthrough v3.0 (PDF)

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 v2.0 documents, for ease of comparison.

3.2.2 Checklist for migrating a POD from v2.0

Here we list the broad set of tasks needed to update a POD written for v2.0 of the framework to v3.0.

- Update settings and varlist files: In v3.0 these have been combined into a single `settings.jsonc` file. See the settings file guide (page 28), reference (page 91), and example for descriptions of the new format.

- Update references to framework environment variables: See the table below for an overview, and the reference (page 101) for complete information on what environment variables the framework sets. Note that your POD 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 (preferably size within MB range), as well as any code used to perform that data reduction from raw sources.

- 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 v3.0 these tasks are done by the framework: all that your POD needs to do is generate figures of the appropriate formats and names in the specified folders, and the framework will convert and link them appropriately.

---

33 https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example/settings.jsonc
### 3.2.3 Conversion from v2.0 environment variables

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

### 3.3 POD development checklist

This section lists all the steps that need to be taken in order to submit a POD for inclusion in the MDTF framework.

#### 3.3.1 Code and documentation submission

The material in this section must be submitted through a pull request[^34] to the NOAA-GFDL GitHub repo[^35]. This is described in Git-based development workflow (page 42).

The example POD[^36] should be used as a reference for how each component of the submission should be structured.

[^35]: https://github.com/NOAA-GFDL/MDTF-diagnostics
[^36]: https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example
POD source code

All scripts should be placed in a subdirectory of `diagnostics/`. Among the scripts, there should be 1) a main driver script, 2) a template html, and 3) a `settings.jsonc` file. The POD directory and html template should be named after your POD’s short name.

- For instance, `diagnostics/convective_transition_diag/` contains its driver script `convective_transition_diag.py`, `convective_transition_diag.html`, and `settings.jsonc`, etc.

- The framework will call the driver script, which calls the other scripts in the same POD directory.

- If you need a new Conda environment, add a new `.yml` file to `src/conda/`, and install the environment using the `conda_env_setup.sh` script as described in the Getting Started (page 3).

POD settings file

The format of this file is described in POD settings file summary (page 28) and in more detail in Diagnostic settings file format (page 91).

POD html template for output

- The html template will be copied by the framework into the output directory to display the figures generated by the POD. You should be able to create a new html template by simply copying and modifying the example templates from existing PODs even without prior knowledge about html syntax.

Preprocessing scripts for digested data

The “digested” supporting data policy is described in Section 3.1.2.

For maintainability and provenance purposes, we request that you include the code used to generate your POD’s “digested” data from raw data sources (any source of data that’s permanently hosted). This code will not be called by the framework and will not be used by end users, so the restrictions and guidelines concerning the POD code don’t apply.

POD documentation

- The documentation for the framework is automatically generated using sphinx37, which works with files in reStructured text38 (reST, `.rst`) format. In order to include documentation for your POD (page 46), we require that it be in this format.

  – Use the example POD documentation39 as a template for the information required for your POD, by modifying its `.rst` source code40. This should include 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.

- The .rst files and all linked figures should be placed in a doc subdirectory under your POD directory (e.g., diagnostics/convective_transition_diag/doc/) and put the .rst file and figures inside.

- The most convenient way to write and debug reST documentation is with an online editor. We recommend https://livesphinx.herokuapp.com/ because it recognizes sphinx-specific commands as well.

- For reference, see the reStructured text introduction\textsuperscript{41}, quick reference\textsuperscript{42} and in-depth guide\textsuperscript{43}.

- Also see a reST syntax comparison\textsuperscript{44} to other text formats you may be familiar with.

  • For maintainability, all scripts should be self-documenting by including in-line comments. The main driver script (e.g., 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.

  • 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 html template (e.g., convective_transition_diag.html).

### Preprocessing script documentation

The “digested” supporting data policy is described in Section 3.1.2.

For maintainability purposes, include all information needed for a third party to reproduce your POD’s digested data from its raw sources in the doc directory. This information is not published on the documentation website and can be in any format. In particular, please document the raw data sources used (DOIs/versioned references preferred) and the dependencies/build instructions (e.g., conda environment) for your preprocessing script.

#### 3.3.2 Sample and supporting data submission

Data hosting for the MDTF framework is currently managed manually. The data is currently hosted via anonymous FTP on UCAR’s servers. Please contact the MDTF team leads via email to arrange a data transfer.

\textsuperscript{41} http://docutils.sourceforge.net/docs/user/rst/quickstart.html
\textsuperscript{42} http://docutils.sourceforge.net/docs/user/rst/quickref.html
\textsuperscript{43} http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html
\textsuperscript{44} http://hyperpolyglot.org/lightweight-markup
Digested observational or supporting data

The “digested” supporting data policy is described in Section 3.1.2.

Create a directory under `inputdata/obs_data/` named after the short name, and put all your digested observation data in (or more generally, any quantities that are independent of the model being analyzed).

- Digested data should be in the form of numerical data, not figures.
- The data files should be small (preferably a few MB) and just enough for producing figures for model comparison.
- If you really cannot reduce the data size or require GB of space, consult with the lead team.

Sample model data

For PODs dealing with atmospheric phenomena, we recommend that you use sample data from the following sources, if applicable:

- A timeslice run of NCAR CAM\textsuperscript{45}
- A timeslice run of GFDL AM\textsuperscript{46} (contact the leads for password).

3.4 Developer quickstart guide

This section contains instructions for beginning to

3.4.1 Developer installation instructions

To download and install the framework for development, follow the instructions for end users given in Quickstart installation instructions (page 3), with the following developer-specific modifications:

Obtaining the source code

POD developers should work from the develop branch\textsuperscript{47} of the framework code. This is the “beta test” version, used for testing changes before releasing them to end users.

Developers may download the code from GitHub as described in Download the framework code and supporting data (page 3), but we strongly recommend that you clone the repo in order to keep up with changes in the develop branch, and to simplify submitting pull requests with your POD’s code. Instructions for how to do this are given in Git-based development workflow (page 42).

\textsuperscript{45} https://www.earthsystemgrid.org/dataset/ucar.cgd.ccsm4.NOAA-MDTF.html
\textsuperscript{46} http://data1.gfdl.noaa.gov/MDTF/
\textsuperscript{47} https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/develop
Installing dependencies via conda

Regardless of development language, we strongly recommend that developers use conda to manage their language and library versions. Note that Conda is not Python-specific, but allows coexisting versioned environments of most scripting languages, including, R\(^{48}\), NCL\(^{49}\), Ruby\(^{50}\), PyFerret\(^{51}\), and more.

We recommend that new PODs be written in Python 3. We provide a developer version of the python3_base environment (described below) that includes Jupyter and other developer-specific tools. This is not installed by default, and must be requested by passing the `--all-dev` flag to the conda setup script:

```
% cd $CODE_ROOT
% ./src/conda/conda_env_setup.sh --all-dev --conda_root $CONDA_ROOT --env_dir $CONDA_ENV_DIR
```

### 3.4.2 POD development using existing Conda environments

To prevent the proliferation of dependencies, we suggest that new POD development use existing Conda environments whenever possible, e.g., python3_base\(^{52}\), NCL_base\(^{53}\), and R_base\(^{54}\) for Python, NCL, and R, respectively.

In case you need any exotic third-party libraries, e.g., a storm tracker, consult with the lead team and create your own Conda environment following instructions (page 23) below.

**Python**

The framework provides the _MDTF_python3_base\(^{55}\) Conda environment (recall the _MDTF prefix for framework-specific environment) as the generic Python environment, which you can install following the instructions (page 5). You can then activate this environment by running in a terminal:

```
% source activate $CONDA_ENV_DIR/_MDTF_python3_base
```

where `$CONDA_ENV_DIR` is the path you used to install the Conda environments. After you’ve finished working under this environment, run `% conda deactivate` or simply close the terminal.

---

\(^{48}\) https://anaconda.org/conda-forge/r-base
\(^{49}\) https://anaconda.org/conda-forge/ncl
\(^{50}\) https://anaconda.org/conda-forge/ruby
\(^{51}\) https://anaconda.org/conda-forge/pyferret
\(^{52}\) https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/develop/src/conda/env_python3_base.yml
\(^{54}\) https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/develop/src/conda/env_R_base.yml
Other languages

The framework also provides the _MDTF_NCL_base\textsuperscript{56} and _MDTF_R_base\textsuperscript{57} Conda environments as the generic NCL and R environments.

3.4.3 POD development using a new Conda environment

If your POD requires languages that aren’t available in an existing environment or third-party libraries unavailable through the common conda-forge\textsuperscript{58} and anaconda\textsuperscript{59} channels, we ask that you notify us (since this situation may be relevant to other developers) and submit a YAML (.yml) file\textsuperscript{60} that creates the environment needed for your POD.

• The new YAML file should be added to src/conda/, where you can find templates for existing environments from which you can create your own.

• The YAML filename should be env$_{your\_POD\_short\_name}.yml$.

• The first entry of the YAML file, name of the environment, should be _MDTF$_{your\_POD\_short\_name}$.

• We recommend listing conda-forge 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 channels) may create incompatibilities.

• We recommend constructing the list of packages manually, by simply searching your POD’s code for import statements referencing third-party libraries. Please do not exporting your development environment with % conda env export, which gives platform-specific version information 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.

• We recommend specifying versions as little as possible, out of consideration for end-users: 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 (e.g., Python 2 vs. 3) or a bug affecting your POD was fixed (e.g., postscript font rendering on Mac OS with older NCL). Conda installs the latest version of each package that’s consistent with all other dependencies.

\textsuperscript{56} https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/develop/src/conda/env_NCL_base.yml

\textsuperscript{57} https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/develop/src/conda/env_R_base.yml

\textsuperscript{58} https://conda-forge.org/feedstocks/

\textsuperscript{59} https://docs.anaconda.com/anaconda/packages/pkg-docs/

\textsuperscript{60} https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-file-manually
Framework interaction with conda environments

As described in Run the MDTF framework on sample data (page 7), when you run the mdtf executable, among other things, it reads pod_list in default_tests.jsonc and executes POD codes accordingly. For a POD included in the list (referred to as $POD_NAME):

1. The framework will first try to look for the YAML file src/conda/env_\$POD_NAME.yml. If it exists, the framework will assume that the corresponding conda environment _MDTF_\$POD_NAME has been installed under $CONDA_ENV_DIR, and will switch to this environment and run the POD.

2. If not, the framework will then look into the POD’s settings.jsonc file in $CODE_ROOT/diagnostics/$POD_NAME/. The runtime_requirements section in settings.jsonc specifies the programming language(s) adopted by the POD:
   a). If purely Python 3, the framework will look for src/conda/env_python3_base.yml and check its content to determine whether the POD’s requirements are met, and then switch to _MDTF_python3_base and run the POD.
   b). Similarly, if NCL or R is used, then NCL_base or R_base.

Note that for the 6 existing PODs depending on NCL (EOF_500hPa, MJO_prop_amp, MJO_suite, MJO_teleconnection, precip_diurnal_cycle, and Wheeler_Kiladis), Python is also used but merely as a wrapper. Thus the framework will switch to _MDTF_NCL_base when seeing both NCL and Python in settings.jsonc.

The framework verifies PODs’ requirements via looking for the YAML files and their contents. Thus if you choose to selectively install conda environments using the --env flag (Install framework dependencies with conda (page 6)), remember to install all the environments needed for the PODs you’re interested in, and that _MDTF_base is mandatory for the framework’s operation.

- For instance, the minimal installation for running the EOF_500hPa and convective_transition_diag PODs requires _MDTF_base (mandatory), _MDTF_NCL_base (because of b), and _MDTF_convective_transition_diag (because of 1). These can be installed by passing base, NCL_base, and convective_transition_diag to the --env flag one at a time (Install framework dependencies with conda (page 6)).

Testing with a new Conda environment

If you’ve updated an existing environment or created a new environment (with corresponding changes to the YAML file), verify that your POD works.

Recall how the framework finds a proper Conda environment for a POD. First, it searches for an environment matching the POD’s short name. If this fails, it then looks into the POD’s settings.jsonc and prepares a generic environment depending on the language(s). Therefore, no additional steps are needed to specify the environment if your new YAML file follows the naming conventions above (in case of a new environment) or your settings.jsonc correctly lists the language(s) (in case of updating an existing environment).

- For an updated environment, first, uninstall it by deleting the corresponding directory under $CONDA_ENV_DIR.

- Re-install the environment using the conda_env_setup.sh script as described in the installation instructions (page 6), or create the new environment for you POD:
• Have the framework run your POD on suitable test data.

  1. Add your POD’s short name to the pod_list section of the configuration input file (template: src/default_tests.jsonc).
  2. Prepare the test data as described in Framework configuration for user model data (page 8).

3.5 POD development guidelines

3.5.1 Admissible languages

The framework itself is written in Python, and can call PODs written in any scripting language. However, Python support by the lead team will be “first among equals” in terms of priority for allocating developer resources, etc.

  • To achieve portability, the MDTF cannot accept PODs written in closed-source languages (e.g., MATLAB and IDL; try Octave and GDL if possible). We also cannot accept PODs written in compiled languages (e.g., C or Fortran): installation would rapidly become impractical if users had to check compilation options for each POD.
  
  • Python is strongly encouraged for new PODs; PODs funded through the CPO grant are requested to be developed in Python. Python version $>= 3.6$ is required. Official support for Python 2 was discontinued as of January 2020.
  
  • If your POD was previously developed in NCL or R (and development is not funded through a CPO grant), you do not need to re-write existing scripts in Python 3 if doing so is likely to introduce new bugs into stable code, especially if you’re unfamiliar with Python.
  
  • If scripts were written in closed-source languages, translation to Python 3.6 or above is required.

3.5.2 Preparation for POD implementation

We assume that, at this point, you have a set of scripts, written in languages consistent with the framework’s open source policy, that a) read in model data, b) perform analysis, and c) output figures. Here are 3 steps to prepare your scripts for POD implementation.

We recommend running the framework on the sample model data again with both save_ps and save_nc in the configuration input src/default_tests.jsonc set to true. This will preserve directories and files created by individual PODs in the output directory, which could come in handy when you go through the instructions below, and help understand how a POD is expected to write output.

  • Give your POD an official name (e.g., Convective Transition; referred to as long_name) and a short name (e.g., convective_transition_diag). The latter will be used consistently to name the directories

---

% cd $CODE_ROOT
% ./src/conda/conda_env_setup.sh --env $your_POD_short_name --conda_root
↪$CONDA_ROOT --env_dir $CONDA_ENV_DIR

---

61 https://www.gnu.org/software/octave/

62 https://github.com/gnudatalanguage/gdl
and files associated with your POD, so it should (1) loosely resemble the long_name, (2) avoid space bar and special characters (@#$%^&*), and (3) not repeat existing PODs’ name (i.e., the directory names under diagnostics/). Try to make your POD’s name specific enough that it will be distinct from PODs contributed now or in the future by other groups working on similar phenomena.

- If you have multiple scripts, organize them so that there is a main driver script calling the other scripts, i.e., a user only needs to execute the driver script to perform all read-in data, analysis, and plotting tasks. This driver script should be named after the POD’s short name (e.g., convective_transition_diag.py).

- You should have no problem getting scripts working as long as you have (1) the location and filenames of model data, (2) the model variable naming convention, and (3) where to output files/figures. The framework will provide these as environment variables that you can access (e.g., using os.environ in Python, or getenv in NCL). DO NOT hard code these paths/filenames/variable naming convention, etc., into your scripts. See the complete list of environment variables supplied by the framework.

- Your scripts should not access the internet or other networked resources.

### 3.5.3 An example of using framework-provided environment variables

The framework provides a collection of environment variables, mostly in the format of strings but also some numbers, so that you can and MUST use in your code to make your POD portable and reusable.

For instance, using 3 of the environment variables provided by the framework, CASENAME, DATADIR, and pr_var, the full path to the hourly precipitation file can be expressed as

```python
MODEL_OUTPUT_DIR = os.environ['DATADIR'] + '/1hr/
pr_filename = os.environ['CASENAME'] + '.' + os.environ['pr_var'] + '.1hr.nc'
pr_filepath = MODEL_OUTPUT_DIR + pr_filename
```

You can then use `pr_filepath` in your code to load the precipitation data.

Note that in Linux shell or NCL, the values of environment variables are accessed via a `$` sign, e.g., `os.environ['CASENAME']` in Python is equivalent to `$CASENAME` in Linux shell/NCL.

### 3.5.4 Relevant environment variables

The environment variables most relevant for a POD’s operation are:

- **POD_HOME**: Path to directory containing POD’s scripts, e.g., diagnostics/convective_transition_diag/.

- **OBS_DATA**: Path to directory containing POD’s supporting/digested observation data, e.g., inputdata/obs_data/convective_transition_diag/.

- **DATADIR**: Path to directory containing model data files for one case/experiment, e.g., inputdata/model/QBOi.EXP1.AMIP.001/.

- **WK_DIR**: Path to directory for POD to output files. Note that this is the only directory a POD is allowed to write its output. E.g., wkdir/MDTF_QBOi.EXP1.AMIP.001_1977_1981/convective_transition_diag/.

26
1. Output figures to $WK_DIR/obs/ and $WK_DIR/model/ respectively.

2. $WK_DIR/obs/PS/ and $WK_DIR/model/PS/: If a POD chooses to save vector-format figures, save them as EPS under these two directories. Files in these locations will be converted by the framework to PNG for HTML output. Caution: avoid using PS because of potential bugs in recent matplotlib and converting to PNG.

3. $WK_DIR/obs/netCDF/ and $WK_DIR/model/netCDF/: If a POD chooses to save digested data for later analysis/plotting, save them in these two directories in NetCDF.

Note that (1) values of POD_HOME, OBS_DATA, and WK_DIR change when the framework executes different PODs; (2) the WK_DIR directory and subdirectories therein are automatically created by the framework. Each POD should output files as described here so that the framework knows where to find what, and also for the ease of code maintenance.

More environment variables for specifying model variable naming convention can be found in the src/filelist_$convention.jsonc files. Also see the list of environment variables supplied by the framework.

### 3.5.5 Guidelines for testing your POD

Test before distribution. Find people (e.g., nearby postdocs/grads and members from other POD-developing groups) who are not involved in your POD’s implementation and are willing to help. Give the tar files and point your GitHub repo to them. Ask them to try running the framework with your POD following the Getting Started instructions. Ask for comments on whether they can understand the documentation.

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? Here are some simple tests you should try:

- Move the inputdata directory around. Your POD should still work by simply updating the values of OBS_DATA_ROOT and MODEL_DATA_ROOT in the configuration input file.

- Try to run your POD with a different set of model data.

- If you have problems getting another set of data, try changing the files’ CASENAME and variable naming convention. The POD should work by updating CASENAME and convention in the configuration input.

- Try your POD on a different machine. Check that your POD can work with reasonable machine configuration and computation power, e.g., can run on a machine with 32 GB memory, and can finish computation in 10 min. Will memory and run time become a problem if one tries your POD on model output of high spatial resolution and temporal frequency (e.g., avoid memory problem by reading in data in segments)? Does it depend on a particular version of a certain library? Consult the lead team if there’s any unsolvable problems.
3.5.6 Other tips on implementation

1. Structure of the code package: Implementing the constituent PODs in accordance with the structure described in earlier sections 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 Getting Started (page 8) regarding the model data filename structure, An example of using framework-provided environment variables (page 26) and POD development checklist (page 18) 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 digested data after analysis: 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 Step 5: Output and cleanup (page 36) 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 POD development checklist (page 18).

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.

See Run the MDTF framework on sample data (page 7) and :doc:` framework operation walkthrough <dev_walkthrough>` for details on how the package is called. See the command line reference (page 88) for documentation on command line options (or run mdtf --help).

Avoid making assumptions about the machine on which the framework 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.

3.6 POD settings file summary

This page gives a quick introduction to how to write the settings file for your POD. See the full documentation (page 91) on this file format for a complete list of all the options you can specify.
3.6.1 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 101) containing paths to the data files and other information specific to the run.

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

- The CF conventions\(^\text{63}\) for standardized variable names and units.

- The netCDF4 (classic) data model, in particular the notions of variables\(^\text{64}\) and dimensions\(^\text{65}\) as they’re used in a netCDF file.

3.6.2 Example

```json
// 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": {
        
    }
}
```

\(^\text{63}\) [http://cfconventions.org/](http://cfconventions.org/)

\(^\text{64}\) [https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcVars.html](https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcVars.html)

\(^\text{65}\) [https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html](https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html)
"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"]
}
}
}

### 3.6.3 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, ocnBgcchem, sealce) 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” or “ncl” etc. but also any utilities such as “ncks”, “cdo”, 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* but not *math*. If no third-party libraries are needed, the value should be an empty list.

---

66 [https://www.python.org/](https://www.python.org/)
67 [https://www.ncl.ucar.edu/](https://www.ncl.ucar.edu/)
69 [https://code.mpimet.mpg.de/projects/cdo](https://code.mpimet.mpg.de/projects/cdo)
70 [https://numpy.org/](https://numpy.org/)
71 [https://docs.python.org/3/library/math.html](https://docs.python.org/3/library/math.html)
3.6.4 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, e.g. “1hr”, “6hr”, “day”, “mon”, …

3.6.5 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](http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html) for that coordinate.
- **units**: The units the diagnostic expects that coordinate to be in (using the syntax of the [UDUnits library](https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax)).
  
  This is optional: if not given, the framework will assume you want CF convention [canonical units](http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html).

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

- **positive**: Either "up" or "down", according to the [CF conventions](http://cfconventions.org/faq.html#vertical_coords_positive_attribute). A pressure axis is always "down" (increasing values are closer to the center of the earth).

3.6.6 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](http://cfconventions.org/Data/cf-standard-names/72/build/cf-standard-name-table.html) 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 101) for details.
- **units**: The units the diagnostic expects the variable to be in (using the syntax of the [UDUnits library](https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax)).
  
  This is optional: if not given, the framework will assume you want CF convention [canonical units](http://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html).

- **dimensions**: List of names of dimensions specified in the “dimensions” section, to specify the coordinate dependence of each variable.

---


73 [https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax](https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax)


75 [http://cfconventions.org/faq.html#vertical_coords_positive_attribute](http://cfconventions.org/faq.html#vertical_coords_positive_attribute)


77 [https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax](https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax)

3.7 Walkthrough of framework operation

In this section, we describe the actions that are taken when the framework is run, focusing on aspects that are relevant for the operation of individual PODs. The Example Diagnostic POD\textsuperscript{79} (short name: example) is used as a concrete example here to illustrate how a POD is implemented and integrated into the framework.

We begin with a reminder that there are 2 essential files for the operation of the framework and POD:

- `src/default_tests.jsonc`: configuration input for the framework.
- `diagnostics/example/settings.jsonc`: settings file for the example POD.

### 3.7.1 Step 1: Framework invocation

The user runs the framework by executing the framework’s main driver script `$CODE_ROOT/mdtf`, rather than executing the PODs directly. This is where the user specifies the model run to be analyzed, and chooses which PODs to run via the `pod_list` section in `default_tests.jsonc`.

- Some of the configuration options can be input through command line, see the command line reference (page 88) or run `% $CODE_ROOT/mdtf --help`.

At this stage, the framework also creates the directory `$OUTPUT_DIR/` (default: `mdtf/wkdir`) and all subdirectories therein for hosting the output files by the framework and PODs from each run.

- If you’ve run the framework with both `save_ps` and `save_nc` in `default_tests.jsonc` set to true, check the output directory structure and files therein.

Note that when running, the framework will keep collecting the messages relevant to individual PODs, including (1) the status of required data and environment, and (2) texts printed out by PODs during execution, and will save them as log files under each POD’s output directory. These log files can be viewed via the

\textsuperscript{79} https://github.com/NOAA-GFDL/MDTF-diagnostics/tree/main/diagnostics/example
top-level results page `index.html` and, together with messages printed in the terminal, are useful for debugging.

**Example diagnostic**

Run the framework using the NCAR-CAM5.timeslice case. After successful execution, open the `index.html` under the output directory in a web browser. The plots links to the webpage produced by the example POD with figures, and log to `example.log` including all example-related messages collected by the framework. The messages displayed in the terminal are not identical to those in the log files, but also provide a status update on the framework-POD operation.

### 3.7.2 Step 2: Data request

Each POD describes the model data it requires as input in the `varlist` section of its `settings.json`, with each entry in `varlist` corresponding to one model data file used by the POD. The framework goes through all the PODs to be run in `pod_list` and assembles a list of required model data from their `varlist`. It then queries the source of the model data (`$DATADIR/`) for the presence of each requested variable with the requested characteristics (e.g., frequency, units, etc.).

- The most important features of `settings.json` are described in the [settings documentation](page 28) and full detail on the [reference page](page 91).

- Variables are specified in `varlist` following [CF convention](http://cfconventions.org/) wherever possible. If your POD requires derived quantities that are not part of the standard model output (e.g., column weighted averages), incorporate necessary preprocessing for computing these from standard output variables into your code. PODs are allowed to request variables outside of the CF conventions (by requiring an exact match on the variable name), but this will severely limit the POD’s application.

- Some of the requested variables may be unavailable or without the requested characteristics (e.g., frequency). You can specify a backup plan for this situation by designating sets of variables as alternates if feasible: when the framework is unable to obtain a variable that has the `alternates` attribute in `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 also unavailable, the framework will skip executing your POD, and an error log will be presented in `index.html`.

Once the framework has determined which PODs are able to run given the model data, it prepares the necessary environment variables, including directory paths and the requested variable names (as defined in `src/filelist_$convention.json`) for PODs’ operation.

- At this step, the framework also checks the PODs’ observational/supporting data under `inputdata/obs_data/`. If the directory of any of the PODs in `pod_list` is missing, the framework would terminate with error messages showing on the terminal. Note that the framework only checks the presence of the directory, but not the files therein.

---

80 [http://cfconventions.org/](http://cfconventions.org/)
Example diagnostic

The example POD uses only one model variable in its varlist\(^1\): surface air temperature, recorded at monthly frequency.

- In the beginning of example.log, the framework reports finding the requested model data file under Found files.
- If the framework could not locate the file, the log would instead record Skipping execution with the reason being missing data.

3.7.3 Step 3: Runtime environment configuration

The framework reads the other parts of your POD’s settings.jsonc, e.g., pod_env_vars, and generates additional environment variables accordingly (on top of those being defined through default_tests.jsonc).

Furthermore, in the runtime_requirements section of settings.jsonc, we request that you provide a list of languages and third-party libraries your POD uses. The framework will check that all these requirements are met by one of the Conda environments under $CONDA_ENV_DIR/.

- The requirements should be satisfied by one of the existing generic Conda environments (updated by you if necessary), or a new environment you created specifically for your POD.
- If there isn’t a suitable environment, the POD will be skipped.

Note that the framework’s information about the Conda environments all comes from the YMAL (.yml) files under src/conda/ (and their contents) by assuming that the corresponding Conda environments have been installed using (thus are consistent with) the YAML files.

- The framework doesn’t directly check files under $CONDA_ENV_DIR/, where the Conda environments locate.
- Therefore, it’s imperative that you keep the Conda environments and the YAML files consistent at all time so the framework can properly function.

Example diagnostic

In its settings.jsonc, the example POD lists its requirements\(^2\): Python 3, and the matplotlib, xarray and netCDF4 third-party libraries for Python. In this case, the framework assigns the POD to run in the generic python3_base\(^3\) environment provided by the framework.

- In example.log, under Env vars: is a comprehensive list of environment variables prepared for the POD by the framework. A great part of them are defined as in src/fieldlist_CMIP.jsonc\(^4\) via setting convention in default_tests.jsonc to CMIP. Some of the environment variables are POD-specific as defined under pod_env_vars\(^5\) in the POD’s settings.jsonc, e.g., EXAMPLE_FAV_COLOR.

---


• In example.log, after `MDTF.py calling POD example`, the framework verifies the Conda-related paths, and makes sure that the runtime_requirements in settings.json are met by the python3_base environment via checking env_python3_base.yml.

3.7.4 Step 4: POD execution

At this point, your POD’s requirements have been met, and the environment variables are set. The framework then activates the right Conda environment, and begins execution of your POD’s code by calling the top-level driver script listed in its settings.json.

• See Relevant environment variables (page 26) for most relevant environment variables, and how your POD is expected to output results.

• All information passed from the framework to your POD is in the form of Unix/Linux shell environment variables; see reference for a complete list of environment variables (another good source is the log files for individual PODs).

• For debugging, we encourage that your POD print out messages of its progress as it runs. All text written to stdout or stderr (i.e., displayed in a terminal) will be captured by the framework and added to a log file available to the users via index.html.

• Properly structure your code/scripts and include error and exception handling mechanisms so that simple issues will not completely shut down the POD’s operation. Here are a few suggestions:

  A. Separate basic and advanced diagnostics. Certain computations (e.g., fitting) may need adjustment or are more likely to fail when model performance out of observed range. Organize your POD scripts so that the basic part can produce results even when the advanced part fails.

  B. If some of the observational data files are missing by accident, the POD should still be able to run analysis and produce figures for model data regardless.

  C. Say a POD reads in multiple variable files and computes statistics for individual variables. If some of the files are missing or corrupted, the POD should still produce results for the rest (note that the framework would skip this POD due to missing data, but PODs should have this robustness property for ease of workarounds or running outside the framework).

  • The framework contains additional exception handling so that if a POD experiences a fatal or unrecoverable error, the rest of the tasks and POD-calls by the framework can continue. The error messages, if any, will be included in the POD’s log file.

In case your POD requires derived quantities that are not part of the standard model output, and you’ve incorporated necessary preprocessing into your code (e.g., compute column average temperature from a vertically-resolved temperature field), one might be interested in saving these derived quantities as intermediate output for later use, and you may include this functionality in your code.

  • Here we are referring to derived quantities gridded in a similar way to model output, instead of highly-digested data that is just enough for making figures.

  • Save these as NetCDF files to the same directory containing the original model files. One file for one variable, following the filename convention spelled out in Getting Started (page 8).

• You must provide an option so that users can choose not to save the files (e.g., because of write permission, disk space, or files are accessed via soft links). Include this option through `pod_env_vars` in your POD’s `settings.json`. With “not to save” as default. You can remind users about this option by printing out messages in the terminal during runtime, or include a reminder in your POD documentation.

**Example diagnostic**

The framework activates the `_MDTF_python3_base` Conda environment, and calls the driver script `example-diag.py` listed in `settings.json`. Take a look at the script and the comments therein.

`example-diag.py` performs tasks roughly in the following order:

1) It reads the model surface air temperature data at `input_path`,
2) Computes the model time average,
3) Saves the model time averages to `$WK_DIR/model/netCDF/temp_means.nc` for later use,
4) Plots model figure `$WK_DIR/model/PS/example_model_plot.eps`,
5) Reads the digested data in time-averaged form at `$OBS_DATA/example_tas_means.nc`, and plots the figure to `$WK_DIR/obs/PS/example_obs_plot.eps`.

Note that these tasks correspond to the code blocks 1) through 5) in the script.

• When the script is called and running, it prints out messages which are saved in `example.log`. These are helpful to determine when and how the POD execution is interrupted if there’s a problem.

• The script is organized to deal with model data first, and then to process digested observations. Thus if something goes wrong with the digested data, the script is still able to produce the html page with model figures. This won’t happen if code block 5) is moved before 4), i.e., well-organized code is more robust and may be able to produce partial results even when it encounters problems.

In code block 7) of `example-diag.py`, we include an example of exception handling by trying to access a non-existent file (the final block is just to confirm that the error would not interrupt the script’s execution because of exception-handling).

• The last few lines of `example.log` demonstrate the script is able to finish execution despite an error having occurred. Exception handling makes code robust.

### 3.7.5 Step 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`

• `$WK_DIR/obs/PS/filename.eps` → `$WK_DIR/obs/filename.png`

The html template for each POD is then copied to `$WK_DIR` by the framework.

---

• In writing the template file all plots should be referenced as relative links to this location, e.g., “<A href=\(\text{model/filename.png}\)>”. See templates from existing PODs.

• Values of all environment variables referenced in the html template are substituted by the framework, allowing you to show the run’s CASENAME, date range, etc. Text you’d like to change at runtime must be changed through environment variables (the v3 framework doesn’t allow other ways to alter the text of your POD’s output webpage at runtime).

• If \texttt{save\_ps} and \texttt{save\_nc} are set to \texttt{false}, the .eps and .nc files will be deleted.

Finally, the framework links your POD’s html page to the top-level \texttt{index.html}, and copies all files to the specified output location (\texttt{OUTPUT\_DIR in default\_tests.jsonc}; same as \texttt{WK\_DIR} by default).

• If \texttt{make\_variab\_tar} in \texttt{default\_tests.jsonc} is set to \texttt{true}, the framework will create a tar file for the output directory, in case you’re working on a server, and have to move the file to a local machine before viewing it.

\section*{Example diagnostic}

Open the html template \texttt{diagnostics/example/example.html} and the output $\texttt{WK\_DIR/example.html}$ in a text editor, and compare. All the environment variables in the template have been substituted, e.g., \{\texttt{EXAMPLE\_FAV\_COLOR}\} becomes blue (defined in \texttt{pod\_env\_vars} in \texttt{settings.jsonc}).

\section*{3.8 POD coding best practices}

In this section we describe issues we’ve seen in POD code that have caused problems in the form of bugs, inefficiencies, or unintended consequences.

\subsection*{3.8.1 All languages}

• PS vs. EPS figures: Save vector plots as .eps (Encapsulated PostScript), not .ps (regular PostScript).

Why: Postscript (.ps) is perhaps the most common vector graphics format, and almost all plotting packages are able to output postscript files. Encapsulated Postscript\textsuperscript{88} (.eps) includes bounding box information that describes the physical extent of the plot’s contents. This is used by the framework to generate bitmap versions of the plots correctly: the framework calls ghostscript\textsuperscript{89} for the conversion, and if not provided with a bounding box ghostscript assumes the graphics use an entire sheet of (letter or A4) paper. This can cause plots to be cut off if they extend outside of this region.

Note that many plotting libraries will set the format of the output file automatically from the filename extension. The framework will process both *.ps and *.eps files.

\textsuperscript{88} \url{https://en.wikipedia.org/wiki/Encapsulated_PostScript}

\textsuperscript{89} \url{https://www.ghostscript.com/}
3.8.2 Python: General

- **Whitespace**: Indent python code with four spaces per indent level.

  Why: Python uses indentation to delineate nesting and scope within a program, and indentation that’s not done consistently is a syntax error. Using four spaces is not required, but is the generally accepted standard.

  Indentation can be configured in most text editors, or fixed with scripts such as `reindent.py` described here\(^90\). We recommend using a linter\(^91\) such as `pylint` to find common bugs and syntax errors.

  Beyond this, we don’t impose requirements on how your code is formatted, but voluntarily following standard best practices (such as described in PEPs\(^92\) or the Google style guide\(^93\)) will make it easier for you and others to understand your code, find bugs, etc.

- **Filesystem commands**: Use commands in the `os`\(^94\) and `shutil`\(^95\) modules to interact with the filesystem, instead of running unix commands using `os.system()`, `commands` (which is deprecated), or `subprocess`.

  Why: Hard-coding unix commands makes code less portable. Calling out to a subprocess introduces overhead and makes error handling and logging more difficult. The main reason, however, is that Python already provides these tools in a portable way. Please see the documentation for the `os`\(^96\) and `shutil`\(^97\) modules, summarized in this table:

<table>
<thead>
<tr>
<th>Task</th>
<th>Recommended function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct a path from dir1, dir2, …, filename</td>
<td><code>os.path.join(dir1, dir2, …, filename)</code></td>
</tr>
<tr>
<td>Split a path into directory and filename</td>
<td><code>os.path.split(path)</code> and related functions in <code>os.path</code>(^100)</td>
</tr>
<tr>
<td>List files in directory dir</td>
<td><code>os.scandir(dir)</code></td>
</tr>
<tr>
<td>Move or rename a file or directory from old_path to new_path</td>
<td><code>shutil.move(old_path, new_path)</code></td>
</tr>
<tr>
<td>Create a directory or sequence of directories dir</td>
<td><code>os.makedirs(dir)</code></td>
</tr>
<tr>
<td>Copy a file from path to new_path</td>
<td><code>shutil.copy2(path, new_path)</code></td>
</tr>
<tr>
<td>Copy a directory dir, and everything inside it, to new_dir</td>
<td><code>shutil.copytree(dir, new_dir)</code></td>
</tr>
<tr>
<td>Delete a single file at path</td>
<td><code>os.remove(path)</code></td>
</tr>
<tr>
<td>Delete a directory dir and everything inside it</td>
<td><code>shutil.rmtree(dir)</code></td>
</tr>
</tbody>
</table>

\(^90\) https://stackoverflow.com/q/1024435
\(^92\) https://www.python.org/dev/peps/pep-0008/
\(^93\) https://github.com/google/styleguide/blob/gh-pages/pyguide.md
\(^94\) https://docs.python.org/3.7/library/os.html
\(^95\) https://docs.python.org/3.7/library/shutil.html
\(^96\) https://docs.python.org/3.7/library/os.html
\(^97\) https://docs.python.org/3.7/library/shutil.html
In particular, using `os.path.join` is more verbose than joining strings but eliminates bugs arising from missing or redundant directory separators.

### 3.8.3 Python: Arrays

To obtain acceptable performance for numerical computation, people use Python interfaces to optimized, compiled code. NumPy is the standard module for manipulating numerical arrays in Python. xarray sits on top of NumPy and provides a higher-level interface to its functionality; any advice about NumPy applies to it as well.

NumPy and xarray both have extensive documentation and many tutorials, such as:

- NumPy’s own basic and intermediate tutorials; xarray’s overview and climate and weather examples;
- A demonstration of the features of xarray using earth science data;
- The 2020 SciPy conference has open-source, interactive tutorials you can work through on your own machine or fully online using Binder. In particular, there are tutorials for NumPy and xarray.
- Eliminate explicit for loops: Use NumPy/xarray functions instead of writing for loops in Python that loop over the indices of your data array. In particular, nested for loops on multidimensional data should never need to be used.

Why: For loops in Python are very slow compared to C or Fortran, because Python is an interpreted language. You can think of the NumPy functions as someone writing those for-loops for you in C, and giving you a way to call it as a Python function.

It’s beyond the scope of this document to cover all possible situations, since this is the main use case for NumPy. We refer to the tutorials above for instructions, and to the following blog posts that discuss this specific issue:

---

98 https://docs.python.org/3.7/library/os.path.html?highlight=os%20path#os.path.join
99 https://docs.python.org/3.7/library/os.path.html?highlight=os%20path#os.path.split
100 https://docs.python.org/3.7/library/os.path.html?highlight=os%20path#os.path
101 https://docs.python.org/3.7/library/os.html#os.scandir
102 https://docs.python.org/3.7/library/shutil.html#shutil.move
103 https://docs.python.org/3.7/library/os.html#os.makedirs
104 https://docs.python.org/3.7/library/shutil.html#shutil.copy2
105 https://docs.python.org/3.7/library/shutil.html#shutil.copytree
106 https://docs.python.org/3.7/library/os.html#os.remove
107 https://docs.python.org/3.7/library/shutil.html#shutil.rmtree
108 https://docs.python.org/3.7/library/os.path.html?highlight=os%20path#os.path.join
115 https://rabernat.github.io/research_computing/xarray.html
117 https://mybinder.org/
• Use xarray with netCDF data:

Why: This is xarray’s use case. You can think of NumPy as implementing multidimensional matrices in the fully general, mathematical sense, and xarray providing the specialization to the case where the matrix contains data on a lat-lon-time-(etc.) grid.

xarray lets you refer to your data with human-readable labels such as ‘latitude,’ rather than having to remember that that’s the second dimension of your array. This bookkeeping is essential when writing code for the MDTF framework, when your POD will be run on data from models you haven’t been able to test on.

In particular, xarray provides seamless support for time axes, with support for all CF convention calendars through the cftime library. You can, eg, subset a range of data between two dates without having to manually convert those dates to array indices.

See the xarray tutorials linked above for more examples of xarray’s features.

• Memory use and views vs. copies: Use scalar indexing and slices (index specifications of the form start_index:stop_index:stride) to get subsets of arrays whenever possible, and only use advanced indexing features (indexing arrays with other arrays) when necessary.

Why: When advanced indexing is used, NumPy will need to create a new copy of the array in memory, which can hurt performance if the array contains a large amount of data. By contrast, slicing or basic indexing is done in-place, without allocating a new array: the NumPy documentation calls this a “view.”

Note that array slices are native Python objects, so you can define a slice in a different place from the array you intend to use it on. Both NumPy and xarray arrays recognize slice objects.

This is easier to understand if you think about NumPy as a wrapper around C-like functions: array indexing in C is implemented with pointer arithmetic, since the array is implemented as a contiguous block of memory. An array slice is just a pointer to the same block of memory, but with different offsets. More complex indexing isn’t guaranteed to follow a regular pattern, so NumPy needs to copy the requested data in that case.

See the following references for more information:

– The NumPy documentation on indexing:

120 https://realpython.com/numpy-array-programming/
122 https://www.pythonlikeyoumeanit.com/Module3_IntroducingNumpy/VectorizedOperations.html
123 https://www.pythonlikeyoumeanit.com/
128 https://docs.python.org/3.7/library/functions.html?highlight=slice#slice
– “Numpy Views vs Copies: Avoiding Costly Mistakes”\textsuperscript{130}, by Jessica Yung;
– “How can I tell if NumPy creates a view or a copy?”\textsuperscript{131} on stackoverflow.

• MaskedArrays instead of NaNs or sentinel values: Use NumPy’s MaskedArrays\textsuperscript{132} for data that may contain missing or invalid values, instead of setting those entries to NaN or a sentinel value.

Why: One sometimes encounters code which sets array entries to fixed “sentinel values” (such as $1.0\times10^{20}$ or NaN\textsuperscript{133}) to indicate missing or invalid data. This is a dangerous and error-prone practice, since it’s frequently not possible to detect if the invalid entries are being used by mistake. For example, computing the variance of a timeseries with missing elements set to $1e+20$ will either result in a floating-point overflow, or return zero.

NumPy provides a better solution in the form of MaskedArrays\textsuperscript{134}, which behave identically to regular arrays but carry an extra boolean mask to indicate valid/invalid status. All the NumPy mathematical functions will automatically use this mask for error propagation. For example\textsuperscript{135}, trying to divide an array element by zero or taking the square root of a negative element will mask it off, indicating that the value is invalid: you don’t need to remember to do these sorts of checks explicitly.

### 3.8.4 Python: Plotting

• Use the ‘Agg’ backend when testing your POD: For reproducibility, set the shell environment variable MPLBACKEND to Agg when testing your POD outside of the framework.

Why: Matplotlib can use a variety of backends\textsuperscript{136}; interfaces to low-level graphics libraries. Some of these are platform-dependent, or require additional libraries that the MDTF framework doesn’t install. In order to achieve cross-platform portability and reproducibility, the framework specifies the ‘Agg’ non-interactive (ie, writing files only) backend for all PODs, by setting the MPLBACKEND environment variable.

When developing your POD, you’ll want an interactive backend – for example, this is automatically set up for you in a Jupyter notebook. When it comes to testing your POD outside of the framework, however, you should be aware of this backend difference.

### 3.8.5 NCL

• Deprecated calendar functions: Check the function reference\textsuperscript{137} to verify that the functions you use are not deprecated in the current version of NCL\textsuperscript{138}. This is especially necessary for date/calendar functions\textsuperscript{139}.

\textsuperscript{130}https://www.jessicayung.com/numpy-views-vs-copies-avoiding-costly-mistakes/
\textsuperscript{131}https://stackoverflow.com/questions/11524664/how-can-i-tell-if-numpy-creates-a-view-or-a-copy
\textsuperscript{133}https://en.wikipedia.org/wiki/NaN
\textsuperscript{134}https://numpy.org/doc/stable/reference/maskedarray.html
\textsuperscript{135}https://numpy.org/doc/stable/reference/maskedarray.generic.html#numerical-operations
\textsuperscript{136}https://matplotlib.org/tutorials/introductory/usage.html#backends
\textsuperscript{137}https://www.ncl.ucar.edu/Document/Functions/index.shtml
\textsuperscript{138}https://www.ncl.ucar.edu/
\textsuperscript{139}https://www.ncl.ucar.edu/Document/Functions/date.shtml
Why: The framework uses a current version of NCL\(^{140}\) (6.6.x), to avoid plotting bugs that were present in earlier versions. This is especially relevant for calendar functions: the \texttt{ut_*} set of functions have been deprecated in favor of counterparts beginning with \texttt{cd_} that take identical arguments (so code can be updated using find/replace). For example, use \texttt{cd_calendar}\(^{141}\) instead of the deprecated \texttt{ut_calendar}\(^{142}\).

This change is necessary because only the \texttt{cd_*} functions support all calendars defined in the CF conventions, which is needed to process data from some models (eg, weather or seasonal models are typically run with a Julian calendar.)

### 3.9 Git-based development workflow

We recommend developers to manage the MDTF package using the GitHub webpage interface:

- If you have no prior experience with GitHub\(^{143}\), create an account first.

- Create a fork of the project by clicking the Fork button in the upper-right corner of NOAA’s MDTF GitHub page\(^{144}\). This will create a copy (also known as repository, or simply repo) in your own GitHub account which you have full control over.

- Before you start working on the code, remember to switch to the develop branch (instead of main) as expected from a POD developer.

It should be easy to figure out how to add/edit files through your repo webpage interface.

- After updating the code in your repo, submit a Pull request so that the changes you have made can be incorporated into the official NOAA’s repo.

- Your changes will not affect the official NOAA’s repo until the pull request is accepted by the lead-team programmer.

Note that if any buttons are missing, try CRTL + + or CRTL + - to adjust the webpage font size so the missing buttons may magically appear.

Managing through the webpage interface as described above is quick and easy. Another approach, unfortunately with a steeper learning curve, is to create a local repo on your machine and manage the code using the git command in a terminal. In the interests of making things self-contained, the rest of this section gives brief step-by-step instructions on git for interested developers.

Before following the instructions below, make sure that a) you’ve created a fork of the project, and b) the git command is available on your machine (installation instructions\(^{145}\)).

---

\(^{140}\) https://www.ncl.ucar.edu/
\(^{141}\) https://www.ncl.ucar.edu/Document/Functions/Built-in/cd_calendar.shtml
\(^{142}\) https://www.ncl.ucar.edu/Document/Functions/Built-in/ut_calendar.shtml
\(^{143}\) https://github.com/
\(^{144}\) https://github.com/NOAA-GFDL/MDTF-diagnostics
\(^{145}\) https://git-scm.com/download/
3.9.1 Some online git resources

If you are new to git and unfamiliar with many of the terminologies, Dangit, Git?!\(^{146}\) provides solutions in plain English to many common mistakes people have made.

There are many comprehensive online git tutorials, such as:

- The official git tutorial\(^ {147}\).
- A more verbose introduction\(^ {148}\) to the ideas behind git and version control.
- A still more detailed walkthrough\(^ {149}\), assuming no prior knowledge.

3.9.2 Set up SSH with GitHub

- You have to generate an SSH key\(^ {150}\) and add it\(^ {151}\) 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.
- When generating the SSH key, you’ll be asked to pick a passphrase (i.e., password).
- The following instructions assume you’ve generated an SSH key. If you’re using manual authentication instead, replace the “git@github.com:” addresses in what follows with “https://github.com/”.

3.9.3 Clone a local repository onto your machine

- Clone your fork onto your computer: `git clone git@github.com:<your_github_account>/MDTF-diagnositics.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-diagnositics`.
- 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-diagnositics.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.

\(^{146}\) https://dangitgit.com/
\(^{147}\) https://git-scm.com/docs/gittutorial
\(^{148}\) https://www.atlassian.com/git/tutorials/what-is-version-control
\(^{149}\) http://swcarpentry.github.io/git-novice/
\(^{151}\) https://help.github.com/en/articles/adding-a-new-ssh-key-to-your-github-account
3.9.4 Start coding

- Switch to 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.
  - If you are unfamiliar with git and want to practice with the commands listed here, we recommend you to create an additional feature branch just for this. Remember: your changes will not affect NOAA’s repo until you’ve submitted a pull request through the GitHub webpage and accepted by the lead-team programmer.
  - If you encounter problems during practice, you can first try looking for plain English instructions to unmess the situation at Dangit, Git?!152.

- Write your code! A useful command is `git status` to remind you what branch you’re on and changes you’ve made (but have not committed yet). `git branch -a` lists all branches with * indicating the branch you’re on.

- If you’ve added new files, `git add --all` before commit the changes.

- Commit changes with `git commit -a`. This creates a snapshot of the code into the history in your local repo.
  - The snapshot will exist until you intentionally delete it (after confirming a warning message). You can always revert to a previous snapshot.
  - Don’t commit code that you know is buggy or non-functional!
  - You’ll be asked to enter a commit message. Good commit messages are key to making the project’s history useful.
  - Write in present tense describing 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.

- When finish updating your feature, 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` so that the changes to your local repo is incorporated to the your GitHub fork (displayed on the webpage).
  - Enter the SSH key passphrase when asked for password.

152 https://dangitgit.com/
• If you haven’t finished working on your feature, you can checkout and update the local feature branch again by repeating the above commands.

• When your feature is ready, submit a pull request by going to the GitHub page of your fork and clicking on the Pull request button. This is your proposal to the maintainers to incorporate your feature into NOAA’s repo.

• When the feature branch is 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.

• 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.
4.1 Summary of MDTF process-oriented diagnostics

The MDTF diagnostics package is a portable framework for running process-oriented diagnostics (PODs) on climate model data. Each POD script targets a specific physical process or emergent behavior, with the goals of determining how accurately the model represents that process, ensuring that models produce the right answers for the right reasons, and identifying gaps in the understanding of phenomena.

The scientific motivation and content behind the framework was described in E. D. Maloney et al. (2019): Process-Oriented Evaluation of Climate and Weather Forecasting Models. BAMS, 100 (9), 1665–1686, doi:10.1175/BAMS-D-18-0042.1.

4.1.1 Convective Transition Diagnostics

J. David Neelin (UCLA) neelin@atmos.ucla.edu

This POD computes statistics that relate precipitation to measures of tropospheric temperature and moisture, as an evaluation of the interaction of parameterized convective processes with the large-scale environment. Here the basic statistics include the conditional average and probability of precipitation, PDF of column water vapor (CWV) for all events and precipitating events, evaluated over tropical oceans. The critical values at which the conditionally averaged precipitation sharply increases as CWV exceeds the critical threshold are also computed (provided the model exhibits such an increase).

<table>
<thead>
<tr>
<th>Variables</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precipitation rate</td>
<td>6-hourly or higher</td>
</tr>
<tr>
<td>Column water vapor</td>
<td>6-hourly or higher</td>
</tr>
</tbody>
</table>

References:


\[^{153}\text{https://doi.org/10.1175/BAMS-D-18-0042.1}\]
4.1.2 Extratropical Variance (EOF 500hPa Height)

CESM/AMWG (NCAR) bundy@ucar.edu

This POD computes the climatological anomalies of 500 hPa geopotential height and calculates the EOFs over the North Atlantic and the North Pacific.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface pressure</td>
<td>Monthly</td>
</tr>
<tr>
<td>Geopotential height</td>
<td>Monthly</td>
</tr>
</tbody>
</table>

4.1.3 MJO Propagation and Amplitude

Xianan Jiang (UCLA) xianan@ucla.edu

This POD calculates the model skill scores of MJO eastward propagation versus winter mean low-level moisture pattern over Indo-Pacific, and compares the simulated amplitude of MJO over the Indian Ocean versus moisture convective adjustment time-scale.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precipitation rate</td>
<td>Daily or higher</td>
</tr>
<tr>
<td>Specific humidity</td>
<td>Daily or higher</td>
</tr>
</tbody>
</table>

References:


4.1.4 MJO Spectra and Phasing

CESM/AMWG (NCAR) bundy@ucar.edu

This PDO computes many of the diagnostics described by the WGNE MJO Task Force and developed by Dennis Shea for observational data. Using daily precipitation, outgoing longwave radiation, zonal wind at 850 and 200 hPa and meridional wind at 200 hPa, the module computes anomalies, bandpass-filters for the 20-100 day period, calculates the MJO Index as defined as the running variance over the bandpass filtered data, performs an EOF analysis, and calculates lag cross-correlations, wave-number frequency spectra and composite life cycles of MJO events.
### 4.1.5 MJO Teleconnections

Eric Maloney (CSU) eric.maloney@colosate.edu

The POD first compares MJO phase (1-8) composites of anomalous 250 hPa geopotential height and precipitation with observations (ERA-Interim/GPCP) and several CMIP5 models (BCC-CSM1.1, CNRM-CM5, GFDL-CM3, MIROC5, MRI-CGCM3, and NorESM1-M). Then, average teleconnection performance across all MJO phases defined using a pattern correlation of geopotential height anomalies is assessed relative to MJO simulation skill and biases in the North Pacific jet zonal winds to determine reasons for possible poor teleconnections. Performance of the candidate model is assessed relative to a cloud of observations and CMIP5 simulations.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precipitation rate</td>
<td>Daily</td>
</tr>
<tr>
<td>OLR</td>
<td>Daily</td>
</tr>
<tr>
<td>U850</td>
<td>Daily</td>
</tr>
<tr>
<td>U200</td>
<td>Daily</td>
</tr>
<tr>
<td>V200</td>
<td>Daily</td>
</tr>
</tbody>
</table>

References:


### 4.1.6 Diurnal Cycle of Precipitation

Rich Neale (NCAR) bundy@ucar.edu

The POD generates a simple representation of the phase (in local time) and amplitude (in mm/day) of total precipitation, comparing a lat-lon model output of total precipitation with observed precipitation derived from the Tropical Rainfall Measuring Mission.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precipitation rate</td>
<td>3-hourly or higher</td>
</tr>
</tbody>
</table>

References:

References:


### 4.1.7 Coupling between Soil Moisture and Evapotranspiration

Alexis M. Berg (Princeton) ab5@princeton.edu

This POD evaluates the relationship between soil moisture and evapotranspiration. It computes the correlation between surface (0–10 cm) soil moisture and evapotranspiration during summertime. It then associates the coupling strength with the simulated precipitation.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soil moisture</td>
<td>Monthly</td>
</tr>
<tr>
<td>Evapotranspiration</td>
<td>Monthly</td>
</tr>
<tr>
<td>Precipitation rate</td>
<td>Monthly</td>
</tr>
</tbody>
</table>

References:


### 4.1.8 Wavenumber-Frequency Spectra

CESM/AMWG (NCAR) bundy@ucar.edu

This POD performs wavenumber frequency spectra analysis (Wheeler and Kiladis) on OLR, Precipitation, 500hPa Omega, 200hPa wind and 850hPa wind.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precipitation rate</td>
<td>Daily</td>
</tr>
<tr>
<td>OLR</td>
<td>Daily</td>
</tr>
<tr>
<td>U850</td>
<td>Daily</td>
</tr>
<tr>
<td>U200</td>
<td>Daily</td>
</tr>
<tr>
<td>W250</td>
<td>Daily</td>
</tr>
</tbody>
</table>

References:

4.2 Example Diagnostic Documentation

Last update: 5/06/2020

This is an example document that you can use as a template for your diagnostics’ own documentation: it describes what information you should provide in each section. For example, if this were a real POD, you’d place a one-paragraph synopsis of your diagnostic here (like an abstract).

It also serves as an example of the RestructuredText (ReST, .rst) format used to generate this page: compare this output with the input source file\(^1\). The easiest way to get started is to copy the source file into the online editor at https://livesphinx.herokuapp.com/ and experiment.

4.2.1 Version & Contact info

Here you should describe who contributed to the diagnostic, and who should be contacted for further information:

• Version/revision information: version 1 (5/06/2020)
• PI (name, affiliation, email)
• Developer/point of contact (name, affiliation, email)
• Other contributors

Open source copyright agreement

The MDTF framework is distributed under the LGPLv3 license (see LICENSE.txt). Unless you’ve distributed your script elsewhere, you don’t need to change this.

4.2.2 Functionality

In this section you should summarize the stages of the calculations your diagnostic performs, and how they translate to the individual source code files provided in your submission. This will, e.g., let maintainers fixing a bug or people with questions about how your code works know where to look.

\(^1\) https://raw.githubusercontent.com/NOAA-GFDL/MDTF-diagnostics/develop/diagnostics/example/doc/example.rst
4.2.3 Required programming language and libraries

In this section you should summarize the programming languages and third-party libraries used by your diagnostic. You also provide this information in the `settings.json` file, but here you can give helpful comments to human maintainers (e.g., “We need at least version 1.5 of this library because we call this function.”)

4.2.4 Required model output variables

In this section you should describe each variable in the input data your diagnostic uses. You also need to provide this in the `settings.json` file, but here you should go into detail on the assumptions your diagnostic makes about the structure of the data.

4.2.5 References

Here you should cite the journal articles providing the scientific basis for your diagnostic. To keep the documentation format used in version 2.0 of the framework, we list references “manually” with the following command:

```latex
.. _ref-Maloney:

```

which produces


which can be cited in text as :ref:`a hyperlink <reference tag>`, which gives a hyperlink (page 51) to the location of the reference on the page. Because references are split between this section and the following “More about this diagnostic” section, unfortunately you’ll have to number references manually.

We don’t enforce any particular bibliographic style, but please provide a hyperlink to the article’s DOI for ease of online access. Hyperlinks are written as `link text <URL>` (text and url enclosed in backticks, followed by two underscores).

\(^\text{155}\) https://doi.org/10.1175/BAMS-D-18-0042.1
4.2.6 More about this diagnostic

In this section, you can go into more detail on the science behind your diagnostic, for example, by copying in relevant text articles you’ve written using th It’s especially helpful if you’re able to teach users how to use your diagnostic’s output, by showing how to interpret example plots.

Instead of doing that here, we provide more examples of RestructuredText syntax that you can customize as needed.

As mentioned above, we recommend the online editor at https://livesphinx.herokuapp.com/, which gives immediate feedback and has support for sphinx-specific commands.

Here’s an introduction\textsuperscript{156} to the RestructuredText format, a quick reference\textsuperscript{157}, and a syntax comparison\textsuperscript{158} to other text formats you may be familiar with.

Links to external sites

URLs written out in the text are linked automatically: https://ncar.ucar.edu/.

To use custom text for the link, use the syntax `link text <https://www.noaa.gov/>`\textsuperscript{159} (text and url enclosed in backticks, followed by two underscores). This produces link text\textsuperscript{159}.

More references and citations

Here’s another reference:

\begin{quote}
\end{quote}

Here’s an example of citing these references:

```
:ref:`Maloney et. al., 2019 <ref-Maloney>`,
:ref:`Charney, Fjørtoft and von Neumann, 1950 <ref-Charney>`
```

produces Maloney et. al., 2019 (page 51), Charney, Fjørtoft and von Neumann, 1950 (page 52).

---
\textsuperscript{156} http://docutils.sourceforge.net/docs/user/rst/quickstart.html
\textsuperscript{157} http://docutils.sourceforge.net/docs/user/rst/quickref.html
\textsuperscript{158} http://hyperpolyglot.org/lightweight-markup
\textsuperscript{159} https://www.noaa.gov/
\textsuperscript{160} https://doi.org/10.3402/tellusa.v2i4.8607
Figures

Images must be provided in either .png or .jpeg formats in order to be displayed properly in both the html and pdf output.

Here’s the syntax for including a figure in the document:

```
.. _my-figure-tag: [only needed for linking to figures]
.. figure:: [path to image file, relative to the source.rst file]
   :align: left
   :width: 75 % [these both need to be indented by three spaces]

   Paragraphs or other text following the figure that are indented by three spaces are treated as a caption/legend, eg:

   - red line: a Gaussian
   - blue line: another Gaussian
```

which produces

![Fig. 1: Paragraphs or other text following the figure that are indented by three spaces are treated as a caption/legend, eg:](image)

- blue line: a Gaussian
- orange line: another Gaussian

The tag lets you refer to figures in the text, e.g. :ref:`Figure 1 <my-figure-tag>` → Figure 1 (page 53).

Equations

Accented and Greek letters can be written directly using Unicode: é, Ω. (Make sure your text editor is saving the file in UTF-8 encoding).

Use the following syntax for superscripts and subscripts in text: 

```
```

→ W m⁻²; CO₂ → CO₂. Note that spaces are needed before and after the forward slashes.

Equations can be written using standard LaTeX\(^\text{161}\) (PDF link) syntax. Short equations in-line with the text can be written as:

```
\text{math}: f = 2 \Omega \sin \phi \rightarrow f = 2\Omega \sin \phi.
```

\(^\text{161}\) https://www.reed.edu/academic_support/pdfs/qskills/latexcheatsheet.pdf
Longer display equations can be written as follows. Note that a blank line is needed after the .. math:: heading and after each equation, with the exception of aligned equations.

\[
\frac{D \mathbf{u}_g}{Dt} + f_0 \hat{k} \times \mathbf{u}_a = 0; \\
\frac{Dh}{Dt} + f \nabla_z \cdot \mathbf{u}_a = 0,
\]

which produces:

\[
\frac{D u_g}{Dt} + f_0 \hat{k} \times u_a = 0; \\
\frac{D h}{Dt} + f \nabla_z \cdot u_a = 0,
\]

where \( u_g = \frac{g}{f_0} \hat{k} \times \nabla_z h \).

The editor at https://livesphinx.herokuapp.com/ can have issues formatting complicated equations, so you may want to check its output with a latex-specific editor, such as overleaf\footnote{https://www.overleaf.com/} or other equation editors\footnote{https://www.codecogs.com/latex/eqneditor.php}.

### 4.3 Convective Transition Diagnostic Package

Last update: 2/26/2019

The convective transition diagnostic package computes statistics that relate precipitation to measures of tropospheric temperature and moisture, as an evaluation of the interaction of parameterized convective processes with the large-scale environment. Here the basic statistics include the conditional average and probability of precipitation, PDF of column water vapor (CWV) for all events and precipitating events, evaluated over tropical oceans. The critical values at which the conditionally averaged precipitation sharply increases as CWV exceeds the critical threshold are also computed (provided the model exhibits such an increase).

#### 4.3.1 Version & Contact info

- Version 1 revision 3 13-Nov-2017 Yi-Hung Kuo (UCLA)
- PI: J. David Neelin (UCLA; neelin@atmos.ucla.edu)
- Current developer: Yi-Hung Kuo (yhkuo@atmos.ucla.edu)
- Contributors: K. A. Schiro (UCLA), B. Langenbrunner (UCLA), F. Ahmed (UCLA), C. Martinez (UCLA), and C.-C. (Jack) Chen (NCAR)
Open source copyright agreement

This package is distributed under the LGPLv3 license (see LICENSE.txt).

4.3.2 Functionality

The currently package consists of following functionalities:

1. Convective Transition Basic Statistics (convecTransBasic.py)
2. Convective Transition Critical Collapse (convecTransCriticalCollapse.py)
3. (*) Moisture Precipitation Joint Probability Density Function (cwvPrecipJPDF.py)
4. (*) Super Critical Precipitation Probability (supCriticPrecipProb.py)

More on the way… (* under development)

As a module of the MDTF code package, all scripts of this package can be found under the `convecutive_transition_diag` directory and pre-digested observational data under `inputdata/obs_data/convective_transition_diag`.

4.3.3 Required programming language and libraries

The is package is written in Python 2, and requires the following Python packages: os, glob, json, Dataset, numpy, scipy, matplotlib, networkx, warnings, numba, & netcdf4. These Python packages are already included in the standard Anaconda installation.

Known issue with matplotlib

The plotting scripts of this POD may not produce the desired figures with the latest version of matplotlib (because of the default size adjustment settings). The matplotlib version comes with the Anaconda 2 installer, version 5.0.1 has been tested. The readers can switch to this older version.

Depending on the platform and Linux distribution/version, a related error may occur with the error message “… ImportError: libcrypto.so.1.0.0: cannot open shared object file: No such file or directory”. One can find the missing object file `libcrypto.so.1.0.0` in the subdirectory `~/anaconda2/pkgs/openssl-1.0.21-h077ae2c_5/lib/`, where `~/anaconda2/` is where Anaconda 2 is installed. The precise names of the object file and openssl-folder may vary. Manually copying the object file to `~/anaconda2/lib/` should solve the error.

---

4.3.4 Required model output variables

The following three 3-D (lat-lon-time) high-frequency model fields are required:

1. Precipitation rate (units: \(\text{mm s}^{-1} = \text{kg m}^{-2} \text{s}^{-1}\); 6-hrly avg. or shorter)
2. Column water vapor (CWV, or precipitable water vapor; units: \(\text{mm} = \text{kg m}^{-2}\))
3. Column-integrated saturation humidity (units: \(\text{mm} = \text{kg m}^{-2}\)) or mass-weighted column average temperature (units: \(\text{K}\)), column: 1000-200 hPa by default. Since variables in (3) are not standard model output, this package will automatically calculate (3) if the following 4-D (lat-lon-pressure-time) model field is available:
4. Air temperature (units: \(\text{K}\))

4.3.5 References


See http://research.atmos.ucla.edu/csi//REF/pub.html for updates.

4.3.6 More about this diagnostic

The current version of the convective transition diagnostic package produces three sets of figures for both pre-digested observations and model output, including (1) basic statistics, (2) collapsed statistics, and (3) critical column water vapor. In the following, we will show an example set of the figures for an uncoupled simulation of the 1° version of the GFDL AM4 (configuration AM4-G9; Zhao et al., 2018a (page 59), 2018b (page 59); see also Kuo et al., in prep (page 81)) that are produced by the package.
1) Basic statistics

Fig. 2: Basic statistics computed using events over tropical western Pacific (20°S-20°N, west to 180°), including (a) conditionally averaged precipitation rate, (b) conditional probability of precipitation > 0.25 mm hr$^{-1}$, (c) PDF of CWV, and (d) PDF of CWV for precipitating events, all as a function of CWV. Here the large markers represent results simulated by the model, and small markers represent the corresponding observations at 1°. The colors indicate the column-integrated saturation humidity which is used as a bulk measure of tropospheric temperature (also shown as triangles). The PDFs in (c) together represent the normalized joint PDF of CWV and bulk temperature. Multiplying (b) and (c) results in PDFs in (d) (not normalized).

The observed (small markers) and simulated precipitation (large markers) in panel (a) sharply picks up as CWV exceeds a certain threshold, known as the critical CWV [see panel (e) below for how it is defined, and panel (i) for the values]. Here, the column integrated saturation humidity $\bar{q}_{sat}$ (units: mm) is used as a bulk measure of the tropospheric temperature. As the bulk tropospheric temperature increases, the pickup of precipitation occurs at higher CWV. The probability of precipitation in panel (b) exhibits a similar pickup behavior. The AM4 model examined here can reasonably simulate the observed pickup of precipitation, with slightly higher probability than observed.

In panel (c), the observed PDFs of CWV display characteristic shapes that depend on the bulk tropospheric temperature. At low temperature, the PDF peaks at a low CWV value, below which the PDF drops rapidly, and above which the PDF decreases slowly until reaching a cutoff. As temperature increases, another peak around critical develops with the low-CWV peak diminishing. The rapid drop of PDF for CWV above critical [see panel (g) below] is consistent with the pickup of precipitation, i.e., precipitation becomes an effective moisture sink in this regime. It has been noted that low-level convergence tends to be associated with high-CWV events, while low-level divergence is associated with low-CWV events. The AM4 model reasonably reproduces the observed CWV PDF with noticeably more above-critical events. However, given the uncertainty associated with the CWV retrievals used here (RSS TMI data products, version 7.1; Wentz et al. 2015 (page 59)), especially at high values, we cannot conclude that the model misbehaves in the high-CWV regime.
2) Collapsed statistics

Fig. 3: Same as the statistics in panels (a)-(d), respectively, but for each bulk tropospheric temperature, shift the CWV by the corresponding critical CWV $w_c$. Here, only the results from observations are presented. In panels (g)-(h), the PDF values are rescaled.

In practice, we define the critical CWV to be value at which the best-fit line to the conditionally averaged precipitation intersects with the CWV axis, as shown in panel (e) [see panels (i)-(j) below for the observed and simulated critical values]. When expressed as a function of CWV − $w_c$, the conditional average and probability of precipitation [panels (e)-(f)] collapse without exhibiting dependence on the bulk temperature (and ocean basin). The rescaled PDFs in panel (g) also collapse for CWV above critical. For the most relevant temperature bins in the tropics ($\bar{q}_{sat} \geq 70$ mm or the mass-weighted column average temperature $\geq 271$ K), the PDF of CWV for precipitating events share a common near-Gaussian core near the critical CWV.

Fig. 4: (i) Critical CWV $w_c$ and (j) the corresponding critical column relative humidity $w_c/\bar{q}_{sat}$. The colored markers represent the results simulated by the AM4 model and observed values in gray.

Compared to the observations, the slope of the best-fit line simulated by the AM4 model is slightly higher than observed [0.76 vs. 0.62 in panel (e); Kuo et al., in prep (page 81)], but within the uncertainty range of observations (Kuo et al. 2018 (page 81)). The simulated statistics are more sensitive to the tropospheric temperature and ocean basin and indicate that there are more above critical events for highest temperature bins in the model. The functional form of the PDFs for precipitating events deviates from Gaussian. The simulated
critical values are consistent with the observed values [panels (i)-(j)]. It has been noted that the dependence of critical values on tropospheric temperature (i.e., critical CWV increases with tropospheric temperature but the corresponding critical column RH $\frac{w_c}{q_{sat}}$ decreases) is a generic consequence of including entrainment in the buoyancy/conditional instability calculation (Sahany et al. 2012 (page 59)).

### 4.3.7 Additional references


### 4.4 ENSO Moist Static Energy Diagnostic Package

Last update 02/22/2019

ENSO moist static energy (MSE) diagnostic package consists of four levels. With a focus on identifying leading processes that determine ENSO-related precipitation anomalies, main module of the POD estimates vertically MSE budget and its variance analysis to account for relative contribution of each MSE term to column MSE. In that pursuit, POD is applied to monthly data (climate model or reanalysis products), and budget terms are estimated for “composite” El Niño or La Nina events. To estimate MSE budget, along with surface and radiation fluxes, 3-dimensional atmospheric variables are required. Hence, ERA-Interim is “considered” as “observations” here, and diagnostics obtained from ERA-Interim are used for model validation. In this general document, brief descriptions of the four levels of the POD are provided but detailed information (e.g., equations and input variables) is provided at each level. For the four levels of diagnostics, selected results are illustrated here.
4.4.1 Version & Contact info

- Version 1, 02/22/2019
- PI: Dr. H. Annamalai (IPRC/SOEST University of Hawaii; hanna@hawaii.edu)
- Current developer: Jan Hafner (IPRC/SOEST University of Hawaii; jhafner@hawaii.edu)

Open source copyright agreement

This package is distributed under the LGPLv3 license (see LICENSE.txt).

4.4.2 Functionality

The currently package consists of following functionalities:

1. Basic ENSO diagnostics performed by script COMPOSITE.py
2. MSE (Moist Static Energy) budget analysis performed by script MSE.py
3. MSE variance diagnostics performed by script MSE_VAR.py
4. MSE scatter plots performed by script SCATTER.py

4.4.3 Required programming language and libraries

This package is coded in Python 2.7.12 and requires the following packages: scipy, numpy, os, math.
The pre-processing and plotting is coded in NCAR Command Language Version 6.3.0.

4.4.4 Required model output variables

The following model fields are required as monthly data:

- 4-D variables (longitude, latitude, pressure level, time):
  1. HGT – geopotential height (m)
  2. U wind component [m/s]
  3. V wind component [m/s]
  4. Temperature [K]
  5. Specific Humidity [kg/kg]
  6. Vertical Velocity [Pa/s]

- 3-D variables (longitude, latitude, time):
  7. Precipitation [kg/m²/s]
  8. Surface Temperature [K]
9. Sensible Heat Flux [W/m²]
10. Latent Heat Flux [W/m²]
11. Net Shortwave Radiative Flux [W/m²]
12. Net Longwave Radiative Flux [W/m²]

Net radiative fluxes [variables 11 and 12] are estimated from the individual seven radiative flux components, namely: (i) Top Of Atmosphere (TOA) Shortwave down; (ii) TOA Shortwave up; (iii) Surface Shortwave down; (iv) Surface Shortwave up; (v) TOA Longwave up (OLR); (vi) Surface Longwave up and (vii) Surface Longwave down.

4.4.5 References


4.4.6 More about this diagnostic

Level 1 – Basic ENSO diagnostics

Composites, regression and correlation etc: Reference index (e.g., Nino3.4 SST)

- Monthly and seasonal averages
- 2 Year life cycle of ENSO: Year(0) and Year(1) Year (0) = developing phase and Year (1) = decaying phase

At this level, POD calculates simple seasonal averages, composites, regression and correlations. Based on a reference ENSO index (e.g., area-averaged SST anomalies over Nino3.4 region), seasonal composites of variables relevant to MSE budget are constructed for the entire 2-year life cycle of ENSO. Here, Y (0) refers to the developing, and Y (1) the decaying phase of ENSO. Fig. 1 shows composite net radiative flux divergence in the column \( F_{\text{rad}} \) for boreal winter (DJF) seasons during El Niño (Fig. 1a) and La Nina (Fig. 1b).

Fig. 5: Figure 1: Boreal winter (DJF) composites of net radiative flux divergence in the column \( F_{\text{rad}} \) constructed from CCSM4 historical simulations (1950-2005): (a) El Niño and (b) La Nina.
Level 2 – MSE (Moist Static Energy) budget analysis (for composite ENSO)

In the deep tropics, weak temperature gradient approximation (WTG) implies that precipitation is largely determined by column MSE [or vertically integrated specific humidity or precipitable water (PW)]. Fig. 2a shows that in regions of organized positive and negative precipitation anomalies along the equatorial Pacific, spatial structure and intensity of MSE (contour) and PW (shading) anomalies bear a “close association”. In this view, climate model biases in column MSE and precipitation are clearly linked and models’ fidelity in representing ENSO-related precipitation anomalies along the equatorial Pacific then requires that models accurately represent processes that determine column MSE anomalies.

In Level 2, for the composites constructed in Level 1, vertically integrated MSE and its budget are estimated (more details on the equations etc are in the README file in Level 2). All the terms are expressed in energy units (W/m²). As an illustration, anomalous MSE export (or vertical advection of MSE) calculated for composite El Niño winter from CCSM4 solutions is shown in Fig. 2b.

Fig. 6: Figure 2: (a) Vertically integrated anomalous MSE (contours, J/m², and scaled by 10E-7) and precipitable water (shaded, mm) and (b) vertical advection of MSE (W/m²). Results are for composite El Niño winters.

More details on Level 2 diagnostics can be found in the README file located in diagnostics/ENSO_MSE/doc/README_LEVEL_02.pdf.
**Level 3 – MSE variance diagnostics (for composite ENSO)**

Once all the individual MSE terms are estimated their relative contributions to column MSE is estimated here. This particular diagnostic is estimated for user-defined regions of interest, and outputs correspond to co- variances scaled by MSE variance (equation details in Level 3 README file). For example, one can estimate this diagnostic for equatorial central and eastern Pacific regions separately to assess the role of different processes in contributing to column MSE anomalies (or precipitation anomalies).

In the current version, the diagnostic is estimated for two “default regions” and one user optional region, and they are:

a) Equatorial Central Pacific 180 –200 E 10 S – 5 N  
b) Equatorial Eastern Pacific 220 –280 E 5 S – 5 N  
c) User prescribed area (for more details see README_LEVEL_03 document)

Here, in Fig. 3 results for both composite El Niño and La Nina winters, and from both equatorial central and eastern Pacific regions obtained from CCSM4 solutions are shown.

![Graphs showing relative contributions of various MSE terms to column MSE averaged for equatorial central (red) and eastern (blue) Pacific regions estimated from CCSM4 historical solutions for composite: (a) El Niño winter and (b) La Nina winter.](diagnostics/ENSO_MSE/doc/README_LEVEL_03.pdf)

More details on Level 3 diagnostics can be found in the README file located in diagnostics/ENSO_MSE/doc/README_LEVEL_03.pdf.
Level 4 – MSE scatter plots (Metrics)

Note that if diagnostics from multiple models are sought to assess systematic errors across all models then the results can be displayed as scatter plots between variables that are physically linked. In this level, one can also estimate “inter-model correlations” and “best fit” regression line, and show them in the plots.

At this level, results from Level 2 (CMIP-era models) are condensed into scatter plots. Specifically, estimates of each MSE budget term (e.g., $F_{rad}$) is plotted against precipitation, and the example shown here in Fig. 4 suggests that error in representing net radiative flux divergence ($F_{rad}$) is systematically tied to error in model simulated precipitation over both the equatorial central and eastern Pacific regions.

Fig. 8: Figure 4. Scatter plots between anomalous net radiative flux divergence ($F_{rad}$) and precipitation for composite El Niño winters estimated from historical simulations of CMIP5 models: (a) Central Pacific and (b) Eastern Pacific. In the panels, inter-model correlations and best regression fit lines are also provided.

doc/README_LEVEL_04.pdf.

4.5 EOF of Geopotential Height Diagnostic Module From NCAR

Last update: 03/11/2019
4.5.1 Contact info

- Current Developer: Dani Coleman (bundy@ucar.edu), NCAR
- Contributors: Dennis Shea, Andrew Gettleman, Jack Chen (NCAR)

This computes the climatological anomalies of 500 hPa geopotential height, then calculates the EOFs using NCL’s eofunc. The code is in NCL and requires model input:

1. monthly averaged surface pressure (ps),
2. monthly averaged geopotential height (zg).

- Generates a netcdf file of climatological anomalies of 500 hPa geopotential height (compute_anomalies.ncl)
- Calculates and plot EOFs of North Atlantic (eof_natlantic.ncl) and North Pacific regions using NCL function eofunc
- Uses pre-made figures of eofs of NCEP observational data for comparison.

4.5.2 Open source copyright agreement

This package is distributed under the LGPLv3 license (see LICENSE.txt).

4.5.3 Functionality

All scripts can be found at: mdtf/MDTF_$ver/var_code/EOF_500hPa

1. Make anomalies (compute_anomalies.ncl)
2. Calculated and plots EOFs in N. Atlantic (eof_natlantic.ncl) and N. Pacific (eof_npacific.ncl)

Preprocessed observational data from NCEP as gif images are located in mdtf/``inputdata/obs_data/EOF_500hPa

Place your input data at: inputdata/model/$model_name/day index.html can be found at: mdtf/MDTF_$ver/wkdir/MDTF_$model_name

4.5.4 Required Programming Language and libraries

All these scripts required NCAR Command Language Version 6.3.0 or higher

Required input data to the module:

1) Monthly averaged surface pressure (ps)
2) Monthly averaged geopotential height (zg)
4.5.5 References

None

4.5.6 More About the Diagnostic

4.6 MJO Propagation and Amplitude Diagnostic Package

Last update: 02/27/2019

This MJO propagation and amplitude diagnostic metrics is mainly motivated by recent multi-model studies that model skill in representing eastward propagation of the MJO is closely related to model winter mean low-level moisture pattern over the Indo-Pacific region, and the model MJO amplitude tends to be tightly associated with the moisture convective adjustment time scale. This package is designed to provide further independent verification of these above processes based on new GCM simulations.

4.6.1 Version & Contact info

- Version 2.0.
- PI and POC: Xianan Jiang (UCLA; xianan@ucla.edu)
- Developers: Alex Gonzalez (agon@iastate.edu), Xianan Jiang (xianan@ucla.edu)
- Contributors: E. Maloney (CSU), D. Waliser (JPL), Ming Zhao (GFDL)

Open source copyright agreement

This package is distributed under the LGPLv3 license (see LICENSE.txt).

4.6.2 Functionality

The currently package (mjo_diag.py) consists of following functionalities:

1. Model skill scores of MJO eastward propagation versus winter mean low-level moisture pattern over Indo-Pacific;
2. Model amplitude of MJO over the Indian Ocean versus moisture convective adjustment time-scale.

As a module of the MDTF code package, all scripts of this package can be found under mdtf/MDTF_$ver/var_code/mjo_diag", and pre-digested observational data under mdtf/inputdata/obs_data/mjo_diag.
4.6.3 **Required programming language and libraries**

This package is mainly written in NCAR Command Language (NCL) which is driven by Python 2. A newer version of the NCL above 6.4 is recommended for implementation of this diagnostic package.

4.6.4 **Required model output variables**

The following 3D (lat-lon-time) or 4D (lat-lon-pressure-time) model fields are required:

1. 3D precipitation rate (units: mm d\(^{-1}\); at least at daily interval or higher-frequency)
2. 4D specific humidity from 1000hPa to 100hPa (g g\(^{-1}\); at least at daily interval or higher-frequency);

Daily 3D column water vapor (CWV, or precipitable water vapor; units: mm = kg m\(^{-2}\)) will be calculated from (2) for calculation of moisture convective time-scale.

4.6.5 **References**

4.6.6 More details about this diagnostic

Fig. 9: a) Winter (Nov-Apr) mean 650-900hPa specific humidity based on ERA-Interim reanalysis; b) Scatter plot of model skill for eastward propagation of the MJO versus model skill of the mean 650-900hPa moisture pattern over the Maritime Continent (MC; red rectangle in a) based on multi-model simulations from the MJOTF/GASS project. Model MJO propagation skill is defined by the pattern correlation of Homvöller diagrams of model simulated rainfall anomalies associated with the MJO against its observed counterpart following Jiang et al. (2015) (page 59). Red (blue) dots denote good (poor) MJO models. c) Scatter plot of MJO amplitude and model convective moisture adjustment time scale in models (black dots) and observations (red dots). The MJO amplitude in each model is defined by the standard deviation of 20-100 day band-pass filtered rainfall over the Indian Ocean (75-85oE; 10oS-10oN) during winter. Convective time scale in a model is defined by the ratio of precipitable water (W) anomaly to precipitation (P) anomaly associated with the MJO and derived by a regression approach. Before conducting the regression, both W and P anomalies are subject to 20-100 day filtering and averaged over the Indian Ocean (75-85oE; 10oS-10oN) region. Adapted from Jiang et al. (2016) (page 59) and Gonzalez and Jiang (2017) (page 81).

This diagnostic metric for MJO propagation is motivated by findings that the horizontal advection of column moist static energy, or equivalently the lower-tropospheric moisture, plays a critical role in driving the eastward propagation of the winter MJO (e.g., Maloney 2009 (page 70); Maloney et al. 2010 (page 70); Kiranmayi and Maloney 2011 (page 70); Sobel et al. 2014 (page 70); Chikira 2014 (page 59); Kim et al. 2014 (page 69); Adames and Wallace 2015 (page 59); Jiang 2017 (page 69); Kim et al. 2017 (page 69)). Under this process, the spatial distribution of the winter mean lower-tropospheric moisture distribution over
the equatorial Indo-Pacific region (Fig. 1a) is crucial for moistening (drying) to the east (west) of the MJO convection through advection by MJO anomalous winds. The critical role of the mean lower-tropospheric moisture pattern for the MJO eastward propagation is confirmed by multi-model simulations from the MJO Task Force / GEWEX GASS MJO model comparison project (Jiang 2017 (page 69); Gonzalez and Jiang 2017 (page 81)). In particular, model skill in representing the 900-650hPa mean moisture pattern over the Maritime Continent region (red rectangle in Fig. 1a) exhibits a high correlation (about 0.8) with model MJO eastward propagation skill in about 25 GCM simulations (Fig. 1b).

On the other hand, the convective moisture adjustment time scale ($\tau$) in a model, defined by the ratio of intraseasonal perturbations of precipitable water and surface precipitation (e.g., Bretherton et al. 2004 (page 59); Peters and Neelin 2006 (page 70); Sobel and Maloney 2013 (page 70)), is selected as a diagnostic metric for model MJO amplitude, which is motivated by the high anti-correlation (-0.72) between $\tau$ and model MJO amplitude across multi-model simulations as indicated in Jiang et al. (2016, Fig. 1c). The convective moisture adjustment time scale depicts how rapidly precipitation must occur to remove excess column water vapor, or alternately the efficiency of surface precipitation generation per unit column water vapor anomaly, and is highly relevant to the convection onset diagnostics described above.

With this diagnostic packet, relationships between model skill in representing MJO eastward propagation and winter mean lower-tropospheric moisture as shown in Fig. 1b, and model MJO amplitude and moisture convective adjustment time scale as in Fig. 1c are examined based on specified model simulations.

4.6.7 References


4.7 MJO Suite Diagnostic Module From NCAR

Last update: 03/11/2019

This module computes many of the diagnostics described by the the US-CLIVAR Madden-Julian Oscillation (MJO) working group and developed by NCAR’s Dennis Shea for observational data. Using daily precipitation, outgoing longwave radiation, zonal wind at 850 and 200 hPa and meridional wind at 200hPa, the module computes anomalies, bandpass-filters for the 20-100 day period, calculates the MJO Index as defined as the running variance over the bandpass filtered data, performs an EOF analysis, and calculates lag cross-correlations, wave-number frequency spectra and composite life cycles of MJO events.
4.7.1 Contact info

- PI: Rich Neale, NCAR
- Current Developer: Dani Coleman (bundy@ucar.edu), NCAR
- Contributors: Dennis Shea, Andrew Gettleman, Jack Chen (NCAR)

Open source copyright agreement

This package is distributed under the LGPLv3 license (see LICENSE.txt).

4.7.2 Functionality

All scripts can be found at: mdtf/MDTF_$ver/var_code/MJO_suite

1. Read in daily model output: daily_netcdf.ncl
2. Computes daily anomalies: daily_anom.ncl
3. Calculates EOFs: mjo_EOF.ncl
4. Creates MJO lag plots: mjo_lag_lat_lon.ncl
5. Calculates MJO spectra: mjo_spectra.ncl
6. Calculates principle component from EOF analysis: mjo_EOF_cal.ncl
7. MJO life cycle composite: mjo_life_cycle_v2.ncl

Preprocessed observational data from NCEP, ERA and TRMM are located in mdtf/inputdata/obs_data/MJO_suite.
Place your input data at: mdtf/inputdata/model/$model_name/day
index.html can be found at: mdtf/MDTF_$ver/wkdir/MDTF_$model_name

4.7.3 Required Programing Language and libraries

All these scripts required NCAR Command Language Version 6.3.0 or higher
The following Python packages are required: os, glob
4.7.4 Required input data to the module

The following five 3-D (lat-lon-time) model fields are required with a daily time output:

1. Precipitation rate (units: mm/s = kg/m$^2$/s) or mm/day with appropriate conversion
2. Outgoing Longwave radiation (units: W/m$^2$)
3. Zonal wind at 850hPa (units: m/s)
4. Zonal wind at 200hPa (units: m/s)
5. Meridional wind at 200hPa (units: m/s)

4.7.5 References


4.7.6 More About the Diagnostic

An extensive explanation of the figures and techniques used to achieve them can be found on Dennis Shea’s NCL MJO CLIVAR page. Some examples are provided below:
Lag Correlation: Lag correlation diagram using on 20-100 day band-pass filtered daily data. The reference time series is the central Indian Ocean regional precipitation time series, which is correlated with precipitation and zonal wind anomalies in specified regions at different lags. Lag-longitude and lag-latitude plots of correlation values for different regions are shown. Color is for precipitation correlations while the lagged correlations for the zonal winds are the contours. These are analogous the Figures 5 and 6 in the reference article except they are for one season.
Wavenumber-Frequency Spectra: The wavenumber - frequency spectra for each season, with vertical reference lines indicating at for 30 and 80 days.

EOF analysis (univariate): Conventional (covariance) univariate EOF analysis for 20-100 day band-pass filtered data of outgoing longwave radiation over 30S to 30N.

Composite Life-cycles: The first and second principle components of the EOF analysis are used to derive the appropriate MJO phase category. The size of the reference anomaly wind vector is in the upper right. The phase (eg P3, means “Phase 3”) and the number of days used to create the composite are at the lower right.
4.8 MJO Teleconnection Diagnostic Package

Last Update: 2/1/2019

The teleconnection diagnostics first generate maps of MJO phase composites of 250 hPa geopotential height and precipitation for observations and several CMIP5 models, putting behavior of the candidate model within this cloud of models and observations. Then, average teleconnection performance across all MJO phases defined using a pattern correlation of geopotential height anomalies is assessed relative to 1) MJO simulation skill and 2) biases in the North Pacific jet zonal winds to determine reasons for possible poor teleconnections. Performance of the candidate model is assessed relative to a cloud of observations and CMIP5 simulations.
4.8.1 Contact info

- PI: Eric D. Maloney (eric.maloney@colostate.edu), Colorado State University
- Current Developer: Bohar Singh (bohar.singh@colostate.edu), Colorado State University
- Contributors: Stephanie Henderson (University of Wisconsin–Madison), Bohar Singh (CSU)

Open source copyright agreement

This package is distributed under the LGPLv3 license (see LICENSE.txt).

4.8.2 Functionality

1. Calculation of RMM indices for a new model will be saved in wkdir/casename/MJO_teleconnection/model/netCDF in txt format (mjo_diag_RMM_MDTF.ncl)
2. Z250 phase composite for all MJO phases (mjo_diag_geop_hgt_comp_MDTF.ncl)
3. Pattern correlation with observation (ERA-I Z250) (mjo_diag_Corr_MDTF.ncl)
4. Precipitation (30S-30N) phase composite for all MJO phases (mjo_diag_prec_comp_MDTF.ncl)
5. Extended winter wave number-frequency power spectrum of precipitation to get the ratio of eastward and westward propagation power (mjo_diag_EWR_MDTF.ncl)
6. Area averaged DJF mean U250 error (model-observation) over Pacific Ocean (15N80N,120E-120W) (mjo_diag_U250_MDTF.ncl)
7. ncl script to plot teleconnection skill v/s MJO skill (mjo_diag_fig1_MDTF.ncl)
8. ncl script to plot teleconnection skill v/s basic state skill (mjo_diag_fig1_MDTF.ncl)

All scripts can be found at: mdtf/MDTF_$ver/var_code/MJO_teleconnection

Preprocessed data for 10 CMIP5 models and observations data is located at mdtf/MDTF_$ver/obs_data/MJO_teleconnection

Keep your input data at: mdtf/MDTF_$ver/$model_name/day

Index.html can be found at: mdtf/MDTF_$ver/wkdir/MDTF_$model_name

4.8.3 Required Programming Language and libraries

All these scripts required NCAR Command Language Version 6.3.0 or higher in addition to ncl
The following Python packages are required: os, glob, json, Dataset, numpy, scipy, matplotlib & networkx, warnings, numba, netcdf4
Please use Anaconda: These Python packages are already included in the standard installation
4.8.4 Required input data to the module

The following five 3-D (lat-lon-time) model fields are required:

1. precipitation rate (units: mm/s = kg/m²/s) or mm/day with appropriate conversion, daily avg
2. Outgoing Longwave radiation (units: W/m²)
3. U850 wind (units: m/s)
4. U250 wind (units: m/s) (Note: U250 wind is used instead of u200 for RMM index calculation)
5. Z250 (units:m)

Please change the variable names and conversion factor corresponding to your data before running the MJO teleconnection diagnostics at: var_code/util/set_variables_CESM.py

Please condense each input variable into a single file.

4.8.5 References


4.8.6 More About the Diagnostic

Henderson et al (2017) (page 81) documented reasons for MJO midlatitude teleconnection errors in CMIP5 models. Since MJO teleconnections have significant impacts on atmospheric rivers, blocking, and other extreme events in the midlatitudes, teleconnection errors in models have important implications for the subseasonal prediction of midlatitude weather extremes (e.g. Henderson et al. 2016; Mundhenk et al. 2018; Baggett et al. 2017). In addition to extended analyses of stationary wavenumber biases and use of a linear baroclinic model to diagnose CMIP model biases, Henderson et al (2017) (page 81) developed diagnostics linking teleconnection biases to biases in the position and extent of the North Pacific jet.

The first diagnostic in this POD presents MJO composite 250 hPa geopotential height anomalies for ERA-I, the candidate model (upper right), and six other CMIP5 models assessed to have good MJO performance. All composites are generated as a function of MJO phase as defined according to Wheeler and Hendon (2004). An example of this diagnostic is presented in Figure 1 for phase 1 of the MJO.

77
Fig. 10: Figure 1. MJO phase 1 pentad composites of anomalous 250-hPa geopotential height, where a pentad denotes a 5-day mean, in this case the average of lag days 0 - 4 following an MJO phase. ERA-I is shown in the upper left, and the candidate model in the upper right. Positive geopotential height anomalies are in red solid contours, and negative anomalies are in blue dashed contours. Contours are plotted every 10 m, and the zero contour is omitted. Anomalies found to be 95% significantly different from zero are dotted. The color shading shows the anomalous tropical precipitation composite during MJO phase 1.

The diagnostic next assesses teleconnection performance versus measures of basic state fidelity and MJO skill. Figure 2 from Henderson et al (2017) (page 81) contains two panels, each having MJO teleconnection performance during December-February on the y-axis. In Figure 2a, the x axis represents an MJO skill
metric. While Figure 2a shows a relationship between MJO skill and teleconnection performance, even models with a good MJO can have poor teleconnection performance. For only the models assessed to have a sufficiently good MJO, Figure 2b assesses the relationship between teleconnection performance and biases in the North Pacific zonal flow. Plus signs are a measure of the total root mean squared (RMS) error of the 250 hPa zonal flow over the region 15°N – 60°N, 110°E – 120°W, and the filled circle provides a measure of the RMS error in the length of the North Pacific subtropical jet. Both measures are correlated with MJO teleconnection performance, although biases in the jet provides a somewhat better metric (r = -0.7 versus -0.6 for the total RMS). Subsequent analysis showed that models with a jet that extends too far east tend to have degraded teleconnection performance. Model physics appears to play a key role in the extent of the Pacific jet, as was demonstrated by Neelin et al. (2016) in diagnosing projected California precipitation changes between CMIP3 and CMIP5 models into the late 21st Century. The Pod developed here places the candidate model in question into the cloud of other models on Figure 2, with separate links on the POD site for left and right panels of Figure 2.

Fig. 11: Figure 2. From Henderson et al (2017) (page 81). December-February teleconnection performance averaged across all MJO phases (y-axis) versus (a) MJO skill (MJO E/W ratio) and (b) the RMS error of the 250-hPa December-February zonal wind. The observed E/W precipitation (GPCP) ratio is provided (dashed line, left panel). The MJO skill is derived as the ratio of eastward to westward power of equatorial precipitation in the 30-60 day, zonal wavenumbers 1-3 band (e.g. Ahn et al. 2017). Teleconnection performance was determined as pattern correlation of North Pacific and North America (15°S - 80°N, 130°E - 60°W) MJO composite 250 hPa geopotential height anomalies between CMIP models and ERA-i reanalysis averaged over all MJO phases. In panel (b), the crosses show the model 250hPa zonal wind RMS error over the full Pacific basin, while the closed circles indicate the longitudinal RMS error of the subtropical jet.
4.9 Phase and Amplitude of Precipitation Diurnal Cycle

Last update: 03/11/2019

The diurnal cycle package generates a simple representation of the phase (in local time) and amplitude (in mm/day) of total precipitation, comparing a lat-lon model output of total precipitation with observed precipitation derived from the Tropical Rainfall Measuring Mission (TRMM: https://pmm.nasa.gov/TRMM/mission-overview) satellite derived 3B42 product.

4.9.1 Version & Contact info

- Version 1:
  - Current Developer: Rich Neale

Open source copyright agreement

This package is provided under the LGPLv3 license (see LICENSE.txt).

4.9.2 Functionality

a) Computation of the diurnal cycle (local time phase) and amplitude of the first diurnal harmonic of total precipitation from TRMM (a 3-hourly product) with 3-hourly mean or instantaneous output from a model b) Plotting of a lat lon field of phase (with a cyclic color label) and hue (saturation of phase color) for a number of pre-specified regions and seasons. c) Computation of the variance explained by the diurnal harmonic on a gridded lat lon d) Computation of the mean precipitation over the analysis period on a gridded lat lon

4.9.3 Required programming language and libraries

Requires NCAR Command Language (NCL) and associated numerical and graphical capabilities. License agreement https://www.ncl.ucar.edu/Download/NCL_binary_license.shtml and licenses therewithin.

4.9.4 Required model output variables

Code requires the input of total precipitation either as a single variable or one that can be derived from a convective and large-scale combination.
4.9.5 References


4.9.6 More about this diagnostic

Fig. 12: Figure 1: Diurnal precipitation analysis for the tropics in June/July/August for output from a default CESM case, showing (a) the local timing maximum of the peak in the first harmonic of diurnal rainfall, (b) the variance explained by the first harmonic of the diurnal rainfall variability (%), and the mean precipitation over the analysis period (mm/day).

This diagnostic set provides an analysis of the variation in diurnal peak of rainfall over both land and ocean. The land analysis in particular gives insight into the mechanisms of rainfall production during the day. A recognized bias of many climate models is that they produce a peak in rainfall that is too early in the day.
(~noon) when the peak should be later in the evening. This is frequently interpreted as being due to short-comings in the representations of the atmospheric physics and land-atmosphere interactions. Additionally, there is more detail relating to the impact of orographic and coastal regions that can amplify, weaken and in a number of other ways, modify the precipitation diurnal cycle.

4.10 Soil moisture-Evapotranspiration Coupling Diagnostic Package

Last update: 6/28/2019

The Soil moisture-Evapotranspiration (SM-ET) Coupling Diagnostic Package evaluates the relationship between SM and ET in the summertime. It computes the correlation between surface (top-10cm) SM and ET, at the interannual timescale, using summertime-mean values. Positive correlation values indicate that, at the interannual time scale (from one summer to the next), soil moisture variability controls ET variability. This can generally be expected to occur when soil moisture availability is the limiting factor for ET. Conversely, negative values indicate that ET variations drive variations in soil moisture levels, which can be expected to occur in regions where soil moisture is plentiful and the limiting factor for ET becomes atmospheric evaporative demand (radiation, temperature); it also reflects the anticorrelation between precipitation, which drives soil moisture, and radiation, which drives ET. In addition to its sign, the correlation value quantifies how much of ET interannual variability is explained by soil moisture variations (if the correlation is positive; vice versa if it is negative)—in other words, the tightness of the SM–ET relationship. Considering seasonal means removes issues associated with the coseasonality of soil moisture and ET, while still reflecting the overall (i.e., seasonally integrated) dependence of ET on soil moisture throughout the whole season. See Berg and Sheffield (2018) for further details.

4.10.1 Contact info

- PIs of the project: Eric Wood, Princeton University (efwood@princeton.edu);
- previous PI Justin Sheffield, formerly at Princeton University, now at University of Southampton, UK (justin.sheffield@soton.ac.uk).
- Current developer: Alexis Berg (ab5@princeton.edu)

Open source copyright agreement

This package is distributed under the LGPLv3 license (see LICENSE.txt).
4.10.2 Functionality

The currently package consists of: - a Python script (SM_ET_coupling.py), which sets up the directories and calls... - ... an R script (SM_ET_coupling.R) which reads the data, performs the calculations and generates the plots.

As a module of the MDTF code package, all scripts of this package can be found under mdtf/MDTF_$ver/var_code/SM_ET_coupling and pre-digested observational data (in RData format) under mdtf/inputdata/obs_data/SM_ET_coupling Place your input data at: mdtf/inputdata/model/$model_name/mon/

4.10.3 Required programming language and libraries

Python and R are required to run the diagnostic.

The part of the package written in Python requires packages os and subprocess. These Python packages are already included in the standard Anaconda installation The R script requires packages ColorRamps, maps, fields, akima and ncdf4. R version 3.4 was used to develop this package, but it should work on older and more recent R versions.

4.10.4 Required model output variables

The following three 3-D (lat-lon-time), monthly model fields are required:

- surface soil moisture (“mrsos” in CMIP5 conventions)
- land evaporation (“evspsbl”) or latent heat flux (“hfls”)
- precipitation (“pr”)

The observational estimate from GLEAM (see below) is for 1980-2014; therefore, the model data should cover the same time period, as the background climate, and thus the SM-ET coupling, could be different if the model data covers another period (although we attempt to control for precipitation differences between model and observations – see below). Note that 2014 is the end year of the historical period of CMIP6 historical simulations. 1980 is the beginning of the GLEAM data. Note that, by default, the R script will read the whole monthly model file provided as input. We thus recommend that users truncate their model files to cover precisely the period 1980-2014.

4.10.5 More about the diagnostic

a) Choice of reference dataset

With respect to SM-ET coupling, an observational value of the metric is difficult to obtain, because of the challenges associated with measuring soil moisture and evapotranspiration extensively over continents, at the required spatial and temporal scales. Global observational products of ET and soil moisture do exist (in particular from remote sensing), but are plagued by numerous uncertainties or shortcomings limiting their use, here, to compute SM-ET coupling in a straightforward manner. Calculating SM-ET coupling from various datasets combining modeling and observations, such as reanalyses (e.g., MERRA2, JRA55, ERA-I)
or land surface models driven by observations (e.g., GLADS, GLDAS2), yields estimates of SM-ET coupling that exhibit significant spread (comparable in some regions to the spread across CMIP5 models), even though their representation of the driving surface climate (e.g., precipitation) is very comparable. This diversity is not necessarily surprising, given that SM-ET coupling largely remains, in these types of products, a product of the underlying land model used to create the dataset. In this context, we eventually decided to use the GLEAM (Global Land Evaporation Amsterdam Model) dataset (Martens et al. 2017; see https://www.gleam.eu/), as a reference, provided along the SM-ET coupling metric here in the diagnostic package. GLEAM is a global, gridded land surface dataset based on remote sensing covering 1980-2017 (here we only use 1980-2014, so that, in particular, the latest CMIP6 simulations, which extend to 2014, can be compared to this data). While the control of SM on ET in the GLEAM dataset ultimately remains a property of the modeling assumptions underlying this product, GLEAM is the only product, to our knowledge, assimilating available remote sensing observations of both soil moisture and vegetation, thus providing a dataset including both observationally-constrained and mutually-consistent SM and ET. That being said, caution should be exerted when comparing the model results to this estimate, which is only provided as a tentative reference, not an observational truth.

b) Correction for precipitation differences

In Berg and Sheffield (2018) (page 81), we found that across CMIP5 models, differences in summertime precipitation explained a significant part of model differences in SM-ET coupling. In other words, in a given location – for instance, a semi-arid location models with more precipitation have less positive SM-ET coupling – i.e., ET is less limited by soil moisture (see Figure 3 in Berg and Sheffield 2018). However, mean precipitation did not explain all of the differences across models, which we interpreted as reflecting model differences, for a given amount of precipitation, in the treatment of land surface processes related to vegetation and hydrology. In the diagnostic package here, summertime precipitation differences between the model and the observations (GLEAM over 1980-2014) are provided as a plot. Assuming that, to first approximation, precipitation differences are independent from the surface, we attempt to control for precipitation differences between model and observations in the package by using the regression across CMIP5 models between mean summertime precipitation and SM-ET coupling established in Berg and Sheffield (2018) (page 81; Figure 3). In other words, the coupling calculated for the model, when correcting for precipitation differences, is the coupling that would have existed in the model if precipitation were correct (i.e., equal to the observations in GLEAM). For instance, in regions where the model produces too much rainfall, the correction will tend to increase the estimate of SM-ET coupling (since, if precipitation was more realistic, it would be lower and soil moisture control on ET would thus be greater). This correction is tentative, as it assumes that the relationship across CMIP5 models between precipitation and SM-ET coupling is realistic, in the sense that it says something about the physics of the real world.

4.10.6 References


4.11 Wavenumber Frequency Spectra Diagnostic Module From NCAR

Last update: 03/11/2019

Produces wavenumber-frequency spectra for OLR, Precipitation, 500hPa Omega, 200hPa wind and 850hPa Wind.

4.11.1 Contact info

- Current Developer: Dani Coleman (bundy@ucar.edu), NCAR
- Contributors: Dennis Shea, Andrew Gettleman, Jack Chen, Rich Neale (NCAR)

Open source copyright agreement

This package is distributed under the LGPLv3 license (see LICENSE.txt).

4.11.2 Functionality

Python code calls NCL wkSpaceTime_driver.ncl code for each of the variables in turn. Preprocessed observational data in the form of gif figures from NCEP precipitation, OLR, Omega and winds, and TRMM precipitation are in the mdtf/inputdata/obs_data/Wheeler_Kiladis directory. Place your input data at: mdtf/inputdata/model/$model_name/day index.html can be found at: mdtf/MDTF_$ver/wkdir/MDTF_$model_name

4.11.3 Required Programing Language and libraries

All these scripts required NCAR Command Language Version 6.3.0 or higher

4.11.4 Required input data to the module

Daily U200, U850, OMEGA500, OLR, PRECT
4.11.5 References


4.11.6 More About the Diagnostic
5.1 Command-line option reference

5.1.1 Usage

```
mdtf [options] [INPUT_FILE] [CASE_ROOT_DIR]
mdtf info [TOPIC]
```

If the framework was installed to use Conda\(^\text{165}\) (recommended), the top-level `mdtf` driver script is created at install time. It sets the conda environment and calls `src/mdtf.py`\(^\text{166}\), the top-level script for non-conda installations.

The first form of the command runs diagnostics on data at `CASE_ROOT_DIR`, using configuration set on the command line or in `INPUT_FILE`.

- **INPUT_FILE**: Path to a user configuration file that sets options listed here. This can be a JSON file of the form given in `src/default_tests.jsonc`\(^\text{167}\) (which is intended to be copied and used as a template), or a text file containing command-line arguments. Options set explicitly on the command line will still override settings in this file.

- **CASE_ROOT_DIR**: Root directory of model data to analyze.

The second form of the command prints information about available diagnostics; run without an argument (`'mdtf info'`) to see a list of available topics.

5.1.2 General options

- `-h, --help`: Show a help message, potentially more up-to-date than this page, along with your site’s default values for these options.

- `--version`: Show program’s version number and exit.

- `--input-file, --input_file, -f <INPUT_FILE>`: Alternate way to specify user configuration file (with a flag instead of a positional argument.)

---

\(^\text{165}\) https://docs.conda.io/en/latest/

\(^\text{166}\) https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/mdtf.py

\(^\text{167}\) https://github.com/NOAA-GFDL/MDTF-diagnostics/blob/main/src/default_tests.jsonc
5.1.3 Paths

Parent directories of input and output data. Note that all the paths below should be on a local filesystem. Environment variables in paths (eg $HOME) are resolved according to the shell context mdtf was called from. Relative paths are resolved relative to the repo directory.

- **--MODEL-DATA-ROOT, --MODEL_DATA_ROOT <DIR>**: Directory to store input data from different models. Depending on the choice of data_manager (see below), input model data will typically be copied from a remote filesystem to this location.

- **--OBS-DATA-ROOT, --OBS_DATA_ROOT <DIR>**: Directory containing observational data used by individual PODs. Currently, this must be downloaded manually as part of the framework installation. See Section 1.2.1 of the installation guide (page 3) for instructions.

- **--WORKING-DIR, --WORKING_DIR <DIR>**: Working directory.

- **--OUTPUT-DIR, --OUTPUT_DIR, -o <DIR>**: Destination for output files. Currently this must be on the same filesystem as WORKING_DIR.

5.1.4 Model data

- **--CASE-ROOT-DIR, --CASE_ROOT_DIR <DIR>**: Alternate method to specify root directory of model data (with a flag instead of a positional argument.)

- **--FIRSTYR, -Y <year>**: Starting year of analysis period.

- **--LASTYR, -Z <year>**: Ending year of analysis period (inclusive).

- **--CASENAME, -n <name>**: Identifier used to label the input model dataset.

- **--convention, -c <convention>**: Variable name/unit convention used in model data. Defaults to CMIP6-style conventions.

- **--model, -m <source_id>**: Model name (only used in retrieving data from a CMIP6 directory structure).

- **--experiment, -e <experiment_id>**: Experiment ID (only used in retrieving data from a CMIP6 directory structure).

5.1.5 Model data retrieval settings

- **--data-manager, --data_manager <DATA_MANAGER>** Method used to fetch model data.

- **--file-transfer-timeout, --file_transfer_timeout <seconds>** Time (in seconds) to wait before giving up on transferring a data file to the local filesystem. Set to zero to wait indefinitely.

- **--keep-temp, --keep_temp**: Set flag to retain local copies of fetched model data (in MODEL_DATA_ROOT) between runs of the framework. Default is false. This can be useful when you need to run a diagnostic repeatedly for development purposes and the model data hosted remotely.
5.1.6 Diagnostics

- `--pods`, `-p` [...]: List of diagnostics to run on model data, separated by spaces. This can be all (the default), one or more POD names\(^{168}\), or one or more modeling realms. Run `mdtf info pods` for a list of installed PODs.

5.1.7 Runtime settings

- `--environment-manager`, `--environment_manager <ENVIRONMENT_MANAGER>`: Method to manage POD runtime dependencies.
- `--conda-root`, `--conda_root <DIR>`: Path to the Anaconda installation. Only used if `environment_manager = 'Conda'`. Set equal to ‘’ to use conda from your system’s $PATH.
- `--conda-env-root`, `--conda_env_root <DIR>`: Root directory for Anaconda environment installs. Only used if `environment_manager = 'Conda'`. Set equal to ‘’ to install in your system’s default location.
- `--venv-root`, `--venv_root <DIR>`: Root directory for python virtual environments. Only used if `environment_manager = 'Virtualenv'`. Set equal to ‘’ to install in your system’s default location.
- `--r-lib-root`, `--r_lib_root <DIR>`: Root directory for R packages requested by PODs. Only used if `environment_manager = 'Virtualenv'`. Set equal to ‘’ to install in your system library.

5.1.8 Output settings

- `--save-ps`, `--save_ps`: Set flag to have PODs save postscript figures in addition to bitmaps.
- `--save-nc`, `--save_nc`: Set flag to have PODs save netCDF files of processed data.
- `--save-non-nc`, `--save_non_nc`: Set flag to save all processed data except netcdf files.
- `--overwrite`: Set flag to overwrite results in OUTPUT_DIR. Default is false: Runs of the framework are saved as directories with the name `MDTF_<CASENAME>_<FIRSTYR>_<LASTYR>`, so if a directory with that name is found in OUTPUT_DIR, the current results will be saved as `MDTF_<CASENAME>_<FIRSTYR>_<LASTYR>.1`, `MDTF_<CASENAME>_<FIRSTYR>_<LASTYR>.2`, etc.

5.1.9 Debugging settings

- `--verbose`, `-v`: Increase log verbosity level.
- `--test-mode`, `--test_mode`: Set flag for framework test. Data is fetched but PODs are not run.
- `--dry-run`, `--dry_run`: Set flag for framework test. No external commands are run and no remote data is copied. Implies `test_mode`.

\(^{168}\) https://github.com/tsjackson-noaa/MDTF-diagnostics/tree/main/diagnostics
5.2 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\(^\text{169}\) format, with the addition that any text to the right of `//` is treated as a comment and ignored (sometimes called the "JSONC" format).

5.2.1 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.

5.2.2 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...
    }
}
```

(continues on next page)

\(^{169}\) https://en.wikipedia.org/wiki/JSON#Data_types_and_syntax

91
At the top level, the settings file is an object (page 91) containing four required entries, described in detail below.

- **settings** (page 92): properties that label the diagnostic and describe its runtime requirements.
- **data** (page 94): properties that apply to all the data your diagnostic is requesting.
- **dimensions** (page 96): properties that apply to the dimensions (in netCDF 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 98): properties that describe the individual variables your diagnostic operates on. Each variable should be listed as a separate entry here.

### 5.2.3 Settings section

This section is an object (page 91) containing properties that label the diagnostic and describe its runtime requirements.

**Example**

```json
"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"
   }
}
```

170 [https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html](https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html)
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 91) (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 91), 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 “python171” or “ncl172” etc. but also any utilities such as “ncks173”, “cdo174”, etc.
- The value corresponding to each key is an array (page 91) (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, you need to list numpy175 but not math176. 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 versions177. For now, please communicate your diagnostic’s version requirements to the MDTF organizers.

pod_env_vars: object (page 91), 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.

---

171 https://www.python.org/
172 https://www.ncl.ucar.edu/
173 http://nco.sourceforge.net/
174 https://code.mpimet.mpg.de/projects/cdo
175 https://numpy.org/
176 https://docs.python.org/3/library/math.html
5.2.4 Data section

This section is an object (page 91) 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)
  - "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](https://www.unidata.ucar.edu/software/netcdf/docs/faq.html) (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 96) 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...
rename_variables: Boolean. Optional: assumed false if not specified. If set to true, the framework will change the name of all variables 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 101) 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, e.g., xarray.

min_duration, max_duration: Unit-ful quantities (page 91). 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 (e.g., 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 (e.g., 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 98). 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: e.g., it should not assume that 3D data will be presented as (time, lat, lon).

frequency, min_frequency, max_frequency: Unit-ful quantities (page 91). Time frequency at which the data is provided. Either frequency or the min/max pair, or both, is required:

- 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.

5.2.5 Dimensions section

This section is an object (page 91) contains properties that apply to the dimensions of model data. “Dimensions” are meant in the sense of the netCDF data model: informally, they are “coordinate axes” holding the values of independent variables that the dependent variable is sampled at.

All dimensions (page 99) and scalar coordinates (page 100) 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. 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
  },
  "time": {
    "standard_name": "time",
    "units": "days",
    "calendar": "noleap",
    "need_bounds": false
  }
}
```

180 https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcDims.html
181 http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#_independent_latitude_longitude_vertical_and_time_axes
Latitude and Longitude

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

units: Optional. String, following syntax of the UDUnits library\(^{182}\). Units the diagnostic expects the dimension to be in. Currently the framework only supports decimal degrees_north and degrees_east, respectively.

range: Array (page 91) (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\(^{183}\): 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 (e.g. 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\(^{184}\) 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\(^{185}\): the bounds attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

Z axis (height/depth, pressure, …)

standard_name: Required, string. Standard name\(^{186}\) of the variable as defined by the CF conventions\(^{187}\), or a commonly used synonym as employed in the CMIP6 MIP tables.

units: Optional. String, following syntax of the UDUnits library\(^{188}\). Units the diagnostic expects the dimension to be in. If not provided, the framework will assume CF convention canonical units\(^{189}\).

---

\(^{182}\) [https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax](https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax)

\(^{183}\) [http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries](http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries)

\(^{184}\) [http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#calendar](http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#calendar)

\(^{185}\) [http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries](http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries)


\(^{188}\) [https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax](https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax)

**positive**: String, required. Must be "up" or "down", according to the CF conventions\(^{190}\). 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\(^{191}\): 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\(^{192}\) of the variable as defined by the CF conventions\(^{193}\), or a commonly used synonym as employed in the CMIP6 MIP tables.

**units**: Optional. String, following syntax of the UDUnits library\(^{194}\). Units the diagnostic expects the dimension to be in. If not provided, the framework will assume CF convention canonical units\(^{195}\).

**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\(^{196}\): the bounds attribute of this dimension will be set to the name of another netCDF variable containing the bounds information.

### 5.2.6 Varlist section

This section is an object (page 91) 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\(^{197}\): 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.

---

\(^{190}\) [http://cfconventions.org/faq.html#vertical_coords_positive_attribute](http://cfconventions.org/faq.html#vertical_coords_positive_attribute)

\(^{191}\) [http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries](http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries)


\(^{193}\) [http://cfconventions.org/](http://cfconventions.org/)

\(^{194}\) [https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax](https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax)


\(^{196}\) [http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries](http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#cell-boundaries)

\(^{197}\) [https://www.unidata.ucar.edu/software/netcdf/workshops/2010/datamodels/NcVars.html](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 91) containing properties specific to that variable:

- **standard_name**: String, required. Standard name of the variable as defined by the CF conventions, or a commonly used synonym as employed in the CMIP6 MIP tables (e.g. “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 101) 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. Units the diagnostic expects the variable to be in. If not provided, the framework will assume CF convention canonical units.

- **dimensions**: Required. List of strings, which must be selected the keys of entries in the dimensions section. Dimensions of the array containing the variable’s data. Note that the framework

---

199 http://cfconventions.org/
200 https://www.unidata.ucar.edu/software/udunits/udunits-2.0.4/udunits2lib.html#Syntax
will not reorder dimensions (transpose) unless \texttt{dimensions\_ordered} is additionally set to \texttt{true}.

\textbf{dimensions\_ordered}: Boolean. Optional: assumed \texttt{false} if not specified. If \texttt{true}, the framework will ensure that the dimensions of this variable's array are given in the same order as listed in \texttt{dimensions}. If set to \texttt{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 \texttt{description} (page 95) there).

\textbf{scalar\_coordinates}: \texttt{object} (page 91), optional. This implements what the CF conventions refer to as "scalar coordinates\textsuperscript{202}", 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 \texttt{dimensions} (page 96) 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 \texttt{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.

\textbf{frequency, min\_frequency, max\_frequency}: Unit-ful quantities (page 91). Optional. Time frequency at which the variable's data is provided. If given here, overrides the values set globally in the data section (see \texttt{description} (page 95) there).

\textbf{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 \texttt{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.

\textbf{alternates}: \texttt{Array} (page 91) (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 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.)

\textsuperscript{202} http://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#scalar-coordinate-variables
5.3 MDTF Environment variables

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

5.3.1 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 28) 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\(^203\) 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.

5.3.2 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.

- `$WK_DIR/obs/netCDF` and `$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.

5.3.3 Model run information

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

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

5.3.4 Locations of model data files

These are set depending on the data your diagnostic requests in its settings file (page 28). 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.)


5.3.5 Names of variables and dimensions

These are set depending on the data your diagnostic requests in its settings file (page 28). 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 91).

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 91).

### 5.3.6 Simple example

We only give the relevant parts of the settings file (page 91) below.

```json
"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.`
5.3.7 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.

```json
"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.

CHAPTER SIX

INTERNAL FRAMEWORK CODE

src.mdtf (page 105)
src.data_manager (page 106)
src.environment_manager (page 109)
src.shared_diagnostic (page 111)
src.util (page 115)  
Common functions and classes used in multiple places in the MDTF code.

6.1 src.mdtf module

class src.mdtf.MDTFFramework(code_root, defaults_rel_path)
    Bases: object

    __init__(code_root, defaults_rel_path)
        Initial dispatch of CLI args: are we printing help info or running framework.

    cleanup_tempdirs(signum=None, frame=None)

    _framework_init(code_root, defaults_rel_path)

    _cli_pre_parse_hook(cli_obj)

    _cli_post_parse_hook(cli_obj)

    static _populate_from_cli(cli_obj, group_nm, target_d=None)

    parse_mdtf_args(cli_obj, config)
        Parse script options returned by the CLI. For greater customizability, most of the functionality is spun out into sub-methods.

    parse_env_vars(cli_obj, config)

    parse_pod_list(cli_obj, config)

    parse_case_list(cli_obj, config)

    parse_case(case_tup, cli_d, cli_obj, config)

204 https://docs.python.org/3.7/library/functions.html#object
set_case_pod_list(case, cli_obj, config)
parse_paths(cli_obj, config)
_post_parse_hook(cli_obj, config)
verify_paths(config)
_print_config(cli_obj, config)
_dispatch_search = [<module 'src.data_manager' from '/home/docs/checkouts/readthedocs.org/user_builds/mdtf-diagnostics/checkouts/latest/src/netcdf_helper.py'>
manual_dispatch(config)
main_loop()

6.2 src.data_manager module

exception src.data_manager.DataQueryFailure(dataset, msg="")
   Bases: Exception

   Exception signaling a failure to find requested data in the remote location.
   Raised by queryData() to signal failure of a data query. Should be caught properly in planData() or fetchData().

   __init__(dataset, msg="")
       Initialize self. See help(type(self)) for accurate signature.

exception src.data_manager.DataAccessError(dataset, msg="")
   Bases: Exception

   Exception signaling a failure to obtain data from the remote location.

   __init__(dataset, msg="")
       Initialize self. See help(type(self)) for accurate signature.

class src.data_manager.DataSet(*args, **kwargs)
   Bases: src.util.NameSpace (page 116)

   Class to describe datasets.

   https://stackoverflow.com/a/48806603 for implementation.

   __init__(*args, **kwargs)
       Initialize self. See help(type(self)) for accurate signature.

copy() → a shallow copy of D

classmethod from_pod_varlist(pod_convention, var, dm_args)

   property is_static
       Check for time-independent data (‘fx’ in CMIP6 DRS.) Do the comparison by checking

---

205 https://docs.python.org/3.7/library/exceptions.html#Exception
206 https://docs.python.org/3.7/library/exceptions.html#Exception
date_range against the placeholder value because that’s unique – we may be using a different DateFrequency depending on the data source.

_freeze()
   Return immutable representation of (current) attributes.
   Exclude attributes starting with ‘_’ from the comparison, in case we want DataSets with different timestamps, temporary directories, etc. to compare as equal.

class src.data_manager.DataManager(case_dict, DateFreqMixin=None)
   Bases: object
   __init__(case_dict, DateFreqMixin=None)
      Initialize self. See help(type(self)) for accurate signature.
   iter_pods()
      Generator iterating over all pods which haven’t been skipped due to requirement errors.
   iter_vars()
      Generator iterating over all variables in all pods which haven’t been skipped due to requirement errors.
   setUp(verbose=0)
   _setup_pod(pod)
   static dataset_key(dataset)
      Return immutable representation of DataSet. Two DataSets should have the same key.
   local_path(data_key)
      Returns the absolute path of the local copy of the file for dataset.
      This determines the local model data directory structure, which is
      $MODEL_DATA_ROOT/<CASENAME>/<freq>/<CASENAME>.<var name>.<freq>.nc'.
      Files not following this convention won’t be found.
   _build_data_dicts()
   fetch_data()
   _fetch_exception_handler(exc)
   _query_data()
   _iter_populated_varlist(var_iter, pod_name)
      Generator function yielding either a variable, its alternates if the variable was not found in the
      data query, or DataQueryFailure if the variable request can’t be satisfied with found data.
   remote_data_list()
      Process list of requested data to make data fetching efficient.
      This is intended as a hook to be used by subclasses. Default behavior is to delete from the list
      duplicate datasets and datasets where a local copy of the data already exists and is current (as
determined by local_data_is_current()).
      Returns: collection of DataSet objects.

207 https://docs.python.org/3.7/library/functions.html#object
_fetch_order_function = None

local_data_is_current(dataset)
    Determine if local copy of data needs to be refreshed.
    
    This is intended as a hook to be used by subclasses. Default is to always return False, ie always
    fetch remote data.
    
    Returns: True if local copy of data exists and remote copy hasn’t been updated.

plan_data_fetch_hook()

preprocess_local_data(*args, **kwargs)

abstract query_dataset(dataset)

abstract fetch_dataset(dataset)

tearDown()

_make_html(clean_up=True)

_backup_config_file(config)
    Record settings in file variab_dir/config_save.json for rerunning

_make_tar_file(tar_dest_dir)
    Make tar file of web/bitmap output.

_copy_to_output()

_abc_impl = <_abc_data object>

class src.data_manager.LocalfileDataManager(case_dict, DateFreqMixin=None)
    Bases: src.data_manager.DataManager (page 107)

    _abc_impl = <_abc_data object>

class DataKey(name_in_model, date_freq)
    Bases: tuple

    _asdict()
        Return a new OrderedDict which maps field names to their values.

    _fields = ('name_in_model', 'date_freq')

    _fields_defaults = {}

classmethod _make(iterable)
    Make a new DataKey object from a sequence or iterable

    _replace(**kwds)
        Return a new DataKey object replacing specified fields with new values

    property date_freq
        Alias for field number 1

    property name_in_model
        Alias for field number 0

---

[208] https://docs.python.org/3.7/library/stdtypes.html#tuple
dataset_key(dataset)
    Return immutable representation of DataSet. Two DataSets should have the same key

query_dataset(dataset)

local_data_is_current(dataset)
    Determine if local copy of data needs to be refreshed.

    This is intended as a hook to be used by subclasses. Default is to always return False, ie always
    fetch remote data.

    Returns: True if local copy of data exists and remote copy hasn’t been updated.

fetch_dataset(dataset)

6.3 src.environment_manager module

class src.environment_manager.EnvironmentManager(verbose=0)
    Bases: object

    __init__(verbose=0)
        Initialize self. See help(type(self)) for accurate signature.

    abstract create_environment(env_name)
    abstract set_pod_env(pod)
    abstract activate_env_commands(env_name)
    abstract deactivate_env_commands(env_name)
    abstract destroy_environment(env_name)

setUp()
    run(verbos=0)

spawn_subprocess(cmd_list, env_name, env=None, cwd=None, stdout=None, stderr=None)
    tearDown()
    subprocess_cleanup(signum=None, frame=None)

    _abc_impl = <_abc_data object>

class src.environment_manager.NoneEnvironmentManager(verbos=0)
    Bases: src.environment_manager.EnvironmentManager (page 109)

    create_environment(env_name)
    destroy_environment(env_name)
    set_pod_env(pod)
    activate_env_commands(env_name)

209 https://docs.python.org/3.7/library/functions.html#object
deactivate_env_commands(env_name)

_abc_impl = _abc_data object_

class src.environment_manager.VirtualenvEnvironmentManager(verbose=0)
    Bases: src.environment_manager.EnvironmentManager (page 109)

    __init__(verbose=0)
        Initialize self. See help(type(self)) for accurate signature.

create_environment(env_name)

    _create_py_venv(env_name)

    _create_r_venv(env_name)

destroy_environment(env_name)

set_pod_env(pod)

activate_env_commands(env_name)

deactivate_env_commands(env_name)

    _abc_impl = _abc_data object_

class src.environment_manager.CcondaEnvironmentManager(verbose=0)
    Bases: src.environment_manager.EnvironmentManager (page 109)

    env_name_prefix = '_MDTF_'

    __init__(verbose=0)
        Initialize self. See help(type(self)) for accurate signature.

create_environment(env_name)

    _call_conda_create(env_name)

create_all_environments()

destroy_environment(env_name)

set_pod_env(pod)

    _abc_impl = _abc_data object_

activate_env_commands(env_name)

        Source conda_init.sh to set things that aren’t set b/c we aren’t in an interactive shell.

deactivate_env_commands(env_name)
6.4 src.shared_diagnostic module

exception src.shared_diagnostic.PodRequirementFailure(pod, msg=None)

Bases: Exception

Exception raised if POD doesn’t have required resoruces to run.

__init__ (pod, msg=None)

Initialize self. See help(type(self)) for accurate signature.

class src.shared_diagnostic.Diagnostic(pod_name, verbose=0)

Bases: object

Class holding configuration for a diagnostic script.

This is the analogue of TestCase in the xUnit analogy.

Object attributes are read from entries in the settings section of the POD’s settings.json file upon initialization.

Attributes

- driver (str) – Filename of the top-level driver script for the POD.
- long_name (str) – POD’s name used for display purposes. May contain spaces.
- description (str) – Short description of POD inserted by the link in the top-level index.html file.
- required_programs (list of str, optional) – List of executables required by the POD (typically language interpreters). validate_environment.sh will make sure these are on the environment’s $PATH before the POD is run.
- required_ncl_scripts (list of str, optional) – List of NCL scripts required by the POD, if any. validate_environment.sh will make sure these are on the environment’s $PATH before the POD is run.

__init__ (pod_name, verbose=0)

POD initializer. Given a POD name, we attempt to read a settings.json file in a subdirectory of /diagnostics by that name and parse the contents.

Parameters

- pod_name (str) – Name of the POD to initialize.
- verbose (int, optional) – Logging verbosity level. Default 0.
iter_vars_and_alts()
Generator iterating over all variables and alternates in POD’s varlist.

_parse_pod_settings(settings, verbose=0)
Private method called by __init__().

Parameters
- settings (dict) – Contents of the settings portion of the POD’s settings.json file.
- verbose (int, optional) – Logging verbosity level. Default 0.

Returns Dict of parsed settings.

_parse_pod_varlist(varlist, verbose=0)
Private method called by __init__().

Parameters
- varlist (list of dict) – Contents of the varlist portion of the POD’s settings.json file.
- verbose (int, optional) – Logging verbosity level. Default 0.

Returns varlist

setUp(verbos=0)
Perform filesystem operations and checks prior to running the POD.
In order, this 1) sets environment variables specific to the POD, 2) creates POD-specific working directories, and 3) checks for the existence of the POD’s driver script.

Note: The existence of data files is checked with data_manager.DataManager.fetchData() and the runtime environment is validated separately as a function of environment_manager.EnvironmentManager.run(). This is because each POD is run in a subprocess (due to the necessity of supporting multiple languages) so the validation must take place in that subprocess.

Raises: PodRequirementFailure if requirements aren’t met. This is re-raised from the _check_pod_driver() and _check_for_varlist_files() subroutines.

_set_pod_env_vars(verbos=0)
Private method called by setUp(). Sets all environment variables for POD.

Parameters verbose (int, optional) – Logging verbosity level. Default 0.

_setup_pod_directories(verbos=0)
Private method called by setUp().

https://docs.python.org/3.7/library/stdtypes.html#dict
https://docs.python.org/3.7/library/functions.html#int
https://docs.python.org/3.7/library/stdtypes.html#list
https://docs.python.org/3.7/library/stdtypes.html#dict
https://docs.python.org/3.7/library/functions.html#int
https://docs.python.org/3.7/library/functions.html#int
https://docs.python.org/3.7/library/functions.html#int
Parameters `verbose (int, optional)` – Logging verbosity level. Default 0.

`_check_pod_driver (verbose=0)`

Private method called by `setUp()`.

Parameters `verbose (int, optional)` – Logging verbosity level. Default 0.

Raises: `PodRequirementFailure` if driver script can’t be found.

`_check_for_varlist_files (varlist, verbose=0)`

Verify that all data files needed by a POD exist locally.

Private method called by `fetchData()`.

Parameters

- `varlist (list of dict)` – Contents of the varlist portion of the POD’s settings.json file.
- `verbose (int, optional)` – Logging verbosity level. Default 0.

Returns: `tuple of found and missing file lists`. Note that this is called recursively.

`run_commands ()`

Produces the shell command(s) to run the POD. Called by `environment_manager.EnvironmentManager.run()`.

Returns Command-line invocation to run the POD.

Return type (list of str)

`validate_commands ()`

Produces the shell command(s) to validate the POD’s runtime environment (ie, check for all requested third-party module dependencies.)

Called by `environment_manager.EnvironmentManager.run()`. Dependencies are passed as arguments to the shell script `src/validate_environment.sh`, which is invoked in the POD’s subprocess before the POD is run.

Returns

Command-line invocation to validate the POD’s runtime environment.

Return type (list of str)
tearDown(verbose=0)
Performs cleanup tasks when the POD has finished running.

In order, this 1) creates the POD’s HTML output page from its included template, replacing CASENAME and other template variables with their current values, and adds a link to the POD’s page from the top-level HTML report; 2) converts the POD’s output plots (in PS or EPS vector format) to a bitmap format for webpage display; 3) Copies all requested files to the output directory and deletes temporary files.

Parameters verbose (int, optional) – Logging verbosity level. Default 0.

make_pod_html()
Perform templating on POD’s html results page(s).

A wrapper for append_html_template(). Looks for all html files in POD_CODE_DIR, templates them, and copies them to POD_WK_DIR, respecting subdirectory structure (see doc for recursive_copy()).

append_result_link(error=None)
Update the top level index.html page with a link to this POD’s results.

This simply appends one of two html fragments to index.html: pod_result_snippet.html if the POD completed successfully, or pod_error_snippet.html if an exception was raised during the POD’s setup or execution.

Parameters error (default None) – Exception object (if any) that was raised during POD’s attempted execution. If this is None, assume that POD ran successfully.

verify_pod_links()
Check for missing files linked to from POD’s html page.

See documentation for LinkVerifier. This method calls LinkVerifier to check existence of all files linked to from the POD’s own top-level html page (after templating). If any files are missing, an error message listing them is written to the run’s index.html (located in src/html/pod_missing_snippet.html).

convert_pod_figures(src_subdir, dest_subdir)
Convert all vector graphics in POD_WK_DIR/subdir to .png files using ghostscript.

All vector graphics files (identified by extension) in any subdirectory of POD_WK_DIR/src_subdir are converted to .png files by running ghostscript in a subprocess. Ghostscript is included in the _MDTF_base conda environment. Afterwards, any bitmap files (identified by extension) in any subdirectory of POD_WK_DIR/src_subdir are moved to POD_WK_DIR/dest_subdir, preserving and subdirectories (see doc for recursive_copy()).

Parameters

- src_subdir – Subdirectory tree of POD_WK_DIR to search for vector graphics files.
- dest_subdir – Subdirectory tree of POD_WK_DIR to move converted bitmap files to.

`cleanup_pod_files()`
Copy and remove remaining files to POD_WK_DIR.

In order, this 1) copies .pdf documentation (if any) from POD_CODE_DIR/doc, 2) copies any bitmap figures in any subdirectory of POD_OBS_DATA to POD_WK_DIR/obs (needed for legacy PODs without digested observational data), 3) removes vector graphics if requested, 4) removes netCDF scratch files in POD_WK_DIR if requested.

Settings are set at runtime, when ConfigManager is initialized.

## 6.5 src.util module

Common functions and classes used in multiple places in the MDTF code. Specifically, util.py implements general functionality that’s not MDTF-specific.

```python
class src.util._Singleton
    Bases: type

    Private metaclass that creates a Singleton base class when called. This version is copied from https://stackoverflow.com/a/6798042 and should be compatible with both Python 2 and 3.

    _instances = {}

class src.util.Singleton(*args, **kwargs)
    Bases: src.util.SingletonMeta

    Parent class defining the Singleton pattern. We use this as safer way to pass around global state.

    classmethod _reset()
        Private method of all Singleton-derived classes added for use in unit testing only. Calling this method on test teardown deletes the instance, so that tests coming afterward will initialize the Singleton correctly, instead of getting the state set during previous tests.

class src.util.ExceptionPropagatingThread(group=None, target=None, name=None, args=(), kw_args=(), daemon=None)
    Bases: threading.Thread

    Class to propagate exceptions raised in a child thread back to the caller thread when the child is join()ed. Adapted from https://stackoverflow.com/a/31614591.

    run()
        Method representing the thread’s activity.

        You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kw_args arguments, respectively.
```

---

240 https://docs.python.org/3.7/library/functions.html#type
242 https://docs.python.org/3.7/library/threading.html#threading.Thread
join(timeout=None)
    Wait until the thread terminates.

    This blocks the calling thread until the thread whose join() method is called terminates – either
    normally or through an unhandled exception or until the optional timeout occurs.

    When the timeout argument is present and not None, it should be a floating point number speci-
    fying a timeout for the operation in seconds (or fractions thereof). As join() always returns None,
    you must call is_alive() after join() to decide whether a timeout happened – if the thread is still
    alive, the join() call timed out.

    When the timeout argument is not present or None, the operation will block until the thread
    terminates.

    A thread can be join()ed many times.

    join() raises a RuntimeError if an attempt is made to join the current thread as that would cause
    a deadlock. It is also an error to join() a thread before it has been started and attempts to do so
    raises the same exception.

class src.util.MultiMap(*args, **kwargs)
    Bases: collections.defaultdict

    Extension of the dict class that allows doing dictionary lookups from either keys or values.

    Syntax for lookup from keys is unchanged, bd['key'] = 'val', while lookup from values is done
    on the inverse attribute and returns a set of matching keys if more than one match is present: bd.
    inverse['val'] = ['key1', 'key2']. See https://stackoverflow.com/a/21894086.

    __init__(*args, **kwargs)
        Initialize MultiMap by passing an ordinary dict.

    get_(key)

to_dict()

inverse()

inverse_get_(val)

class src.util.NameSpace
    Bases: dict

    A dictionary that provides attribute-style access.

    For example, d['key'] = value becomes d.key = value. All methods of dict are supported.

    Note: recursive access (d.key.subkey, as in C-style languages) is not supported.

    Implementation is based on https://github.com/Infinidat/munch.

243 https://docs.python.org/3.7/library/collections.html#collections.defaultdict
244 https://docs.python.org/3.7/library/stdtypes.html#dict
245 https://docs.python.org/3.7/library/stdtypes.html#dict
246 https://docs.python.org/3.7/library/stdtypes.html#dict
247 https://docs.python.org/3.7/library/stdtypes.html#dict
toDict()
   Recursively converts a NameSpace back into a dictionary.

classmethod _toDict(x)
   Recursively converts a NameSpace back into a dictionary. nb. As dicts are not hashable, they
   cannot be nested in sets/frozensets.

classmethod fromDict(x)
   Recursively transforms a dictionary into a NameSpace via copy. nb. As dicts are not hashable,
   they cannot be nested in sets/frozensets.

copy() \rightarrow a shallow copy of D

_freeze()
   Return immutable representation of (current) attributes.

   We do this to enable comparison of two Namespaces, which otherwise would be done by the
default method of testing if the two objects refer to the same location in memory. See https://
   stackoverflow.com/a/45170549.

src.util.strip_comments(str_, delimiter=None)

src.util.read_json(file_path)

src.util.parse_json(str_)

src.util.write_json(struct, file_path, verbose=0, sort_keys=False)
   Wrapping file I/O simplifies unit testing.
   Parameters
   • struct (dict\_) –
   • file_path (str\_) – path of the JSON file to write.
   • verbose (int\_), optional – Logging verbosity level. Default 0.

src.util.pretty_print_json(struct, sort_keys=False)
   Pseudo-YAML output for human-readable debugging output only - not valid JSON

src.util.find_files(src_dirs, filename_globs)
   Return list of files in src_dirs matching any of filename_globs.
   Wraps glob.glob for the use cases encountered in cleaning up POD output.
   Parameters
   • src_dirs – Directory, or a list of directories, to search for files in. The function
     will also search all subdirectories.
   • filename_globs – Glob, or a list of globs, for filenames to match. This is a shell
     globbing pattern, not a full regex.

\(^{248}\) https://docs.python.org/3.7/library/stdtypes.html#dict
\(^{249}\) https://docs.python.org/3.7/library/stdtypes.html#str
\(^{250}\) https://docs.python.org/3.7/library/functions.html#int
Returns: list\(^{251}\) of paths to files matching any of the criteria. If no files are found, the list is empty.

src.util.recursive_copy(src_files, src_root, dest_root, copy_function=None, overwrite=False)
Copy src_files to dest_root, preserving relative subdirectory structure.

Copies a subset of files in a directory subtree rooted at src_root to an identical subtree structure rooted
at dest_root, creating any subdirectories as needed. For example, recursive_copy(‘/A/B/C.txt’, ‘/A’, ‘/D’) will first create the destination subdirectory /D/B and copy ‘/A/B/C.txt’ to /D/B/C.txt.

Parameters

- **src_files** – Absolute path, or list of absolute paths, to files to copy.
- **src_root** – Root subtree of all files in src_files. Raises a ValueError if all files in
src_files are not contained in the src_root directory.
- **dest_root** – Destination directory in which to create the copied subtree.
- **copy_function** – Function to use to copy individual files. Must take two ar-
guments, the source and destination paths, respectively. Defaults to shutil.copy2().
- **overwrite** – Boolean, deafault False. If False, raise an OSError if any destination
files already exist, otherwise silently overwrite.

src.util.resolve_path(path, root_path='', env=None)
Abbreviation to resolve relative paths.

Parameters

- **path** (str\(^{252}\)) – path to resolve.
- **root_path** (str\(^{253}\), optional) – root path to resolve path with. If not given, re-
solves relative to os.getcwd.

Returns: Absolute version of path, relative to root_path if given, otherwise relative to os.getcwd.

src.util.check_executable(exec_name)
Tests if <exec_name> is found on the current $PATH.

Parameters **exec_name** (str\(^{254}\)) – Name of the executable to search for.

Returns: bool\(^{255}\) True/false if executable was found on $PATH.

src.util.poll_command(command, shell=False, env=None)
Runs a shell command and prints stdout in real-time.

Optional ability to pass a different environment to the subprocess. See documentation for the Python2 subprocess\(^{256}\) module.

---

\(^{251}\) [https://docs.python.org/3.7/library/stdtypes.html#list](https://docs.python.org/3.7/library/stdtypes.html#list)

\(^{252}\) [https://docs.python.org/3.7/library/stdtypes.html#str](https://docs.python.org/3.7/library/stdtypes.html#str)

\(^{253}\) [https://docs.python.org/3.7/library/stdtypes.html#str](https://docs.python.org/3.7/library/stdtypes.html#str)

\(^{254}\) [https://docs.python.org/3.7/library/stdtypes.html#str](https://docs.python.org/3.7/library/stdtypes.html#str)

\(^{255}\) [https://docs.python.org/3.7/library/functions.html#bool](https://docs.python.org/3.7/library/functions.html#bool)

\(^{256}\) [https://docs.python.org/2/library/subprocess.html](https://docs.python.org/2/library/subprocess.html)
Parameters

- **command** – list of command + arguments, or the same as a single string. See subprocess syntax. Note this interacts with the shell setting.
- **shell** (bool\(^{257}\), optional) – shell flag, passed to Popen, default False.
- **env** (dict\(^{258}\), optional) – environment variables to set, passed to Popen, default None.

**exception** src.util.TimeoutAlarm

Bases: Exception\(^{259}\)

src.util.run_command(command, env=None, cwd=None, timeout=0, dry_run=False)

Subprocess wrapper to facilitate running single command without starting a shell.

---

Note: We hope to save some process overhead by not running the command in a shell, but this means the command can’t use piping, quoting, environment variables, or filename globbing etc.

See documentation for the Python2 subprocess\(^{260}\) module.

Parameters

- **command** (list of str\(^{261}\)) – List of commands to execute
- **env** (dict\(^{262}\), optional) – environment variables to set, passed to Popen, default None.
- **cwd** (str\(^{263}\), optional) – child processes’ working directory, passed to Popen. Default is None, which uses parent processes’ directory.
- **timeout** (int\(^{264}\), optional) – Optionally, kill the command’s subprocess and raise a CalledProcessError if the command doesn’t finish in timeout seconds.

Returns list\(^{265}\) of str\(^{266}\) containing output that was written to stdout by each command.

Note: this is split on newlines after the fact.

Raises CalledProcessError – If any commands return with nonzero exit code. Stderr for that command is stored in output attribute.

src.util.run_shell_command(command, env=None, cwd=None, dry_run=False)

Subprocess wrapper to facilitate running shell commands.

See documentation for the Python2 subprocess\(^{267}\) module.

---

\(^{257}\) https://docs.python.org/3.7/library/functions.html#bool

\(^{258}\) https://docs.python.org/3.7/library/stdtypes.html#dict

\(^{259}\) https://docs.python.org/3.7/library/exceptions.html#Exception

\(^{260}\) https://docs.python.org/2/library/subprocess.html

\(^{261}\) https://docs.python.org/3.7/library/stdtypes.html#str

\(^{262}\) https://docs.python.org/3.7/library/stdtypes.html#dict

\(^{263}\) https://docs.python.org/3.7/library/stdtypes.html#str

\(^{264}\) https://docs.python.org/3.7/library/functions.html#int

\(^{265}\) https://docs.python.org/3.7/library/stdtypes.html#list

\(^{266}\) https://docs.python.org/3.7/library/stdtypes.html#str

\(^{267}\) https://docs.python.org/2/library/subprocess.html
Parameters

- **commands** (list of `str`\(^{268}\)) – List of commands to execute
- **env** (`dict`\(^{269}\), optional) – Environment variables to set, passed to Popen, default None.
- **cwd** (`str`\(^{270}\), optional) – Child processes’ working directory, passed to Popen. Default is None, which uses parent processes’ directory.

Returns `list`\(^{271}\) of `str`\(^{272}\) containing output that was written to stdout by each command. Note: this is split on newlines after the fact, so if commands give != 1 lines of output this will not map to the list of commands given.

Raises **CalledProcessError** – If any commands return with nonzero exit code. Stderr for that command is stored in output attribute.

```python
src.util.is_iterable(obj)
src.util.coerce_to_iter(obj, coll_type=<class 'list'>)
src.util.coerce_from_iter(obj)
src.util.filter_kwars(kwarg_dict, function)
    Given a dict of kwargs, return only those kwargs accepted by function.
src.util.signal_logger(caller_name, signum=None, frame=None)
    Lookup signal name from number; [https://stackoverflow.com/a/2549950](https://stackoverflow.com/a/2549950).
```

---

\(^{268}\) [https://docs.python.org/3.7/library/stdtypes.html#str](https://docs.python.org/3.7/library/stdtypes.html#str)
\(^{269}\) [https://docs.python.org/3.7/library/stdtypes.html#dict](https://docs.python.org/3.7/library/stdtypes.html#dict)
\(^{270}\) [https://docs.python.org/3.7/library/stdtypes.html#str](https://docs.python.org/3.7/library/stdtypes.html#str)
\(^{271}\) [https://docs.python.org/3.7/library/stdtypes.html#list](https://docs.python.org/3.7/library/stdtypes.html#list)
\(^{272}\) [https://docs.python.org/3.7/library/stdtypes.html#str](https://docs.python.org/3.7/library/stdtypes.html#str)
ACKNOWLEDGEMENTS

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

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

7.1 Disclaimer

This repository is a scientific product and is not an official communication of the National Oceanic and Atmospheric Administration, or the United States Department of Commerce. All NOAA GitHub project code is provided on an ‘as is’ basis and the user assumes responsibility for its use. Any claims against the Department of Commerce or Department of Commerce bureaus stemming from the use of this GitHub project will be governed by all applicable Federal law. Any reference to specific commercial products, processes, or services by service mark, trademark, manufacturer, or otherwise, does not constitute or imply their endorsement, recommendation or favoring by the Department of Commerce. The Department of Commerce seal and logo, or the seal and logo of a DOC bureau, shall not be used in any manner to imply endorsement of any commercial product or activity by DOC or the United States Government.

273 https://www.noaa.gov/
274 https://cpo.noaa.gov/Meet-the-Divisions/Earth-System-Science-and-Modeling/MAPP
275 https://www.ucla.edu/
276 https://www.gfdl.noaa.gov/
277 https://ncar.ucar.edu/
278 https://www.colostate.edu/
279 https://www.llnl.gov/
280 https://www.energy.gov/
S

src.data_manager, 106
src.environment_manager, 109
src.mdtf, 105
src.shared_diagnostic, 111
src.util, 115
Symbols

_Singleton (class in src.util), 115
__init__() (src.data_manager.DataAccessError method), 106
__init__() (src.data_manager.DataManager method), 107
__init__() (src.data_manager.DataQueryFailure method), 106
__init__() (src.data_manager.DataSet method), 106
__init__() (src.environment_manager.CondaEnvironmentManager method), 110
__init__() (src.environment_manager.EnvironmentManager method), 109
__init__() (src.environment_manager.VirtualenvEnvironmentManager method), 110
__init__() (src.mdtf.MDTFFramework method), 105
__init__() (src.shared_diagnostic.Diagnostic method), 111
__init__() (src.shared_diagnostic.PodRequirementFailure method), 111
__init__() (src.util.MultiMap method), 116
_abc_impl (src.data_manager.DataManager attribute), 108
_abc_impl (src.data_manager.LocalfileDataManager attribute), 108
_abc_impl (src.environment_manager.CondaEnvironmentManager attribute), 110
_abc_impl (src.environment_manager.EnvironmentManager attribute), 109
_abc_impl (src.environment_manager.NoneEnvironmentManager attribute), 110
_abc_impl (src.environment_manager.VirtualenvEnvironmentManager attribute), 110
_asdict() (src.data_manager.LocalfileDataManager.DataKey method), 108
_backup_config_file() (src.data_manager.DataManager method), 108
_build_data_dicts() (src.data_manager.DataManager method), 107
_call_conda_create() (src.environment_manager.CondaEnvironmentManager method), 110
_check_for_varlist_files() (src.shared_diagnostic.Diagnostic method), 113
_check_pod_driver() (src.shared_diagnostic.Diagnostic method), 113
_cli_post_parse_hook() (src.mdtf.MDTFFramework method), 105
_cli_pre_parse_hook() (src.mdtf.MDTFFramework method), 105
_copy_to_output() (src.data_manager.DataManager method), 108
_create_py_venv() (src.environment_manager.VirtualenvEnvironmentManager method), 110
_create_r_venv() (src.environment_manager.VirtualenvEnvironmentManager method), 110
_dispatch_search (src.mdtf.MDTFFramework attribute), 106
_fetch_exception_handler() (src.data_manager.DataManager method), 107
_fetch_order_function (src.data_manager.DataManager attribute), 108
_fields (src.data_manager.LocalfileDataManager.DataKey attribute), 108
_fields_defaults (src.data_manager.LocalfileDataManager.DataKey attribute), 108
_framework_init() (src.mdtf.MDTFFramework method), 105
_freeze() (src.data_manager.DataSet method), 107
_freeze() (src.util.NameSpace method), 117
_instances (src._Singleton attribute), 115
_iter_populated_varlist() (src.data_manager.DataManager method), 107
_make() (src.data_manager.LocalfileDataManager.Data class method), 108
_make_html() (src.data_manager.DataManager method), 108
_make_tar_file() (src.data_manager.DataManager method), 108
_parse_pod_settings() (src.shared_diagnostic.Diagnostic method), 112
_parse_pod_varlist() (src.shared_diagnostic.Diagnostic method), 112
_populate_from_cli() (src.mdtf.MDTFFramework static method), 105
_post_parse_hook() (src.mdtf.MDTFFramework method), 106
_print_config() (src.mdtf.MDTFFramework method), 106
_query_data() (src.data_manager.DataManager method), 107
_replace() (src.data_manager.LocalfileDataManager.DataKey method), 108
_reset() (src.util.Singleton class method), 115
_set_pod_env_vars() (src.shared_diagnostic.Diagnostic method), 112
_setup_pod() (src.data_manager.DataManager method), 107
_setup_pod_directories() (src.shared_diagnostic.Diagnostic method), 112
_toDict() (src.util.NameSpace class method), 117
activate_env_commands() (src.environment_manager.CcondaEnvironmentManager method), 110
activate_env_commands() (src.environment_manager.EnvironmentManager method), 109
activate_env_commands() (src.environment_manager.NoneEnvironmentManager method), 109
activate_env_commands() (src.environment_manager.VirtualenvEnvironmentManager method), 110
append_result_link() (src.shared_diagnostic.Diagnostic method), 114
check_executable() (in module src.util), 118
cleanup_pod_files() (src.shared_diagnostic.Diagnostic method), 115
cleanup_tempdirs() (src.mdtf.MDTFFramework method), 105
coerce_from_iter() (in module src.util), 120
coerce_to_iter() (in module src.util), 120
CondaEnvironmentManager (class in src.environment_manager), 110
copy() (src.data_manager.DataSet method), 106
copy() (src.util.NameSpace method), 117
create_all_environments() (src.environment_manager.CcondaEnvironmentManager method), 110
create_environment() (src.environment_manager.CcondaEnvironmentManager method), 110
create_environment() (src.environment_manager.EnvironmentManager method), 109
create_environment() (src.environment_manager.NoneEnvironmentManager method), 109
<table>
<thead>
<tr>
<th>Function/Publication/Attribute</th>
<th>Class/Module Source</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create_environment()</code></td>
<td><code>src.environment_manager.VirtaulenvEnvironmentManager Method</code>, 110</td>
</tr>
<tr>
<td><code>DataAccessError</code></td>
<td>106</td>
</tr>
<tr>
<td><code>DataManager</code></td>
<td>(class in <code>src.data_manager</code>), 107</td>
</tr>
<tr>
<td><code>DataQueryFailure</code></td>
<td>106</td>
</tr>
<tr>
<td><code>DataSet</code></td>
<td>(class in <code>src.data_manager</code>), 106</td>
</tr>
<tr>
<td><code>dataset_key()</code></td>
<td>(src.data_manager DataManager static method), 107</td>
</tr>
<tr>
<td><code>date_freq()</code></td>
<td>(src.data_manager.LocalfileDataManager DataKey property), 108</td>
</tr>
<tr>
<td><code>deactivate_env_commands()</code></td>
<td>(src.environment_manager.CondaEnvironmentManager method), 110</td>
</tr>
<tr>
<td><code>destroy_environment()</code></td>
<td>(src.environment_manager.CondaEnvironmentManager method), 110</td>
</tr>
<tr>
<td><code>destroy_environment()</code></td>
<td>(src.environment_manager.EnvironmentManager method), 109</td>
</tr>
<tr>
<td><code>destroy_environment()</code></td>
<td>(src.environment_manager.NoneEnvironmentManager method), 109</td>
</tr>
<tr>
<td><code>destroy_environment()</code></td>
<td>(src.environment_manager.VirtaulenvEnvironmentManager method), 110</td>
</tr>
<tr>
<td><code>Diagnostic</code></td>
<td>(class in <code>src.shared_diagnostic</code>), 111</td>
</tr>
<tr>
<td><code>env_name_prefix</code></td>
<td>(src.environment_manager.CondaEnvironmentManager attribute), 110</td>
</tr>
<tr>
<td><code>EnvironmentManager</code></td>
<td>(class in <code>src.environment_manager</code>), 109</td>
</tr>
<tr>
<td><code>ExceptionPropagatingThread</code></td>
<td>(class in <code>src.util</code>), 115</td>
</tr>
<tr>
<td><code>fetch_data()</code></td>
<td>(src.data_manager DataManager method), 107</td>
</tr>
<tr>
<td><code>fetch_dataset()</code></td>
<td>(src.data_manager DataManager method), 108</td>
</tr>
<tr>
<td><code>fetch_dataset()</code></td>
<td>(src.data_manager.LocalfileDataManager method), 109</td>
</tr>
<tr>
<td><code>filter_kwargs()</code></td>
<td>(in module <code>src.util</code>), 120</td>
</tr>
<tr>
<td><code>find_files()</code></td>
<td>(in module <code>src.util</code>), 117</td>
</tr>
<tr>
<td><code>from_pod_varlist()</code></td>
<td>(src.data_manager.DataSet class method), 106</td>
</tr>
<tr>
<td><code>fromDict()</code></td>
<td>(src.util.NameSpace class method), 117</td>
</tr>
<tr>
<td><code>get_()</code></td>
<td>(src.util.MultiMap method), 116</td>
</tr>
<tr>
<td><code>inverse()</code></td>
<td>(src.util.MultiMap method), 116</td>
</tr>
<tr>
<td><code>inverse_get_()</code></td>
<td>(src.util.MultiMap method), 116</td>
</tr>
<tr>
<td><code>is_iterable()</code></td>
<td>(in module <code>src.util</code>), 120</td>
</tr>
<tr>
<td><code>is_static()</code></td>
<td>(src.data_manager.DataSet property), 106</td>
</tr>
<tr>
<td><code>iter_pods()</code></td>
<td>(src.data_manager DataManager method), 107</td>
</tr>
<tr>
<td><code>iter_vars()</code></td>
<td>(src.data_manager DataManager method), 107</td>
</tr>
<tr>
<td><code>local_data_is_current()</code></td>
<td>(src.data_manager.LocalfileDataManager DataManager method), 108</td>
</tr>
<tr>
<td><code>local_data_is_current()</code></td>
<td>(src.data_manager.LocalfileDataManager DataManager method), 109</td>
</tr>
<tr>
<td><code>inverse()</code></td>
<td>(in module <code>src.util</code>), 120</td>
</tr>
<tr>
<td><code>is_iterable()</code></td>
<td>(in module <code>src.util</code>), 120</td>
</tr>
<tr>
<td><code>is_static()</code></td>
<td>(src.data_manager.DataSet property), 106</td>
</tr>
<tr>
<td><code>iter_pods()</code></td>
<td>(src.data_manager DataManager method), 107</td>
</tr>
<tr>
<td><code>iter_vars()</code></td>
<td>(src.data_manager DataManager method), 107</td>
</tr>
<tr>
<td><code>local_data_is_current()</code></td>
<td>(src.data_manager.LocalfileDataManager DataManager method), 108</td>
</tr>
<tr>
<td><code>local_data_is_current()</code></td>
<td>(src.data_manager.LocalfileDataManager DataManager method), 109</td>
</tr>
<tr>
<td>M</td>
<td>Q</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>main_loop() (src.mdtf.MDTFFramework method), 106</td>
<td>query_dataset() (src.data_manager.DataManager method), 108</td>
</tr>
<tr>
<td>make_pod_html() (src.shared_diagnostic.Diagnostic method), 114</td>
<td>query_dataset() (src.data_manager.LocalfileDataManager method), 109</td>
</tr>
<tr>
<td>manual_dispatch() (src.mdtf.MDTFFramework method), 106</td>
<td></td>
</tr>
<tr>
<td>MDTFFramework (class in src.mdtf), 105</td>
<td></td>
</tr>
<tr>
<td>module</td>
<td></td>
</tr>
<tr>
<td>src.data_manager, 106</td>
<td></td>
</tr>
<tr>
<td>src.environment_manager, 109</td>
<td></td>
</tr>
<tr>
<td>src.mdtf, 105</td>
<td></td>
</tr>
<tr>
<td>src.shared_diagnostic, 111</td>
<td></td>
</tr>
<tr>
<td>src.util, 115</td>
<td></td>
</tr>
<tr>
<td>MultiMap (class in src.util), 116</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Q</td>
</tr>
<tr>
<td>name_in_model() (src.data_manager.LocalfileDataManager.DataKey property), 108</td>
<td></td>
</tr>
<tr>
<td>NameSpace (class in src.util), 116</td>
<td></td>
</tr>
<tr>
<td>NoneEnvironmentManager (class in src.environment_manager), 109</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>S</td>
</tr>
<tr>
<td>parse_case() (src.mdtf.MDTFFramework method), 105</td>
<td>set_case_pod_list() (src.mdtf.MDTFFramework method), 105</td>
</tr>
<tr>
<td>parse_case_list() (src.mdtf.MDTFFramework method), 105</td>
<td></td>
</tr>
<tr>
<td>parse_env_vars() (src.mdtf.MDTFFramework method), 105</td>
<td></td>
</tr>
<tr>
<td>parse_json() (in module src.util), 117</td>
<td></td>
</tr>
<tr>
<td>parse_mdtf_args() (src.mdtf.MDTFFramework method), 105</td>
<td></td>
</tr>
<tr>
<td>parse_paths() (src.mdtf.MDTFFramework method), 106</td>
<td></td>
</tr>
<tr>
<td>parse_pod_list() (src.mdtf.MDTFFramework method), 105</td>
<td></td>
</tr>
<tr>
<td>plan_data_fetch_hook() (src.data_manager.DataManager method), 108</td>
<td></td>
</tr>
<tr>
<td>PodRequirementFailure, 111</td>
<td></td>
</tr>
<tr>
<td>poll_command() (in module src.util), 118</td>
<td></td>
</tr>
<tr>
<td>preprocess_local_data() (src.data_manager.DataManager method), 108</td>
<td></td>
</tr>
<tr>
<td>pretty_print_json() (in module src.util), 117</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>read_json() (in module src.util), 117</td>
<td>set_pod_env() (src.environment_manager.CondaEnvironmentManager method), 110</td>
</tr>
<tr>
<td>recursive_copy() (in module src.util), 118</td>
<td></td>
</tr>
<tr>
<td>remote_data_list() (src.data_manager.DataManager method), 107</td>
<td></td>
</tr>
<tr>
<td>resolve_path() (in module src.util), 118</td>
<td></td>
</tr>
<tr>
<td>run() (src.environment_manager.EnvironmentManager method), 109</td>
<td></td>
</tr>
<tr>
<td>run() (src.util.ExceptionPropagatingThread method), 115</td>
<td></td>
</tr>
<tr>
<td>run_commands() (src.shared_diagnostic.Diagnostic method), 113</td>
<td></td>
</tr>
<tr>
<td>run_shell_command() (in module src.util), 119</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>set_up() (src.data_manager.DataManager method), 107</td>
<td></td>
</tr>
<tr>
<td>set_up() (src.environment_manager.EnvironmentManager method), 109</td>
<td></td>
</tr>
<tr>
<td>set_up() (src.shared_diagnostic.Diagnostic method), 112</td>
<td></td>
</tr>
<tr>
<td>signal_logger() (in module src.util), 120</td>
<td></td>
</tr>
<tr>
<td>Singleton (class in src.util), 115</td>
<td></td>
</tr>
<tr>
<td>spawn_subprocess() (src.environment_manager.EnvironmentManager method), 109</td>
<td></td>
</tr>
<tr>
<td>src.data_manager module, 106</td>
<td></td>
</tr>
<tr>
<td>src.environment_manager</td>
<td></td>
</tr>
</tbody>
</table>
module, 109
src.mdtf
    module, 105
src.shared_diagnostic
    module, 111
src.util
    module, 115
strip_comments() (in module src.util), 117
subprocess_cleanup()
    (src.environment_manager.EnvironmentManager
    method), 109

t
tearDown() (src.data_manager.DataManager
    method), 108
tearDown() (src.environment_manager.EnvironmentManager
    method), 109
tearDown() (src.shared_diagnostic.Diagnostic
    method), 113
TimeoutAlarm, 119
to_dict() (src.util.MultiMap method), 116
toDict() (src.util.NameSpace method), 116

v
validate_commands()
    (src.shared_diagnostic.Diagnostic method), 113
verify_paths() (src.mdtf.MDTFFramework
    method), 106
verify_pod_links()
    (src.shared_diagnostic.Diagnostic method), 114
VirtualenvEnvironmentManager (class in
    src.environment_manager), 110

w
write_json() (in module src.util), 117