v2.0 document: Developers_walkthrough_v2.0.pdf

Developers' walkthrough for the MDTF Diagnostic Framework

Yi-Hung Kuo (a), Dani Coleman (b), Chih-Chieh Jack Chen (b), Andrew Gettelman (b), J. David Neelin (a), Eric Maloney (c).
(a: UCLA; b: NCAR; c: CSU)
Last update: XXX

This document describes the operation of the MDTF framework from the point of view of a diagnostic developer, and summarizes steps that a developer should take to enable their analysis script to work within the MDTF framework. Full reference documentation for developers is provided at https://mdtf-diagnostics.readthedocs.io/ , and we only mention the most relevant information here.

Instructions for obtaining, installing and running the MDTF framework are covered in the "Getting Started" document. The scientific content and purpose of contributed diagnostic scripts is also outside the scope of this document; see Maloney et al., Process-Oriented Evaluation of Climate and Weather Forecasting Models, BAMS 100(9), 1665-1686 (2019), doi.org/10.1175/BAMS-D-18-0042.1.

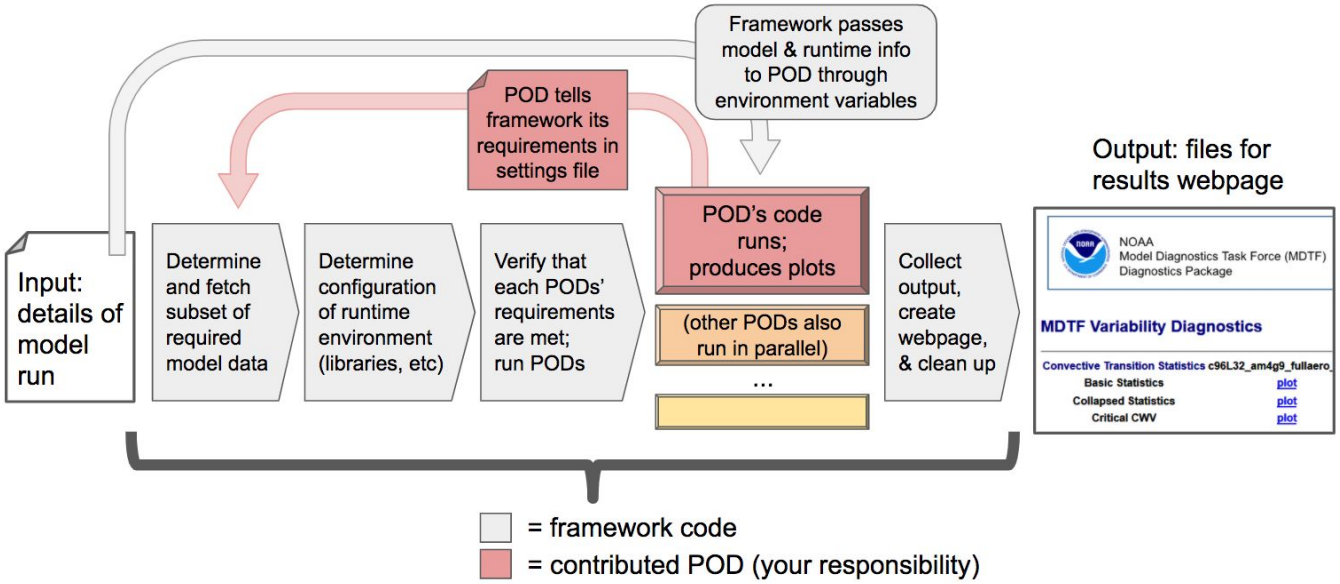
TODO: summarize remaining sections

1. Framework overview

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.

The MDTF Diagnostic Framework consists of multiple Process-Oriented Diagnostic (POD) modules, each of which is developed by an individual research group. For clarity, the framework is the structure provided by the Model Diagnostics Task Force, and the PODs (or modules) are developed by individual groups (or developers). PODs are developed and run independently of each other. 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.)



As summarized in the figure above, the changes needed to convert an existing analysis script for use in the framework are:

1) 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.
2) 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).
3) Provide a template web page which links to, and briefly describes, the plots generated by the script.

Each of these are described in more detail below.

1. Get started with development

The first step is to have a tested analysis script to be added to the framework. You may be adapting existing code you used in one of your publications; if you are developing a POD from scratch, we recommend that you focus on that first (using output from a model you're familiar with) and adapt it for the framework once it's been tested and debugged, rather than try to do two things at once.

1.1 Get started with open-source community development

The next step is to install the MDTF framework locally. General instructions for doing this are provided in the Getting Started document; however we request that developers use git and

github to obtain the framework and contribute their POD's code, following standard open source development practices. git is an open-source tool used in all aspects of modern software development, allowing large code projects to be developed by many contributors independently (eg, as with the Linux kernel). Github is a website providing free hosting to projects that use git. Because of the wide adoption of git and github, questions not specific to the MDTF framework can usually be quickly answered through google.

https://mdtf-diagnostics.readthedocs.io/en/latest/sphinx/dev_git_intro.html .

(1) Register for an account at github.com.
(2) Create a fork of the project by clicking the button in the upper-right corner of the framework's Github page (https://github.com/NOAA-GFDL/MDTF-diagnostics). This will create a copy of the framework in your own account which you have full control over.
(3) Download the framework code to your computer through, eg, `git clone https://github.com/`<your account name>`/MDTF-diagnostics.git`
(4) You need to tell your local installation about NOAA's repo if you wish to contribute changes back to the code base. To do this, change to the "MDTF-diagnostics" directory and type `git remote add upstream https://github.com/NOAA-GFDL/MDTF-diagnostics.git`. Now you have two remote repos: `origin`, your Github fork which you can read and write to, and `upstream`, NOAA's code base which you can only read from.
(5) Start from the framework's "develop" branch: `git checkout develop`.
(6) 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 copy these changes from your local copy to your github account with `git push origin develop`.
(7) 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.
(8) Download observational and sample model data and configure the framework as described in the Getting Started document. Ensure that the framework operates correctly by verifying the operation of existing PODs on the sample model data.

1.2 Development considerations

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

(1) Scope of the analysis your POD conducts.
    (a) See the BAMS article referenced above for the project's scientific goals and what is meant by a "process oriented diagnostic." We encourage PODs to have a specific, focused scope.

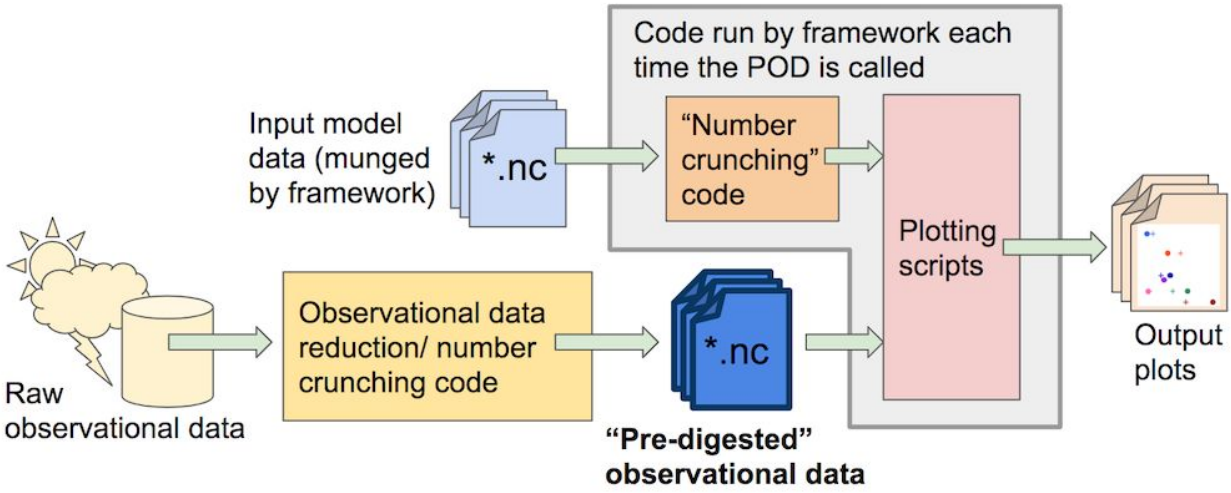
(b) PODs should be relatively lightweight in terms of computation and memory requirements (eg, run time measured in minutes, not hours): this is to enable rapid feedback and iteration cycles to assist users in model development.
(c) Bear in mind that your POD may be run on model output of potentially any date range and spatial resolution. Your POD should not require strong assumptions about these quantities, or other details of the model's operation.

(2) Choice of language(s):
(a) In order to accomplish our goal of portability, the MDTF cannot accept PODs written in closed-source languages (eg MATLAB; depending on your use case, it may be feasible to port MATLAB code to Octave).
(b) We also do not accept PODs written in compiled languages (C or Fortran): installation would rapidly become impractical if the user had to check compilation options for each POD. See below for options if your POD requires the performance of a compiled language.
(c) We require that PODs that are funded through the CPO grant be developed in Python, specifically Python >= 3.6 (official support for Python 2 was discontinued as of January 2020). While the framework is able to call PODs written in any scripting language, Python support will be "first among equals" in terms of priority for allocating developer resources, etc.
(d) At the same time, if your POD development is not being funded, we recommend against rewriting existing analysis scripts in Python. Doing so is likely to introduce new bugs into stable code, especially if you're unfamiliar with the language.

(3) Scope of your POD's code.
(a) As described above, your POD should accept model data as input and express the results of its analysis in a series of figures, which are presented to the user in a web page.
(b) 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. Because your POD may be run on, essentially, any model, you should be careful about the assumptions your code makes about the layout of these files. Supporting data may be in any format and will not be modified by the framework.
(c) The above data sources are your POD's only input: you may provide options in the settings file for the user to configure when the POD is installed, but these cannot be changed each time the POD is run. Furthermore, your POD should not access the internet or other networked resources.
(d) The output of your POD should be a series of figures in vector format (.eps or .ps), written to a specific working directory (described below). Optionally, we encourage POD developers to also save relevant output data (eg, the output data being plotted) as netcdf files, to give users the ability to take the POD's output and perform further analysis on it.

(4) Observational and supporting data; code organization.
(a) In order to make your code run faster for the users, we request that you separate any calculations that don't depend on the model data (eg. pre-processing of

observational data), and instead save the end result of these calculations in data files for your POD to read when it is run. We refer to this as “pre-digested observational data,” but it refers to any quantities that are independent of the model being analyzed.



(b) 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 “pre-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).
(c) Pre-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.
(d) In order to keep the amount of supporting data needed by the framework manageable, we request that you limit the total amount of pre-digested data you supply to no more than a few gigabytes.
(e) TODO: insert ref & explanation for PCMDIobs

2. Walkthrough of framework operation

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

2.1 Framework invocation

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

See section 3 of the Getting Started for more details on how the package is called. See https://github.com/NOAA-GFDL/MDTF-diagnostics/wiki/Command-line-reference for documentation on command line options (or run `mdtf --help`).

2.2 Data request

Each POD describes the model data it requires as input in the "varlist" section of its settings file (described in https://mdtf-diagnostics.readthedocs.io/en/latest/sphinx/dev_settings_quick.html#varlist-section and in full detail at https://mdtf-diagnostics.readthedocs.io/en/latest/sphinx/dev_settings_format.html#varlist-section .)
Each entry in the varlist section corresponds to one model data file used by the POD.

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

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

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

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

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

2.2.1 Example diagnostic

The example diagnostic uses only one model variable: surface air temperature, recorded at monthly frequency.

2.3 Runtime environment configuration

In the first section of your POD's settings file, we request that you provide a list of programs your POD uses to run (names of interpreters for scripting languages, as well as any utility programs) and any third-party libraries they use (see https://mdtf-diagnostics.readthedocs.io/en/latest/sphinx/dev_settings_quick.html#settings-section).

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

2.3.1 Example diagnostic

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

2.4 POD execution

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

All information is passed from the framework to your POD in the form of unix shell environment variables; see https://mdtf-diagnostics.readthedocs.io/en/latest/sphinx/dev_envvars.html .

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

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

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

2.4.1 POD execution: paths

Recall that installing the code will create a directory titled “MDTF-diagnostics” containing the files listed on the github page. Below we refer to this MDTF-diagnostics directory as $CODE_ROOT. It contains the following subdirectories:

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

The most important environment variables set by the framework describe the location of resources your POD needs. To achieve the design goal of portability, you should ensure that no paths are hard-coded in your POD, for any reason. Instead, they should reference one of the following variable names:

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

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

2.4.2 Example diagnostic

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

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

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

2.5 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 formula:

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

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

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

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

3. Development checklist

The following are the necessary 6 steps for the module implementation and integration into the framework. Here the POD name tag is convective_transition_diag[1]. It is to be replicated with different POD name tags. All the modules currently included in the code package have the same structure, and hence the descriptions below apply:

1. Provide all the scripts for the convective_transition_diag POD in the sub-directory DIAG_HOME/var_code/convective_transition_diag. Among the provided scripts, there should be a template html file convective_transition_diag.html[2], and a main script convective_transition_diag.py that calls the other scripts in the same sub-directory for analyzing, plotting, and finalizing html.

2. Provide all the pre-digested observation data/figures in the sub-directory DATA_IN/obs_data/convective_transition_diag.

5. Provide documentation following the templates:

1) Provide a comprehensive POD documentation, including a one-paragraph synopsis of the POD, developers' contact information, required programming language and libraries, and model output variables, a brief summary of the presented diagnostics as well as references in which more in-depth discussions can be found (see an example).

2) All scripts should be self-documenting by including in-line comments. The main script convective_transition_diag.py should contain a comprehensive header providing information that contains the same items as in the POD documentation, except for the "More about this diagnostic" section.

3) The one-paragraph POD synopsis (in the POD documentation) as well as a link to the Full Documentation should be placed at the top of the template convective_transition_diag.html (see example).

6. Test before distribution. It is important that developers test their POD before sending it to the MDTF contact. Please take the time to go through the following procedures:

---

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

[2] One can create a new html template by simply copying and modifying the example templates in DIAG_HOME/var_code/html/html_template_examples. Note that scripts therein are exact replications of the html-related scripts in the example PODs, serving merely as a reference, and are not called by mdtf.py.

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

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

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

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

4. Summary of things to be aware of

Here is a short list of tips on implementation:

I. Structure of the code package – Implementing the constituent PODs in accordance with the structure described in sections 2 and 3 makes it easy to pass the package (or just part of it) to other groups.

II. Robustness to model file/variable names – Each POD should be robust to modest changes in the file/variable names of the model output; see section 5 regarding the model output filename structure, and section 6 regarding using the environment variables and robustness tests. Also, it would be easier to apply the code package to a broader range of model output.

III. Save intermediate output – Can be used, e.g. to save time when there is a substantial computation that can be re-used when re-running or re-plotting diagnostics. See section 3.I regarding where to save the output.

IV. Self-documenting – For maintenance and adaptation, to provide references on the scientific underpinnings, and for the code package to work out of the box without support. See step 5 in section 2.

V. 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.

VI. Basic vs. advanced diagnostics (within a POD) – Separate parts of diagnostics, e.g, those might need adjustment when model performance out of obs range.

VII. Avoid special characters (!@#$%^&*) in file/script name

Appendix 1: Environment Variables used by MDTF code package with their default settings

See https://mdtf-diagnostics.readthedocs.io/en/latest/sphinx/dev_envvars.html.

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.

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.

Model run information

- CASENAME: User-provided label describing the run of model data being analyzed.
- FIRSTYR, LASTYR: Four-digit years describing the analysis period.

Locations of model data files

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

Names of variables and dimensions

These are set depending on the data your diagnostic requests in its settings file. 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.

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.

Conversion from v2.0 environment variables

The paths referred to by environment variables have been changed to be specific to your POD. The variables themselves have been renamed to avoid confusion.

Path description	v2.0 environment variable expression	Equivalent new variable
Top-level code repository	$DIAG_HOME	No variable set: PODs should not access files outside of their own source code directory at $POD_HOME
POD's source code	$VARCODE/<pod name>	$POD_HOME
POD's observational/supporting data	$VARDATA/<pod name>	$OBS_DATA
POD's working directory	$variab_dir/<pod name>	$WK_DIR
Path to requested netcdf data file for <variable name> at date frequency <freq>	$CASENAME/<freq>/$CASENAME.<variable name>.<freq>.nc	