aospy Documentation

Release 0.0

Spencer Hill, Spencer Clark

Feb 03, 2017

Contents

aospy is an open source Python package for automating computations that use gridded climate data (namely data stored as netCDF files) and the management of the results of those computations.

Once a user describes where their data is stored on disk using aospy's built-in tools, they can subsequently use the provided main script at any time to fire off calculations to be performed in parallel using the permutation of an arbitrary number of climate models, simulations, variables to be computed, date ranges, sub-annual-sampling, and many other parameters. Their results get saved in a highly structured directory format as netCDF files.

The eventual goal is for aospy to become the "industry standard" for gridded climate data analysis and, in so doing, accelerate progress in climate science and make the results of climate research more easily reproducible and shareable. aospy relies heavily on the xarray package.

CHAPTER 1

Documentation

1.1 What's New

1.1.1 v0.1.1

This release includes fixes for a number of bugs mistakenly introduced in the refactoring of the variable loading step of `calc.py` (PR90), as well as support for xarray version 0.9.1.

Ehancements

- Support for xarray version 0.9.1 and require it or a later xarray version. By Spencer Clark and Spencer Hill.
- Better support for variable names relating to "bounds" dimension of input data files. "bnds", "bounds", and "nv" now all supported (PR140). By Spencer Hill.
- When coercing dims of input data to aospy's internal names, for scalars change only the name; for non-scalars change the name, force them to have a coord, and copy over their attrs (PR140). By Spencer Hill.

Bug fixes

- Fix bug involving loading data that has dims that lack coords (which is possible as of xarray v0.9.0). By Spencer Hill.
- Fix an instance where the name for pressure half levels was mistakenly replaced with the name for the pressure full levels (PR126). By Spencer Clark.
- Prevent workaround for dates outside the `pd.Timestamp` valid range from being applied to dates within the `pd.Timestamp` valid range (PR128). By Spencer Clark.
- Ensure that all DataArrays associated with `aospy.Var` objects have a time weights coordinate with CF-compliant time units. This allows them to be cast as the type `np.timedelta64`, and be safely converted to have units of days before taking time-weighted averages (PR128). By Spencer Clark.
- Fix a bug where the time weights were not subset in time prior to taking a time weighted average; this caused computed seasonal averages to be too small. To prevent this from failing silently again, we now raise a `ValueError` if the time coordinate of the time weights is not identical to the time coordinate of the array associated with the `aospy.Var` (PR128). By Spencer Clark.
- Enable calculations to be completed using data saved as a single time-slice on disk (fixes GH132 through PR135). By Spencer Clark.

- Fix bug where workaround for dates outside the `pd.Timestamp` valid range caused a mismatch between the data loaded and the data requested (fixes GH138 through PR139). By Spencer Clark.

1.1.2 v0.1 (24 January 2017)

- Initial release!
- Contributors:
  - Spencer Hill
  - Spencer Clark

1.2 Overview: Why aospy?

1.2.1 Motivations

Climate models generally output a wide array of useful quantities, but almost invariably not all needed quantities are directly outputted. Even for those that are, further slicing and dicing in time and/or space are required. Moreover, these multiple computations and spatiotemporal reductions are needed for not just a single simulation but across multiple models, simulations, time durations, subsets of the annual cycle, and so on.

Performing these computations across all of the desired parameter combinations quickly becomes impractical without some automation. But even once some automation is in place, the resulting plethora of data quickly becomes unusable unless it is easily found and imbued with sufficient metadata to describe precisely what it is and how it was computed.

1.2.2 What aospy does

aospy provides functionality that enables users to perform commonly needed tasks in climate, weather, and related sciences as:

- Repeating a calculation across multiple simulations in a single climate model
- Repeating a calculation across the same simulations performed in multiple models, even if variable names or other quantities differ among the models or simulations
- Repeating a calculation across multiple timespans, both in terms of the start date and end date and in terms of sub-annual sampling: e.g. annual mean, seasonal-means, monthly means.
- Computing multiple statistical (e.g. mean, standard deviation) and physical (e.g. column integrals or averages; zonal integrals or averages) reductions on any given computation, both on a gridpoint-by-gridpoint basis and over an arbitrary number of geographical regions
- Any combination of the above, plus many more!

1.2.3 Design philosophy

Key to enabling this automation and handling of model- and simulation-level idiosyncrasies is separating

1. Code that describes the data that you want to work with
2. Code that specifies any physical calculations you eventually want to perform
3. Code that specifies the set of computations the user wishes to perform at a given time

For (1), the user defines objects at three distinct levels: *Proj*, *Model*, and *Run*, that specify where the data is located that you want to work with. For (2), the user defines *Var* objects that describe the physical quantities to be computed, including any functions that transform one or more directly model-outputted quantities into the ultimately desired quantity, as well as *Region* objects that describe any geographical regions over which to perform averages. Once these objects have been defined, the user can proceed with (3) via a simple script that specifies any models, simulations, physical quantities, etc. to be performed.

The run script can be modified and re-submitted as further calculations are desired. Similarly, new objects can be defined at any time describing new simulations, models, or variables. More detailed instructions are available in the "Using aospy" section.

1.2.4 Open Science & Reproducible Research

aospy promotes open science and reproducible research in multiple ways:

- By separating code that describes where your data is from the code used to compute particular physical quantities, the latter can be written in a generic form that closely mimics the physical form of the particular expression. The resulting code is easier to read and debut and therefore to share with others.
- By enabling automation of calculations across an arbitrary number of parameter combinations, aospy facilitates more rigorous analyses and/or analyses encompassing a larger span of input data than would otherwise be possible.
- By outputting the results of calculations as netCDF files in a highly organized directory structure with lots of metadata embued within the file path, file name, and the netCDF file itself, aospy facilitates the sharing of data with others.

  It also enhances the usability of one's own data, providing a remedy to the familiar refrain among scientists along the lines of "What is this data1.txt file that was created six months ago? Better just delete it and re-do the calculations to be sure.")

1.3 Using aospy

This section provides a high-level summary of how to use aospy. See the Examples section and associated Jupyter Notebook for concrete examples.

Note: There is a non-trivial amount of effort required (mainly in creating and populating your object library, described below) before you will be able to perform any calculations. However, once the object library is in place, there is essentially no limit to the number of calculations that can be performed on your data either together at the same time or at different times. In other words, the spinup time should be well worth it.

1.3.1 Your aospy object library

The first step is writing the code that describes your data and the quantities you eventually want to compute using it. We refer to this code collectively as your "object library".

Describing your data on disk

aospy needs to know where the data you want to use is located on disk and how it is organized across different simulations, models, and projects. This involves a hierarchy of three classes, `Proj`, `Model`, and `Run`.

1. `Proj`: This represents a single project that involves analysis of data from one or more models and simulations.
2. `Model`: This represents a single climate model, other numerical model, observational data source, etc.
3. `Run`: This represents a single simulation, version of observational data, etc.

So each user's object library will contain one or more `Proj` objects, each of which will have one or more child `Model` objects, which in turn will each have one or more child `Run` objects.

Note: Currently, the Proj-Model-Run hierarchy is rigid, in that each Run has a parent Model, and each Model has a parent Proj. For small projects, this can lead to a lot of boilerplate code. Work is ongoing to relax this constraint to facilitate easier exploratory analysis.

Physical variables

The `Var` class is used to represent physical variables, e.g. precipitation rate or potential temperature. This includes both variables which are directly available in netCDF files (e.g. they were directly outputted by your climate model) as well as those fields that must be computed from other variables (e.g. they weren't directly outputted but can be computed from other variables that were outputted).

Geographical regions

The `Region` class is used to define geographical regions over which quantities can be averaged (in addition to gridpoint-by-gridpoint values). Like `Var` objects, they are more generic than the objects of the `Proj` - `Model` - `Run` hierarchy, in that they correspond to the generic physical quantities/regions rather than the data of a particular project, model, or simulation.

1.3.2 Configuring your object library

Required components

In order for your object library to work with the main script, it must include the following two objects:

1. `projs` : A container of `Proj` objects
2. `variables` : A container of `Var` objects

(The `Model`, `Run`, and `Region` objects are all included within their parent `Proj` objects and thus don't require analogous top-level containers.)

These must be accessible from the object library's toplevel namespace, i.e. the Python commands `import my_obj_lib.projs` and `import my_obj_lib.variables` must work, where `my_obj_lib` is the name you've given to your library. Which leads to the next topic: how to structure your object library within one or more `.py` files.

File/directory structure

The simplest way to structure your object library is to define everything in a single module (i.e. a single `.py` file). This works great for small projects and for initially trying out aospy.

As an object library grows, however, it can become desirable to split it into multiple `.py` files. This effectively changes it from a module to a proper Python package. Python packages require a specific directory structure and specification

of things to include at each level via `__init__.py` files. See the official documentation on packages for further guidance.

For an example of a large object library that is structured as a proper package, see here.

Making your object library visible to Python

Whether it is structured as a single module or as a proper package, you'll likely have to add the directory containing your object library to the `PYTHONPATH` environment variable in order for Python to be able to import it:

```
export PYTHONPATH=/path/to/your/object/library:${PYTHONPATH}
```

Of course, replace `/path/to/your/object/library` with the actual path to yours. This command places your object library at the front of the `PYTHONPATH` environment variable, which is essentially the first place where Python looks to find packages and modules to be imported.

Note: It's convenient to copy this command into your shell profile (e.g., for the bash shell on Linux or Mac, `~/.bash_profile`) so that you don't have to call it again in every new terminal session.

Note: For object libraries structured as packages, it is also possible to properly install your object library by creating a properly set-up `setup.py` file and `python setup.py install`. But unless you're prevented from modifying `PYTHONPATH` for some reason, there's no advantage of this versus the simpler `PYTHONPATH` alternative above.

Once this has been done, you should be able to import your object library from within Python via `import my_obj_lib`, where `my_obj_lib` is the name you've given to your library. You will not be able to use the main script until this works.

1.3.3 Executing calculations

The main script contents

Calculations are performed by specifying in a "main script" the desired parameters and then running the script.

We provide a template main script within aospy. You should copy it to the location of your choice and in the copy replace the given names with the names of your own project, model, etc. objects that you want to perform computations on. (If you accidentally change the original, you can always get a fresh copy from Github.)

Except where noted otherwise in the template script's comments, all parameters should be submitted as lists, even if they are a single element. E.g. `models = ['name-of-my-model']`.

Note: Although the main script is the recommended way to perform calculations, it's possible to submit calculations by other means. For example, one could explicitly create `Calc` objects and call their `compute` method, as is done in the example Jupyter notebook.

Running the main script

Once the main script parameters are all modified as desired, execute the script from the command line as follows

```
/path/to/your/main.py
```

This should generate a text summary of the specified parameters and a prompt as to whether to proceed or not with the calculations. An affirmative response then triggers the calculations to execute.

Specifically, the parameters are permuted over all possible combinations. So, for example, if two model names and three variable names were listed and all other parameters had only one element, six calculations would be generated and executed. There is no limit to the number of permutations.

Note: You can also call the main script interactively within an IPython session via `%run /path/to/your/main.py` or, from the command line, run the script and then start an interactive IPython session via `ipython -i /path/to/your/main.py`.

As the calculations are performed, logging information will be printed to the terminal displaying their progress.

Parallelized calculations

The calculations generated by the main script can be executed in parallel provided the optional dependency `multiprocess` is installed. (It is available via pip: `pip install multiprocess`.) Otherwise, or if the user sets `parallelize` to `False` in the main script, the calculations will be executed one-by-one.

Particularly on instititutional clusters with many cores, this parallelization yields an impressive speed-up when multiple calculations are generated.

Note: When calculations are performed in parallel, often the logging information from different calculations running simultameously end up interwoven with one another, leading to output that is confusing to follow. Work is ongoing to improve the logging output when the computations are parallelized.

Finding the output

aospy saves the results of all calculations as netCDF files and embeds metadata describing it within the netCDF files, in their filenames, and in the directory structure within which they are saved.

- Directory structure: `/path/to/aospy-rootdir/projname/modelname/runname/varname`
- File name : `varname.intvl_out.dtype_out_time.'from_'intvl_in'_'dtype_in_time.model.run.date`

See the API reference documentation of `CalcInterface` for explanation of each of these components of the path and file name.

Under the hood

The main script encodes each permutation of the input parameters into a `CalcInterface` object. This object, in turn, is used to instantiate a `Calc` object. The `Calc` object, in turn, performs the calculation.

Unlike `Proj`, `Model`, `Run`, `Var`, and `Region`, these objects are not intended to be saved in `.py` files for continual re-use. Instead, they are generated as needed, perform their desired tasks, and then go away.

See the API reference documentation for further details.

1.4 Examples

aospy comes with some sample data files, which can be used to illustrate some of its basic features. These files contain monthly mean time series output of two variables (the large-scale and convective components of the total precipitation rate) from an idealized aquaplanet climate model. A simple computation one could seek to do from this model output would be to compute some statistics of the total precipitation rate (large-scale plus convective).

Here's a quick summary of the included data:

```
In [1]: import xarray as xr

In [2]: xr.open_mfdataset('../aospy/test/data/netcdf/000[4-6]0101.precip_monthly.nc',
   ...:                   decode_times=False)
   ...:
Out[2]:
<xarray.Dataset>
Dimensions:             (lat: 64, latb: 65, lon: 128, lonb: 129, nv: 2, time: 36)
Coordinates:
  * lonb                (lonb) float64 -1.406 1.406 4.219 7.031 9.844 12.66 ...
  * lon                 (lon) float64 0.0 2.812 5.625 8.438 11.25 14.06 16.88 ...
  * latb                (latb) float64 -90.0 -86.58 -83.76 -80.96 -78.16 ...
  * lat                 (lat) float64 -87.86 -85.1 -82.31 -79.53 -76.74 ...
  * nv                  (nv) float64 1.0 2.0
  * time                (time) float64 1.111e+03 1.139e+03 1.17e+03 1.2e+03 ...
Data variables:
    time_bounds         (time, nv) float64 1.095e+03 1.126e+03 1.126e+03 ...
    condensation_rain   (time, lat, lon) float64 5.768e-06 5.784e-06 ...
    convection_rain     (time, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    average_DT          (time) float64 31.0 28.0 31.0 30.0 31.0 30.0 31.0 ...
```

In this particular model, the large-scale component of the precipitation rate is called "condensation_rain" and the convective component is called "convection_rain."

1.4.1 Defining the simulation metadata

The first step is to create an `aospy.Run` object that stores metadata about this simulation. This includes giving it a name, a description, and specifying where files are located through a DataLoader.

```
In [3]: from datetime import datetime

In [4]: from aospy import Run

In [5]: from aospy.data_loader import DictDataLoader

In [6]: file_map = {'monthly': '../aospy/test/data/netcdf/000[4-6]0101.precip_monthly.nc'}

In [7]: example_run = Run(
   ...:     name='example_run',
   ...:     description=(
   ...:         'Control simulation of the idealized moist model'
   ...:     ),
   ...:     data_loader=DictDataLoader(file_map)
   ...: )
   ...:
```

We then need to associate this `Run` with an `aospy.Model` object:

```
In [8]: from aospy import Model

In [9]: example_model = Model(
   ...:     name='example_model',
   ...:     grid_file_paths=(
   ...:         '../aospy/test/data/netcdf/00040101.precip_monthly.nc',
   ...:         '../aospy/test/data/netcdf/im.landmask.nc'
   ...:     ),
   ...:     runs=[example_run]
   ...: )
   ...:
```

Finally, we need to associate the `Model` object with an `aospy.Proj` object. Here we can specify the location that aospy will save its output files.

```
In [10]: from aospy import Proj

In [11]: example_proj = Proj(
   ....:     'example_proj',
   ....:     direc_out='example-output',
   ....:     tar_direc_out='example-tar-output',
   ....:     models=(example_model,)
   ....: )
   ....:
```

Now the metadata associated with this simulation is fully defined. We can move on to computing the total precipitation.

1.4.2 Computing the annual mean total precipitation rate

We can start by defining a simple python function that computes the total precipitation from condensation and convection rain arguments:

```
In [12]: def total_precipitation(condensation_rain, convection_rain):
   ....:     return condensation_rain + convection_rain
   ....:
```

To hook this function into the aospy framework, we need to connect it to an `aospy.Var` object, as well as define the `Var` objects it depends on (variables that are natively stored in model output files).

```
In [13]: from aospy import Var

In [14]: condensation_rain = Var(
   ....:     name='condensation_rain',
   ....:     alt_names=('prec_ls',),
   ....:     def_time=True,
   ....:     description=('condensation rain'),
   ....: )
   ....:

In [15]: convection_rain = Var(
   ....:     name='convection_rain',
   ....:     alt_names=('prec_conv',),
   ....:     def_time=True,
   ....:     description=('convection rain'),
   ....: )
   ....:

In [16]: precip = Var(
```

```
   ....:     name='total_precipitation',
   ....:     def_time=True,
   ....:     description=('total precipitation rate'),
   ....:     func=total_precipitation,
   ....:     variables=(condensation_rain, convection_rain)
   ....: )
   ....:
```

Here the func attribute of the precip `Var` object is the function we defined, and the variables attribute is a tuple containing the `Var` objects the function depends on, in the order of the function's call signature.

If we'd like to compute the time-mean total precipitation rate from year four to year six using aospy, we can create an `aospy.Calc` object. This is currently done through passing an `aospy.CalcInterface` object to a `Calc` object; once created, the computation can be submitted by simply calling the compute function of `Calc`.

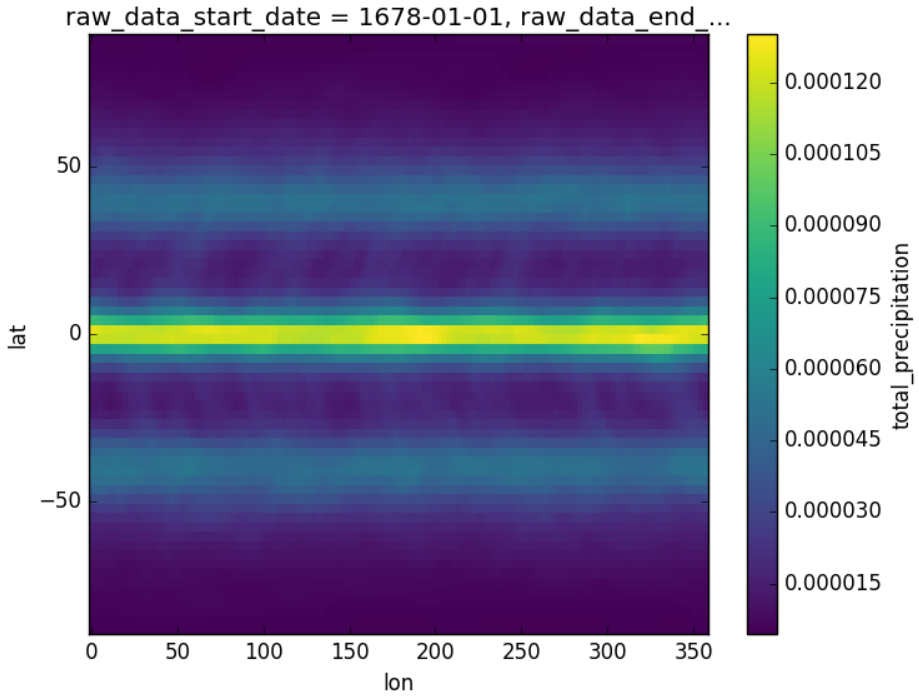
```
In [17]: from aospy import CalcInterface, Calc

In [18]: calc_int = CalcInterface(
   ....:     proj=example_proj,
   ....:     model=example_model,
   ....:     run=example_run,
   ....:     var=precip,
   ....:     date_range=(datetime(4, 1, 1), datetime(6, 12, 31)),
   ....:     intvl_in='monthly',
   ....:     dtype_in_time='ts',
   ....:     intvl_out='ann',
   ....:     dtype_out_time='av'
   ....: )
   ....:

In [19]: Calc(calc_int).compute()
```

The result is stored in a netcdf file, whose path and filename contains metadata about where it came from:

```
In [20]: calc_int.path_out['av']
Out[20]: 'example-output/example_proj/example_model/example_run/total_precipitation/total_precipitat
```

Using xarray we can open and plot the results of the calculation:

```
In [21]: xr.open_dataset(calc_int.path_out['av']).total_precipitation.plot()
Out[21]: <matplotlib.collections.QuadMesh at 0x7f2b31e2e1d0>
```



1.4.3 Computing the global annual mean total precipitation rate

Not only does aospy enable reductions along the time dimension, it also enables area weighted regional averages. As a simple introduction, we'll show how to compute the global mean total precipitation rate from this `Run`. To do so, we'll make use of the infrastructure defined above, and also define an `aospy.Region` object:

```
In [22]: from aospy import Region

In [23]: globe = Region(
   ....:     name='globe',
   ....:     description='Entire globe',
   ....:     lat_bounds=(-90, 90),
   ....:     lon_bounds=(0, 360),
   ....:     do_land_mask=False
   ....: )
   ....:
```

To compute the global annual mean total precipitation rate, we can now create another `Calc` object:

```
In [24]: calc_int = CalcInterface(
   ....:     proj=example_proj,
   ....:     model=example_model,
   ....:     run=example_run,
   ....:     var=precip,
   ....:     date_range=(datetime(4, 1, 1), datetime(6, 12, 31)),
   ....:     intvl_in='monthly',
   ....:     dtype_in_time='ts',
   ....:     intvl_out='ann',
   ....:     dtype_out_time='reg.av',
   ....:     region={'globe': globe}
   ....: )
```

```
   ....:

In [25]: Calc(calc_int).compute()
```

This produces a new file, located in:

```
In [26]: calc_int.path_out['reg.av']
Out[26]: 'example-output/example_proj/example_model/example_run/total_precipitation/total_precipitat
```

We find that the global annual mean total precipitation rate for this run (converting to units of mm per day) is:

```
In [27]: xr.open_dataset(calc_int.path_out['reg.av']).globe * 86400.
Out[27]:
<xarray.DataArray 'globe' ()>
array(3.025191320965487)
Coordinates:
    raw_data_start_date  datetime64[ns] 1678-01-01
    raw_data_end_date    datetime64[ns] 1681-01-01
    subset_start_date    datetime64[ns] 1678-01-01
    subset_end_date      datetime64[ns] 1680-12-31
```

1.5 Installation

1.5.1 Supported platforms

- Operating system: Windows, MacOS/Mac OS X, and Linux
- Python: 2.7, 3.4, 3.5, and 3.6

Note: For users that are new to Python, we recommend installing the free Anaconda distribution of Python, which works on Mac, Linux, and Windows and both on normal computers and institutional clusters and doesn't require root permissions.

1.5.2 Recommended installation method: pip

aospy is available from the official Python Packaging Index (PyPI) via `pip`:

```
pip install aospy
```

Note: We are currently working on adding aospy to conda-forge such that it will be installable via conda (i.e. `conda install aospy -c conda-forge`).

1.5.3 Alternative method: clone from Github

You can also clone the Github repo

```
git clone https://www.github.com/spencerahill/aospy/aospy.git
cd aospy
python setup.py install
```

1.5.4 Verifying proper installation

Once installed via either method, you can run aospy's suite of tests using py.test. From the top-level directory of the aospy installation

```
pip install pytest  # if you don't have it already
py.test aospy
```

If this results in any error messages or test failures, something has gone wrong.

1.5.5 Troubleshooting

Please open an Issue on Github if you have problems with any of these installation methods.

1.6 API Reference

Here we provide the reference documentation for aospy's public API. If you are new to the package and/or just trying to get a feel for the overall workflow, you are better off starting in the main documentation sections.

Warning: aospy is under active development. While we strive to maintain backwards compatibility, it is likely that some breaking changes to the codebase will occur in the future as aospy is improved.

1.6.1 Core Hierarchy for Input Data

aospy provides three classes for specifying the location and characteristics of data saved on disk as netCDF files that the user wishes to use as input data for aospy calculations: `Proj`, `Model`, and `Run`.

Proj

class `aospy.proj.`**`Proj`**(*name*, *description=None*, *models=None*, *default_models='all'*, *regions=None*, *direc_out=''*, *tar_direc_out=''*)

An object that describes a single project that will use aospy.

This is the top-level class in the aospy hierarchy of data representations. It is meant to contain all of the *Model*, *Run*, and *Region* objects that are of relevance to a particular research project. (Any of these may be used by multiple *Proj* objects.)

The *Proj* class itself provides little functionality, but it is an important means of organizing a user's work across different projects. In particular, the output of all calculations using *aospy.Calc* are saved in a directory structure whose root is that of the *Proj* object specified for that calculation.

Attributes

name	(str) The run's name
description	(str) A description of the run
direc_out, tar_direc_out	(str) The paths to the root directories of, respectively, the standard and .tar versions of the output of aospy calculations saved to disk.
models	(dict) A dictionary with entries of the form `{model_obj.name: model_obj}`, for each of this `Proj`'s child model objects
default_models	(dict) The default model objects on which to perform calculations via *aospy.Calc* if not otherwise specified
regions	(dict) A dictionary with entries of the form `{regin_obj.name: region_obj}`, for each of this `Proj`'s child region objects

__init__(*name*, *description=None*, *models=None*, *default_models='all'*, *regions=None*, *direc_out=''*, *tar_direc_out=''*)

Parameters **name** : str

The project's name. This should be unique from that of any other *Proj* objects being used.

description : str, optional

A description of the model. This is not used internally by aospy; it is solely for the user's information.

regions : {None, sequence of aospy.Region objects}, optional

The desired regions over which to perform region-average calculations.

models : {None, sequence of aospy.Model objects}, optional

The child Model objects of this project.

default_models : {None, sequence of aospy.Run objects}, optional

The subset of this Model's runs over which to perform calculations by default.

direc_out, tar_direc_out : str

Path to the root directories of where, respectively, regular output and a .tar-version of the output will be saved to disk.

See also:

`aospy.Model`, `aospy.Region`, `aospy.Run`

Model

class `aospy.model.`**Model**(*name=None*, *description=None*, *proj=None*, *grid_file_paths=None*, *default_start_date=None*, *default_end_date=None*, *runs=None*, *default_runs=None*, *load_grid_data=False*)

An object that describes a single climate or weather model.

Each *Model* object is associated with a parent *Proj* object and also with one or more child *Run* objects.

If aospy is being used to work with non climate- or weather-model data, the *Model* object can be used e.g. to represent a gridded observational product, with its child *Run* objects representing different released versions of that dataset.

Attributes

name	(str) The model's name
description	(str) A description of the model
proj	({None, aospy.Proj}) The model's parent aospy.Proj object
runs	(dict) A dictionary with entries of the form `{run_obj.name: run_obj}`, for each of this model's child Run objects
default_runs	(dict) The default subset of chidl run objects on which to perform calculations via *aospy.Calc* with this model if not otherwise specified
grid_file_paths	(list) The paths to netCDF files stored on disk from which the model's coordinate data can be taken.
default_start_date, default_end_date	(datetime.datetime) The default start and end dates of any calculations using this Model

__init__(*name=None*, *description=None*, *proj=None*, *grid_file_paths=None*, *default_start_date=None*, *default_end_date=None*, *runs=None*, *default_runs=None*, *load_grid_data=False*)

Parameters **name** : str

The model's name. This must be unique from that of any other *Model* objects being used by the parent *Proj*.

description : str, optional

A description of the model. This is not used internally by aospy; it is solely for the user's information.

proj : {None, aospy.Proj}, optional

The parent Proj object. When the parent *Proj* object is instantiated with this Model included in its *models* attribute, this will be over-written with that *Proj* object.

grid_file_paths : {None, sequence of strings}, optional

The paths to netCDF files stored on disk from which the model's coordinate data can be taken.

default_start_date, default_end_date : {None, *datetime.datetime*}, optional

Default start and end dates of calculations to be performed using this Model.

runs : {None, sequence of aospy.Run objects}, optional

The child run objects of this Model

default_runs : {None, sequence of aospy.Run objects}, optional

The subset of this Model's runs over which to perform calculations by default.

load_grid_data : bool, optional (default False)

Whether or not to load the grid data specified by 'grid_file_paths' upon initilization

See also:

`aospy.DataLoader`, `aospy.Proj`, `aospy.Run`

set_grid_data()

Populate the attrs that hold grid data.

Run

class aospy.run.**Run**(*name=None*, *description=None*, *proj=None*, *default_start_date=None*, *default_end_date=None*, *data_loader=None*)

An object that describes a single model 'run' (i.e. simulation).

Each *Run* object is associated with a parent *Model* object. This parent attribute is not set by *Run* itself, however; it is set during the instantation of the parent *Model* object.

If aospy is being used to work with non climate-model data, the *Run* object can be used e.g. to represent different versions of a gridded observational data product, with the parent *Model* representing that data product more generally.

Attributes

name	(str) The run's name
description	(str) A description of the run
proj	({None, aospy.Proj}) The run's parent aospy.Proj object
default_start_date, default_end_date	(datetime.datetime) The default start and end dates of any calculations using this Run
data_loader	(aospy.DataLoader) The aospy.DataLoader object used to find data on disk corresponding to this Run object

__init__(*name=None*, *description=None*, *proj=None*, *default_start_date=None*, *default_end_date=None*, *data_loader=None*)

Instantiate a *Run* object.

Parameters **name** : str

The run's name. This must be unique from that of any other *Run* objects being used by the parent *Model*.

description : str, optional

A description of the model. This is not used internally by aospy; it is solely for the user's information.

proj : {None, aospy.Proj}, optional

The parent Proj object.

data_loader : aospy.DataLoader

The *DataLoader* object used to find the data on disk to be used as inputs for aospy calculations for this run.

default_start_date, default_end_date : datetime.datetime, optional

Default start and end dates of calculations to be performed using this Model.

See also:

aospy.DataLoader, aospy.Model

1.6.2 DataLoaders

Run objects rely on a helper "data loader" to specify how to find their underlying data that is saved on disk. This mapping of variables, time ranges, and potentially other parameters to the location of the corresponding data on disk can differ among modeling centers or even between different models at the same center.

Currently supported data loader types are `DictDataLoader`, `NestedDictDataLoader`, and `GFDLDataLoader` Each of these inherit from the abstract base `DataLoader` class.

class `aospy.data_loader.`**`DataLoader`**
A fundamental DataLoader object

`__init__``()`
x.__init__(...) initializes x; see help(type(x)) for signature

`load_variable`(*var=None*, *start_date=None*, *end_date=None*, *time_offset=None*, ***DataAttrs*)
Load a DataArray for requested variable and time range.

Automatically renames all grid attributes to match aospy conventions.

Parameters **var** : Var

aospy Var object

start_date : datetime.datetime

start date for interval

end_date : datetime.datetime

end date for interval

time_offset : dict

Option to add a time offset to the time coordinate to correct for incorrect metadata.

****DataAttrs**

Attributes needed to identify a unique set of files to load from

Returns **da** : DataArray

DataArray for the specified variable, date range, and interval in

class `aospy.data_loader.`**`DictDataLoader`**(*file_map=None*)
A DataLoader that uses a dict mapping lists of files to string tags

This is the simplest DataLoader; it is useful for instance if one is dealing with raw model history files, which tend to group all variables of a single output interval into single filesets. The intvl_in parameter is a string description of the time frequency of the data one is referencing (e.g. 'monthly', 'daily', '3-hourly'). In principle, one can give it any string value.

Parameters **file_map** : dict

A dict mapping an input interval to a list of files

Examples

Case of two sets of files, one with monthly average output, and one with 3-hourly output.

```
>>> file_map = {'monthly': '000[4-6]0101.atmos_month.nc',
...             '3hr': '000[4-6]0101.atmos_8xday.nc'}
>>> data_loader = DictDataLoader(file_map)
```

`__init__`(*file_map=None*)
Create a new DictDataLoader

class `aospy.data_loader.`**`NestedDictDataLoader`**(*file_map=None*)
DataLoader that uses a nested dictionary mapping to load files

This is the most flexible existing type of DataLoader; it allows for the specification of different sets of files for different variables. The intvl_in parameter is a string description of the time frequency of the data one is referencing (e.g. 'monthly', 'daily', '3-hourly'). In principle, one can give it any string value. The variable name can be any variable name in your aospy object library (including alternative names).

Parameters **file_map** : dict

A dict mapping intvl_in to dictionaries mapping Var objects to lists of files

Examples

Case of a set of monthly average files for large scale precipitation, and another monthly average set of files for convective precipitation.

```
>>> file_map = {'monthly': {'precl': '000[4-6]0101.precl.nc',
...                         'precc': '000[4-6]0101.precc.nc'}}
>>> data_loader = NestedDictDataLoader(file_map)
```

__init__ (*file_map=None*)
Create a new NestedDictDataLoader

class aospy.data_loader.**GFDLDataLoader**(*template=None*, *data_direc=None*, *data_dur=None*, *data_start_date=None*, *data_end_date=None*)

DataLoader for NOAA GFDL model output

This is an example of a domain-specific custom DataLoader, designed specifically for finding files output by the Geophysical Fluid Dynamics Laboratory's model history file post-processing tools.

Parameters **template** : GFDLDataLoader

Optional argument to specify a base GFDLDataLoader to inherit parameters from

data_direc : str

Root directory of data files

data_dur : int

Number of years included per post-processed file

data_start_date : datetime.datetime

Start date of data files

data_end_date : datetime.datetime

End date of data files

Examples

Case without a template to start from.

```
>>> base = GFDLDataLoader(data_direc='/archive/control/pp', data_dur=5,
...                       data_start_date=datetime(2000, 1, 1),
...                       data_end_date=datetime(2010, 12, 31))
```

Case with a starting template.

```
>>> data_loader = GFDLDataLoader(base, data_direc='/archive/2xCO2/pp')
```

__init__(*template=None*, *data_direc=None*, *data_dur=None*, *data_start_date=None*, *data_end_date=None*)
Create a new GFDLDataLoader

1.6.3 Variables and Regions

The `Var` and `Region` classes are used to represent, respectively, physical quantities the user wishes to be able to compute and geographical regions over which the user wishes to aggregate their calculations.

Whereas the `Proj` - `Model` - `Run` hierarchy is used to describe the data resulting from particular model simulations, `Var` and `Region` represent the properties of generic physical entities that do not depend on the underlying data.

Var

class `aospy.var.`**Var**(*name*, *alt_names=None*, *func=None*, *variables=None*, *func_input_dtype='DataArray'*, *units=''*, *plot_units=''*, *plot_units_conv=1*, *domain='atmos'*, *description=''*, *def_time=False*, *def_vert=False*, *def_lat=False*, *def_lon=False*, *math_str=False*, *colormap='RdBu_r'*, *valid_range=None*)
An object representing a physical quantity to be computed.

Attributes

name	(str) The variable's name
alt_names	(tuple of strings) All other names that the variable may be referred to in the input data
names	(tuple of strings) The combination of *name* and *alt_names*
description	(str) A description of the variable
func	(function) The function with which to compute the variable
variables	(sequence of aospy.Var objects) The variables passed to *func* to compute it
func_input_dtype	({'DataArray', 'Dataset', 'numpy'}) The datatype expected by *func* of its arguments
units	(aospy.units.Units object) The variable's physical units
domain	(str) The physical domain of the variable, e.g. 'atmos', 'ocean', or 'land'
def_time, def_vert, def_lat, def_lon	(bool) Whether the variable is defined, respectively, in time, vertically, in latitude, and in longitude
math_str	(str) The mathematical representation of the variable
colormap	(str) The name of the default colormap to be used in plots of this variable
valid_range	(length-2 tuple) The range of values outside which to flag as unphysical/erroneous

__init__(*name*, *alt_names=None*, *func=None*, *variables=None*, *func_input_dtype='DataArray'*, *units=''*, *plot_units=''*, *plot_units_conv=1*, *domain='atmos'*, *description=''*, *def_time=False*, *def_vert=False*, *def_lat=False*, *def_lon=False*, *math_str=False*, *colormap='RdBu_r'*, *valid_range=None*)
Instantiate a Var object.

Parameters **name** : str

The variable's name

alt_names : tuple of strings

All other names that the variable might be referred to in any input data. Each of these should be unique to this variable in order to avoid loading the wrong quantity.

description : str

A description of the variable

func : function

The function with which to compute the variable

variables : sequence of aospy.Var objects

The variables passed to *func* to compute it. Order matters: whenever calculations are performed to generate data corresponding to this Var, the data corresponding to the elements of *variables* will be passed to *self.function* in the same order.

func_input_dtype : {None, 'DataArray', 'Dataset', 'numpy'}

The datatype expected by *func* of its arguments

units : aospy.units.Units object

The variable's physical units

domain : str

The physical domain of the variable, e.g. 'atmos', 'ocean', or 'land'. This is only used by aospy by some types of *DataLoader*, including *GFDLDataLoader*.

def_time, def_vert, def_lat, def_lon : bool

Whether the variable is defined, respectively, in time, vertically, in latitude, and in longitude

math_str : str

The mathematical representation of the variable. This is typically a raw string of LaTeX math-mode, e.g. r'$T_mathrm{sfc}$' for surface temperature.

colormap : str

(Currently not used by aospy) The name of the default colormap to be used in plots of this variable.

valid_range : length-2 tuple

The range of values outside which to flag as unphysical/erroneous

mask_unphysical(*data*)
Mask data array where values are outside physically valid range.

to_plot_units(*data*, *dtype_vert=False*)
Convert the given data to plotting units.

Region

class aospy.region.**Region**(*name=''*, *description=''*, *lon_bounds=[]*, *lat_bounds=[]*, *mask_bounds=[]*, *do_land_mask=False*)
Geographical region over which to perform averages and other reductions.

Each *Proj* object includes a list of *Region* objects, which is used by *Calc* to determine which regions over which to perform time reductions over region-average quantities.

Region boundaries are specified as either a single "rectangle" in latitude and longitude or the union of multiple such rectangles. In addition, a land or ocean mask can be applied.

See also:

aospy.Calc.region_calcs

Attributes

name	(str) The region's name
description	(str) A description of the region
mask_bounds	(tuple) The coordinates defininining the lat-lon rectangle(s) that define(s) the region's boundaries
do_land_mask	Whether values occurring over land, ocean, or neither are excluded from the region, and whether the included points must be strictly land or ocean or if fractional land/ocean values are included.

__init__(*name=''*, *description=''*, *lon_bounds=[]*, *lat_bounds=[]*, *mask_bounds=[]*, *do_land_mask=False*)

Instantiate a Region object.

If a region spans across the endpoint of the data's longitude array (i.e. it crosses the Prime Meridian for data with longitudes spanning 0 to 360), it must be defined as the union of two sections extending to the east and to the west of the Prime Meridian.

Parameters **name** : str

The region's name. This must be unique from that of any other *Region* objects being used by the overlying *Proj*.

description : str, optional

A description of the region. This is not used internally by aospy; it is solely for the user's information.

lon_bounds, lat_bounds : length-2 sequence, optional

The longitude and latitude bounds of the region. If the region boundaries are more complicated than a single lat-lon rectangle, use *mask_bounds* instead.

mask_bounds : sequence, optional

Each element is a length-2 tuple of the format *(lat_bounds, lon_bounds)*, where each of *lat_bounds* and *lon_bounds* are of the form described above.

do_land_mask : { False, True, 'ocean', 'strict_land', 'strict_ocean'},

optional

Determines what, if any, land mask is applied in addition to the mask defining the region's boundaries. Default *False*.

- True: apply the data's full land mask
- False: apply no mask
- 'ocean': mask out land rather than ocean
- 'strict_land': mask out all points that are not 100% land
- 'strict_ocean': mask out all points that are not 100% ocean

Examples

Define a region spanning the entire globe

```
>>> globe = Region(name='globe', lat_bounds=(-90, 90),
...                lon_bounds=(0, 360), do_land_mask=False)
```

Define a region corresponding to the Sahel region of Africa, which we'll define as land points within 10N-20N latitude and 18W-40E longitude. Because this region crosses the 0 degrees longitude point, it has to be defined using *mask_bounds* as the union of two lat-lon rectangles.

```
>>> sahel = Region(name='sahel', do_land_mask=True,
...                mask_bounds=[((10, 20), (0, 40)),
...                             ((10, 20), (342, 360))])
```

av(*data*)
Time average of region-average time-series.

mask_var(*data*)
Mask the data of the given variable outside the region.

std(*data*)
Standard deviation of region-average time-series.

ts(*data*)
Create time-series of region-average data.

1.6.4 Calculations

`Calc` is the engine that combines the user's specifications of (1) the data on disk via `Proj`, `Model`, and `Run`, (2) the physical quantity to compute and regions to aggregate over via `Var` and `Region`, and (3) the desired date range, time reduction method, and other characteristics to actually perform the calculation

Whereas `Proj`, `Model`, `Run`, `Var`, and `Region` are all intended to be saved in `.py` files for reuse, `Calc` objects are intended to be generated dynamically by a main script and then not retained after they have written their outputs to disk following the user's specifications.

Moreover, if the `main.py` script is used to execute calculations, no direct interfacing with `Calc` or it's helper class, `CalcInterface` is required by the user, in which case this section should be skipped entirely.

Also included is the `find_obj` module, which enables aospy e.g. in the main script to find objects in the user's object library that the user specifies via their string names rather than having to import the objects themselves.

CalcInterface and Calc

class aospy.calc.**CalcInterface**(*proj=None*, *model=None*, *run=None*, *ens_mem=None*, *var=None*, *date_range=None*, *region=None*, *intvl_in=None*, *intvl_out=None*, *dtype_in_time=None*, *dtype_in_vert=None*, *dtype_out_time=None*, *dtype_out_vert=None*, *level=None*, *time_offset=None*, *verbose=True*)
Interface to the Calc class.

__init__(*proj=None*, *model=None*, *run=None*, *ens_mem=None*, *var=None*, *date_range=None*, *region=None*, *intvl_in=None*, *intvl_out=None*, *dtype_in_time=None*, *dtype_in_vert=None*, *dtype_out_time=None*, *dtype_out_vert=None*, *level=None*, *time_offset=None*, *verbose=True*)
Instantiate a CalcInterface object.

Parameters **proj** : aospy.Proj object

The project for this calculation.

model : aospy.Model object

The model for this calculation.

run : aospy.Run object

The run for this calculation.

var : aospy.Var object

The variable for this calculation.

ens_mem : Currently not supported.

This will eventually be used to specify particular ensemble members of multi-member ensemble simulations.

region : sequence of aospy.Region objects

The region(s) over which any regional reductions will be performed.

date_range : tuple of datetime.datetime objects

The range of dates over which to perform calculations.

intvl_in : {None, 'annual', 'monthly', 'daily', '6hr', '3hr'}, optional

The time resolution of the input data.

dtype_in_time : {None, 'inst', 'ts', 'av', 'av_ts'}, optional

What the time axis of the input data represents:

- 'inst' : Timeseries of instantaneous values
- 'ts' : Timeseries of averages over the period of each time-index
- 'av' : A single value averaged over a date range

dtype_in_vert : {None, 'pressure', 'sigma'}, optional

The vertical coordinate system used by the input data:

- None : not defined vertically
- 'pressure' : pressure coordinates
- 'sigma' : hybrid sigma-pressure coordinates

intvl_out : {'ann', season-string, month-integer}

The sub-annual time interval over which to compute:

- 'ann' : Annual mean
- season-string : E.g. 'JJA' for June-July-August
- month-integer : 1 for January, 2 for February, etc.

dtype_out_time : tuple with elements being one or more of:

- Gridpoint-by-gridpoint output:
  - 'av' : Gridpoint-by-gridpoint time-average
  - 'std' : Gridpoint-by-gridpoint temporal standard deviation
  - 'ts' : Gridpoint-by-gridpoint time-series
- Averages over each region specified via *region*:
  - 'reg.av', 'reg.std', 'reg.ts' : analogous to 'av', 'std', 'ts'

dtype_out_vert : {None, 'vert_av', 'vert_int'}, optional

How to reduce the data vertically:

- None : no vertical reduction (i.e. output is defined vertically)
- 'vert_av' : mass-weighted vertical average
- 'vert_int' : mass-weighted vertical integral

time_offset : {None, dict}, optional

How to offset input data in time to correct for metadata errors

- None : no time offset applied
- dict : e.g. `{'hours': -3}` to offset times by -3 hours See *aospy.utils.times.apply_time_offset()*.

class aospy.calc.**Calc**(*calc_interface*)

Class for executing, saving, and loading a single computation.

Calc objects are instantiated with a single argument: a *CalcInterface* object that includes all of the parameters necessary to determine what calculations to perform.

__init__(*calc_interface*)

ARR_XARRAY_NAME = 'aospy_result'

compute(*save_files=True*, *save_tar_files=True*)

Perform all desired calculations on the data and save externally.

key = 'level'

load(*dtype_out_time*, *dtype_out_vert=False*, *region=False*, *time=False*, *vert=False*, *lat=False*, *lon=False*, *plot_units=False*, *mask_unphysical=False*)

Load the data from the object if possible or from disk.

region_calcs(*arr*, *func*, *n=0*)

Perform a calculation for all regions.

save(*data*, *dtype_out_time*, *dtype_out_vert=False*, *save_files=True*, *save_tar_files=False*)

Save aospy data to data_out attr and to an external file.

find_obj

Get aospy objects from an object containing them given their name string.

aospy.find_obj.**to_iterable**(*obj*)

Return the object if already iterable, otherwise return it as a list.

aospy.find_obj.**to_model**(*model*, *proj*, *projs_module*)

Convert string of a Model name to a Model instance.

aospy.find_obj.**to_proj**(*proj*, *projs_module*)

Convert string of an aospy.Proj name to an aospy.Proj instance.

aospy.find_obj.**to_region**(*region*, *regions_module*, *proj=False*)

Convert string of an aospy.Region name to an aospy.Region instance.

aospy.find_obj.**to_run**(*run*, *model*, *proj*, *projs_module*)

Convert string matching an aospy.run name to an aospy.run instance.

aospy.find_obj.**to_var**(*var*, *vars_module*)

Convert string of an aospy.var name to an aospy.var instance.

operator

Warning: The `operator` module is in the process of being re-vamped and is therefore currently not supported.

Abstraction of mathematical operators to be between aospy objects.

class `aospy.operator.`**`Operator`**(*operator*, *objects*)

1.6.5 Units and Constants

aospy provides the classes `Constant` and `Units` for representing, respectively, physical constants (e.g. Earth's gravitational acceleration at the surface = 9.81 m/s^2) and physical units (e.g. meters per second squared in that example).

aospy comes with several commonly used constants saved within the `constants` module in which the `Constant` class is also defined. In contrast, there are no pre-defined `Units` objects; the user must define any `Units` objects they wish to use (e.g. to populate the `units` attribute of their `Var` objects).

Similarly, whereas these baked-in `Constant` objects are used by aospy in various places, aospy currently does not actually use the `Var.units` attribute or the `Units` class more generally; they are for the user's own informational purposes.

constants

Classes and objects pertaining to physical constants.

class `aospy.constants.`**`Constant`**(*value*, *units*, *description=''*)
Physical constants used in atmospheric and oceanic sciences.

units

Functionality for representing physical units, e.g. meters.

class `aospy.units.`**`Units`**(*units=''*, *plot_units=False*, *plot_units_conv=1.0*, *vert_int_units=False*, *vert_int_plot_units=False*, *vert_int_plot_units_conv=False*)
String representation of physical units and conversion methods.

Note: There has been discussion of implementing units-handling upstream within xarray (see here). If and when that happens, the `Units` class will likely be deprecated and replaced with the upstream version.

1.6.6 Utilities

aospy includes a number of utility functions that are used internally and may also be useful to users for their own purposes. These include functions pertaining to input/output (IO), time arrays, andvertical coordinates.

utils.io

Utility functions for data input and output.

`aospy.utils.io.`**`data_in_label`**(*intvl_in*, *dtype_in_time*, *dtype_in_vert=False*)
Create string label specifying the input data of a calculation.

aospy.utils.io.**data_name_gfdl**(*name*, *domain*, *data_type*, *intvl_type*, *data_yr*, *intvl*, *data_in_start_yr*, *data_in_dur*)
Determine the filename of GFDL model data output.

aospy.utils.io.**data_out_label**(*time_intvl*, *dtype_time*, *dtype_vert=False*)

aospy.utils.io.**dict_name_keys**(*objs*)
Create dict whose keys are the 'name' attr of the objects.

aospy.utils.io.**dmget**(*files_list*)
Call GFDL command 'dmget' to access archived files.

aospy.utils.io.**ens_label**(*ens_mem*)
Create label of the ensemble member for aospy data I/O.

aospy.utils.io.**get_parent_attr**(*obj*, *attr*, *strict=False*)
Search recursively through an object and its parent for an attribute.

Check if the object has the given attribute and it is non-empty. If not, check each parent object for the attribute and use the first one found.

aospy.utils.io.**time_label**(*intvl*, *return_val=True*)
Create time interval label for aospy data I/O.

aospy.utils.io.**to_dup_list**(*x*, *n*, *single_to_list=True*)
Convert singleton or iterable into length-n list. If the input is a list, with length-1, its lone value gets duplicated n times. If the input is a list with length-n, leave it the same. If the input is any other data type, replicate it as a length-n list.

aospy.utils.io.**yr_label**(*yr_range*)
Create label of start and end years for aospy data I/O.

utils.times

Utility functions for handling times, dates, etc.

aospy.utils.times.**add_uniform_time_weights**(*ds*)
Append uniform time weights to a Dataset.

All DataArrays with a time coordinate require a time weights coordinate. For Datasets read in without a time bounds coordinate or explicit time weights built in, aospy adds uniform time weights at each point in the time coordinate.

Parameters **ds** : Dataset

Input data

Returns Dataset

aospy.utils.times.**apply_time_offset**(*time*, *years=0*, *months=0*, *days=0*, *hours=0*)
Apply a specified offset to the given time array.

This is useful for GFDL model output of instantaneous values. For example, 3 hourly data postprocessed to netCDF files spanning 1 year each will actually have time values that are offset by 3 hours, such that the first value is for 1 Jan 03:00 and the last value is 1 Jan 00:00 of the subsequent year. This causes problems in xarray, e.g. when trying to group by month. It is resolved by manually subtracting off those three hours, such that the dates span from 1 Jan 00:00 to 31 Dec 21:00 as desired.

Parameters **time** : xarray.DataArray representing a timeseries

years, months, days, hours : int, optional

The number of years, months, days, and hours, respectively, to offset the time array by. Positive values move the times later.

Returns pandas.DatetimeIndex

Examples

Case of a length-1 input time array:

```
>>> times = xr.DataArray(datetime.datetime(1899, 12, 31, 21))
>>> apply_time_offset(times)
Timestamp('1900-01-01 00:00:00')
```

Case of input time array with length greater than one:

```
>>> times = xr.DataArray([datetime.datetime(1899, 12, 31, 21),
...                       datetime.datetime(1899, 1, 31, 21)])
>>> apply_time_offset(times)
DatetimeIndex(['1900-01-01', '1899-02-01'], dtype='datetime64[ns]',
              freq=None)
```

aospy.utils.times.**assert_matching_time_coord**(*arr1*, *arr2*)

Check to see if two DataArrays have the same time coordinate.

Parameters **arr1** : DataArray or Dataset

First DataArray or Dataset

arr2 : DataArray or Dataset

Second DataArray or Dataset

Raises **ValueError**

If the time coordinates are not identical between the two Datasets

aospy.utils.times.**convert_scalar_to_indexable_coord**(*scalar_da*)

Convert a scalar coordinate to an indexable one.

In xarray, scalar coordinates cannot be indexed. This converts a scalar coordinate-containing `DataArray` to one that can be indexed using `da.sel` and `da.isel`.

Parameters **scalar_da** : DataArray

Must contain a scalar coordinate

Returns DataArray

aospy.utils.times.**datetime_or_default**(*date*, *default*)

Return a datetime.datetime object or a default.

Parameters **date** : *None* or datetime-like object

default : The value to return if *date* is *None*

Returns *default* if *date* is *None*, otherwise returns the result of

utils.times.ensure_datetime(date)

aospy.utils.times.**ensure_datetime**(*obj*)

Return the object if it is of type datetime.datetime; else raise.

Parameters **obj** : Object to be tested.

Returns The original object if it is a datetime.datetime object.

Raises TypeError if 'obj' is not of type 'datetime.datetime'.

aospy.utils.times.**ensure_time_as_dim**(*ds*)

Ensures that time is an indexable dimension on relevant quantites

In xarray, scalar coordinates cannot be indexed. We rely on indexing in the time dimension throughout the code; therefore we need this helper method to (if needed) convert a scalar time coordinate to a dimension.

Note that this must be applied before CF-conventions are decoded; otherwise it casts `np.datetime64[ns]` as `int` values.

Parameters **ds** : Dataset

Dataset with a time coordinate

Returns Dataset

aospy.utils.times.**ensure_time_avg_has_cf_metadata**(*ds*)

Add time interval length and bounds coordinates for time avg data.

If the Dataset or DataArray contains time average data, enforce that there are coordinates that track the lower and upper bounds of the time intervals, and that there is a coordinate that tracks the amount of time per time average interval.

CF conventions require that a quantity stored as time averages over time intervals must have time and time_bounds coordinates *[R1]*. aospy further requires AVERAGE_DT for time average data, for accurate time-weighted averages, which can be inferred from the CF-required time_bounds coordinate if needed. This step should be done prior to decoding CF metadata with xarray to ensure proper computed timedeltas for different calendar types.

Parameters **ds** : Dataset or DataArray

Input data

Returns Dataset or DataArray

Time average metadata attributes added if needed.

aospy.utils.times.**extract_months**(*time*, *months*)

Extract times within specified months of the year.

Parameters **time** : xarray.DataArray

Array of times that can be represented by numpy.datetime64 objects (i.e. the year is between 1678 and 2262).

months : Desired months of the year to include

Returns xarray.DataArray of the desired times

aospy.utils.times.**month_indices**(*months*)

Convert string labels for months to integer indices.

Parameters **months** : str, int

If int, number of the desired month, where January=1, February=2, etc. If str, must match either 'ann' or some subset of 'jfmamjjasond'. If 'ann', use all months. Otherwise, use the specified months.

Returns np.ndarray of integers corresponding to desired month indices

Raises **TypeError** : If *months* is not an int or str

See also:

`_month_conditional`

aospy.utils.times.**monthly_mean_at_each_ind**(*monthly_means*, *sub_monthly_timeseries*)
Copy monthly mean over each time index in that month.

Parameters **monthly_means** : xarray.DataArray

array of monthly means

sub_monthly_timeseries : xarray.DataArray

array of a timeseries at sub-monthly time resolution

Returns xarray.DataArray with eath monthly mean value from *monthly_means* repeated

at each time within that month from *sub_monthly_timeseries*

See also:

monthly_mean_ts Create timeseries of monthly mean values

aospy.utils.times.**monthly_mean_ts**(*arr*)
Convert a sub-monthly time-series into one of monthly means.

Also drops any months with no data in the original DataArray.

Parameters **arr** : xarray.DataArray

Timeseries of sub-monthly temporal resolution data

Returns xarray.DataArray

Array resampled to comprise monthly means

See also:

monthly_mean_at_each_ind Copy monthly means to each submonthly time

aospy.utils.times.**numpy_datetime_range_workaround**(*date*, *min_year*)
Reset a date to earliest allowable year if outside of valid range.

Hack to address np.datetime64, and therefore pandas and xarray, not supporting dates outside the range 1677-09-21 and 2262-04-11 due to nanosecond precision. See e.g. https://github.com/spencerahill/aospy/issues/96.

Parameters **date** : datetime.datetime object

min_year : int

Year in the units attribute of the raw loaded data

Returns datetime.datetime object

Original datetime.datetime object if the original date is within the permissible dates, otherwise a datetime.datetime object with the year offset to the earliest allowable year.

aospy.utils.times.**numpy_datetime_workaround_encode_cf**(*ds*)
Generate CF-compliant units for out-of-range dates.

Hack to address np.datetime64, and therefore pandas and xarray, not supporting dates outside the range 1677-09-21 and 2262-04-11 due to nanosecond precision. See e.g. https://github.com/spencerahill/aospy/issues/96.

Specifically, we coerce the data such that, when decoded, the earliest value starts in 1678 but with its month, day, and shorter timescales (hours, minutes, seconds, etc.) intact and with the time-spacing between values intact.

Parameters **ds** : xarray.Dataset

Returns xarray.Dataset, int

Dataset with time units adjusted as needed, and minimum year in loaded data.

aospy.utils.times.**sel_time**(*da*, *start_date*, *end_date*)
Subset a DataArray or Dataset for a given date range.

Ensures that data are present for full extent of requested range. Appends start and end date of the subset to the DataArray.

Parameters **da** : DataArray or Dataset

data to subset

start_date : np.datetime64

start of date interval

end_date : np.datetime64

end of date interval

Returns **da** : DataArray or Dataset

subsetted data

Raises **AssertionError**

if data for requested range do not exist for part or all of requested range

utils.vertcoord

Utility functions for dealing with vertical coordinates.

aospy.utils.vertcoord.**d_deta_from_pfull**(*arr*)
Compute $partial/partialeta$ of the array on full hybrid levels.

eta is the model vertical coordinate, and its value is assumed to simply increment by 1 from 0 at the surface upwards. The data to be differenced is assumed to be defined at full pressure levels.

Parameters **arr** : xarray.DataArray containing the 'pfull' dim

Returns **deriv** : xarray.DataArray with the derivative along 'pfull' computed via

2nd order centered differencing.

aospy.utils.vertcoord.**d_deta_from_phalf**(*arr*, *pfull_coord*)
Compute pressure level thickness from half level pressures.

aospy.utils.vertcoord.**does_coord_increase_w_index**(*arr*)
Determine if the array values increase with the index.

Useful, e.g., for pressure, which sometimes is indexed surface to TOA and sometimes the opposite.

aospy.utils.vertcoord.**dp_from_p**(*p*, *ps*, *p_top=0.0*, *p_bot=110000.0*)
Get level thickness of pressure data, incorporating surface pressure.

Level edges are defined as halfway between the levels, as well as the user- specified uppermost and lowermost values. The dp of levels whose bottom pressure is less than the surface pressure is not changed by ps, since they don't intersect the surface. If ps is in between a level's top and bottom pressures, then its dp becomes the pressure difference between its top and ps. If ps is less than a level's top and bottom pressures, then that level is underground and its values are masked.

Note that postprocessing routines (e.g. at GFDL) typically mask out data wherever the surface pressure is less than the level's given value, not the level's upper edge. This masks out more levels than the

aospy.utils.vertcoord.**dp_from_ps**(*bk*, *pk*, *ps*, *pfull_coord*)
Compute pressure level thickness from surface pressure

aospy.utils.vertcoord.**get_dim_name**(*arr*, *names*)
Determine if an object has an attribute name matching a given list.

aospy.utils.vertcoord.**int_dp_g**(*arr*, *dp*)
Mass weighted integral.

aospy.utils.vertcoord.**integrate**(*arr*, *ddim*, *dim=False*, *is_pressure=False*)
Integrate along the given dimension.

aospy.utils.vertcoord.**level_thickness**(*p*, *p_top=0.0*, *p_bot=101325.0*)
Calculates the thickness, in Pa, of each pressure level.

Assumes that the pressure values given are at the center of that model level, except for the lowest value (typically 1000 hPa), which is the bottom boundary. The uppermost level extends to 0 hPa.

Unlike *dp_from_p*, this does not incorporate the surface pressure.

aospy.utils.vertcoord.**pfull_from_ps**(*bk*, *pk*, *ps*, *pfull_coord*)
Compute pressure at full levels from surface pressure.

aospy.utils.vertcoord.**phalf_from_ps**(*bk*, *pk*, *ps*)
Compute pressure of half levels of hybrid sigma-pressure coordinates.

aospy.utils.vertcoord.**replace_coord**(*arr*, *old_dim*, *new_dim*, *new_coord*)
Replace a coordinate with new one; new and old must have same shape.

aospy.utils.vertcoord.**to_hpa**(*arr*)
Convert pressure array from Pa to hPa (if needed).

aospy.utils.vertcoord.**to_pascal**(*arr*, *is_dp=False*)
Force data with units either hPa or Pa to be in Pa.

aospy.utils.vertcoord.**to_pfull_from_phalf**(*arr*, *pfull_coord*)
Compute data at full pressure levels from values at half levels.

aospy.utils.vertcoord.**to_phalf_from_pfull**(*arr*, *val_toa=0*, *val_sfc=0*)
Compute data at half pressure levels from values at full levels.

Could be the pressure array itself, but it could also be any other data defined at pressure levels. Requires specification of values at surface and top of atmosphere.

aospy.utils.vertcoord.**to_radians**(*arr*, *is_delta=False*)
Force data with units either degrees or radians to be radians.

aospy.utils.vertcoord.**vert_coord_name**(*arr*)

CHAPTER 2

Get in touch

- Troubleshooting: We are actively seeking new users and are eager to help you get started with aospy! Usage questions, bug reports, and any other correspondence are all welcome and best placed as Issues on the Github repo.
- Contributing: We are also actively seeking new developers! Please get in touch by opening an Issue or submitting a Pull Request.

CHAPTER 3

License

aospy is freely available under the open source Apache License.

CHAPTER 4

History

aospy was originally created by Spencer Hill as a means of automating calculations required for his Ph.D. work across many climate models and simulations. Starting in 2014, Spencer Clark became aospy's second user and developer. The first official release on PyPI was v0.1 on January 24, 2017.

Bibliography

[R1] http://cfconventions.org/cf-conventions/v1.6.0/cf-conventions.html#_data_representative_of_cells

a

Symbols

A

C

D

E

G

I

K

L