Community Intercomparison Suite Documentation

Release 1.5.4 (Stable)

Centre of Environmental Data Archival

Mar 31, 2017

Contents

Contents:

CHAPTER 1

Installing CIS

A pre-packaged version of CIS is available for installation using conda for 64-bit Linux, Mac OSX and Windows.

Once conda is installed, you can easily install CIS with the following command:

```
$ conda install -c conda-forge cis
```

If you don't already have conda, you must first download and install it. Anaconda is a free conda package that includes Python and many common scientific and data analysis libraries, and is available here. Further documentation on using Anaconda and the features it provides can be found at http://docs.continuum.io/anaconda/index.html.

To check that CIS is installed correctly, simply type `cis version` to display the version number, for example:

```
$ cis version
Using CIS version: V1R4M0 (Stable)
```

In order to upgrade CIS to the latest version use:

```
$ conda update -c conda-forge cis
```

Dependencies

If you choose to install the dependencies yourself, use the following command to check the required dependencies are present:

```
$ python setup.py checkdep
```

CHAPTER 2

What's new in CIS

What's new in CIS 1.5

This page documents the new features added, and bugs fixed in CIS since version 1.4.0. See all changes here: https://github.com/cedadev/cis/compare/1.4.0...1.5.0

CIS 1.5 features

- The biggest change is that CIS can now be used as a Python library, all of the command line tools are now easily available through Python. This allows commands to be run sequentially in memory, slicing of gridded or ungridded datasets and easy integration with other Python packages such as Iris and Pandas.
- Taylor diagrams - CIS is now able to plot Taylor diagrams which are an excellent way of quantitatively comparing two or more (collocated) datasets
- All map plots are now able to be plotted in any of the available Cartopy projections, see http://scitools.org.uk/cartopy/docs/latest/crs/projections.html for a full list.

Incompatible changes

- Since aggregation of gridded datasets has quite a different set of options as compared to the aggregation of ungridded datasets, the `aggregate` command has been deprecated for gridded datasets. It is still supported through command line for the time being, but will be removed in future releases. Please use the `collapse` command instead.

Bugs fixed

- [JASCIS-268] The plotting routines have been re-architected to allow easier testing and extension.
- [JASCIS-357] Added deprecation for the aggregation of gridded datasets

- [JASCIS-329] Metadata objects now attempt to use cf_units for all units, but will fall back to strings if needed. In future releases we may insist on plugins providing standard units.

CIS 1.5.1 fixes

- Minor fix in interpreting units when reading some NetCDF data in Python 2
- Fixed an issue where line and scatter plots weren't respecting the yaxis keyword

CIS 1.5.2 fixes

- Gridded and ungridded datasets can now be subset to an arbitrary lat/lon (shapely) shape.
- Slicing and copying Coords now preserves the axis
- Fixed an issue where subsetting gridded data over multiple coordinates sometimes resulted in an error
- CIS will now catch errors when writing out metadata values which might have special types and can't be safely cast (e.g. VALID_RANGE).
- Minor fix for log scale color bars
- Minor fix for parsing the command aliases
- Minor fix for creating data lists from iterators

CIS 1.5.3 fixes

- Fixed a (potentially serious) bug in unit parsing which would convert any string to lowercase.
- [JASCIS-367] Make the name() method more consistent between gridded and ungridded data
- Minor fix when reading variables from PP files with spaces in the name

CIS 1.5.4 fixes

- Minor fix for the info command on Windows

What's new in CIS 1.4

This page documents the new features added, and bugs fixed in CIS since version 1.3.1. See all changes here: https://github.com/cedadev/cis/compare/1.3.1...1.4.0

CIS 1.4 features

- An all new Python interface for subsetting any data read by CIS. Just call the `subset()` method on any CIS GriddedData or UngriddedData object to access the same functionality as through the command line - without reading or writing to disk. See *CIS API* for more details.
- CIS now includes full support for Python => 3.4, as well as Python 2.7
- New verbose and quiet flags allow for control over how much CIS commands output to the screen. The default verbosity has also changed so that by default only warnings and errors will be output to the screen. The full debug output remains for the cis.log file.

- Significant optimizations have been made in gridded -> ungridded collocation which should now be considerably faster. Also, when collocating multiple gridded source datasets the interpolation indices are now cached internally leading to further time savings.
- Any `valid_range` attributes in supported NetCDF or HDF files (including MODIS, CALIOP and CloudSat) files are now automatically respected by CIS. All data values outside of the valid range are masked. Data from NetCDF files with `valid_min` or `valid_max` attributes is also masked appropriately.
- CloudSat `missing` and `missop` attributes are now read and combined to mask out values which don't conform to the inequality defined.
- [JASCIS-342] The extrapolation modes are now consistent across both gridded->gridded and gridded->ungridded collocation modes. The default is no extrapolation (gridded->gridded would previously extrapolate). This can still be overridden by the user.
- [JASCIS-128] If the output file already exists the user is now prompted to overwrite it. This prompt can be disabled by using the –force-overwrite argument, or setting the `CIS_FORCE_OVERWRITE` environment variable to 'TRUE'.

Incompatible changes

- To accommodate the new verbose flags (-v) the info command now takes a single datagroup argument, and optional variable names, as reflected in the updated documentation.
- CIS no longer prepends ungridded output files with 'cis-'. Instead CIS creates a global attribute in the output file called source which contains 'CIS<version>'. This is checked in the updated CIS plugin when reading any NetCDF file.

Note: While this is much neater going forward and will hopefully save a lot of head scratching it will mean CIS is unable to read old files produced by CIS automatically. All commands can be forced to use the CIS product by including the product=cis keyword argument. Alternatively you can update the data file manually using the following command: `ncatted -O -a source,global,a,c,"CIS" in.nc`

Bugs fixed

- [JASCIS-34] MODIS L3 data is now correctly treated as gridded data.
- [JASCIS-345] Product regular expression matching now matches the whole string rather than just the start.
- [JASCIS-360] Collocation now correctly applies the 'missing_data_for_missing_sample' logic for all collocations.
- [JASCIS-361] Fixed the CloudSat scale and offset transformation so that they are now applied correctly.
- [JASCIS-281] Fixed a caching error when aggregating multiple ungridded datasets which could lead to incorrect values
- CIS no longer crashes when the CIS_PLUGIN_HOME path cannot be found

What's new in CIS 1.3

This page documents the new features added, and bugs fixed in CIS since version 1.2. See all changes here: https://github.com/cedadev/cis/compare/1.2.1...1.3.0

CIS 1.3 features

- Some significant optimisations have been made in reading Caliop, CCI and Aeronet datasets, there have also been speed improvements for ungridded data subsetting
- New Pandas interface allows the easy creation of DataFrames through the 'as_data_frame' method on Gridded or Ungridded data. Pandas is an extensive python library providing many powerful data analysis algorithms and routines.
- Compatibility updates for newer versions of Numpy and SciPy. The minimum require version of SciPy is now 0.16.0
- Swapped out Basemap plotting routines for Cartopy. This removed a dependancy (as Cartopy was already required by Iris), and has given us more flexibility for plotting different projections in the future
- Plots now automatically try to use the most appropriate resolution background images for plots over coastlines NASA blue marble images.
- 'scatter_overlay' plots have been completely removed (they have been deprecated for the last two versions), the same functionality can be achieved through the more generic 'overlay' plots.
- Update to the UngriddedData.coord() and .coords() API to match the changes in IRIS >=1.8. This allows users to also search for coordinates by supplying a `Coord` instance to compare against. Currently this only compares standard names, but this may be extended in the future.

Bugs fixed

- JASCIS-279 - This release removes the basemap dependency and means we can use a much newer version of GEOS which doesn't clash with the SciTools version
- JASCIS-267 - Fixed ASCII file reading to be compatible with Numpy 1.9
- JASCIS-259 - Fixed Stats unit tests to reflect updates in SciPy (>0.15.0) linear regression routines for masked arrays
- JASCIS-211 - Subsetting now accepts variable names (rather than axes shorthands) more consistently, the docs have been updated to make the dangers of relying on axes shorthands clear and an error is now thrown if a specific subset coordinate is not found.
- JASCIS-275 - The ungridded subsetting is now done array-wise rather than element wise giving large performance improvements

CIS 1.3.1 fixes

- JASCIS-231 & JASCIS-209 - CIS now better determines the yaxis when the user specifies the xaxis as 'time' so that overlaying multiple time series is easy
- JASCIS-283 - An issue with setting xmin or xmax using datetimes
- A minor fix to the AerosolCCI product

What's new in CIS 1.2

This page documents the new features added, and bugs fixed in CIS since version 1.1. See all changes here: https://github.com/cedadev/cis/compare/1.1.0...1.2.0

CIS 1.2 features

- All new `cis info` command provides much more detailed information about ungridded data variables and enables multiple variables to be output at a time.
- Updated a number of routines to take advantage of Iris 1.8 features. In particular gridded-gridded collocation using the nearest neighbour kernel should be significantly faster. Iris 1.8 is now the minimum version required for CIS.
- Gridded-ungridded collocation now supports collocation from cubes with hybrid height or hybrid pressure coordinates for both nearest neighbour and linear interpolation kernels.
- Built-in support for reading multiple HadGEM .pp files directly.
- All new API and plugin development documentation, including a number of tutorials

Bugs fixed

- JASCIS-253 - Any ungridded points which contain a NaN in any of its coordinate values will now be ignored by CIS
- JASCIS-250 - Multiple HadGEM files can now be read correctly through the new data plugins.
- JASCIS-197 - Gridded-gridded collocation now respects scalar coordinates
- JASCIS-199 - Aggregation now correctly uses the bounds supplied by the user, even when collapsing to length one coordinates.
- Speed improvement to the ungridded-gridded collocation using linear interpolation
- Several bug fixes for reading multiple GASSP ship files
- Renamed and restructured the collocation modules for consistency
- Many documentation spelling and formatting updates
- Many code formatting updates for PEP8 compliance

CIS 1.2.1 features

- Updated CCI plugin to support Aerosol CCI v3 files.

What's new in CIS 1.1

This page documents the new features added, and bugs fixed in CIS since version 1.0. For more detail see all changes here: https://github.com/cedadev/cis/compare/1.0.0...1.1.0

CIS 1.1 features

- JASMIN-CIS is now called CIS, and the packages, modules and documentation have been renamed accordingly.
- Conda packages are now available to allow much easier installation of CIS, and across more platforms: Linux, OSX and Windows.
- PyHDF is now an optional dependency. This makes the installation of CIS on e.g. Windows much easier when HDF reading is not required.

Bugs fixed

- JASCIS-243 - Error when reading multiple GASSP aircraft files
- JASCIS-139 - Updated ungridded aggregation to rename any variables which clash with coordinate variables, as this breaks during the output otherwise.
- Compatibility fixes for Numpy versions >1.8 and Python-NetCDF versions >1.1.
- Fix Caliop pressure units which were stored as hPA, but need to be hPa to conform to CF.
- The integration test data has been moved completely out of the repository - making the download quicker and less bloated. It's location can be specified by setting the CIS_DATA_HOME environment variable.
- A test runner has been created to allow easy running of the unit and integration test.

What's new in CIS 1.1.1

This section documents changes in CIS since version 1.1, these were primarily bug fixes and documentation updates. See all changes here: https://github.com/cedadev/cis/compare/1.1.0...1.1.1

Bugs fixed

- JASCIS-181 - Updated eval documentation
- JASCIS-239 - Documented the requirement of PyHamCrest for running tests
- JASCIS-249 - CIS will now accept variables and filenames (such as Windows paths) which include a colon as long as they are escaped with a backslash. E.g. `cis plot my_var:C\:\my_file.nc`.
- Occasionally HDF will exit when reading an invalid HDF file without throwing any exceptions. To protect against this the HDF reader will now insist on an .hdf extension for any files it reads.

CHAPTER 3

What kind of data can CIS deal with?

Writing

When creating files from a CIS command, CIS uses the NetCDF 4 classic format. Output files are always suffixed with `.nc`.

Reading

CIS has built-in support for NetCDF and HDF4 file formats. That said, most data requires some sort of pre-processing before being ready to be plotted or analysed (this could be scale factors or offsets needing to applied, or even just knowing what the dependencies between variables are). For that reason, the way CIS deals with reading in data files is via the concept of "data products". Each product has its own very specific way of reading and interpreting the data in order for it to be ready to be plotted, analysed, etc.

So far, CIS can read the following ungridded data files:

Dataset	Product name	Type	File Signature
AERONET	Aeronet	Ground-stations	*.lev20
Aerosol CCI	Aerosol_CCI	Satellite	*ESACCI*AEROSOL*
CALIOP L1	Caliop_L1	Satellite	CAL_LID_L1-ValStage1-V3*.hdf
CALIOP L2	Caliop_L2	Satellite	CAL_LID_L2_05kmAPro-Prov-V3*.hdf
Cloud-Sat	CloudSat	Satellite	*_CS_*GRANULE*.hdf
Flight cam-paigns	NCAR_NetCDF_RAF	Aircraft	RF*.nc
MODIS L2	MODIS_L2	Satellite	*MYD06_L2*.hdf, *MOD06_L2*.hdf, *MYD04_L2*.hdf, *MOD04_L2*.hdf, *MYDATML2.*.hdf, *MODATML2*.hdf
Cloud CCI	Cloud_CCI	Satellite	*ESACCI*CLOUD*
CSV data-points	ASCII_Hyperpoints	N/A	*.txt
CIS un-gridded	cis	CIS output	*.nc containing the attribute Source = CIS(version)
NCAR-RAF	NCAR_NetCDF_RAF	Aircraft	*.nc containing the attribute Conventions with the value NCAR-RAF/nimbus
GASSP	NCAR_NetCDF_RAF	Aircraft	*.nc containing the attribute GASSP_Version
GASSP	NCAR_NetCDF_RAF	Ship	*.nc containing the attribute GASSP_Version, with no altitude
GASSP	NCAR_NetCDF_RAF	Ground-station	*.nc containing the attribute GASSP_Version, with attributes Station_Lat, Station_Lon and Station_Altitude

It can also read the following gridded data types:

Dataset	Product name	Type	File Signature
MODIS L3 daily	MODIS_L3	Satellite	*MYD08_D3*.hdf, *MOD08_D3*.hdf, *MOD08_E3*.hdf
HadGEM pp data	HadGEM_PP	Gridded Model Data	*.pp
Net_CDF Gridded Data	NetCDF_Gridded	Gridded Model Data	*.nc (this is the default for NetCDF Files that do not match any other signature)

The file signature is used to automatically recognise which product definition to use. Note the product can overridden easily by being specified at the command line.

This is of course far from being an exhaustive list of what's out there. To cope with this, a "plugin" architecture has been designed so that the user can readily use their own data product reading routines, without even having to change the code - see the *plugin development* page for more information. There are also mechanisms to allow you to overwrite default behaviour if the built-in products listed above do not achieve the desired results.

Datagroups

Most CIS commands operate on a 'datagroup', which is a unit of data containing one or more similar variables and one or more files from which those variables should be taken. A datagroup represents closely related data from a specific

instrument or model and as such is associated with only one data product.

A datagroup is specified with the syntax:

`<variable>...:<filename>[:product=<productname>]` where:

- `<variable>` is a mandatory argument specifying the variable or variable names to use. This should be the name of the variable as described in the file, e.g. the NetCDF variable name or HDF SDS/VDATA variable name. Multiple variables may be specified by commas, and variables may be wildcarded using any wildcards compatible with the python module glob, so that *, *?* and *[]* can all be used

> **Attention:** When specifying multiple variables, it is essential that they be on the same grid (i.e. use the same coordinates).

- `<filenames>` is a mandatory argument used to specify the files to read the variable from. These can be specified as a comma seperated list of the following possibilities:

  1. a single filename - this should be the full path to the file
  2. a single directory - all files in this directory will be read
  3. a wildcarded filename - A filename with any wildcards compatible with the python module glob, so that *, ? and [] can all be used. E.g., `/path/to/my/test*file_[0-9]`.

> **Attention:** When multiple files are specified (whether through use of commas, pointing at a directory, or wildcarding), then all those files must contain all of the specified variables, and the files should be 'compatible' - it should be possible to aggregate them together using a shared dimension - typically time (in a NetCDF file this is usually the unlimited dimension). So selecting multiple monthly files for a model run would be OK, but selecting files from two different datatypes would not be OK.

- `<productname>` is an optional argument used to specify the type of files being read. If omitted, the program will attempt to figure out which product to use based on the filename. See *Reading* to see a list of available products and their file signatures.

For example:

```
illum:20080620072500-ESACCI-L2_CLOUD-CLD_PRODUCTS-MODIS-AQUA-fv1.0.nc
Cloud_Fraction_*:MOD*,MODIS_dir/:product=MODIS_L2
```

Some file paths or variable names might contain colons (:), these need to be escaped so that CIS can tell the difference between it and the colons used to separate Datagroup elements. Simply use a backslash () to escape these characters. For example:

```
"TOTAL RAINFALL RATE\: LS+CONV KG/M2/S:C\:\My files\MODIS_dir:product=MODIS_L2"
```

Notice that we have used outer quotes to allow for the spaces in the variable and file names, and used the backslashes to escape the colons.

Reading hybrid height data with separate orography data

CIS supports the reading of gridded data containing hybrid height and pressure fields, with an orography field supplied in a separate file. The file containing the orography field (which should be properly referenced from a formula term in the data file) can just be appended to the list of files to be read in and CIS will attempt to create an appropriate altitude dimension.

Reading NetCDF4 Hierarchical Groups

CIS supports the reading of NetCDF4 hierarchical groups. These can be specified on the command line in the format `<group>/<variable_name>`, e.g. `AVHRR/Ch4CentralWavenumber`. Groups can be nested to any required depth like `<group1>/<group2...>/<variable_name>`.

CIS currently does not support writing out of NetCDF4 groups, so any groups read in will be output 'flat'.

Reading groups in user-developed product plugins

Most of the methods in the *cis.data_io.netcdf* module support netCDF4 groups using the syntax described above - users should use this module when designing their own plugins to ensure support for groups.

CHAPTER 4

Using the command line

Run the following command to print help and check that it runs: `cis --help`

The following should be displayed:

```
usage: cis [-h] [-v | -q] [--force-overwrite]
           {plot,info,col,aggregate,subset,eval,stats,version} ...

positional arguments:
  {plot,info,col,aggregate,subset,eval,stats,version}
    plot                Create plots
    info                Get information about a file
    col                 Perform collocation
    aggregate           Perform aggregation
    subset              Perform subsetting
    eval                Evaluate a numeric expression
    stats               Perform statistical comparison of two datasets
    version             Display the CIS version number

optional arguments:
  -h, --help            Show this help message and exit
  -v, --verbose         Increase the level of logging information output to
                        screen to include 'Info' statements
  -vv                   All log messages will be output to the screen including 'Debug
↪' statements
  -q, --quiet           Suppress all output to the screen, only 'Error'
                        messages will be displayed (which are always fatal).
  --force-overwrite     Do not prompt when an output file already exists -
                        always overwrite. This can also be set by setting the
                        'CIS_FORCE_OVERWRITE' environment variable to 'TRUE'
```

There are 8 commands the program can execute:

- `plot` which is used to plot the data
- `info` which prints information about a given input file

- `col` which is used to perform collocation on data
- `aggregate` which is used to perform aggregation along coordinates in the data
- `subset` which is used to perform subsetting of the data
- `eval` which is used to evaluate a numeric expression on data
- `stats` which is used to perform a statistical comparison of two datasets
- `version` which is used to display the version number of CIS

If an error occurs while running any of these commands, you may wish to increase the level of output using the verbose option, or check the log file 'cis.log'; the default location for this is the current user's home directory.

LSF Batch Job Submission

CIS jobs may be submitted to an LSF type batch submission system (e.g. the JASMIN environment) by using the command `cis.lsf` instead of cis. In this case the job will be sent to the batch system and any output will be written to the log file.

CHAPTER 5

CIS as a Python library (API)

Main API

As a command line tool, CIS has not been designed with a python API in mind. There are however some utility functions that may provide a useful start for those who wish to use CIS as a python library. For example, the functions in the base cis module provide a straightforward way to load your data. They can be easily import using, for example: `from cis import read_data`. One of the advantages of using CIS as a Python library is that you are able to perform multiple operations in one go, that is without writing to disk in between. In certain cases this may provide a significant speed-up.

Note: This section of the documentation expects a greater level of Python experience than the other sections. There are many helpful Python guides and tutorials available around the web if you wish to learn more.

The `read_data()` function is a simple way to read a single gridded or ungridded data object (e.g. a NetCDF variable) from one or more files. CIS will determine the best way to interpret the datafile by comparing the file signature with the built-in data reading plugins and any user defined plugins. Specifying a particular `product` allows the user to override this automatic detection.

`cis.`**`read_data`**(*filenames*, *variable*, *product=None*)

Read a specific variable from a list of files Files can be either gridded or ungridded but not a mix of both. First tries to read data as gridded, if that fails, tries as ungridded.

Parameters

- **`filenames`** (*`string or list`*) – The filenames of the files to read. This can be either a single filename as a string, a comma separated list, or a `list` of string filenames. Filenames can include directories which will be expanded to include all files in that directory, or wildcards such as `*` or `?`.
- **`variable`** (*`str`*) – The variable to read from the files
- **`product`** (*`str`*) – The name of the data reading plugin to use to read the data (e.g. `Cloud_CCI`).

Returns The specified data as either a `GriddedData` or `UngriddedData` object.

The `read_data_list()` function is very similar to `read_data()` except that it allows the user to specify more than one variable name. This function returns a list of data objects, either all of which will be gridded, or all ungridded, but not a mix. For ungridded data lists it is assumed that all objects share the same coordinates.

`cis.`**`read_data_list`**(*filenames*, *variables*, *product=None*, *aliases=None*)

Read multiple data objects from a list of files. Files can be either gridded or ungridded but not a mix of both.

Parameters

- **`filenames`** (*`string or list`*) – The filenames of the files to read. This can be either a single filename as a string, a comma separated list, or a `list` of string filenames. Filenames can include directories which will be expanded to include all files in that directory, or wildcards such as `*` or `?`.
- **`variables`** (*`string or list`*) – One or more variables to read from the files
- **`product`** (*`str`*) – The name of the data reading plugin to use to read the data (e.g. `Cloud_CCI`).
- **`aliases`** (*`string or list`*) – List of aliases to put on each variable's data object as an alternative means of identifying them.

Returns A list of the data read out (either a `GriddedDataList` or `UngriddedDataList` depending on the type of data contained in the files)

The `get_variables()` function returns a list of variable names from one or more specified files. This can be useful to inspect a set of files before calling the read routines described above.

`cis.`**`get_variables`**(*filenames*, *product=None*, *type=None*)

Get a list of variables names from a list of files. Files can be either gridded or ungridded but not a mix of both.

Parameters

- **`filenames`** (*`string or list`*) – The filenames of the files to read. This can be either a single filename as a string, a comma separated list, or a `list` of string filenames. Filenames can include directories which will be expanded to include all files in that directory, or wildcards such as `*` or `?`.
- **`product`** (*`str`*) – The name of the data reading plugin to use to read the data (e.g. `Cloud_CCI`).
- **`type`** (*`str`*) – The type of HDF data to read, i.e. 'VD' or 'SD'

Returns A list of the variables

Data Objects

Each of the above methods return either `GriddedData` or `UngriddedData` objects. These objects are the main data handling objects used within CIS, and their main methods are discussed in the following section. These classes share a common interface, defined by the `CommonData` class, which is detailed below. For technical reasons some methods which are common to both `GriddedData` and `UngriddedData` are not defined in the `CommonData` interface. The most useful of these methods are probably `summary()` and `save_data()`.

These objects can also be 'sliced' analogously to the underlying numpy arrays, and will return a *copy* of the requested data as a new `CommonData` object with the correct data, coordinates and metadata.

class `cis.data_io.common_data.`**`CommonData`**

Interface of common methods implemented for gridded and ungridded data.

alias
Return an alias for the variable name. This is an alternative name by which this data object may be identified if, for example, the actual variable name is not valid for some use (such as performing a python evaluation).

Returns The alias

Return type str

as_data_frame(*copy*)
Convert a CommonData object to a Pandas DataFrame.

Parameters **copy** – Create a copy of the data for the new DataFrame? Default is True.

Returns A Pandas DataFrame representing the data and coordinates. Note that this won't include any metadata.

collocated_onto(*sample*, *how=''*, *kernel=None*, *missing_data_for_missing_sample=True*, *fill_value=None*, *var_name=''*, *var_long_name=''*, *var_units=''*, ***kwargs*)
Collocate the CommonData object with another CommonData object using the specified collocator and kernel.

Parameters

- **sample** (*CommonData*) – The sample data to collocate onto
- **how** (*str*) – Collocation method (e.g. lin, nn, bin or box)
- **or cis.collocation.col_framework.Kernel kernel** (*str*) –
- **missing_data_for_missing_sample** (*bool*) – Should missing values in sample data be ignored for collocation?
- **fill_value** (*float*) – Value to use for missing data
- **var_name** (*str*) – The output variable name
- **var_long_name** (*str*) – The output variable's long name
- **var_units** (*str*) – The output variable's units
- **kwargs** – Constraint arguments such as h_sep, a_sep, etc.

Return CommonData The collocated dataset

get_all_points()
Returns a list-like object allowing access to all points as HyperPoints. The object should allow iteration over points and access to individual points.

Returns list-like object of data points

get_coordinates_points()
Returns a list-like object allowing access to the coordinates of all points as HyperPoints. The object should allow iteration over points and access to individual points.

Returns list-like object of data points

get_non_masked_points()
Returns a list-like object allowing access to all points as HyperPoints. The object should allow iteration over non-masked points and access to individual points.

Returns list-like object of data points

history
Return the associated history of the object

Returns The history

Return type str

is_gridded()
Returns value indicating whether the data/coordinates are gridded.

plot(**args*, ***kwargs*)
Plot the data. A matplotlib Axes is created if none is provided.

The default method for series data is 'line', otherwise (for e.g. a map plot) is 'scatter2d' for UngriddedData and 'heatmap' for GriddedData.

Parameters **how** (*string*) – The method to use, one of: "contour", "contourf", "heatmap", "line", "scatter", "scatter2d",

"comparativescatter", "histogram", "histogram2d" or "taylor" :param Axes ax: A matplotlib axes on which to draw the plot :param Coord or CommonData xaxis: The data to plot on the x axis :param Coord or CommonData yaxis: The data to plot on the y axis :param string or cartopy.crs.Projection projection: The projection to use for map plots (default is PlateCaree) :param float central_longitude: The central longitude to use for PlateCaree (if no other projection specified) :param string label: A label for the data. This is used for the title, colorbar or legend depending on plot type :param args: Other plot-specific args :param kwargs: Other plot-specific kwargs :return Axes: The matplotlib Axes on which the plot was drawn

sampled_from(*data*, *how=''*, *kernel=None*, *missing_data_for_missing_sample=True*, *fill_value=None*, *var_name=''*, *var_long_name=''*, *var_units=''*, ***kwargs*)
Collocate the CommonData object with another CommonData object using the specified collocator and kernel

Parameters

- **or CommonDataList data** (*CommonData*) – The data to resample
- **how** (*str*) – Collocation method (e.g. lin, nn, bin or box)
- **or cis.collocation.col_framework.Kernel kernel** (*str*) –
- **missing_data_for_missing_sample** (*bool*) – Should missing values in sample data be ignored for collocation?
- **fill_value** (*float*) – Value to use for missing data
- **var_name** (*str*) – The output variable name
- **var_long_name** (*str*) – The output variable's long name
- **var_units** (*str*) – The output variable's units
- **kwargs** – Constraint arguments such as h_sep, a_sep, etc.

Return CommonData The collocated dataset

set_longitude_range(*range_start*)
Rotates the longitude coordinate array and changes its values by 360 as necessary to force the values to be within a 360 range starting at the specified value. :param range_start: starting value of required longitude range

subset(***kwargs*)
Subset the CommonData object based on the specified constraints. Constraints on arbitrary coordinates are specified using keyword arguments. Each constraint must have two entries (a maximum and a minimum) although one of these can be None. Datetime objects can be used to specify upper and lower datetime limits, or a single PartialDateTime object can be used to specify a datetime range.

The keyword keys are used to find the relevant coordinate, they are looked for in order of name, standard_name, axis and var_name.

For example:

data.subset(time=[datetime.datetime(1984, 8, 28), datetime.datetime(1984, 8, 29)], altitude=[45.0, 75.0])

Will subset the data from the start of the 28th of August 1984, to the end of the 29th, and between altitudes of 45 and 75 (in whatever units ares used for that Coordinate).

And: data.subset(time=[PartialDateTime(1984, 9)])

Will subset the data to all of September 1984.

Parameters **kwargs** – The constraint arguments

Return CommonData The subset of the data

var_name

Return the variable name associated with this data object

Returns The variable name

Pandas

All `CommonData` objects can be converted to Pandas DataFrames using the `as_data_frame()` methods. This provides an easy interface to the powerful statistical tools available in Pandas.

Analysis Methods

Collocation

Each data object provides both `collocated_onto()` and `sampled_from()` methods, which are different ways of calling the collocation depending on whether the object being called is the source or the sample. For example the function performed by the command line:

```
$ cis col Temperature:2010.nc 2009.nc:variable=Temperature
```

can be performed in Python using:

```
temperature_2010 = cis.read_data('Temperature', '2010.nc')
temperature_2009 = cis.read_data('Temperature', '2009.nc')
temperature_2010.sampled_from(temperature_2009)
```

or, equivalently:

```
temperature_2009.collocated_onto(temperature_2010)
```

Aggregation

`UngriddedData` objects provide the `aggregate()` method to allow easy aggregation. Each dimension of the desired grid is specified as a keyword and the start, end and step as the argument (as a tuple, list or slice).

For example:

```
data.aggregate(x=[-180, 180, 360], y=slice(-90, 90, 10))
```

or:

```
data.aggregate(how='mean', t=[PartialDateTime(2008,9), timedelta(days=1))
```

Datetime objects can be used to specify upper and lower datetime limits, or a single PartialDateTime object can be used to specify a datetime range. The gridstep can be specified as a DateTimeDelta object.

The keyword keys are used to find the relevant coordinate, they are looked for in order of name, standard_name, axis and var_name.

`GriddedData` objects provide the `collapsed()` method which shadows the Iris method of the same name. Our implementation is a slight extension of the Iris method which allows partial collapsing of multi-dimensional auxilliary coordinates.

Subsetting

All objects have a `subset()` method for easily subsetting data across arbitrary dimensions. Constraints on arbitrary coordinates are specified using keyword arguments. Each constraint must have two entries (a maximum and a minimum) although one of these can be None. Datetime objects can be used to specify upper and lower datetime limits, or a single PartialDateTime object can be used to specify a datetime range.

The keyword keys are used to find the relevant coordinate, they are looked for in order of name, standard_name, axis and var_name.

For example:

```
data.subset(time=[datetime.datetime(1984, 8, 28), datetime.datetime(1984, 8, 29)],
            altitude=[45.0, 75.0])
```

will subset the data from the start of the 28th of August 1984, to the end of the 29th, and between altitudes of 45 and 75 (in whatever units ares used for that Coordinate).

And:

```
data.subset(time=[PartialDateTime(1984, 9)])
```

will subset the data to all of September 1984.

Plotting

Plotting can also easily be performed on these objects. Many options are available depending on the plot type, but CIS will attempt to make a sensible default plot regardless of the datatype or dimensionality. The default method for series data is 'line', otherwise (for e.g. a map plot) is 'scatter2d' for UngriddedData and 'heatmap' for GriddedData.

A matplotlib Axes is created if none is provided, meaning the user is able to reformat, or export the plot however they like.

CHAPTER 6

Getting file information

The info command provides a visual summary of the data within any of the data files CIS supports.

To get this summary, run a command of the format:

```
$ cis info <datagroup> [--type ["VD" | "SD"]]
```

where:

<datagroup> is a *CIS datagroup* specifying the variables and files to read and is of the format `[<variable>...:]<filename>[:product=<productname>]` where:

- `variable` is an optional variable or list of variables to use.
- `filenames` is a mandatory file or list of files to read from.
- `product` is an optional CIS data product to use (see *Data Products*):

Note that the product can only be specified if a variable is specified. See *Datagroups* for a more detailed explanation of datagroups.

`--type` allows the user to list only `SD` or `VD` variables from an HDF file, the default is `All`

Running without a variable (`$ cis info <filenames>`) will print a list of the variables available in those files such as:

```
Trop
latitude
longitude_1
surface
unspecified_1
level6
ht
msl
latitude_1
```

To get more specific information about one or more variables in those files, simply pass those as well:

```
$ cis info var1,var2:<filenames>
```

where `$var1` and `$var2` are the names of the variables to get the information for.

Here is an example output:

```
Ungridded data: SO4 / (ug m-3)
     Shape = (6478,)
     Total number of points = 6478
     Number of non-masked points = 6478
     Long name = Sulphate
     Standard name = SO4
     Units = ug m-3
     Missing value = -9999
     Range = (-0.57346399999999997, 7.0020300000000004)
     History =
     Coordinates:
       time
          Long name = Starting time
          Standard name = time
          Units = days since 1600-01-01 00:00:00
          Calendar = gregorian
          Missing value = -9999
          Range = ('2008-07-10 02:04:35', '2008-07-20 09:50:33')
          History =
       latitude
          Long name = Latitude
          Standard name = latitude
          Units = N degree
          Missing value = -9999
          Range = (4.0211802, 7.14886)
          History =
       longitude
          Long name = Longitude
          Standard name = longitude
          Units = E degree
          Missing value = -9999
          Range = (114.439, 119.733)
          History =
       altitude
          Long name = Altitude
          Standard name = altitude
          Units = m
          Missing value = -9999
          Range = (51.164299, 6532.6401)
          History =
```

CHAPTER 7

Subsetting

Subsetting allows the reduction of data by extracting variables and restricting them to ranges of one or more coordinates.

To perform subsetting, run a command of the format:

```
$ cis subset <datagroup> <limits> [-o <outputfile>]
```

where:

<datagroup> is a *CIS datagroup* specifying the variables and files to read and is of the format `<variable>...:<filename>[:product=<productname>]` where:

- `variable` is a mandatory variable or list of variables to use.
- `filenames` is a mandatory file or list of files to read from.
- `product` is an optional CIS data product to use (see *Data Products*):

See *Datagroups* for a more detailed explanation of datagroups.

<limits> is a comma separated sequence of one or more coordinate range assignments of the form `variable=[start,end]` or `variable=[value]` in which

- `variable` is the name of the variable to be subsetted, this can be the variable name (as it is in the data file) or it's CF standard name. It is also possible to use axes name shorthands such as `x`, `y`, `t`, `z` and `p` - which *usually* refer to longitude, latitude, time, altitude and pressure respectively. However this approach can lead to confusion as these shorthands can be overridden by the files themselves, or the data readers, and may not always behave as expected. For example when specifying 'z' for a gridded hybrid pressure file, this may well refer to sigma levels rather than altitude, and 'p' may not be found at all (it isn't possible to subset over hybrid coordinates). For this reason it is often safer to use variable names explicitly.
- `start` is the value at the start of the coordinate range to be included
- `end` is the value at the end of the coordinate range to be included
- `value` is taken as the start and end value.

Note: Longitude coordinates are considered to be circular, so that -10 is equivalent to 350. The start and end must describe a monotonically increasing coordinate range, so `x=[90,-90]` is invalid, but could be specified using `x=[90,270]`. The range between the start and end must not be greater than 360 degrees. The output coordinates will be on the requested grid, not the grid of the source data.

Note: An arbitrary lat/lon shape can also be provided using the `shape` limit and passing a valid WKT string as the argument, e.g. `shape=POLYGON((-10 50, 0 60, 10 50, 0 40, -10 50))`. See e.g. https://en.wikipedia.org/wiki/Well-known_text for a description of the WKT format.

Note: Date/times are specified in the format: `YYYY-MM-DDThh:mm:ss` in which `YYYY-MM-DD` is a date and `hh:mm:ss` is a time. A colon or space can be used instead of the 'T' separator (but if a space is used, the argument must be quoted). Any trailing components of the date/time may be omitted. When a date/time is used as a range start, the earliest date/time compatible with the supplied components is used (e.g., `2010-04` is treated as `2010-04-01T00:00:00`) and when used as a range end, the latest compatible date/time is used. Including optional and alternative components, the syntax is `YYYY[-MM[-DD[{T|:| }hh[:mm[:ss]]]]]`. When the `t=[value]` form is used, value is interpreted as both the start and end value, as described above, giving a range spanning the specified date/time, e.g., `t=[2010]` gives a range spanning the whole of the year 2010.

`outputfile` is an optional argument to specify the name to use for the file output. This is automatically given a `.nc` extension. The default filename is `out.nc`.

A full example would be:

```
$ cis subset solar_3:xglnwa.pm.k8dec-k9nov.col.tm.nc longitude=[0,180],latitude=[0,
↪90] -o Xglnwa-solar_3
```

Gridded netCDF data is output as gridded data, while ungridded and non-netCDF gridded data is output as ungridded data.

Examples

Below are examples of subsetting using each of the supported products (together with a command to plot the output):

```
$ cis subset AO2CO2:RF04.20090114.192600_035100.PNI.nc time=[2009-01-14:19:26:00,2009-
↪01-14:19:36:00] -o RF04-AO2CO2-out
$ cis plot AO2CO2:RF04-AO2CO2-out.nc

$ cis subset IO_RVOD_ice_water_content:2007180125457_06221_CS_2B-CWC-RVOD_GRANULE_P_
↪R04_E02.hdf t=[2007-06-29:13:00,2007-06-29:13:30] -o CloudSAT-out
$ cis plot IO_RVOD_ice_water_content:CloudSAT-out.nc --xaxis=time --yaxis=altitude

$ cis subset Cloud_Top_Temperature:MYD06_L2.A2011100.1720.051.2011102130126.hdf x=[-
↪50,-40],y=[0,10] -o MODIS_L2-out
$ cis plot Cloud_Top_Temperature:MODIS_L2-out.nc

$ cis subset cwp:20080620072500-ESACCI-L2_CLOUD-CLD_PRODUCTS-MODIS-AQUA-fv1.0.nc␣
↪x=[85,90],y=[-3,3] -o Cloud_CCI-out
```

```
$ cis plot atmosphere_mass_content_of_cloud_liquid_water:Cloud_CCI-out.nc

$ cis subset AOD870:20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-
↪fv02.02.nc x=[-5,20],y=[15,25] -o Aerosol_CCI-out
$ cis plot atmosphere_optical_thickness_due_to_aerosol:Aerosol_CCI-out.nc

$ cis subset 440675Angstrom:920801_121229_Abracos_Hill.lev20 t=[2002] -o Aeronet-out
$ cis plot 440675Angstrom:Aeronet-out.nc --xaxis=time --yaxis=440675Angstrom

$ cis subset solar_3:xglnwa.pm.k8dec-k9nov.vprof.tm.nc y=[0,90] -o Xglnwa_vprof-out
$ cis plot solar_3:Xglnwa_vprof-out.nc

$ cis subset solar_3:xglnwa.pm.k8dec-k9nov.col.tm.nc x=[0,180],y=[0,90] -o Xglnwa-out
$ cis plot solar_3:Xglnwa-out.nc

$ cis subset Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.hdf␣
↪x=[0,179.9],y=[0,90] -o MODIS_L3-out
$ cis plot Cloud_Top_Temperature_Mean_Mean:MODIS_L3-out.nc
```

The files used above can be found at:

```
/group_workspaces/jasmin/cis/jasmin_cis_repo_test_files/
  2007180125457_06221_CS_2B-CWC-RVOD_GRANULE_P_R04_E02.hdf
  20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.02.nc
  20080620072500-ESACCI-L2_CLOUD-CLD_PRODUCTS-MODIS-AQUA-fv1.0.nc
  MOD08_E3.A2010009.005.2010026072315.hdf
  MYD06_L2.A2011100.1720.051.2011102130126.hdf
  RF04.20090114.192600_035100.PNI.nc
  xglnwa.pm.k8dec-k9nov.col.tm.nc
  xglnwa.pm.k8dec-k9nov.vprof.tm.nc
/group_workspaces/jasmin/cis/data/aeoronet/AOT/LEV20/ALL_POINTS/
  920801_121229_Abracos_Hill.lev20
```

CHAPTER 8

Aggregation

The Community Intercomparison Suite (CIS) has the ability to aggregate both gridded and ungridded data along one or more coordinates. For example, you might aggregate a dataset over the longitude coordinate to produce an averaged measurement of variation over latitude.

CIS supports 'collapse' of a coordinate - where all values in that dimension are aggregated so that the coordinate no longer exists - and 'aggregation' - where a coordinate is aggregated into bins of fixed size, so that the coordinate still exists but is on a coarser grid. Aggregation is currently only supported for ungridded data. The output of either type of aggregation is always a CF compliant gridded NetCDF file.

The aggregation command has the following syntax:

```
$ cis <collapse|aggregate> <datagroup>[:options] <grid> [-o <outputfile>]
```

where:

<datagroup> is a *CIS datagroup* specifying the variables and files to read and is of the format `<variable>...:<filename>[:product=<productname>]` where:

- `<variable>` is a mandatory variable or list of variables to use.
- `<filenames>` is a mandatory file or list of files to read from.
- `<productname>` is an optional CIS data product to use (see *Data Products*):

See *Datagroups* for a more detailed explanation of datagroups.

<options> Optional arguments given as `keyword=value` in a comma separated list. Options are:

- `kernel=<kernel>` - the method by which the value in each aggregation cell is determined. `<kernel>` should be one of:
  - `sum` - return the sum of all of the data points in that aggregation cell.
  - `mean` - use the mean value of all the data points in that aggregation cell. For gridded data, this mean is weighted to take into account differing cell areas due to the projection of lat/lon lines on the Earth.
  - `min` - use the lowest valid value of all the data points in that aggregate cell.
  - `max` - use the highest valid value of all the data points in that aggregate cell.

  - `moments` - In addition to returning the mean value of each cell (weighted where applicable), this kernel also outputs the number of points used to calculate that mean and the standard deviation of those values, each as a separate variable in the output file.

  If not specified the default is `moments`.

- `product=<productname>` is an optional argument used to specify the type of files being read. If omitted, CIS will attempt to figure out which product to use based on the filename. See *Reading* to see a list of available product names and their file signatures.

<grid> This mandatory argument specifies the coordinates to aggregate over and whether they should be completely collapsed or aggregated into bins. Multiple coordinates can be aggregated over, in which case they should be separated by commas. Coordinates may be identified using their variable names (e.g. `latitude`), standard names, or using the axes shorthands: `x`, `y`, `t`, `z` and `p` which refer to longitude, latitude, time, altitude and pressure respectively.

Note: The axes shorthands are used throughout the examples here, but should be used with care, as the expected coordinate may not always be chosen. For example when specifying 'z' for a gridded hybrid height file, this may well refer to model level number rather than altitude. For this reason it is often safer to use variable names explicitly.

- *Complete collapse* - To perform a complete collapse of a coordinate, simply provide the name of the coordinate(s) as a comma separated list - e.g. `x,y` will aggregate data completely over both latitude and longitude. For ungridded data this will result in length one coordinates with bounds reflecting the maximum and minimum values of the collapsed coordinate.
- *Partial collapse* - To aggregate a coordinate into bins, specify the start, end and step size of those bins in the form `coordinate=[start,end,step]`. The step may be missed out, in which case the bin will span the whole range given. Partial collapse is currently only supported for ungridded data.

  Longitude coordinates are considered to be circular, so that -10 is equivalent to 350. The start and end must describe a monotonically increasing coordinate range, so `x=[90,-90,10]` is invalid, but could be specified using `x=[90,270,10]`. The range between the start and end must not be greater than 360 degrees.

Complete and partial collapses may be mixed where applicable - for example, to completely collapse time and to aggregate latitude on a grid from -45 degrees to 45 degrees, using a step size of 10 degrees:

```
t,y=[-45,45,10]
```

Note: For ungridded data, if a coordinate is left unspecified it is collapsed completely. This is in contrast to gridded data where a coordinate left unspecified is not used in the aggregation at all.

Note: The range specified is the very start and end of the grid, the actual midpoints of the aggregation cells will start at `start + delta/2`.

Date/times:

Date/times are specified in the format: `YYYY-MM-DDThh:mm:ss` in which `YYYY-MM-DD` is a date and `hh:mm:ss` is a time. A colon or space can be used instead of the 'T' separator (but if a space is used, the argument must be quoted). Any trailing components of the date/time may be omitted. When a date/time is used as a range start, the earliest date/time compatible with the supplied components is used (e.g., `2010-04` is treated as `2010-04-01T00:00:00`) and when used as a range end, the latest compatible date/time is used. Including optional and alternative components, the syntax is `YYYY[-MM[-DD[{T|:| }hh[:mm[:ss]]]]]`.

Date/time steps are specified in the ISO 8601 format `PnYnMnDTnHnMnS`, where any particular time period is optional, for example `P1MT30M` would specify a time interval of 1 month and 30 minutes. Years and months are treated as calendar years and months, meaning they are not necessarily fixed in length. For example a date interval of 1 year and 1 month would mean going from 12:00 15th April 2013 to 12:00 15th May 2013. The are two exceptions to this, in rare cases such as starting at 30th January and going forward 1 month, the month is instead treated as a period of 28 days. Also, for the purposes of finding midpoints for the start in a month the month is always treated as 30 days. For example, to start on the 3rd November 2011 at 12:00 and aggregate over each month up to 3rd January 2013 at 12:00:

- `t=[2011-11-03T12:00,2013-01,P1M]`

Multi-dimensional gridded coordinates

Some gridded coordinates can span multiple dimensions, such as hybrid height. These coordinates can be aggregated over as normal, but note that if you only aggregate over a subset of the dimensions a mean kernel will always be used, and no area weighting will be taken into account.

<outputfile> is an optional argument to specify the name to use for the file output. This is automatically given a `.nc` extension if not present. This must not be the same file path as any of the input files. If not supplied, the default filename is `out.nc`.

A full example would be:

```
$ cis aggregate rsutcs:rsutcs_Amon_HadGEM2-A_sstClim_r1i1p1_*.nc:product=NetCDF_
↪Gridded,kernel=mean t,y=[-90,90,20],x -o rsutcs-mean
```

Conditional Aggregation

Sometimes you may want to perform an aggregation over all the points that meet a certain criteria - for example, aggregating satellite data only where the cloud cover fraction is below a certain threshold. This is possible by performing a CIS evaluation on your data first - see *Using Evaluation for Conditional Aggregation*

Aggregation Examples

Ungridded aggregation

Aircraft Track

Original data:

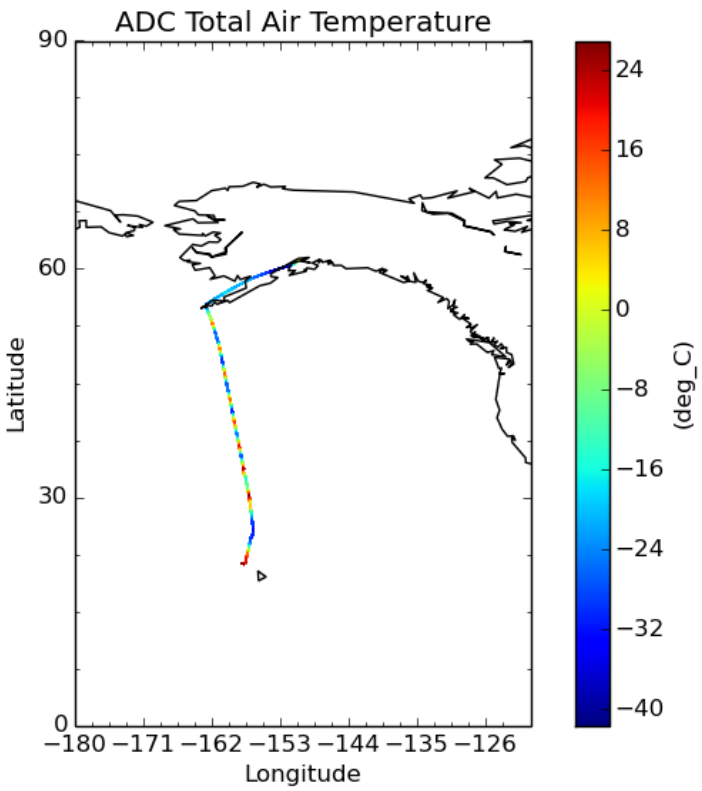
```
$ cis plot TT_A:RF04.20090114.192600_035100.PNI.nc --xmin -180 --xmax -120 --ymin 0 --
↪ymax 90
```

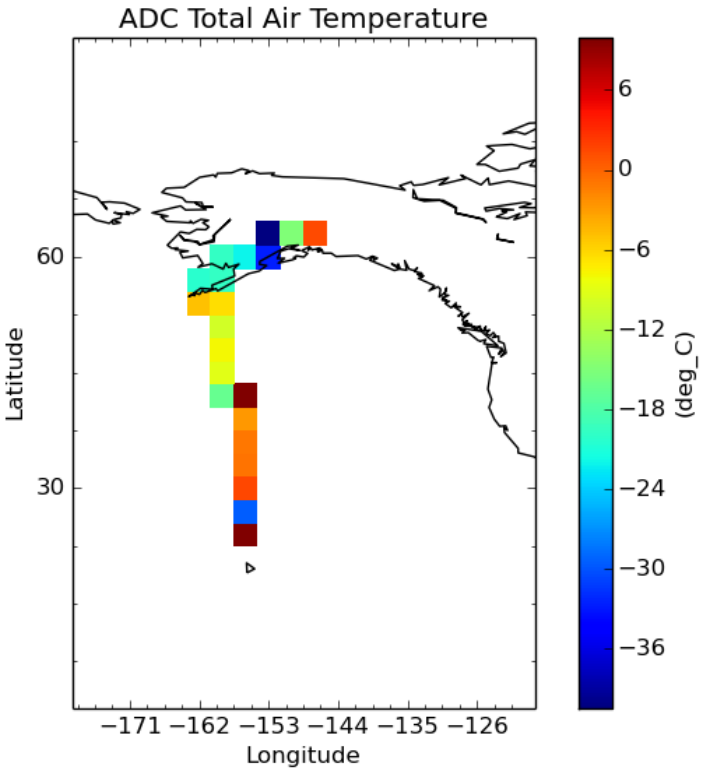
Aggregating onto a coarse grid:

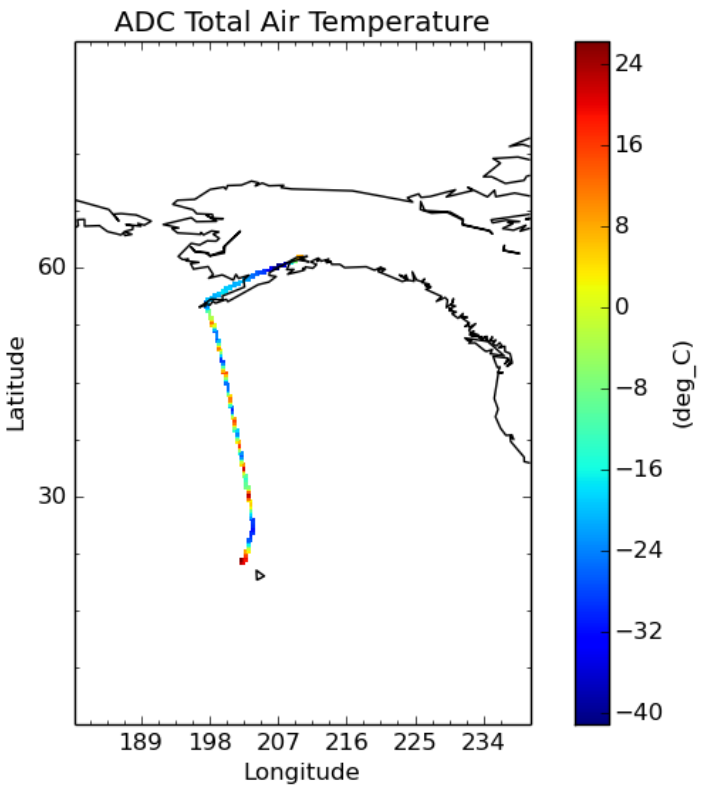
```
$ cis aggregate TT_A:RF04.20090114.192600_035100.PNI.nc x=[-180,-120,3],y=[0,90,3] -o␣
↪NCAR_RAF-1
$ cis plot TT_A:NCAR_RAF-1.nc
```

Aggregating onto a fine grid:

```
$ cis aggregate TT_A:RF04.20090114.192600_035100.PNI.nc x=[180,240,0.3],y=[0,90,0.3] -
↪o NCAR_RAF-2
$ cis plot TT_A:NCAR_RAF-2.nc
```


ADC Total Air Temperature
90
60
30
0
Latitude
−180 −171 −162 −153 −144 −135 −126
Longitude
24
16
8
0
−8
−16
−24
−32
−40
(deg_C)


ADC Total Air Temperature
60
30
Latitude
−171 −162 −153 −144 −135 −126
Longitude
6
0
−6
−12
−18
−24
−30
−36
(deg_C)



Aggregating with altitude and time:

```
$ cis aggregate TT_A:RF04.20090114.192600_035100.PNI.nc t=[2009-01-14T19:30,2009-01-
↪15T03:45,30M],z=[0,15000,1000] -o NCAR_RAF-3
$ cis plot TT_A:NCAR_RAF-3.nc --xaxis time --yaxis altitude
```

Aggregating with altitude and pressure:

```
$ cis aggregate TT_A:RF04.20090114.192600_035100.PNI.nc p=[100,1100,20],z=[0,15000,
↪500] -o NCAR_RAF-4
$ cis plot TT_A:NCAR_RAF-4.nc --xaxis altitude --yaxis air_pressure --logy
```

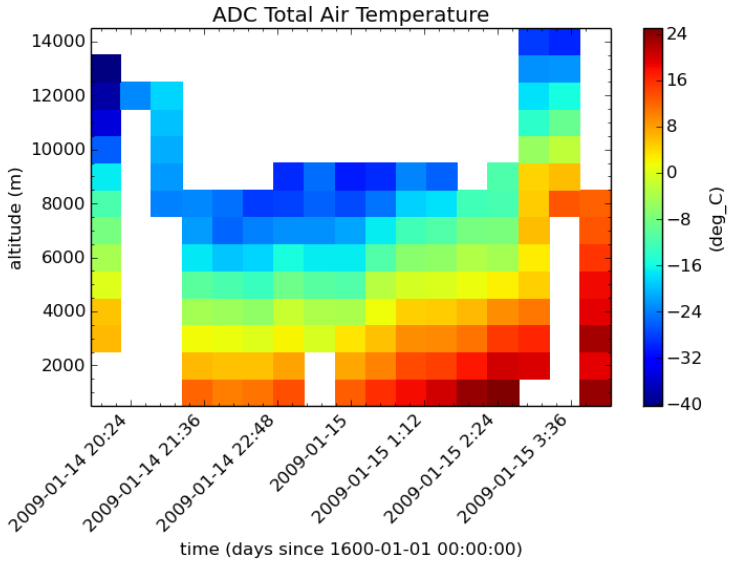
MODIS L3 Data

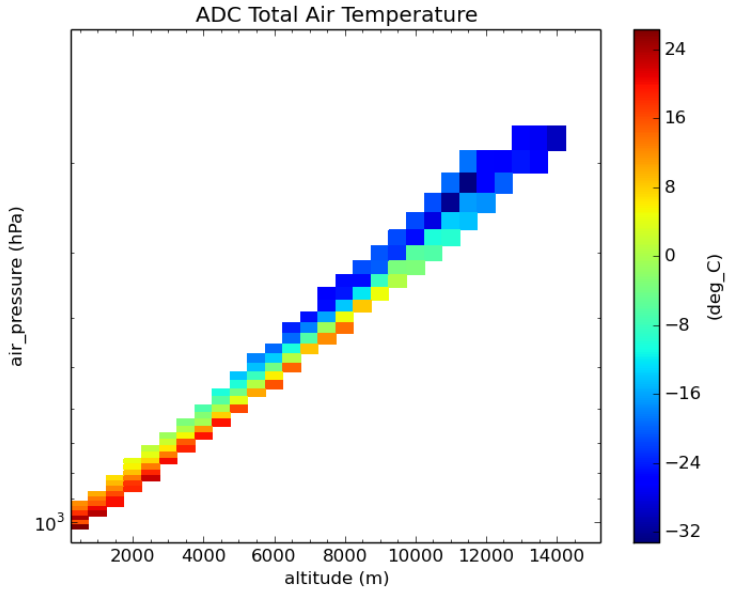
Original data:

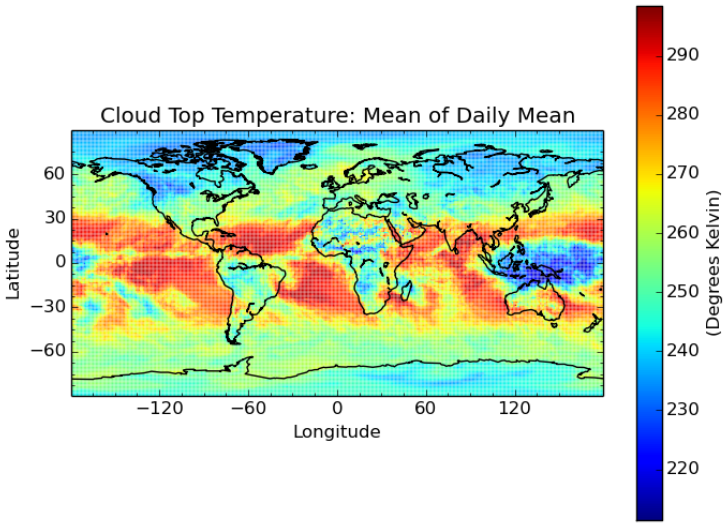
```
$ cis plot Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.hdf
```

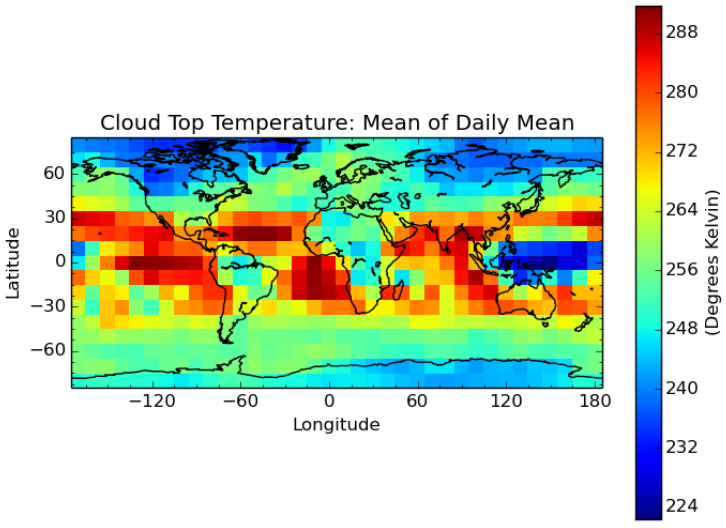
Aggregating with a mean kernel:

```
$ cis aggregate Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.
↪hdf x=[-180,180,10],y=[-90,90,10] -o cloud-mean
$ cis plot Cloud_Top_Temperature_Mean_Mean:cloud-mean.nc
```


ADC Total Air Temperature
altitude (m)
14000
12000
10000
8000
6000
4000
2000
2009-01-14 20:24
2009-01-14 21:36
2009-01-14 22:48
2009-01-15
2009-01-15 1:12
2009-01-15 2:24
2009-01-15 3:36
time (days since 1600-01-01 00:00:00)
24
16
8
0
−8
−16
−24
−32
−40
(deg_C)

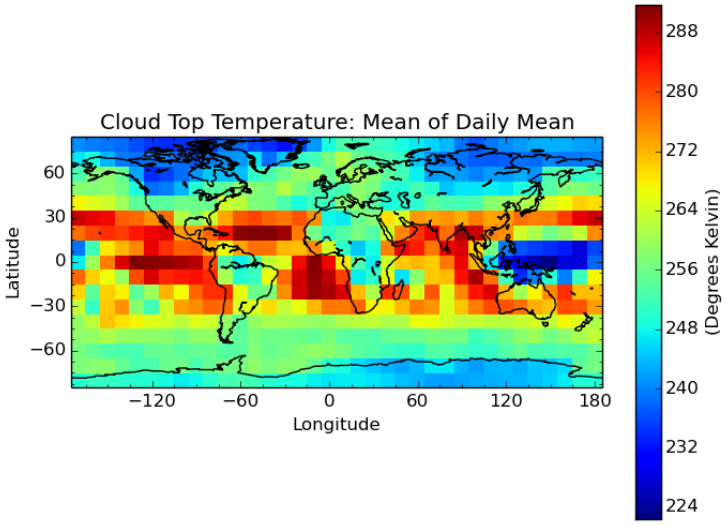

ADC Total Air Temperature
air_pressure (hPa)
10^3
2000
4000
6000
8000
10000
12000
14000
altitude (m)
24
16
8
0
−8
−16
−24
−32
(deg_C)


Cloud Top Temperature: Mean of Daily Mean
Latitude
60
30
0
−30
−60
−120
−60
0
60
120
Longitude
290
280
270
260
250
240
230
220
(Degrees Kelvin)


Cloud Top Temperature: Mean of Daily Mean
Latitude
60
30
0
−30
−60
−120
−60
0
60
120
180
Longitude
288
280
272
264
256
248
240
232
224
(Degrees Kelvin)

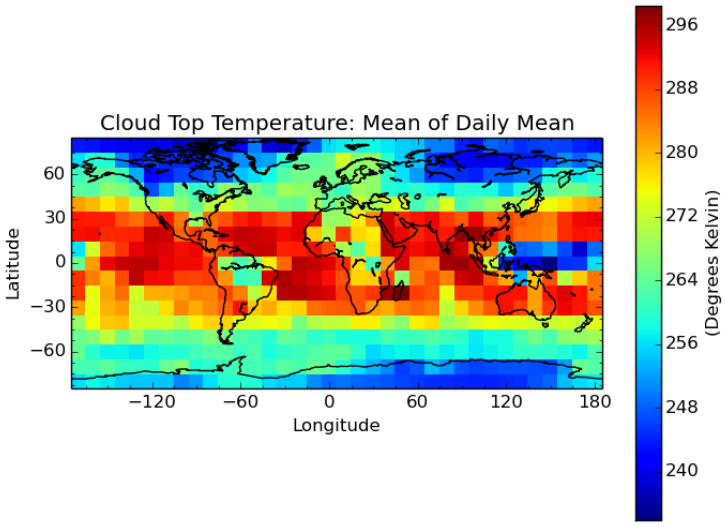
Aggregating with the standard deviation kernel:

```
$ cis aggregate Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.
↪hdf:kernel=stddev x=[-180,180,10],y=[-90,90,10] -o cloud-stddev
$ cis plot Cloud_Top_Temperature_Mean_Mean:cloud-stddev.nc &
```



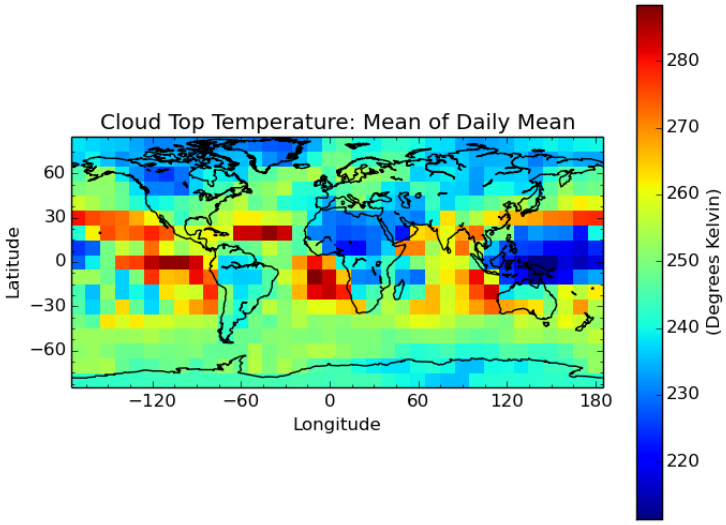
Aggregating with the maximum kernel:

```
$ cis aggregate Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.
↪hdf:kernel=max x=[-180,180,10],y=[-90,90,10] -o cloud-max
$ cis plot Cloud_Top_Temperature_Mean_Mean:cloud-max.nc
```



Aggregating with the minimum kernel:

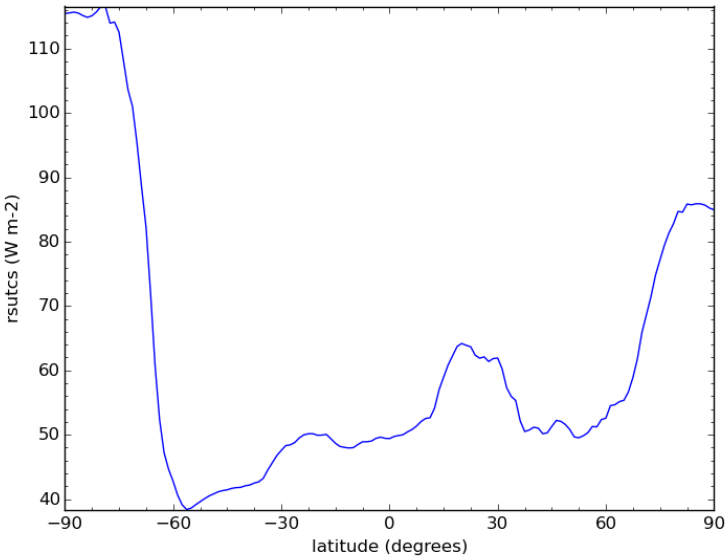
```
$ cis aggregate Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.
↪hdf:kernel=min x=[-180,180,10],y=[-90,90,10] -o cloud-min
$ cis plot Cloud_Top_Temperature_Mean_Mean:cloud-min.nc
```



Gridded collapse

Collapsing 3D model data over time and longitude to produce an averaged measure of variation with latitude:

```
$ cis collapse rsutcs:rsutcs_Amon_HadGEM2-A_sstClim_r1i1p1_185912-188911.
↪nc:kernel=mean t,x -o agg-out.nc
$ cis plot rsutcs:agg-out.nc --xaxis latitude --yaxis rsutcs -o gridded_collapse.png
```



This file can be found in:

```
/group_workspaces/jasmin/cis/data/CMIP5
```

CHAPTER 9

Collocation

One of the key features of the Community Intercomparison Suite (CIS) is the ability to collocate one or more arbitrary data sets onto a common set of coordinates. This page briefly describes how to perform collocation in a number of scenarios.

To perform collocation, run a command of the format:

```
$ cis col <datagroup> <samplegroup> -o <outputfile>
```

where:

<datagroup> is a *CIS datagroup* specifying the variables and files to read and is of the format `<variable>...:<filename>[:product=<productname>]` where:

- `<variable>` is a mandatory variable or list of variables to use.
- `<filenames>` is a mandatory file or list of files to read from.
- `<productname>` is an optional CIS data product to use (see *Data Products*):

See *Datagroups* for a more detailed explanation of datagroups.

<samplegroup> is of the format `<filename>[:<options>]` The available options are described in more detail below. They are entered in a comma separated list, such as `variable=Temperature,collocator=bin,kernel=mean`. Not all combinations of collocator and data are available; see *Available Collocators*.

- `<filename>` is a single filename with the points to collocate onto.
- `variable` is an optional argument used to specify which variable's coordinates to use for collocation. If a variable is specified, a missing value will be set in the output file at every point for which the sample variable has a missing value. If a variable is not specified, non-missing values will be set at all sample points unless collocation at a point does not result in a valid value. This can be overridden by using the `missing_data_for_missing_sample` argument described below.
- `collocator` is an optional argument that specifies the collocation method. Parameters for the collocator, if any, are placed in square brackets after the collocator name, for example,

`collocator=box[fill_value=-999,h_sep=1km]`. If not specified, a *Default Collocator* is identified for your data / sample combination. The collocators available are:

- `bin` For use only with ungridded data and gridded sample points. Data points are placed in bins corresponding to the cell bounds surrounding each grid point. The bounds are taken from the gridded data if they are defined, otherwise the mid-points between grid points are used. The binned points should then be processed by one of the kernels to give a numeric value for each bin.
- `box` For use with gridded and ungridded sample points and data. A search region is defined by the parameters and points within the defined separation of each sample point are associated with the point. The points should then be processed by one of the kernels to give a numeric value for each bin. The parameters defining the search box are:
  - `h_sep` - the horizontal separation. The units can be specified as km or m (for example `h_sep=1.5km`); if none are specified then the default is km.
  - `a_sep` - the altitude separation. The units can be specified as km or m, as for h_sep; if none are specified then the default is m.
  - `p_sep` - the pressure separation. This is not an absolute separation as for h_sep and a_sep, but a relative one, so is specified as a ratio. For example a constraint of p_sep = 2, for a point at 10 hPa, would cover the range 5 hPa < points < 20 hPa. Note that p_sep >= 1.
  - `t_sep` - the time separation. This can be specified in years, months, days, hours, minutes or seconds using `PnYnMnDTnHnMnS` (the T separator can be replaced with a colon or a space, but if using a space quotes are required). For example to specify a time separation of one and a half months and thirty minutes you could use `t_sep=P1M15DT30M`. It is worth noting that the units for time comparison are fractional days, so that years are converted to the number of days in a Gregorian year, and months are 1/12th of a Gregorian year.

  If `h_sep` is specified, a k-d tree index based on longitudes and latitudes of data points is used to speed up the search for points. It h_sep is not specified, an exhaustive search is performed for points satisfying the other separation constraints.
- `lin` For use with gridded source data only. A value is calculated by linear interpolation for each sample point. The extrapolation mode can be controlled with the `extrapolate` keyword. The default mode is not to extrapolate values for sample points outside of the gridded data source (masking them in the output instead). Setting `extrapolate=True` will override this and instruct the kernel to extrapolate these values outside of the data source instead.
- `nn` For use with gridded source data only. The data point closest to each sample point is found, and the data value is set at the sample point. As with linear interpolation the extrapolation mode can be controlled with the `extrapolate` keyword.
- `dummy` For use with ungridded data only. Returns the source data as the collocated data irrespective of the sample points. This might be useful if variables from the original sample file are wanted in the output file but are already on the correct sample points.

Collocators have the following general optional parameters, which can be used in addition to any specific ones listed above:

- `fill_value` - The numerical value to apply to the collocated point if there are no points which satisfy the constraint.
- `var_name` - Specifies the name of the variable in the resulting NetCDF file.
- `var_long_name` - Specifies the variable's long name.
- `var_units` - Specifies the variable's units.

  - `missing_data_for_missing_sample` - Allows the user to specify explicitly whether masked sample data points should be used for sampling. This only applies when a variable has been specified in the samplegroup.

- `kernel` is used to specify the kernel to use for collocation methods that create an intermediate set of points for further processing, that is box and bin. The default kernel for box and bin is *moments*. The built-in kernel methods currently available are:
  - `moments` - **Default**. This is an averaging kernel that returns the mean, standard deviation and the number of points remaining after the specified constraint has been applied. This can be used for gridded or ungridded sample points where the collocator is one of 'bin' or 'box'. The names of the variables in the output file are the name of the input variable with a suffix to identify which quantity they represent:
    - *Mean* - no suffix - the mean value of all data points which were mapped to that sample grid point (data points with missing values are excluded)
    - *Standard Deviation* - suffix: `_std_dev` - The corrected sample standard deviation (i.e. 1 degree of freedom) of all the data points mapped to that sample grid point (data points with missing values are excluded)
    - *Number of points* - suffix: `_num_points` - The number of data points mapped to that sample grid point (data points with missing values are excluded)
  - `mean` - an averaging kernel that returns the mean values of any points found by the collocation method
  - `nn_t` (or `nn_time`) - nearest neighbour in time algorithm
  - `nn_h` (or `nn_horizontal`) - nearest neighbour in horizontal distance
  - `nn_a` (or `nn_altitude`) - nearest neighbour in altitude
  - `nn_p` (or `nn_pressure`) - nearest neighbour in pressure (as in a vertical coordinate). Note that similarly to the `p_sep` constraint that this works on the ratio of pressure, so the nearest neighbour to a point with a value of 10 hPa, out of a choice of 5 hPa and 19 hPa, would be 19 hPa, as 19/10 < 10/5.
- `product` is an optional argument used to specify the type of files being read. If omitted, the program will attempt to determine which product to use based on the filename, as listed at *Reading*.

<outputfile> is an optional argument specifying the file to output to. This will be automatically given a `.nc` extension if not present. This must not be the same file path as any of the input files. If not provided, the default output filename is *out.nc*

A full example would be:

```
$ cis col rain:"my_data_??.*" my_sample_file:collocator=box[h_sep=50km,t_sep=6000S],
↪kernel=nn_t -o my_col
```

Warning: When collocating two data sets with different spatio-temporal domains, the sampling points should be within the spatio-temporal domain of the source data. Otherwise, depending on the collocation options selected, strange artifacts can occur, particularly with linear interpolation. Spatio-temporal domains can be reduced in CIS with *Aggregation* or *Subsetting*.

Available Collocators and Kernels

Collocation type			
(data -> sample)	Available Collocators	Default Collocator	Default Kernel
Gridded -> gridded	`lin`, `nn`, `box`	`lin`	*None*
Ungridded -> gridded	`bin`, `box`	`bin`	`moments`
Gridded -> ungridded	`lin`, `nn`	`lin`	*None*
Ungridded -> ungridded	`box`	`box`	`moments`

Collocation output files

Output data files are suffixed with `.nc` (so there is no need to specify the extension in the output parameter).

It is worth noting that in the process of collocation all of the data and sample points are represented as 1-d lists, so any structural information about the input files is lost. This is done to ensure consistency in the collocation output. This means, however, that input files which may have been plotable as, for example, a heatmap may not be after collocation. In this situation plotting the data as a scatter plot will yield the required results.

Each collocated output variable has a history attributed created (or appended to) which contains all of the parameters and file names which went into creating it. An example might be:

```
double mass_fraction_of_cloud_liquid_water_in_air(pixel_number) ;
    ...
    mass_fraction_of_cloud_liquid_water_in_air:history = "Collocated onto sampling␣
↪from:   [\'/test/test_files/RF04.20090114.192600_035100.PNI.nc\'] using CIS version␣
↪V0R4M4\n",
        "variable: mass_fraction_of_cloud_liquid_water_in_air\n",
        "with files: [\'/test/test_files/xenida.pah9440.nc\']\n",
        "using collocator: DifferenceCollocator\n",
        "collocator parameters: {}\n",
        "constraint method: None\n",
        "constraint parameters: None\n",
        "kernel: None\n",
        "kernel parameters: None" ;
    mass_fraction_of_cloud_liquid_water_in_air:shape = 30301 ;
double difference(pixel_number) ;
    ...
```

Writing your own plugins

The collocation framework was designed to make it easy to write your own plugins. Plugins can be written to create new kernels, new constraint methods and even whole collocation methods. See the *analysis plugin development* section for more details.

CHAPTER 10

Collocation Examples

Ungridded to Ungridded Collocation Examples

Ungridded data with vertical component

First subset two Caliop data files:

```
$ cis subset Temperature:CAL_LID_L2_05kmAPro-Prov-V3-01.2009-12-31T23-36-08ZN.hdf␣
↪x=[170,180],y=[60,80],z=[28000,29000],p=[13,15] -o 2009
$ cis subset Temperature:CAL_LID_L2_05kmAPro-Prov-V3-01.2010-01-01T00-22-28ZD.hdf␣
↪x=[170,180],y=[60,80],z=[28000,29000],p=[12,13.62] -o 2010
```

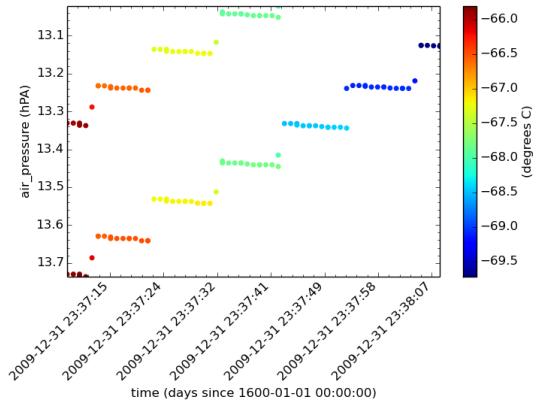
Results of subset can be plotted with:

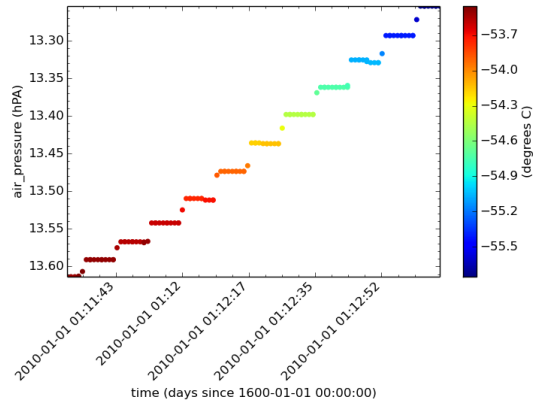
```
$ cis plot Temperature:2009.nc --itemwidth 25 --xaxis time --yaxis air_pressure
$ cis plot Temperature:2010.nc --itemwidth 25 --xaxis time --yaxis air_pressure
```

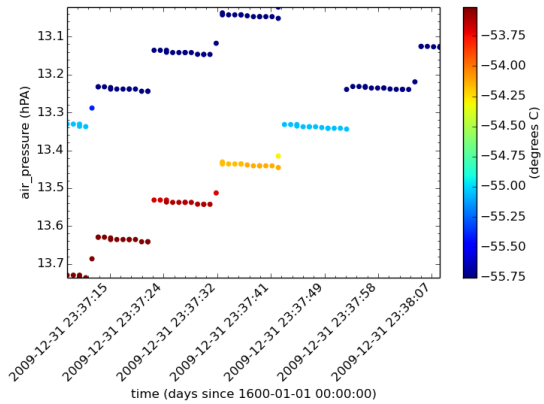
Then collocate data, and plot output:

```
$ cis col Temperature:2010.nc 2009.nc:collocator=box[p_sep=1.1],kernel=nn_p
$ cis plot Temperature:out.nc --itemwidth 25 --xaxis time --yaxis air_pressure
```

The output for the two subset data files, and the collocated data should look like:







File Locations

The files used above can be found at:

```
/group_workspaces/jasmin/cis/data/caliop/CAL-LID-L2-05km-APro
```

Ungridded data collocation using k-D tree indexing

These examples show the syntax for using the k-D tree optimisation of the separation constraint. The indexing is only by horizontal position.

Nearest-Neighbour Kernel

The first example is of Aerosol CCI data on to the points of a MODIS L3 file (which is an ungridded data file but with points lying on a grid).

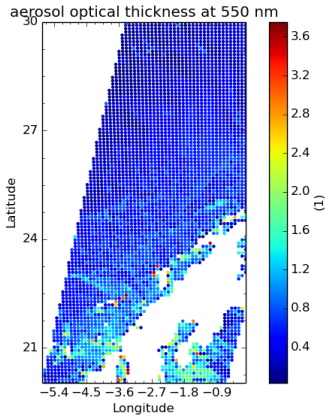
Subset to a relevant region:

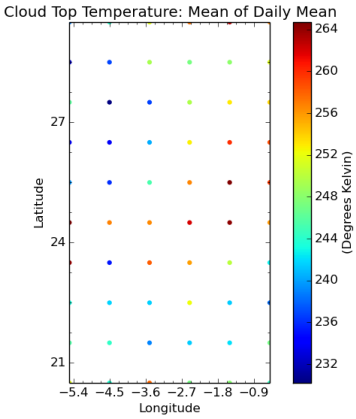
```
$ cis subset AOD550:20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-
↪fv02.02.nc x=[-6,0],y=[20,30] -o AOD550n_3
$ cis subset Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.hdf␣
↪x=[-6,0],y=[20,30] -o MOD08n_3
```

The results of subsetting can be plotted with:

```
$ cis plot AOD550:AOD550n_3.nc --itemwidth 10
$ cis plot Cloud_Top_Temperature_Mean_Mean:MOD08n_3.nc --itemwidth 20
```

These should look like:





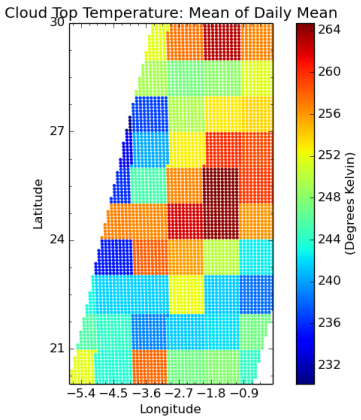
To collocate with the nearest-neighbour kernel use:

```
$ cis col Cloud_Top_Temperature_Mean_Mean:MOD08n_3.nc AOD550n_3.nc:collocator=box[h_
↪sep=150],kernel=nn_h -o MOD08_on_AOD550_nn_kdt
```

This can be plotted with:

```
$ cis plot Cloud_Top_Temperature_Mean_Mean:MOD08_on_AOD550_nn_kdt.nc --itemwidth 10
```

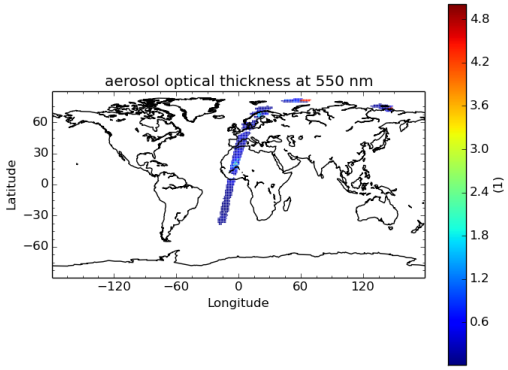
The sample points are more closely spaced than the data points, hence a patchwork effect is produced.



Collocating the full Aerosol CCI file on to the MODIS L3 with:

```
$ cis col AOD550:20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.
↪02.nc MOD08_E3.A2010009.005.2010026072315.hdf:variable=Cloud_Top_Temperature_Mean_
↪Mean,collocator=box[h_sep=150],kernel=nn_h -o AOD550_on_MOD08_kdt_nn_full
```

gives the following result



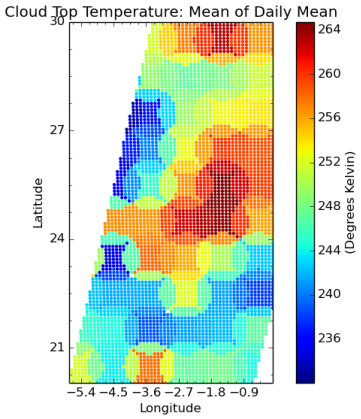
Mean Kernel

This example is similar to the first nearest-neighbour collocation above:

```
$ cis col Cloud_Top_Temperature_Mean_Mean:MOD08n_3.nc AOD550n_3.nc:collocator=box[h_
↪sep=75],kernel=mean -o MOD08_on_AOD550_hsep_75km
```

Plotting this again gives a granular result:

```
$ cis plot Cloud_Top_Temperature_Mean_Mean:MOD08_on_AOD550_hsep_75km.nc --itemwidth 10
```

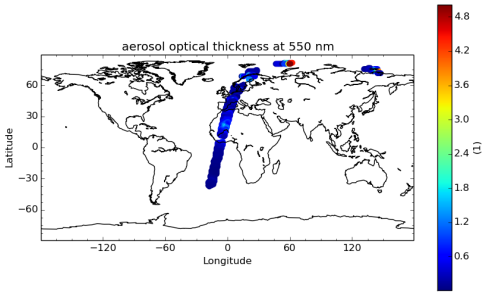


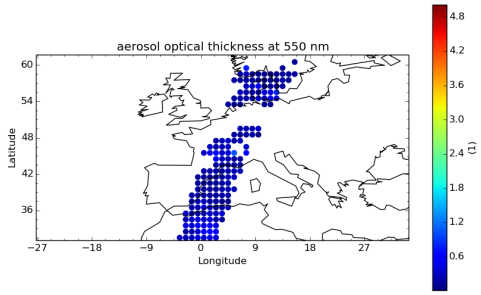
This example collocates the Aerosol CCI data on to the MODIS L3 grid:

```
$ cis col AOD550:20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.
↪02.nc MOD08_E3.A2010009.005.2010026072315.hdf:variable=Cloud_Top_Temperature_Mean_
↪Mean,collocator=box[h_sep=50,fill_value=-999],kernel=mean -o AOD550_on_MOD08_kdt_
↪hsep_50km_full
```

This can be plotted as follows, with the full image and zoomed into a representative section show below:

```
$ cis plot AOD550:AOD550_on_MOD08_kdt_hsep_50km_full.nc --itemwidth 50
```



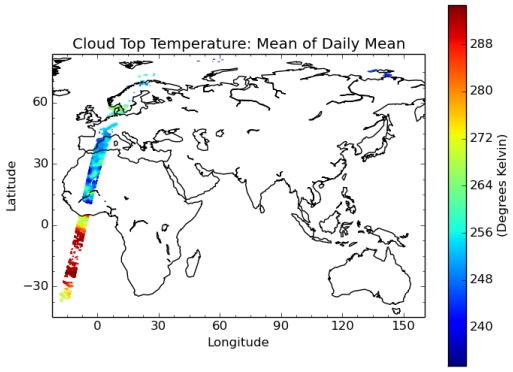


The reverse collocation can be performed with this command (taking about 7 minutes):

```
$ cis col Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.hdf␣
↪20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.02.
↪nc:variable=AOD550,collocator=box[h_sep=100,fill_value=-999],kernel=mean -o MOD08_
↪on_AOD550_kdt_hsep_100km_var_full
```

Plotting it with this command gives the result below:

```
$ cis plot Cloud_Top_Temperature_Mean_Mean:MOD08_on_AOD550_kdt_hsep_100km_var_full.nc
```

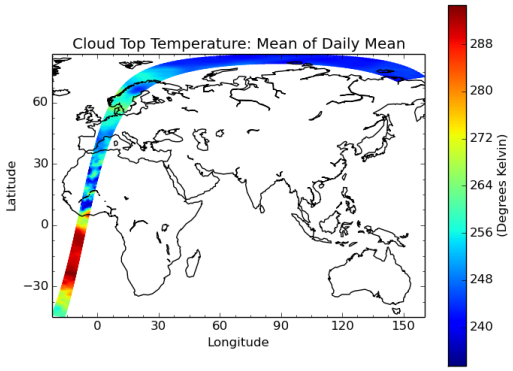


Omitting the variable option in the sample group gives collocated values over a full satellite track (taking about 30 minutes):

```
$ cis col Cloud_Top_Temperature_Mean_Mean:MOD08_E3.A2010009.005.2010026072315.hdf␣
↪20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.02.
↪nc:collocator=box[h_sep=100,fill_value=-999],kernel=mean -o MOD08_on_AOD550_kdt_
↪hsep_100km_full
```

Plotting it with this command gives the result below:

```
$ cis plot Cloud_Top_Temperature_Mean_Mean:MOD08_on_AOD550_kdt_hsep_100km_full.nc
```



File Locations

The files used above can be found at:

```
/group_workspaces/jasmin/cis/jasmin_cis_repo_test_files/
  20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.02.nc
  MOD08_E3.A2010009.005.2010026072315.hdf
```

Examples of collocation of ungridded data on to gridded

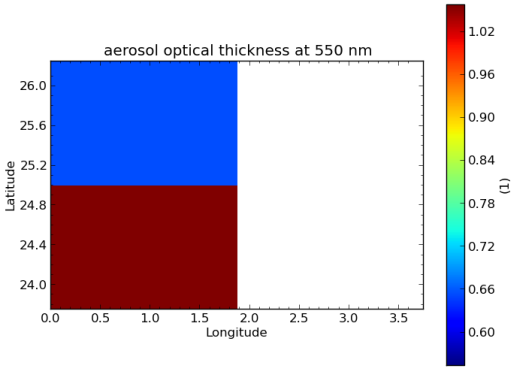
Simple Example of Aerosol CCI Data on to a 4x4 Grid

This is a trivial example that collocates on to a 4x4 spatial grid at a single time:

```
$ cis subset tas:tas_day_HadGEM2-ES_rcp45_r1i1p1_20051201-20151130.nc x=[0,2],y=[24,
↪26],t=[2008-06-12T1,2008-06-12] -o tas_day_HadGEM2-ES_rcp45_r1i1p1_20051201-
↪20151130.nc -o tas_1

$ cis subset AOD550:20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-
↪fv02.02.nc x=[0,2],y=[24,26] -o AOD550n_1

$ cis col AOD550:AOD550n_1.nc tas_1.nc:collocator=bin[fill_value=-9999.0],kernel=mean␣
↪-o AOD550_on_tas_1

$ cis plot AOD550:AOD550_on_tas_1.nc
```

Note that for ungridded gridded collocation, and the collocator must be one bin or box and a kernel such as "mean" must be used.

The plotted image looks like:



Aerosol CCI with Three Time Steps

This example involves collocation on to a grid with three time steps. The ungridded data all has times within the middle step, so the output has missing values for all grid points with the time equal to the first or third value. This can be seen using ncdump:

```
$ cis subset tas:tas_day_HadGEM2-ES_rcp45_r1i1p1_20051201-20151130.nc x=[-6,-.0001],
↪y=[20,30],t=[2008-06-11T1,2008-06-13] -o tas_3day

$ cis subset AOD550:20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-
↪fv02.02.nc x=[-6,0],y=[20,30] -o AOD550n_3
```

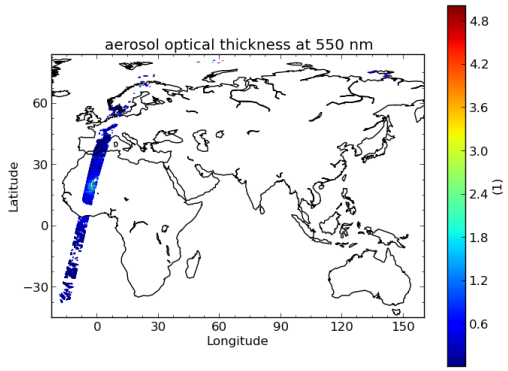
```
$ cis col AOD550:AOD550n_3.nc tas_3day.nc:collocator=bin[fill_value=-9999.0],
↪kernel=mean -o AOD550_on_tas_3day

$ ncdump AOD550_on_tas_3day.nc |less
```

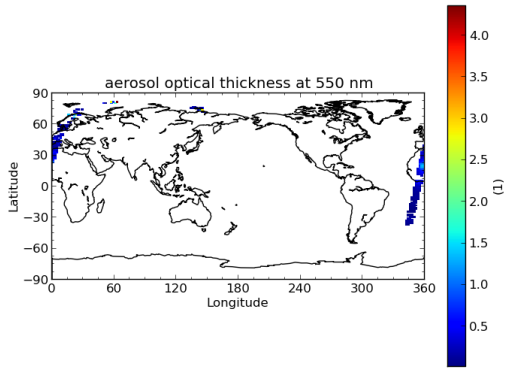
Aerosol CCI with One Time Step

This is as above but subsetting the grid to one time step so that the output can be plotted directly:

```
$ cis subset tas:tas_day_HadGEM2-ES_rcp45_r1i1p1_20051201-20151130.nc t=[2008-06-12T1,
↪2008-06-12] -o tas_2008-06-12

$ cis col AOD550:20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.
↪02.nc tas_2008-06-12.nc:collocator=bin[fill_value=-9999.0],kernel=mean -o AOD550_on_
↪tas_1day

$ cis plot AOD550:AOD550_on_tas_1day.nc
$ cis plot AOD550:20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.
↪02.nc
$ cis plot tas:tas_2008-06-12.nc
```

These are the plots before and after collocation:



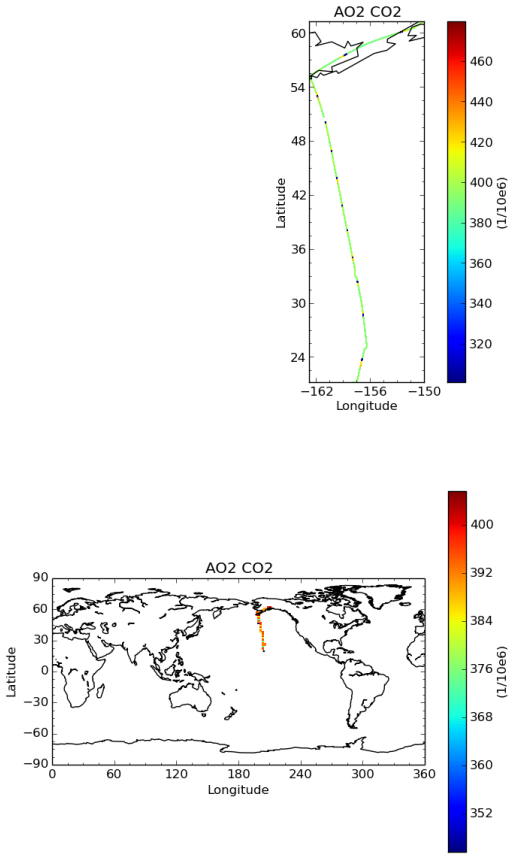


Example with NCAR RAF Data

This example uses the data in RF04.20090114.192600_035100.PNI.nc. However, this file does not have standard_name or units accepted as valid by Iris. These were modified using ncdump and ncgen, giving RF04_fixed_AO2CO2.nc:

```
$ cis subset tas:tas_day_HadGEM2-ES_rcp45_r1i1p1_20051201-20151130.nc t=[2009-01-14T1,
↪2009-01-14] -o tas_2009-01-14

$ cis col AO2CO2:RF04_fixed_AO2CO2.nc tas_2009-01-14.nc:collocator=bin[fill_value=-
↪9999.0],kernel=mean -o RF04_on_tas

$ cis plot AO2CO2:RF04_on_tas.nc:product=NetCDF_Gridded
```

These are the plots before and after collocation:



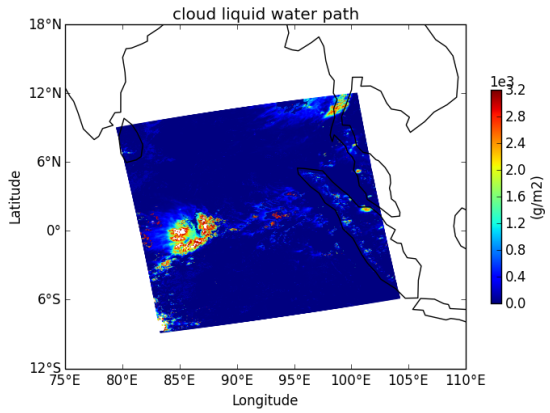
Cloud CCI with One Time Step

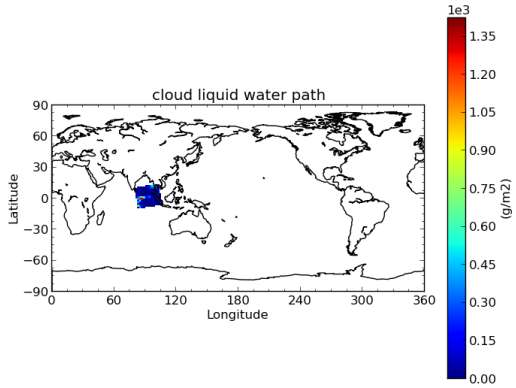
This is analogous to the Aerosol CCI example:

```
$ cis subset tas:tas_day_HadGEM2-ES_rcp45_r1i1p1_20051201-20151130.nc t=[2008-06-20T1,
↪2008-06-20] -o tas_2008-06-20

$ cis col cwp:20080620072500-ESACCI-L2_CLOUD-CLD_PRODUCTS-MODIS-AQUA-fv1.0.nc tas_
↪2008-06-20.nc:collocator=bin[fill_value=-9999.0],kernel=mean -o Cloud_CCI_on_tas
```

```
$ cis plot cwp:Cloud_CCI_on_tas.nc
$ cis plot cwp:20080620072500-ESACCI-L2_CLOUD-CLD_PRODUCTS-MODIS-AQUA-fv1.0.nc
```

These are the plots before and after collocation:





File Locations

The files used above can be found at:

```
/group_workspaces/jasmin/cis/jasmin_cis_repo_test_files/
  20080612093821-ESACCI-L2P_AEROSOL-ALL-AATSR_ENVISAT-ORAC_32855-fv02.02.nc
  20080620072500-ESACCI-L2_CLOUD-CLD_PRODUCTS-MODIS-AQUA-fv1.0.nc
  RF04.20090114.192600_035100.PNI.nc
/group_workspaces/jasmin/cis/example_data/
  RF04_fixed_AO2CO2.nc
/group_workspaces/jasmin/cis/gridded-test-data/cmip5.output1.MOHC.HadGEM2-ES.rcp45.
↪day.atmos.day.r1i1p1.v20111128/
  tas_day_HadGEM2-ES_rcp45_r1i1p1_20051201-20151130.nc
```

Examples of Gridded to Gridded Collocation

Example of Gridded Data onto a Finer Grid

First to show original data subset to a single time slice:

```
$ cis subset rsutcs:rsutcs_Amon_HadGEM2-A_sstClim_r1i1p1_185912-188911.nc t=[1859-12-
↪12] -o sub1
```

Plot for subset data:

```
$ cis plot rsutcs:sub1.nc
```

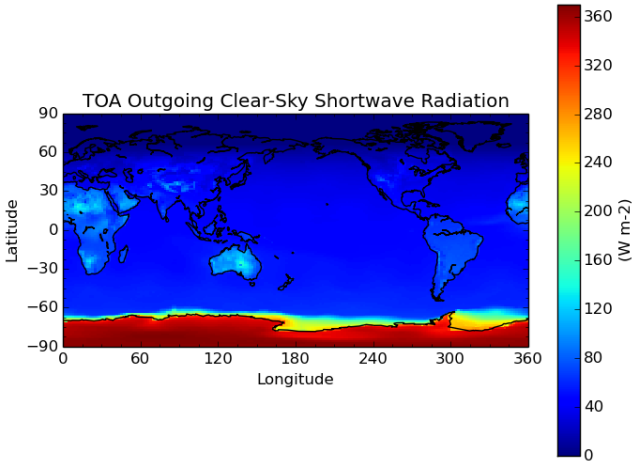
Collocate onto a finer grid, which was created using nearest neighbour:

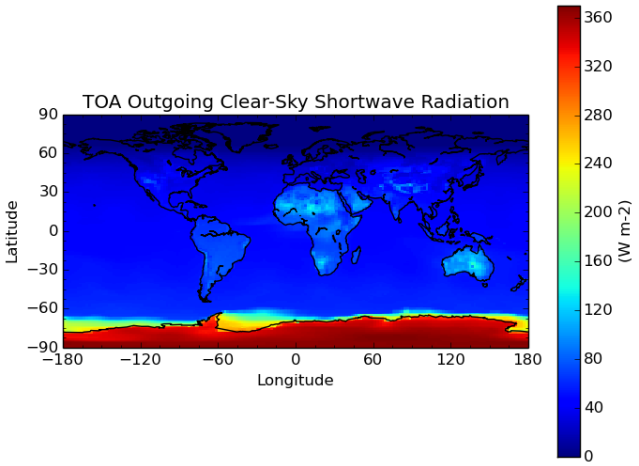
```
$ cis col rsutcs:rsutcs_Amon_HadGEM2-A_sstClim_r1i1p1_185912-188911.nc dummy_high_res_
↪cube_-180_180.nc:collocator=nn -o 2
$ cis subset rsutcs:2.nc t=[1859-12-12] -o sub2
$ cis plot rsutcs:sub2.nc
```

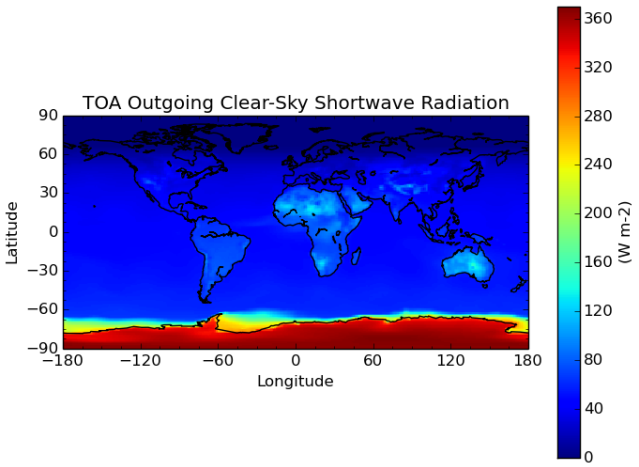
Collocate onto a finer grid, which was created using linear interpolation:

```
$ cis col rsutcs:rsutcs_Amon_HadGEM2-A_sstClim_r1i1p1_185912-188911.nc dummy_high_res_
↪cube_-180_180.nc:collocator=lin -o 3
$ cis subset rsutcs:3.nc t=[1859-12-12] -o sub3
$ cis plot rsutcs:sub3.nc
```

Before, after nearest neighbour and after linear interpolation:







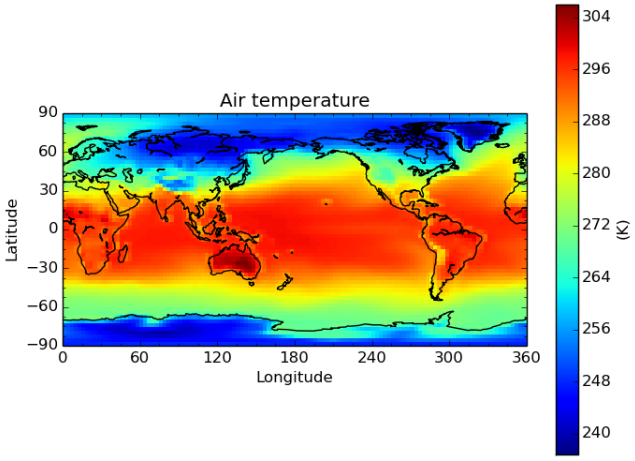
4D Gridded Data with latitude, longitude, air_pressure and time to a New Grid

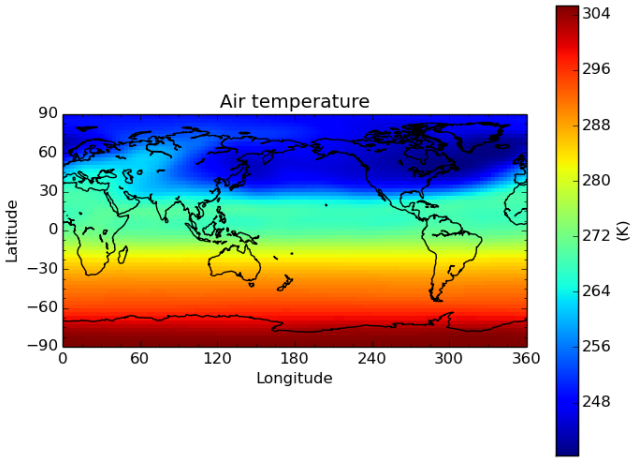
```
$ cis col temp:aerocom.INCA.A2.RAD-CTRL.monthly.temp.2006-fixed.nc dummy_low_res_cube_
↪4D.nc:collocator=lin -o 4D-col
```

Note the file `aerocom.INCA.A2.RAD-CTRL.monthly.temp.2006-fixed.nc` has the standard name of `presnivs` changed to `air_pressure`, in order to be read correctly.

Slices at Different Pressures

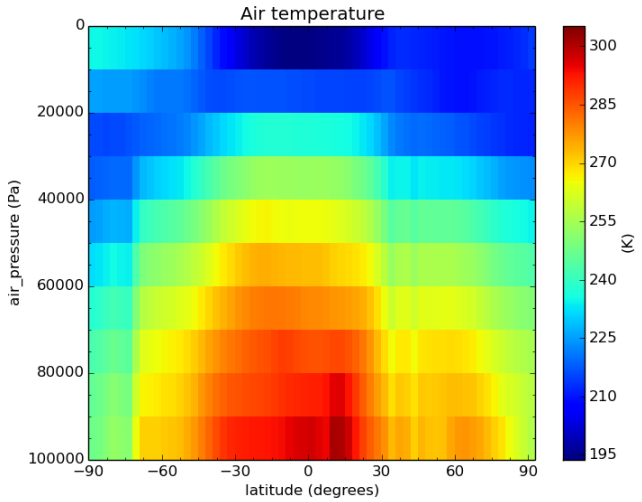
```
$ cis subset temp:4D-col.nc t=[2006-01],z=[100000] -o sub9
$ cis plot temp:sub9.nc
$ cis subset temp:4D-col.nc t=[2006-01],z=[0] -o sub10
$ cis plot temp:sub10.nc
```

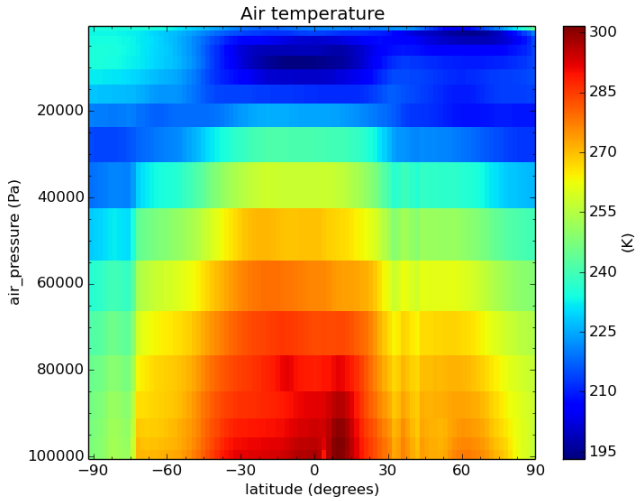




Pressure against time

```
$ cis subset temp:4D-col.nc x=[0],t=[2006-01] -o sub11
$ cis plot temp:sub11.nc --xaxis latitude --yaxis air_pressure
$ cis subset temp:aerocom.INCA.A2.RAD-CTRL.monthly.temp.2006-fixed.nc x=[0],t=[2006-
↪01] -o sub12
$ cis plot temp:sub12.nc --xaxis latitude --yaxis air_pressure
```





File Locations

The files used above can be found at:

```
/group_workspaces/jasmin/cis/sprint_reviews/SR4-IB/gridded_col2
```

CHAPTER 11

Plotting

Plotting is straightforward:

```
$ cis plot variable:filenames
```

This will attempt to locate the variable `variable` in all of the specified `filenames`, work out its metadata, such as units, labels, etc. and the appropriate chart type to plot, so that a line graph is used for two dimensional data, a scatter plot is used for three dimensional ungridded data and a heatmap for three dimensional gridded data. Other types of chart can be specified using the `--type` option. Available types are:

`line` a simple line plot, for three dimensional data the third variable is represented by the line colour

`scatter` a scatter plot

`scatter2d` a scatter plot with two coordinate axis and the data represented by the colour of the marker

`heatmap` a heatmap especially suitable for gridded data

`contour` a standard contour plot, see *contour options*

`contourf` a filled contour plot, see *contour options*

`histogram2d`

`histogram`

`comparativescatter` allows two variables to be plotted against each other, specified as `cis plot variable1:filename1 variable2:filename2 --type comparativescatter`

`taylor` a Taylor diagram for comparing collocated datasets. See Taylor, K. E. (2001), 'Summarizing multiple aspects of model performance in a single diagram', J. Geophys. Res., 106(D7), 7183–7192, doi:10.1029/2000JD900719 for a detailed description.

Note that `filenames` is a non-optional argument used to specify the files to read the variable from. These can be specified as a comma separated list of the following possibilities:

1. A single filename - this should be the full path to the file
2. A single directory - all files in this directory will be read

3. A wildcarded filename - A filename with any wildcards compatible with the python module glob, so that *, ? and [] can all be used. For example `/path/to/my/test*file_[0-9]`.

Note that when using option 2, the filenames in the directory will be automatically sorted into alphabetical order. When using option 3, the filenames matching the wildcard will also be sorted into alphabetical order. The order of the comma separated list will however remain as the user specified, e.g.:

```
$ cis plot $var:filename1,filename2,wildc*rd,/my/dir/,filename3
```

would read `filename1`, then `filename2`, then all the files that match `wildc*rd` (in alphabetical order), then all the files in the directory `/my/dir/` (in alphabetical order) and then finally `filename3`.

Plot Options

There are a number of optional arguments, which should be entered as a comma separated list after the mandatory arguments, for example `variable:filename:product=Cis,edgecolor=black`. The options are:

color colour of markers, e.g. for scatter plot points or contour lines, see *Available Colours and Markers*

cmap colour map to use, e.g. for contour lines or heatmap, see *Available Colours and Markers*

vmin the minimum value for the colourmap

vmax the maximum value for the colourmap

edgecolor colour of scatter marker edges (can be used to differentiate scatter markers with a colourmap from the background plot)

itemstyle shape of scatter marker, see *Available Colours and Markers*

itemwidth width of an item. Units are points in the case of a line, and points^2 in the case of a scatter point

label name of datagroup for the legend

product the data product to use for the plot

type the type of plot for that layer. This can't be set if the global type has been set.

alpha the transparancy of that layer

cbarlabel The label for the colorbar

cbarorient The orientation of the colour bar, either horizontal or vertical

nocolourbar Hides the colour bar on a 3D plot

cbarscale this can be used to change the size of the colourbar when plotting and defaults to 0.55 for vertical colorbars, 1.0 for horizontal.

Additional datagroup options for contour plots only:

contnlevels the number of levels for the contour plot

contlevels a list of levels for the contour plot, e.g. `contlevels=[0,1,3,10]`

contlabel options are `true` or `false`, if true then contour labels are shown

contwidth width of the contour lines

Note that `label` refers to the label the plot will have on the legend, for example if a multi-series line graph or scatter plot is plotted. To set the labels of the axes, use `--xlabel` and `--ylabel`. `--cbarlabel` can be used to set the label on the colour bar.

The axes can be specified with `--xaxis` and `--yaxis`. Gridded data supports any coordinate axes available in the file, while ungridded data supports the following coordinate options (if available in the data):

- `latitude`
- `longitude`
- `time`
- `altitude`
- `air_pressure`
- `variable` - the variable being plotted

If the product is not specified, the program will attempt to figure out which product should be used based on the filename. See *What kind of data can CIS deal with?* to see a list of available products and their file signatures, or run `cis plot -h`.

Saving to a File

By default a plot will be displayed on screen. To save it to an image file instead, use the `--output` option. Available output types are png, pdf, ps, eps and svg, which can be selected using the appropriate filename extension, for example `--output plot.svg`.

Plot Formatting

There are a number of plot formatting options available:

`--xlabel` The label for the x axis

`--ylabel` The label for the y axis

`--title` The title of the plot

`--fontsize` The size of the font in points

`--cmap` The colour map to be used when plotting a 3D plot, see *Available Colours and Markers*

`--projection` The projection to use for the map-plot. All Cartopy projections are supported, see http://scitools.org.uk/cartopy/docs/latest/crs/projections.html for a full list.

`--height` The height of the plot, in inches

`--width` The width of the plot, in inches

`--xbins` The number of bins on the x axis of a histogram

`--ybins` The number of bins on the y axis of a histogram

`--grid` Shows grid lines

`--coastlinescolour` The colour of the coastlines on a map, see *Available Colours and Markers*

`--nasabluemarble` Use the NASA Blue Marble for the background, instead of coastlines, when doing lat-lon plots

`--bias` Plot the bias between the data sets using specified mechanism. Can be either 'color', 'colour', 'size' or 'flag'

`--solid` Use solid markers

`--extend` Extend plot for negative correlation

--fold Fold plot for negative correlation or large variance

--gammamax Fix maximum extent of radial axis

--stdbiasmax Fix maximum standardised bias

Setting Plot Ranges

The arguments `--xmin`, `--xmax`, `--xstep`, `--ymin`, `--ymax`, `--ystep`, `--vmin`, `--vmax`, `--vstep` can be used to specify the range of values to plot, where x and y correspond to the axes and v corresponds to the colours.

When the arguments refer to dates or times, they should be in the format `YYYY-MM-DDThh:mm:ss`, where the time is optional. A colon or a space is also a valid date and time separator (if using a space quotes are necessary).

The `step` arguments are used to specify the tick spacing on the axes and `vstep` is used to specify the tick spacing on the colorbar.

When the `step` arguments refer to an amount of time, they should be in the ISO 8601 format `PnYnMnDTnHnMnS`, where any particular time group is optional, case does not matter, and `T` can be substituted for either a colon or a space (if using a space quotes are necessary).

For example, to specify a tick spacing of one month and six seconds on the x axis, the following argument should be given: `--xstep 1m6S`

Note: If a value is negative, then an equals sign must be used, e.g. `--xmin=-5`.

To plot using a log scale:

--logx The x axis will be plotted using a log scale of base 10

--logy The y axis will be plotted using a log scale of base 10

--logv The values (colours) will be plotted using a log scale of base 10

Overlaying Multiple Plots

Overlaying multiple plots is straightforward, simply use the plot command as before but specify multiple files and variables, e.g.:

```
$ cis plot $var1:$filename1:edgecolor=black $var2:$filename2:edgecolor=red
```

To plot two variables from the same file, simply use the above command with *$filename1* in place of *$filename2*.

The `type` paramter can be used to specify different types for each layer. For example, to plot a heatmap and a contour plot the following options can be used:

```
cis plot var1:file1:type=heatmap var2:file2:type=contour,color=white --width 20 --
↪height 15 --cbarscale 0.5 -o overlay.png
```

Note that the default plot dimensions are deduced from the first datagroup specified.

Many more examples are available in the *overlay examples* page.

Available Colours and Markers

CIS recognises any valid html colour, specified using its name e.g. *red* for options such as item colour (line/scatter colour) and the colour of the coast lines.

A list of available colour maps for 3D plots, such as heatmaps, scatter and contour plots, can be found here: colour maps.

For a list of available scatter point styles, see here: scatter point styles.

CHAPTER 12

Gallery

A collection of example CIS plots along with the commands used to generate them.

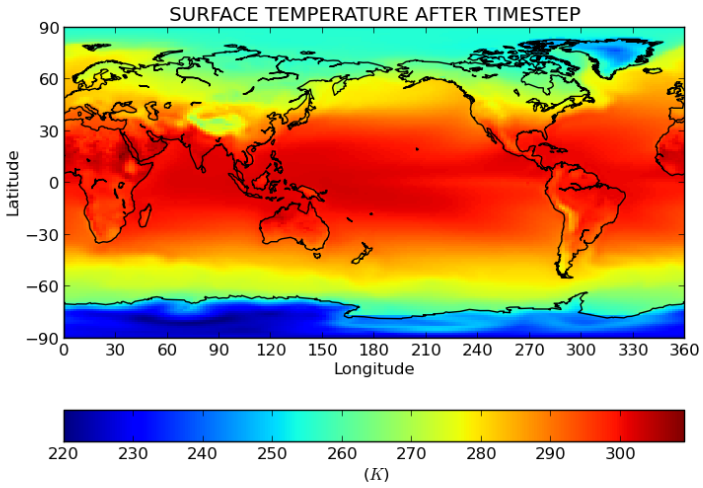


Fig. 12.1: Model output data

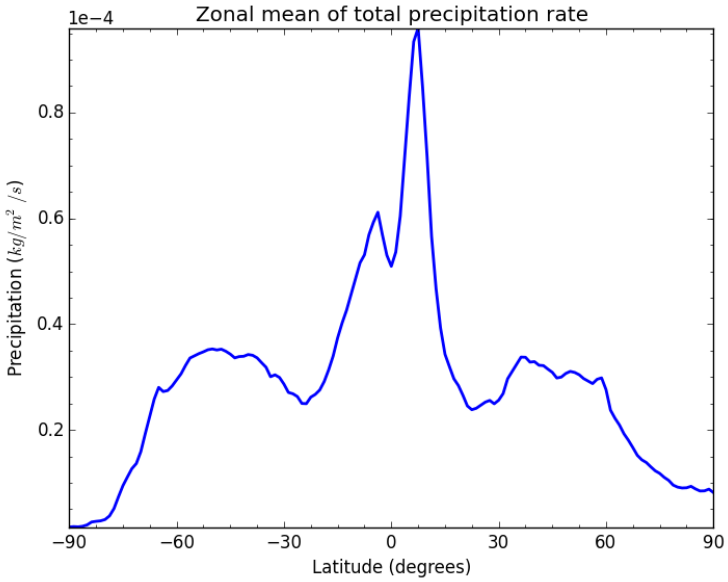


Fig. 12.2: Aggregated model data

% cis plot precip:xenida_zonal.nc –itemwidth=2 –xaxis latitude –xlabel "Latitude (degrees)" –yaxis precip –ylabel "Precipitation ($kg/m^2/s$)" –title "Zonal mean of total precipitation rate" -o line.png

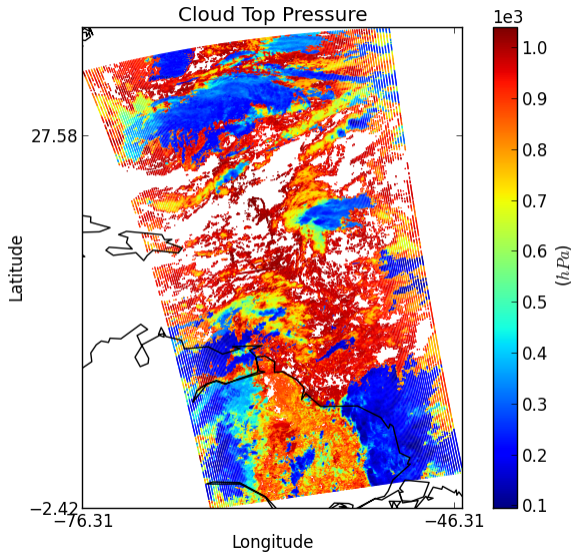


Fig. 12.3: MODIS Level 2

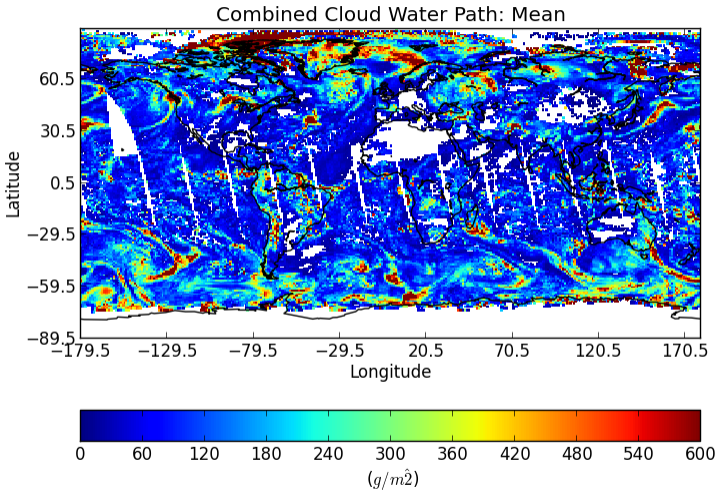


Fig. 12.4: MODIS Level 3

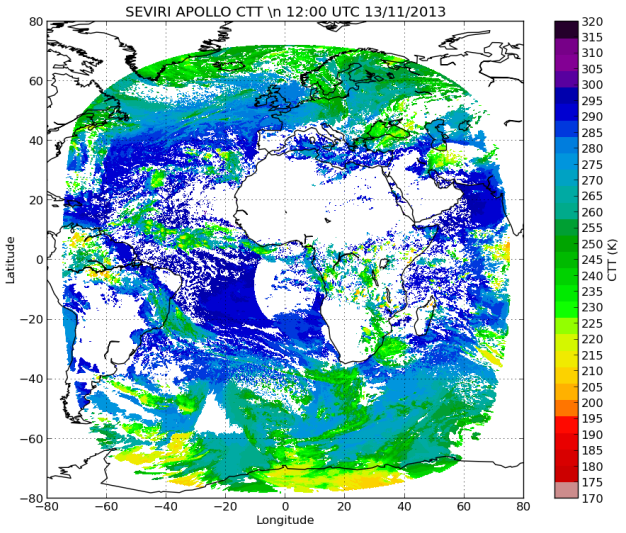


Fig. 12.5: Seviri Cloud top temperature

Fig. 12.6: Aeronet time series

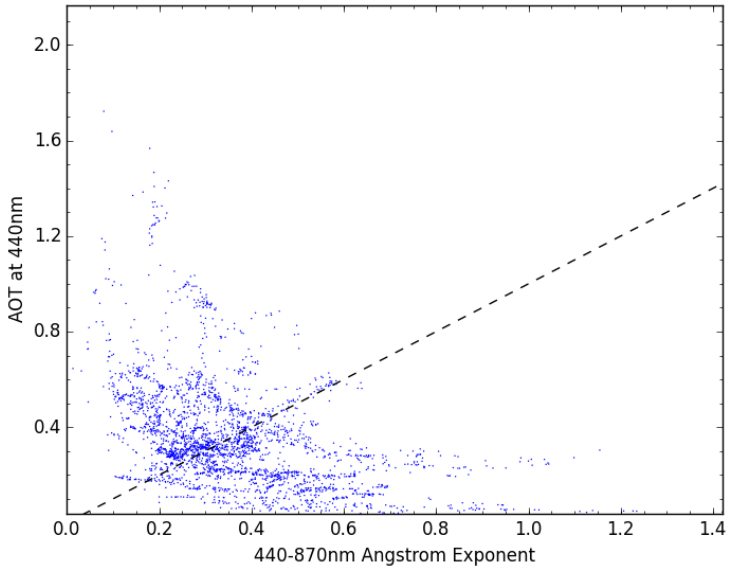


Fig. 12.7: Aeronet comparative scatter

% cis plot 440-870Angstrom:../cis_repo_test_files/920801_091128_Agoufou_small.lev20 AOT_440:../cis_repo_test_files/920801_091128_Agoufou_small.lev20 –xlabel “440-870nm Angstrom Exponent” –ylabel “AOT at 440nm” –title “” –type comparativescatter -o comparative_scatter_Aeronet.png

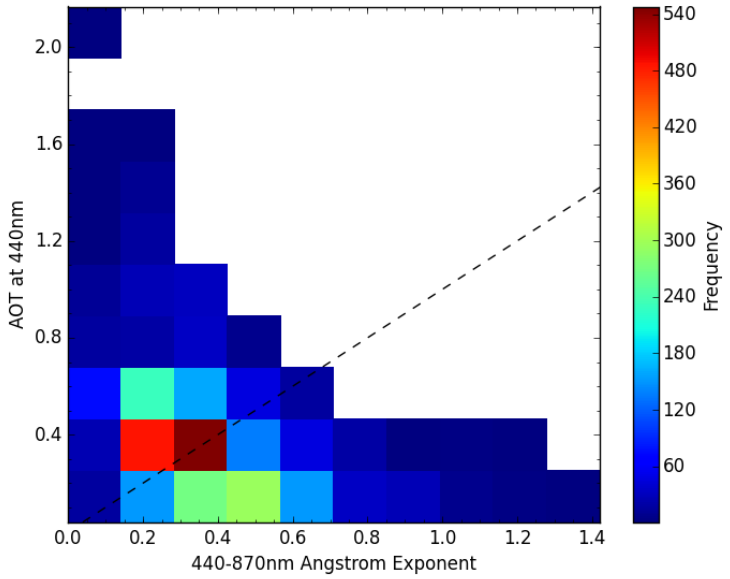


Fig. 12.8: Aeronet comparatice histogram

% cis plot 440-870Angstrom:920801_091128_Agoufou_small.lev20 AOT_440:../cis_repo_test_files/920801_091128_Agoufou_small.lev20 –xlabel “440-870nm Angstrom Exponent” –ylabel “AOT at 440nm” –title “” –type histogram3d -o comparativehistogram3d

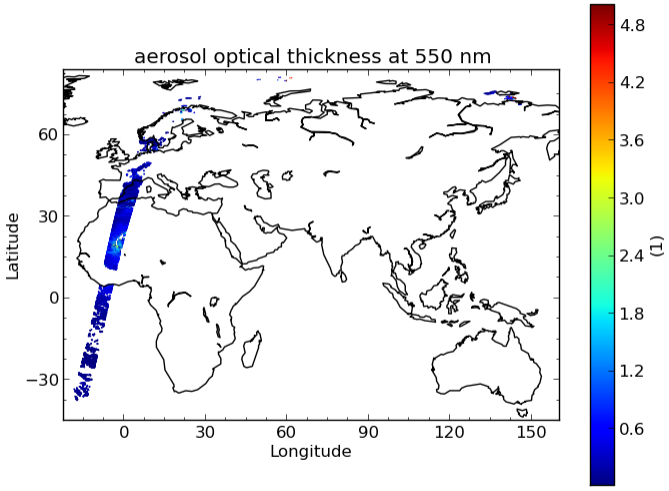


Fig. 12.9: Aerosol CCI

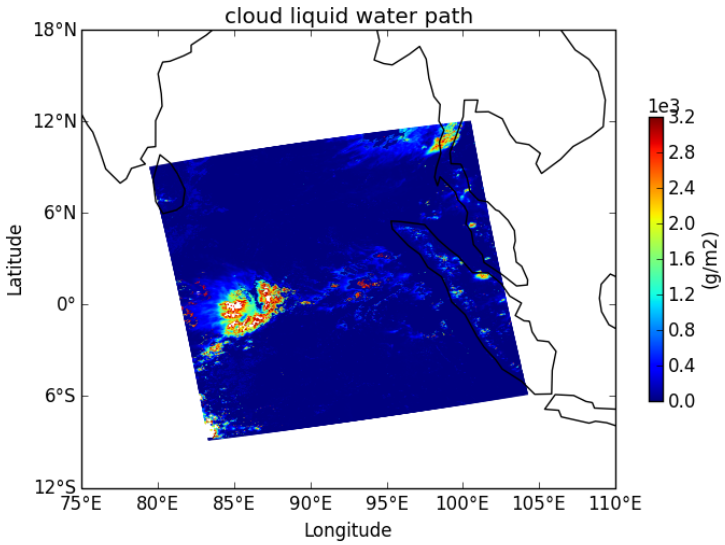


Fig. 12.10: Cloud CCI

%cis plot cwp:20080620072500-ESACCI-L2_CLOUD-CLD_PRODUCTS-MODIS-AQUA-fv1.0.nc -o Cloud_CCI –xmin 75 –xmax 110 –xstep 5

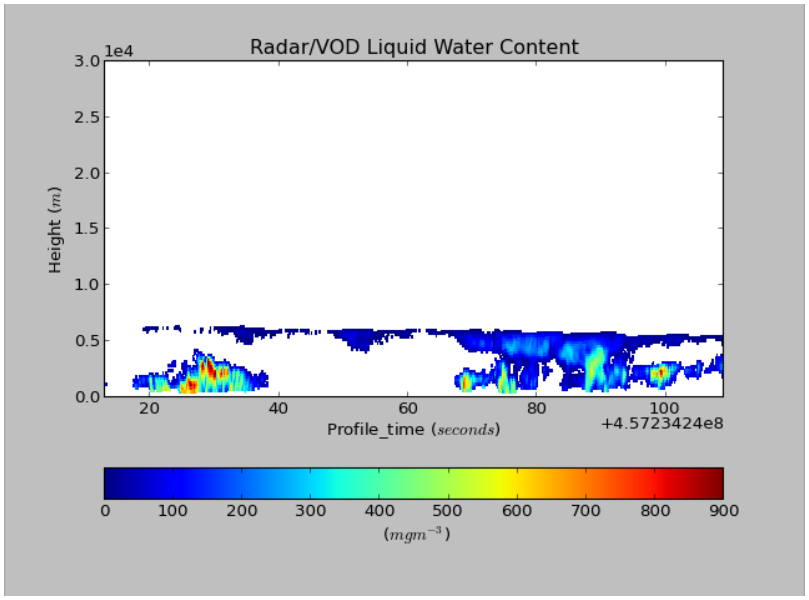


Fig. 12.11: CloudSat Liquid water content

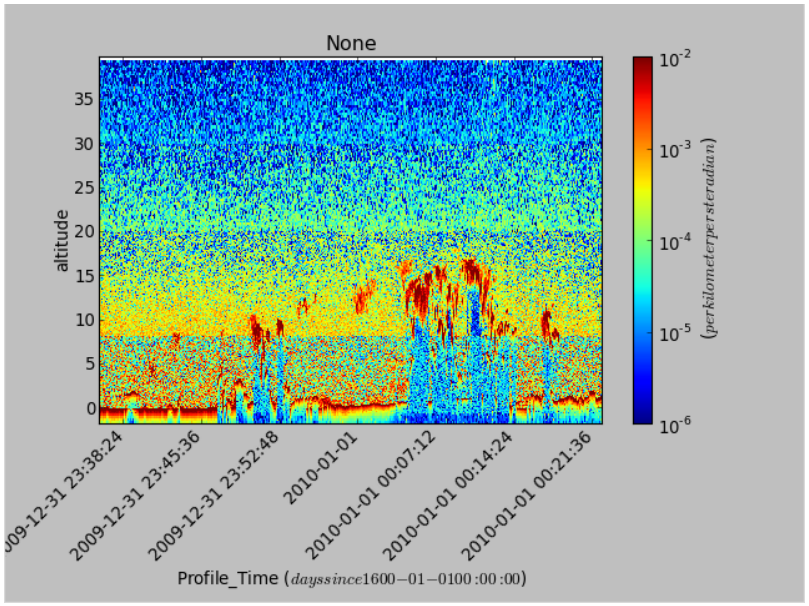


Fig. 12.12: CALIOP Level 1b

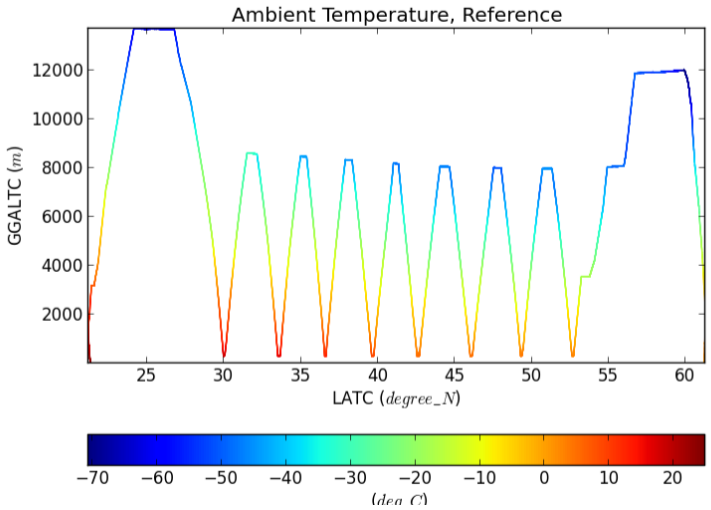


Fig. 12.13: NCAR-RAF ambient temperature

% cis plot ATX:RF04.20090114.192600_035100.PNI.nc –xaxis latitude –xlabel “Latitude (degrees north)” –yaxis altitude –ylabel “Altitude (m)” –cbarlabel “^{circ}C” -o aircraft.png

CHAPTER 13

Evaluation

The Community Intercomparison Suite allows you to perform general arithmetic operations between different variables using the 'eval' command. For example, you might want to calculate the (relative) difference between two variables.

Note: All variables used in a evaluation **must** be of the same shape in order to be compatible, i.e. the same number of points in each dimension, and of the same type (Ungridded or Gridded). This means that, for example, operations between different data products are unlikely to work correctly - performing a collocation or aggregation onto a common grid would be a good pre-processing step.

Warning: This CIS command performs a Python eval() on user input. This has the potential to be a security risk and before deploying CIS to any environment where your user input is untrusted (e.g. if you want to run CIS as a web service) **you must** satisfy yourself that any security risks have been mitigated. CIS implements the following security restrictions on the expression which is evaluated:

- The eval() operates in a restricted namespace that only has access to a select handful of builtins (see *expr* below) - so `__import__`, for example, is unavailable.
- The only module available in the namespace is numpy.
- Any expression containing two consecutive underscores (__) is assumed to be harmful and will not be evaluated.

The evaluate syntax looks like this:

```
$ cis eval <datagroup>... <expr> <units> [-o [<output_var>:]<outputfile>] [--
↪attributes <attributes>]
```

where square brackets denote optional commands and:

<datagroup> is a modified *CIS datagroup* of the format `<variable>[=<alias>]...:<filename>[:product=<productname>]`. One or more datagroups should be given.

- `<variable>` is a mandatory variable or list of variables to use.
- `<alias>` is an optional alternative variable name to use in place of the name given in the file. As you will see in the *expression* section, the variable names given will need to be valid python variable names, which means:

  1. They may use only the characters [A-Z], [a-z] and numbers [0-9] provided they do not start with a number
  2. The only special character which may be used is the underscore (_) - but don't use two consecutively (see *security note*)
  3. Don't use any of the reserved python keywords such as `class` or `and` as variable names (they're OK if they're only part of a name though).
  4. Avoid using names of python builtins like `max` or `abs` (again, it's OK if they're only part of a name).

  So if the variable name in your file violates these rules (e.g. *'550-870Angstrom'*) use an alias:

  ```
  550-870Angstrom=a550to870
  ```

- `<filename>` is a mandatory file or list of files to read from.
- `<productname>` is an optional CIS data product to use (see *Data Products*):

See *Datagroups* for a more detailed explanation of datagroups.

<expr> is the arithmetic expression to evaluate; for example: `variable1+variable2`. Use the following basic rules to get started:

1. Use the variable names (or aliases) as given in the datagroups (they're case-sensitive) - don't enclose them in quotes.
2. If your expression contains whitespace, you'll need to enclose the whole expression in single or double quotes.
3. Construct your expression using plus `+`, minus `-`, times `*`, divide `/` , power `**` (note that you **can't** use `^` for exponents, like you typically can in spreadsheets and some other computer languages). Parentheses `()` can be used to group elements so that your expression is evaluated in the order you intend.

If you need more functionality, you're encountering errors or not getting the answer you expect then you should consider the following.

1. This expression will be evaluated in Python using the eval() method (see *security note*), so the expression must be a valid Python expression.
2. The only Python methods available to you are a trimmed down list of the python builtins: *'abs', 'all', 'any', 'bool', 'cmp', 'divmod', 'enumerate', 'filter', 'int', 'len', 'map', 'max', 'min', 'pow', 'range', 'reduce', 'reversed', 'round', 'sorted', 'sum', 'xrange', 'zip'*.
3. The numpy module is available, so you can use any of its methods e.g. `numpy.mean(variable1)`.
4. For security reasons, double underscores (__) must not appear anywhere in the expression.
5. The expression must produce an output array of the same shape as the input variables.
6. The expression is evaluated at the array level, not at the element level - so the variables in an expression represent numpy arrays, not individual numeric values. This means that `numpy.mean([var1,var2])` will give you a combined average *over the whole of both arrays* (i.e. a single number, not an array), which would be invalid (consider the previous rule). However, you

could add the mean (over the whole array) of one variable to every point on a second variable by doing `var1 + numpy.mean(var2)`.

Note: CIS eval command will flatten ungridded data so that structure present in the input files will be ignored. This allows you to compare ungridded data with different shapes, e.g. (3,5) and (15,)

<units> is a mandatory argument describing the units of the resulting expression. This should be a CF compliant units string, e.g. `"kg m^-3"`. Where this contains spaces, the whole string should be enclosed in quotes.

<outputfile> is an optional argument specifying the file to output to. This will be automatically given a `.nc` extension if not present. This must not be the same file path as any of the input files. If not provided, the default output filename is *out.nc*

- `<output_var>` is an optional prefix to the output file argument to specify the name of the output variable within the output file, e.g. `-o my_new_var:output_filename.nc`. If not provided, the default output variable name is *calculated_variable*

<attributes> is an optional argument allowing users to provide additional metadata to be included in the evaluation output variable. This should be indicated by the attributes flag (`--attributes` or `-a`). The attributes should then follow in comma-separated, key=value pairs, for example `--attributes standard_name=convective_rainfall_amount,echam_version=6.1.00`. Whitespace is permitted in both the names and the values, but then must be enclosed in quotes: `-a "operating system = "AIX 6.1 Power6"`. Colons or equals signs may not be used in attribute names or values.

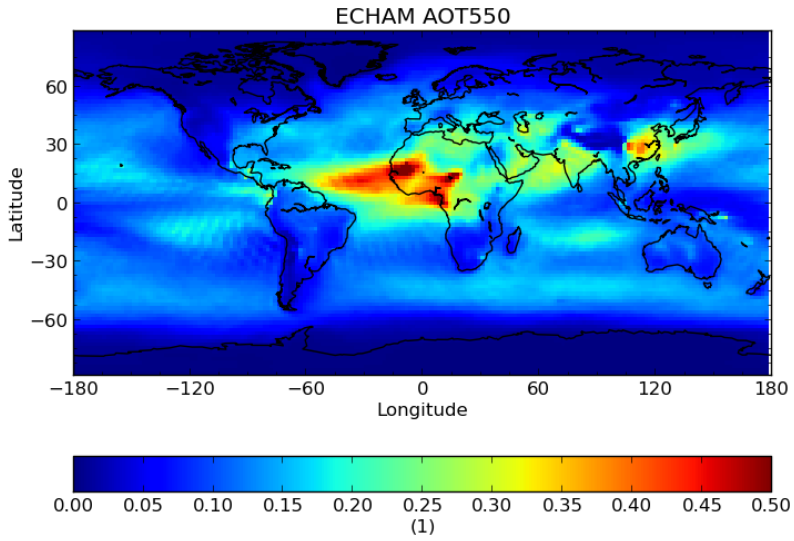
Evaluation Examples

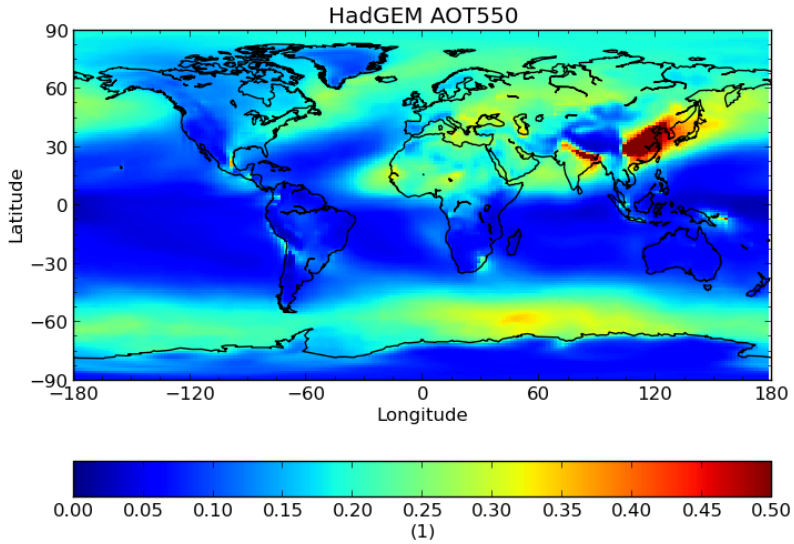
Comparison of annual Aerosol Optical Thickness from models

In this example we compare annual Aerosol Optical Thickness from ECHAM and HadGEM model data. The data used in this example can be found at `/group_workspaces/jasmin/cis/data`.

First we produce annual averages of our data by *aggregating*:

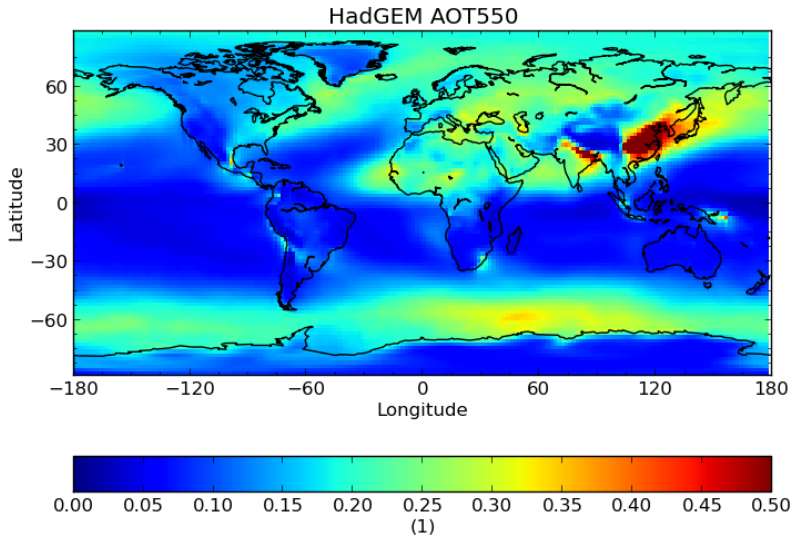
```
$ cis aggregate od550aer:ECHAM_fixed/2007_2D_3hr/od550aer.nc t -o echam-od550aer
$ cis aggregate od550aer:HadGEM_fixed/test_fix/od550aer.nc t -o hadgem-od550aer

$ cis plot od550aer:echam-od550aer.nc --xmin -180 --xmax 180 --cbarorient=horizontal -
↪-title="ECHAM AOT550" --vmin=0 --vmax=0.5
$ cis plot od550aer:hadgem-od550aer.nc --xmin -180 --xmax 180 --cbarorient=horizontal␣
↪--title="HadGEM AOT550" --vmin=0 --vmax=0.5
```





We then linearly interpolate the HadGEM data onto the ECHAM grid:

```
$ cis col od550aer:hadgem-od550aer.nc echam-od550aer.nc:collocator=lin -o hadgem-
↪od550aer-collocated

$ cis plot od550aer:hadgem-od550aer-collocated.nc --xmin -180 --xmax 180 --
↪cbarorient=horizontal --title="HadGEM AOT550" --vmin=0 --vmax=0.5
```

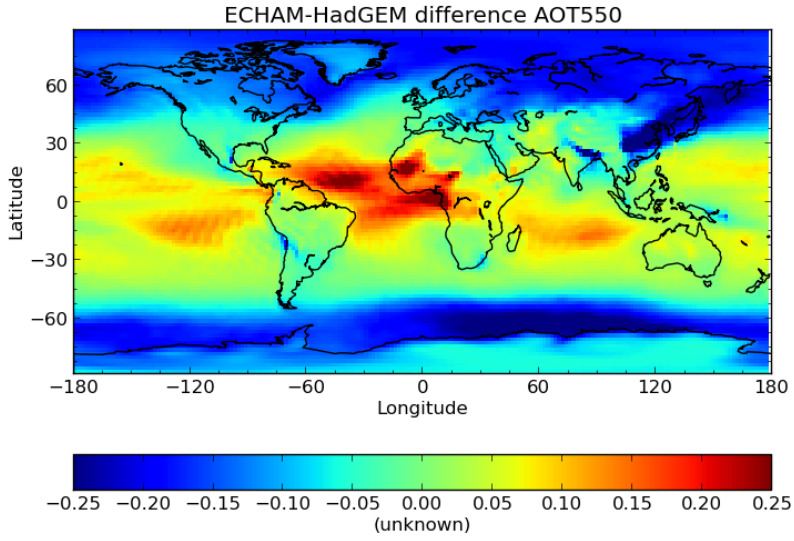


Next we subtract the two fields using:

```
$ cis eval od550aer=a:echam-od550aer.nc od550aer=b:hadgem-od550aer-collocated.nc "a-b
↪" 1 -o modeldifference
```

Finally we plot the evaluated output:

```
$ cis plot calculated_variable:modeldifference.nc --xmin -180 --xmax 180 --
↪cbarorient=horizontal --title="ECHAM-HadGEM difference AOT550" --vmin=-0.25 --
↪vmax=0.2
```



Calculation of Angstrom exponent for AERONET data

AERONET data allows us to calculate Angstrom Exponent (AE) and then compare it against the AE already in the file. They should strongly correlate although it is not expected they will be identical due to averaging etc during production of AERONET datafiles.

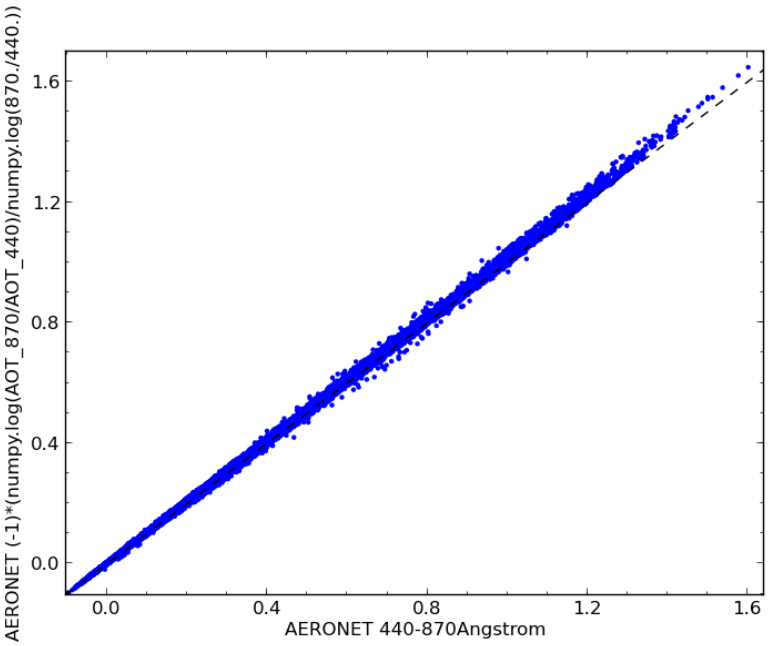
The file agoufou.lev20 refers to `/group_workspaces/jasmin/cis/data/aeronet/AOT/LEV20/ALL_POINTS/920801_121229_Agoufou.lev20`

The AE is calculated using an eval statement:

```
$ cis eval AOT_440,AOT_870:agoufou.lev20 "(-1)* (numpy.log(AOT_870/AOT_440)/numpy.
↪log(870./440.))" 1 -o alfa
```

Plotting it shows the expected correlation:

```
$ cis plot 440-870Angstrom:agoufou.lev20 calculated_variable:alfa.nc --type␣
↪comparativescatter --itemwidth=10 --xlabel="AERONET 440-870Angstrom" --ylabel=
↪"AERONET (-1)*(numpy.log(AOT_870/AOT_440)/numpy.log(870./440.))"
```



This correlation can be confirmed by using the CIS *stats* command:

```
$ cis stats 440-870Angstrom:agoufou.lev20 calculated_variable:alfa.nc

==================================
RESULTS OF STATISTICAL COMPARISON:
==================================
Number of points: 63126
Mean value of dataset 1: 0.290989032142
Mean value of dataset 2: 0.295878214327
Standard deviation for dataset 1: 0.233995525021
Standard deviation for dataset 2: 0.235381075635
Mean of absolute difference: 0.00488918218519
Standard deviation of absolute difference: 0.00546343157047
Mean of relative difference: 0.0284040419499
```

```
Standard deviation of relative difference: 3.95137224542
Spearman's rank coefficient: 0.999750939223
Linear regression gradient: 1.00566622549
Linear regression intercept: 0.003240372714
Linear regression r-value: 0.999746457079
Linear regression standard error: 0.00530006646489
```

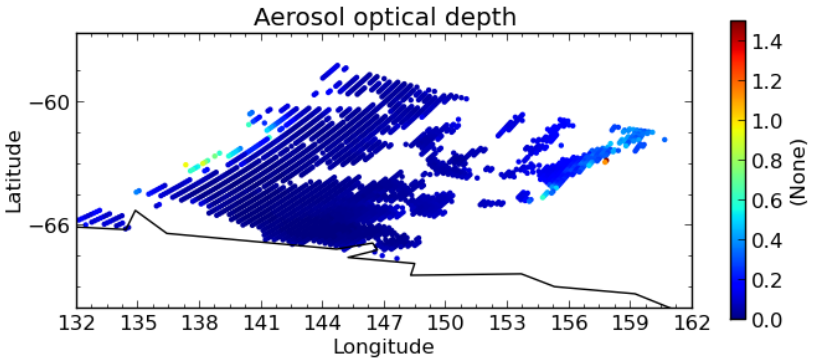
Using Evaluation for Conditional Aggregation

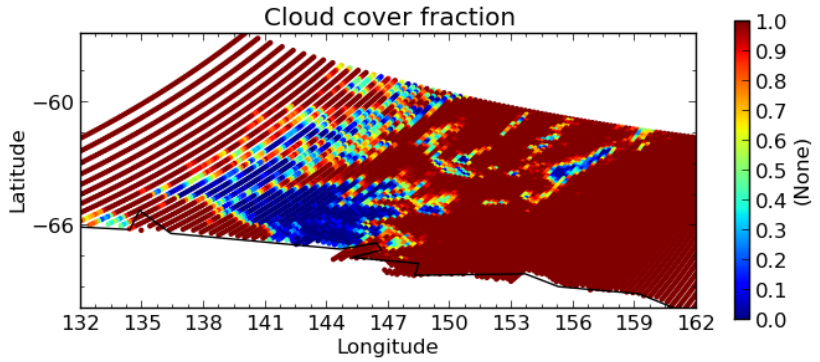
The *eval* command can be combined with other CIS commands to allow you to perform more complex tasks than would otherwise be possible.

For example, you might want to aggregate a satellite measurement of one variable only when the corresponding cloud cover fraction (stored in separate variable) is less than a certain value. The aggregate command doesn't allow this kind of conditional aggregation on its own, but you can use an evaluation to achieve this in two stages.

In this example we use the MODIS file `MOD04_L2.A2010001.2255.005.2010005215814.hdf` in directory `/group_workspaces/jasmin/cis/data/MODIS/MOD04_L2/`. The optical depth and cloud cover variables can be seen in the following two plots:

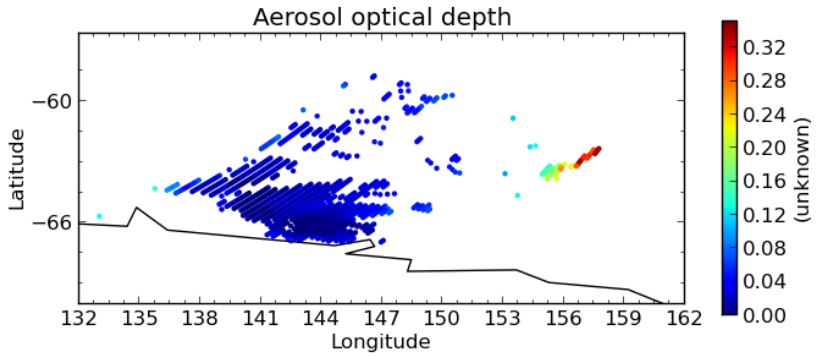
```
$ cis plot Optical_Depth_Land_And_Ocean:MOD04_L2.A2010001.2255.005.2010005215814.hdf -
↪-xmin 132 --xmax 162 --ymin -70 --title "Aerosol optical depth" --cbarscale 0.5 --
↪itemwidth 10 -o cloud_fraction.png
$ cis plot Cloud_Fraction_Ocean:MOD04_L2.A2010001.2255.005.2010005215814.hdf --xmin␣
↪132 --xmax 162 --ymin -70 --title "Cloud cover fraction" --cbarscale 0.5 --
↪itemwidth 10 -o cloud_fraction.png
```





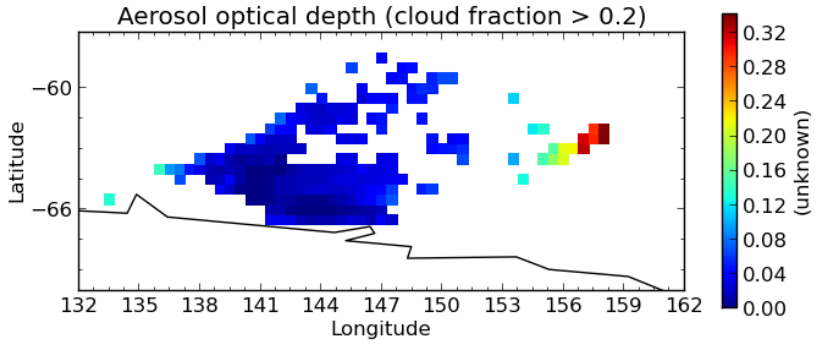
First we perform an evaluation using the numpy.masked_where method to produce an optical depth variable that is masked at all points where the cloud cover is more than 20%:

```
$ cis eval Cloud_Fraction_Ocean=cloud,Optical_Depth_Land_And_Ocean=od:MOD04_L2.
↪A2010001.2255.005.2010005215814.hdf "numpy.ma.masked_where(cloud > 0.2, od)" 1 -o␣
↪od:masked_optical_depth.nc
$ cis plot od:masked_optical_depth.nc --xmin 132 --xmax 162 --ymin -70 --title␣
↪Aerosol optical depth --cbarscale 0.5 --itemwidth 10 -o masked_optical_depth.png'
```



Then we perform an aggregation on this masked output file to give the end result - aerosol optical depth aggregated only using points where the cloud cover is less than 20%:

```
$ cis aggregate od:masked_optical_depth.nc x=[132,162,0.5],y=[-70,-57,0.5] -o␣
↪aggregated_masked_optical_depth
$ cis plot od:aggregated_masked_optical_depth.nc --xmin 132 --xmax 162 --ymin -70 --
↪title "Aerosol optical depth (cloud fraction > 0.2)" --cbarscale 0.5 -o aggregated_
↪aod.png
```


Aerosol optical depth (cloud fraction > 0.2)
Latitude
−60
−66
132
135
138
141
144
147
150
153
156
159
162
Longitude
0.32
0.28
0.24
0.20
0.16
0.12
0.08
0.04
0.00
(unknown)

CHAPTER 14

Statistics

The Community Intercomparison Suite allows you to perform statistical analysis on two variables using the 'stats' command. For example, you might wish to examine the correlation between a model data variable and actual measurements. The 'stats' command will calculate:

1. Number of data points used in the analysis.
2. The mean and standard deviation of each dataset (separately).
3. The mean and standard deviation of the absolute difference (var2 - var1).
4. The mean and standard deviation of the relative difference ((var2 - var1) / var1).
5. The Linear Pearson correlation coefficient.
6. The Spearman Rank correlation coefficient.
7. The coefficients of linear regression (i.e. var2 = a var1 + b), r-value, and standard error of the estimate.

These values will be displayed on screen and can optionally be save as NetCDF output.

Note: Both variables used in a statistical analysis **must** be of the same shape in order to be compatible, i.e. the same number of points in each dimension, and of the same type (ungridded or gridded). This means that, for example, operations between different data products are unlikely to work correctly - performing a collocation or aggregation onto a common grid would be a good pre-processing step.

Note: Only points which have non-missing values for both variables will be included in the analysis. The number of points this includes is part of the output of the stats command.

Warning: Unlike *aggregation*, `stats` does **not** currently use latitude weighting to account for the relative areas of different grid cells.

The statistics syntax looks like this:

```
$ cis stats <datagroup>... [-o <outputfile>]
```

where:

<datagroup> is a *CIS datagroup* specifying the variables and files to read and is of the format `<variable>...:<filename>[:product=<productname>]` where:

- `<variable>` is a mandatory variable or list of variables to use.
- `<filenames>` is a mandatory file or list of files to read from.
- `<productname>` is an optional CIS data product to use (see *Data Products*):

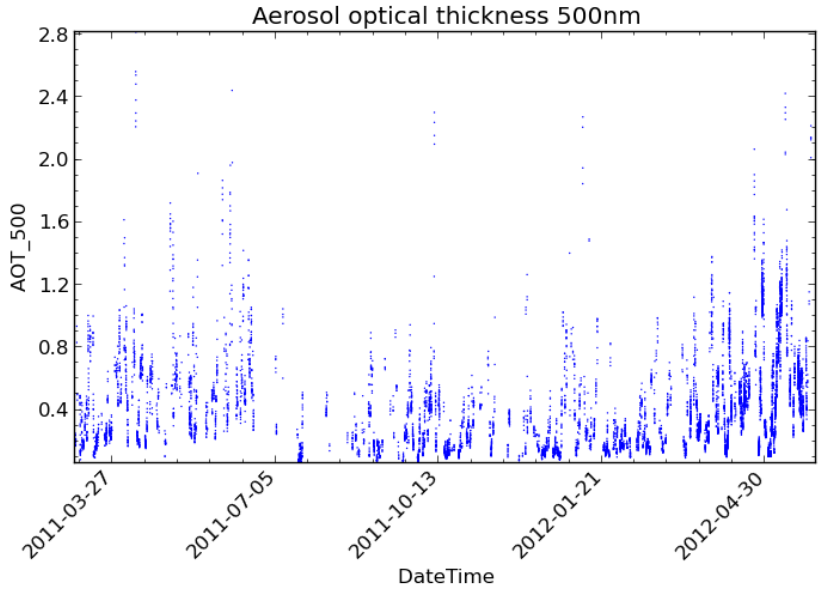
One or more datagroups should be given, but the total number of variables declared in all datagroups must be exactly two. See *Datagroups* for a more detailed explanation of datagroups.

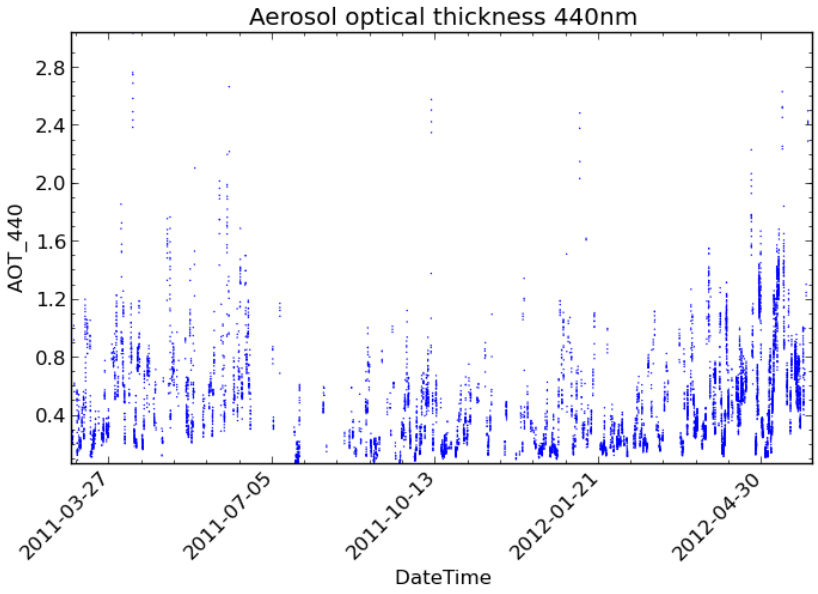
<outputfile> is an optional argument specifying a file to output to. This will be automatically given a `.nc` extension if not present. This must not be the same file path as any of the input files. If not provided, then the output will not be saved to a file and will only be displayed on screen.

Statistics Example

In this example, we perform a statistical comparison of Aeronet aerosol optical thickness at two wavelengths. The data we are using is shown in the following CIS plot commands and can be found at `/group_workspaces/jasmin/cis/data`:

```
$ cis plot AOT_500:aeronet/AOT/LEV20/ALL_POINTS/920801_121229_Yonsei_University.lev20␣
↪--title "Aerosol optical thickness 550nm"
$ cis plot AOT_440:aeronet/AOT/LEV20/ALL_POINTS/920801_121229_Yonsei_University.lev20␣
↪--title "Aerosol optical thickness 440nm"
```





We then perform a statistical comparison of these variables using:

```
$ cis stats AOT_500,AOT_440:aeronet/AOT/LEV20/ALL_POINTS/920801_121229_Yonsei_
↪University.lev20
```

Which gives the following output:

```
=================================================================
RESULTS OF STATISTICAL COMPARISON:
-----------------------------------------------------------------
Compared all points which have non-missing values in both variables
=================================================================
Number of points: 10727
Mean value of dataset 1: 0.427751965508
Mean value of dataset 2: 0.501316673814
Standard deviation for dataset 1: 0.307680514916
Standard deviation for dataset 2: 0.346274598431
Mean of absolute difference: 0.0735647083061
Standard deviation of absolute difference: 0.0455684788406
Mean of relative difference: 0.188097066086
Standard deviation of relative difference: 0.0528621773819
Spearman's rank coefficient: 0.998289763952
Linear regression gradient: 1.12233533743
Linear regression intercept: 0.0212355272705
Linear regression r-value: 0.997245296339
Linear regression standard error: 0.0256834603945
```

CHAPTER 15

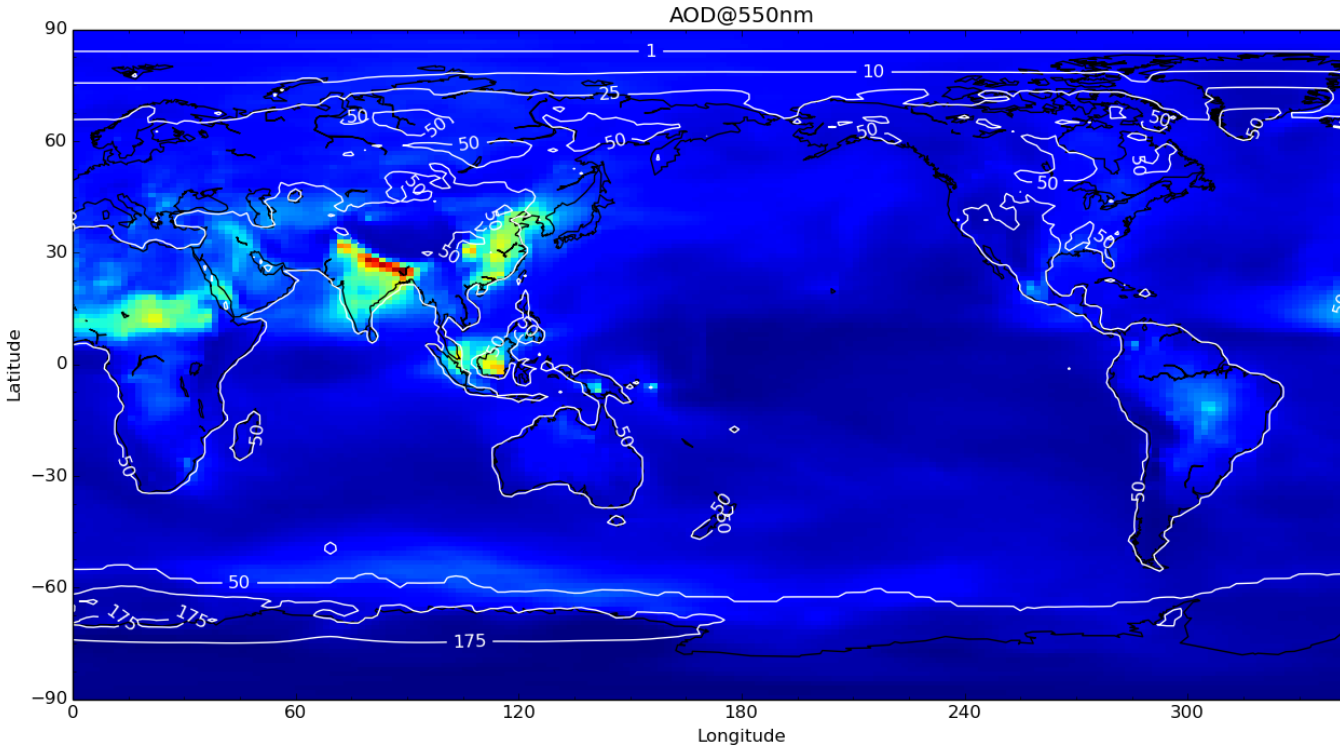
Overlay Plot Examples

First subset some gridded data that will be used for the examples:

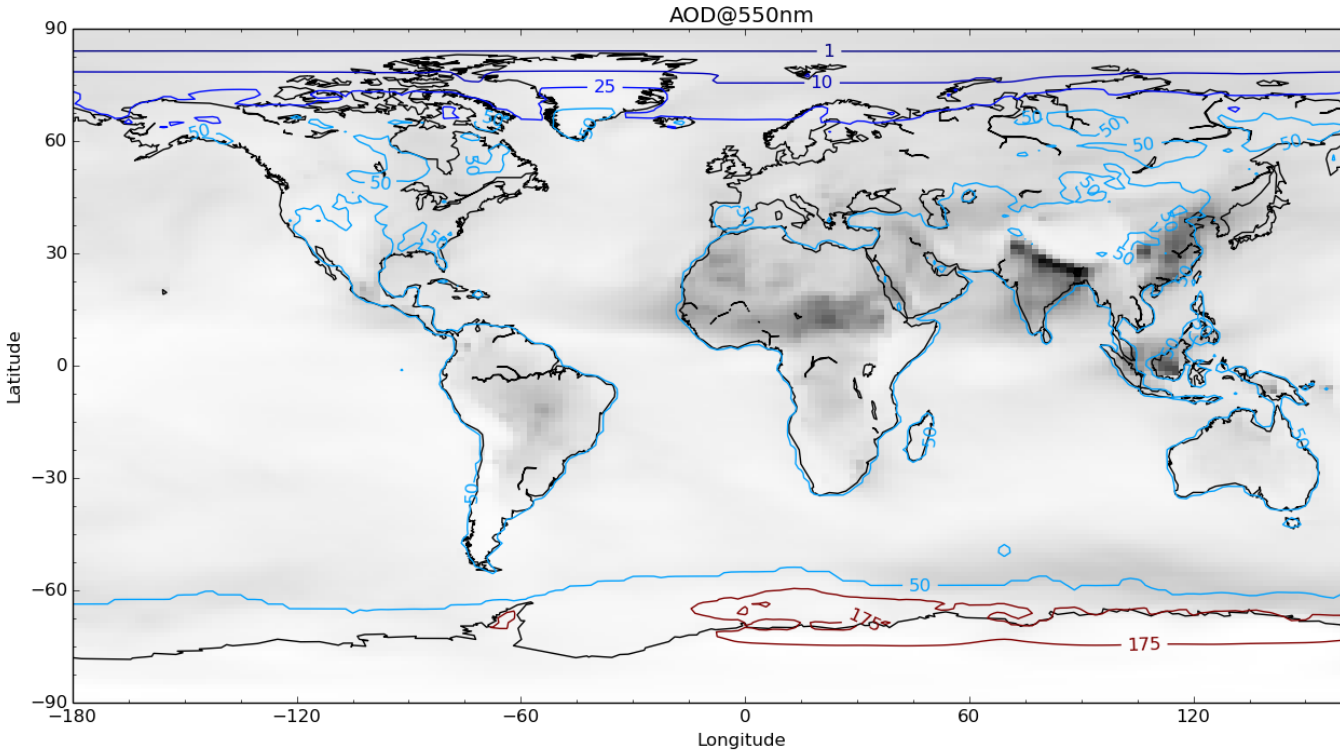
```
cis subset od550aer:aerocom.HadGEM3-A-GLOMAP.A2.CTRL.monthly.od550aer.2006.nc t=[2006-
↪10-13] -o HadGEM_od550aer-subset

cis subset rsutcs:aerocom.HadGEM3-A-GLOMAP.A2.CTRL.monthly.rsutcs.2006.nc t=[2006-10-
↪13] -o HadGEM_rsutcs-subset
```

Contour over heatmap

```
cis plot od550aer:HadGEM_od550aer-subset.nc:type=heatmap rsutcs:HadGEM_rsutcs-subset.
↪nc:type=contour,color=white,contlevels=[1,10,25,50,175] --width 20 --height 15 --
↪cbarscale 0.5 -o overlay1.png
```

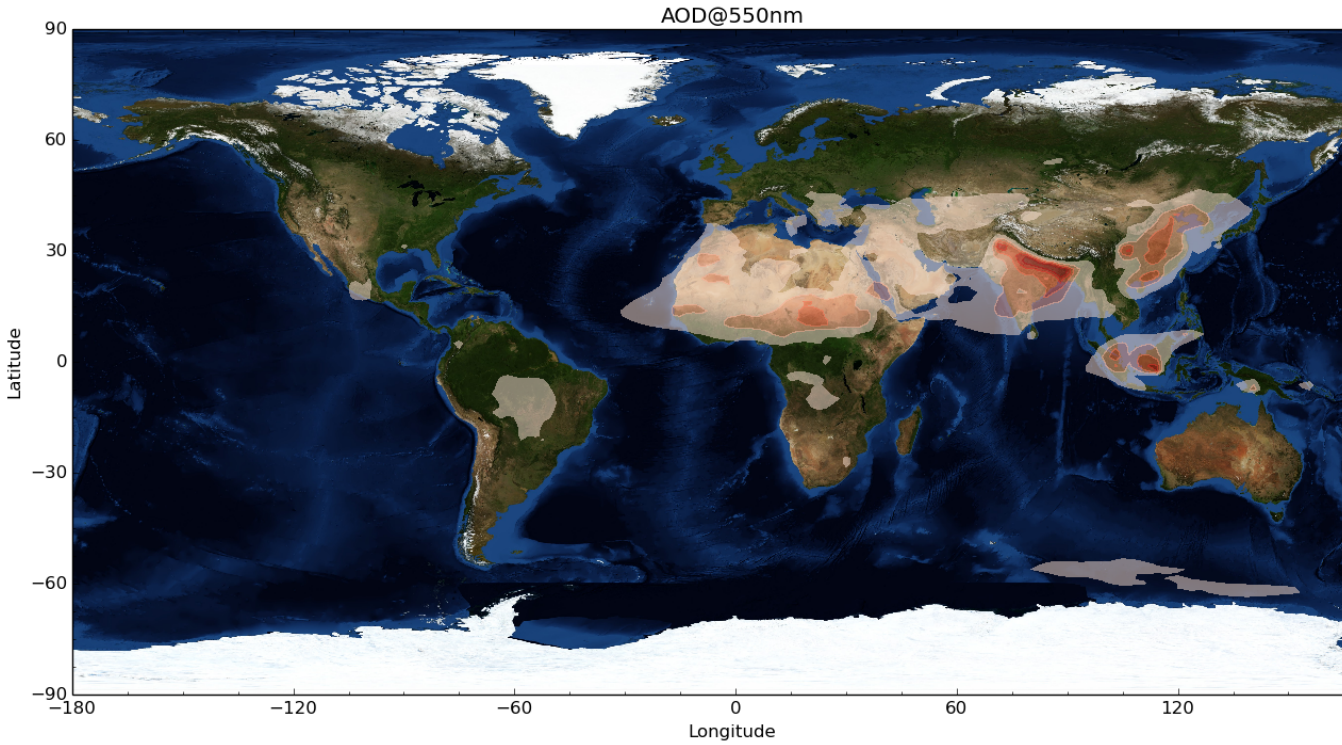



```
cis plot od550aer:HadGEM_od550aer-subset.nc:type=heatmap,cmap=binary rsutcs:HadGEM_
↪rsutcs-subset.nc:type=contour,cmap=jet,contlevels=[1,10,25,50,175] --xmin -180 --
↪xmax 180 --width 20 --height 15 --cbarscale 0.5 -o overlay2.png
```



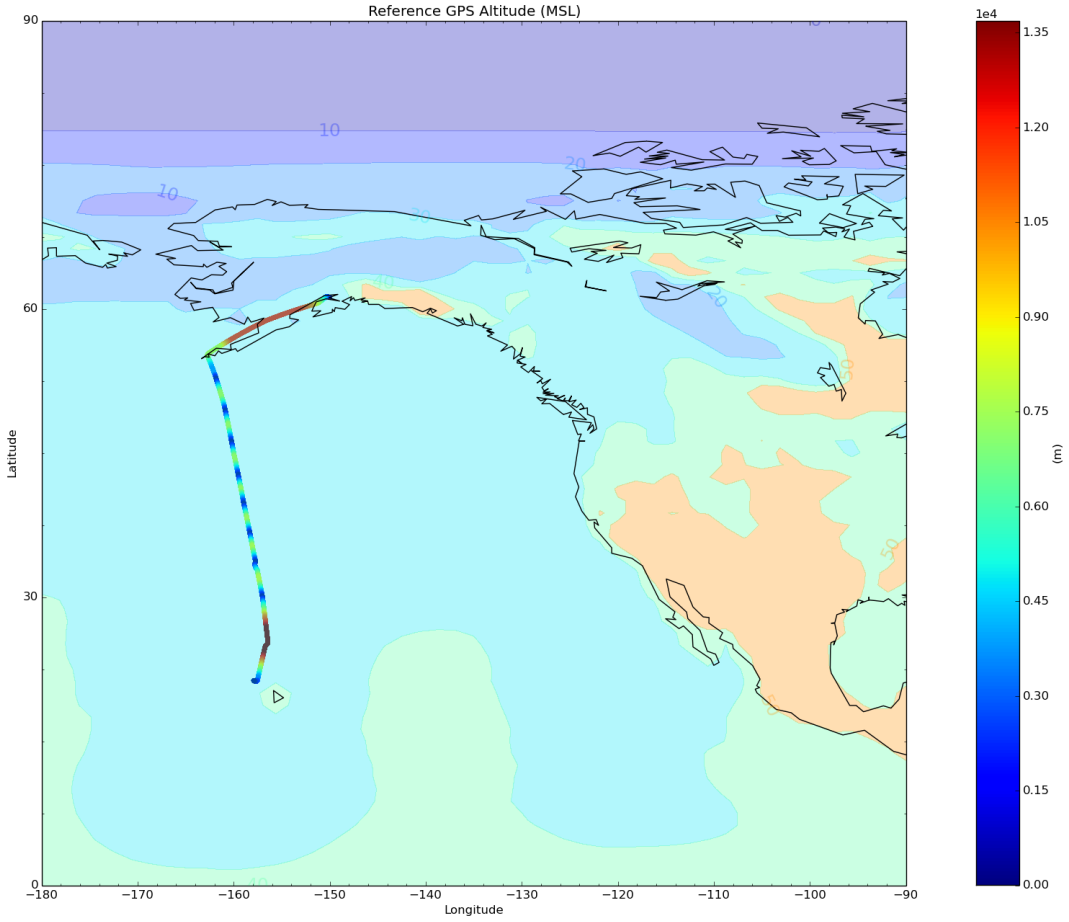
Filled contour with transparency on NASA Blue Marble

```
cis plot od550aer:HadGEM_od550aer-subset.nc:cmap=Reds,type=contourf,transparency=0.5,
↪cmin=0.15 --xmin -180 --xmax 180 --width 20 --height 15 --cbarscale 0.5 --
↪nasabluemarble
```

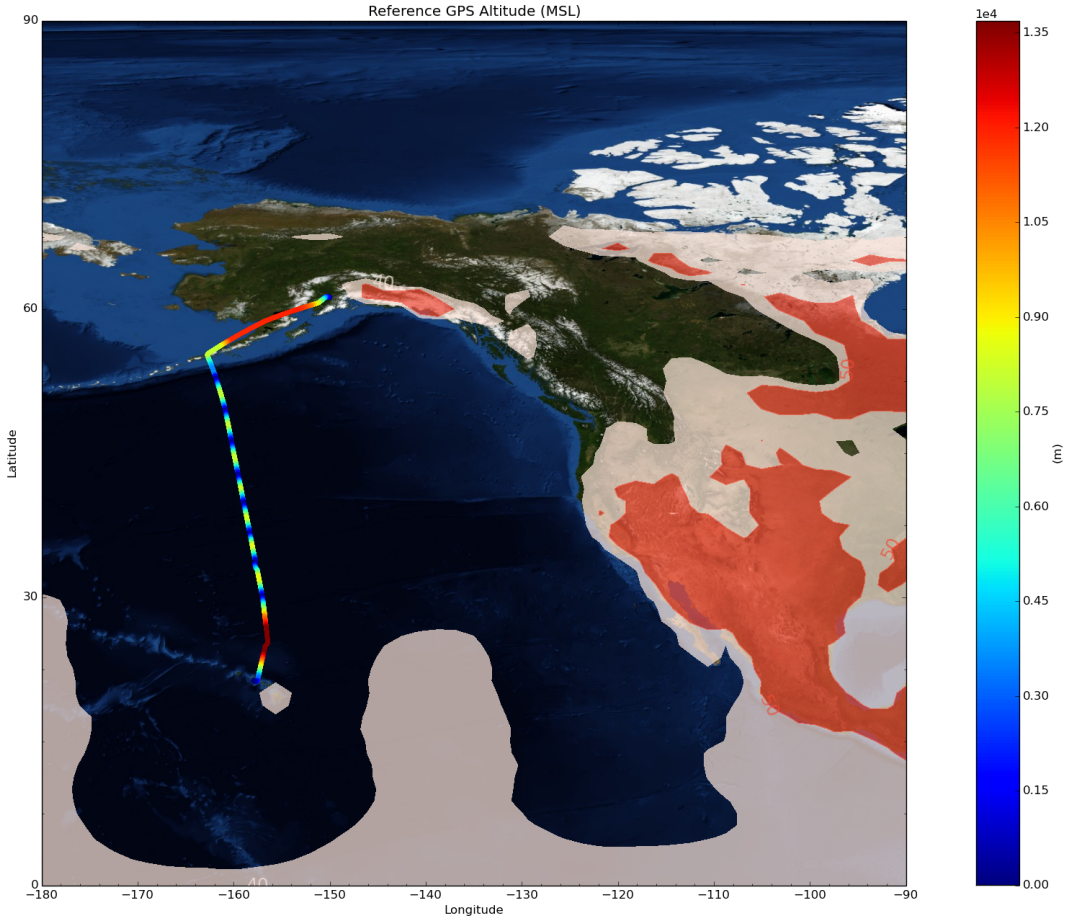


Scatter plus Filled Contour

```
cis subset rsutcs:HadGEM_rsutcs-subset.nc x=[-180,-90],y=[0,90] -o HadGEM_rsutcs-
↪subset2

cis plot GGALT:RF04.20090114.192600_035100.PNI.nc:type=scatter rsutcs:HadGEM_rsutcs-
↪subset2.nc:type=contourf,contlevels=[0,10,20,30,40,50,100],transparency=0.7,
↪contlabel=true,contfontsize=18 --width 20 --height 15 --xaxis longitude --yaxis␣
↪latitude --xmin -180 --xmax -90 --ymin 0 --ymax 90 --itemwidth 20 -o overlay4.png
```




```
cis plot GGALT:RF04.20090114.192600_035100.PNI.nc:type=scatter rsutcs:HadGEM_rsutcs-
↪subset2.nc:type=contourf,contlevels=[40,50,100],transparency=0.3,contlabel=true,
↪contfontsize=18,cmap=Reds --width 20 --height 15 --xaxis longitude --yaxis latitude␣
↪--xmin -180 --xmax -90 --ymin 0 --ymax 90 --itemwidth 20 --nasabluemarble -o␣
↪overlay5.png
```



File Locations

The gridded data files can be found at:

```
/group_workspaces/jasmin/cis/AeroCom/A2/HadGEM3-A-GLOMAP.A2.CTRL/renamed
```

and the ungridded:

```
/group_workspaces/jasmin/cis/jasmin_cis_repo_test_files
```

CHAPTER 16

How can I read my own data?

Introduction

One of the key strengths of CIS is the ability for users to create their own plugins to read data which CIS doesn't currently support. These plugins can then be shared with the community to allow other users access to that data. Although the plugins are written in Python this tutorial assumes no experience in Python. Some programming experience is however assumed.

Note: Any technical details that may be useful to experienced Python programmers will be highlighted in this style - they aren't necessary for completing the tutorial.

Here we describe the process of creating and sharing a plugin. A CIS plugin is simply a python (.py) file with a set of methods (or functions) to describe how the plugin should behave.

Note: The methods for each plugin are described within a Class, this gives the plugin a name and allows CIS to ensure that all of the necessary methods have been implemented.

There are a few methods that the plugin must contain, and some which are optional. A skeleton plugin would look like this:

```
class MyProd(AProduct):
    def get_file_signature(self):
    # Code goes here

    def create_coords(self, filenames):
        ...

    def create_data_object(self, filenames, variable):
        ...
```

Note that in python whitespace matters! When filling in the above methods the code for the method should be indented from the signature by four spaces like this:

```
Class MyProd(AProduct):

    def get_file_signature(self):
        # Code goes here
        foo = bar
```

Note also that the name of the plugin (MyProd) in this case should be changed to describe the data which it will read. (Don't change the AProduct part though – this is important for telling CIS that this is a plugin for reading data.)

Note: The plugin class subclasses `AProduct` which is the abstract class which defines the methods that the plugin needs to override. It also includes a few helper functions for error catching.

When CIS looks for data plugins it searches for all classes which sub-class `AProduct`. There are also plugins available for collocation with their own abstract base classes, so that users can store multiple plugin types in the same plugin directory.

In order to turn the above skeleton into a working plugin we need to fill in each of the methods with the some code, which turns our data into something CIS will understand. Often it is easiest to start from an existing plugin that reads closely matching data. For example creating a plugin to read some other CCI data would probably be easiest to start from the Cloud or Aerosol CCI plugins. We have created three different tutorials to walk you through the creation of some of the existing plugins to try and illustrate the process. The *Easy* tutorial walks through the creation of a basic plugin, the *Medium* tutorial builds on that by creating a plugin which has a bit more detail, and finally the *Advanced* plugin talks through some of the main considerations when creating a large and complicated plugin.

A more general template plugin is included here in case no existing plugin matches your need. We have also created a short reference describing the purpose of each method the plugins implement *here*.

Note: Plugins aren't the only way you can contribute though. CIS is an open source project hosted on GitHub, so please feel free to submit pull-requests for new features or bug-fixes – just check with the community first so that we're not duplicating our effort.

Using and testing your plugin

It is important that CIS knows where to look to find your new plugin, and this is easily done by setting the environment variable CIS_PLUGIN_HOME to point to the directory within which your plugin is stored.

Once you have done this CIS will automatically use your plugin for reading any files which match the file signature you used.

If you have any issues with this (because for example the file signature clashes with a built-in plugin) you can tell CIS to use your plugin when reading data by simply specifying it after the variable and filename in most CIS commands, e.g.:

```
cis subset a_variable:filename.nc:product=MyProd ...
```

Sharing your plugin

This is the easy bit! Once you're happy that your plugin can fairly reliably read a currently unsupported dataset you should share it with the community. Use the upload form here to submit your plugin to the community.

We moderate the plugins we receive to ensure the plugins received are appropriate and meet a minimum level of quality. We're not expecting the plugins to necessarily be production quality code but we do expect them to work for the subset of data they claim to. Having said that, if we feel a plugin provides really a valuable capability and is of high quality we may incorporate that plugin into the core CIS data readers – with credit to the author of course!

Tutorials

Easy

A simple plugin to start with is the plugin for reading native ungridded CIS data.

One of the first things to consider is which type of file our plugin is going to be aimed at reading. It is advisable to not make the definition too broad, it's easy to have multiple plugins so don't try and over complicate the plugin by having it read many different types of file. Roughly, one plugin should describe a set of data with the same metadata.

Since the CIS plugin is designed to read any data which CIS produces, the signature matches any file which ends with *.nc* (we also check the source attribute but that is beyond the scope of this tutorial):

```
def get_file_signature(self):
    return [r'\.nc']
```

This uses a wildcard string to tell CIS which files do and don't match for our product.

Note: For an introduction to regular expressions see, for example, https://docs.python.org/2/howto/regex.html

---

The next step is to complete the `AProduct.create_coords()` method. CIS uses this method to create a set of coordinates from the data, so it needs to return any appropriate coordinates in the shape that CIS expects it.

There are a number of low-level data reading routines within CIS that can help you read in your data. For the CIS plugin (which is reading netCDF data) we use two methods from the `cis.data_io.netcdf` module: `read_many_files_individually` and `get_metadata`. We also import the `Coord` data type, which is where we store the coordinates that we've read, and `UngriddedCoordinates` - which is what we return to CIS.

Note: In python it's very easy to import classes and methods from other modules within your package, and across packages using the from and import commands. The file-reading routines used here are used by many of the other data products. See the *API* section for further details about using CIS as a python library.

Don't worry too much about what these methods do at this stage, just use the import lines below and you should be fine.

```
def create_coords(self, filenames, usr_variable=None):
    from cis.data_io.netcdf import read_many_files_individually, get_metadata
    from cis.data_io.Coord import Coord, CoordList
    from cis.data_io.ungridded_data import UngriddedCoordinates
```

Next, we create a list of netCDF variable names which we know are stored in our file and send that to the file reading routine:

```
var_data = read_many_files_individually(filenames, ["longitude","latitude", "time"])
```

Then we create a `CoordList` to store our coordinates in, a `Coord` for each of those coordinate variables, and then just give them a short label for plotting purposes (x,y,z etc) – it is strongly advisable that you use the standard definitions used below for your axis definitions (and use z for altitude and p for pressure).

```
coords = CoordList()
coords.append(Coord(var_data["longitude",get_metadata(var_data["longitude"][0]),
↪axis="x"))
coords.append(Coord(var_data["latitude",get_metadata(var_data["latitude"][0]),
↪axis="y"))
coords.append(Coord(var_data["time",get_metadata(var_data["time"][0]),axis="t"))
```

That's it, now we can return those coordinates in a way that CIS will understand:

```
return UngriddedCoordinates(coords)
```

The last method we have to write is the `AProduct.create_data_object()` method, which is used by CIS to pull together the coordinates and a particular data variable into an `UngriddedData` object. It's even simpler than the previous method. We can use the same `read_many_files_individually` method as we did before, and this time pass it the variable the user has asked for:

```
def create_data_object(self, filenames, variable):
    from cis.data_io.ungridded_data import UngriddedData
    usr_var_data = read_many_files_individually(filenames,variable)[variable]
```

Then we create the coordinates using the `create_coords()` method we've just written:

```
coords = self.create_coords(filename)
```

And finally we return the ungridded data, this combines the coordinates from the file and the variable requested by the user:

```
return UngriddedData(usr_var_data, get_metadata(usr_var_data[0]),coords)
```

Bringing it all together, tidying it up a bit and including some error catching gives us:

```
import logging
from cis.data_io.products.AProduct import AProduct
from cis.data_io.netcdf import read_many_files_individually, get_metadata

class cis(AProduct):

    def get_file_signature(self):
        return [r'cis\-.*\.nc']

    def create_coords(self, filenames, usr_variable=None):
        from cis.data_io.Coord import Coord, CoordList
        from cis.data_io.ungridded_data import UngriddedCoordinates
        from cis.exceptions import InvalidVariableError

        variables = [("longitude", "x"), ("latitude", "y"), ("altitude", "z"), ("time
↪", "t"), ("air_pressure", "p")]

        logging.info("Listing coordinates: " + str(variables))

        coords = CoordList()
        for variable in variables:
            try:
                var_data = read_many_files_individually(filenames,
↪variable[0])[variable[0]]
                coords.append(Coord(var_data, get_metadata(var_data[0]),
↪axis=variable[1]))
            except InvalidVariableError:
```

```
                pass

        return UngriddedCoordinates(coords)

    def create_data_object(self, filenames, variable):
        from cis.data_io.ungridded_data import UngriddedData
        usr_var_data = read_many_files_individually(filenames,variable)[variable]
        coords = self.create_coords(filename)
        return UngriddedData(usr_var_data, get_metadata(usr_var_data[0]), coords)
```

Medium

For this example we will look at the AERONET data reading plugin. AERONET is a ground based sun-photometer network that produces time-series data for each groundstation in a csv based text file. There is some information about the ground station in the header of the file, and then a table of data with a time column, and a column for each of the measured values.

The `AProduct.get_file_signature()` method is straightforward, so we first consider the `AProduct.create_coords()` method. Here we have actually shifted all of the work to a private method called `_create_coord_list()`, for reasons which we will explain shortly:

```
def create_coords(self, filenames, variable=None):
    return UngriddedCoordinates(self._create_coord_list(filenames))
```

Note: In python there is not really such a thing as a 'private' method as there is in Java and C#, but we can signify that a method shouldn't be accessed externally by starting its name with one or two underscores.

In this method we import an AERONET data reading routine:

```
def _create_coord_list(self, filenames, data=None):
    from cis.data_io.ungridded_data import Metadata
    from cis.data_io.aeronet import load_multiple_aeronet
```

This data reading routine actually performs much of the hard work in reading the AERONET file:

```
if data is None:
    data = load_multiple_aeronet(filenames)
```

Note that we only read the files if Data is None, that is if we haven't been passed any data already.

Note: The load_multiple_aeronet routine uses the numpy genfromtext method to read in the csv file. This is a very useful method for reading text based files as it allows you to define the data formats of each of the columns, tell it which lines to ignore as comments and, optionally, mask out any missing values. This method would provide a useful example for reading different kinds of text based file.

We just have to describe (add metadata to) each of the components in this method:

```
coords = CoordList()
coords.append(Coord(data['longitude'], Metadata(name="Longitude",shape=(len(data),),
↪units="degrees_east", range=(-180, 180))))
coords.append(Coord(data['latitude'], Metadata(name="Latitude",shape=(len(data),),
↪units="degrees_north", range=(-90, 90))))
```

```
coords.append(Coord(data['altitude'], Metadata(name="Altitude",shape=(len(data),),␣
↪units="meters")))
time_coord = Coord(data["datetime"], Metadata(name="DateTime",standard_name='time',␣
↪shape=(len(data),),units="DateTime Object"), "X")
```

Note that we've explicitly added things like units and a shape. These are sometimes already populated for us when reading e.g. NetCDF files, but in the case of AERONET data we have to fill it out 'by hand'.

Internally CIS uses a 'standard' time defined as fractional days since the 1st January 1600, on a Gregorian calendar. This allows us to straightforwardly compare model and measurement times regardless of their reference point. There are many helper methods for converting different date-time formats to this standard time, here we use `Coord.convert_datetime_to_standard_time()`, and then include the coordinate in the coordinate list:

```
time_coord.convert_datetime_to_standard_time()
coords.append(time_coord)
```

Finally we return the coordinates:

```
return coords
```

For the `create_data_object()` method we have the familiar signature and import statements:

```
def create_data_object(self, filenames, variable):
    from cis.data_io.aeronet import load_multiple_aeronet
    from cis.exceptions import InvalidVariableError
```

We can pass the job of reading the data to our AERONET reading routine – catching any errors which occur because the variable doesn't exist.

```
try:
    data_obj = load_multiple_aeronet(filenames, [variable])
except ValueError:
    raise InvalidVariableError(variable + " does not exist in " + str(filenames))
```

Note: Notice here that we're catching a `ValueError` – which Numpy throws when it can't find the specified variable in the data, and rethrowing the same error as an `InvalidVariableError`, so that CIS knows how to deal with it. Any plugins should use this error when a user specifies a variable which isn't within the specified file.

Now we have read the data, we load the coordinate list, but notice that we also pass in the data we've just read. This is why we created a separate coordinate reading routine earlier: The data containing the coordinates has already been read in the line above, so we don't need to read it twice, we just need to pull out the coordinates. This saves time opening the file multiple times, and can be a useful pattern to remember for files which aren't direct access (such as text files).

```
coords = self._create_coord_list(filenames, data_obj)
```

Finally we return the complete data object, including some associated metadata and the coordinates.

```
return UngriddedData(data_obj[variable], Metadata(name=variable, long_name=variable,␣
↪shape=(len(data_obj),), missing_value=-999.0), coords)
```

Here's the plugin in full:

```
class Aeronet(AProduct):

```

```
    def get_file_signature(self):
        return [r'.*\.lev20']

    def _create_coord_list(self, filenames, data=None):
        from cis.data_io.ungridded_data import Metadata
        from cis.data_io.aeronet import load_multiple_aeronet

        if data is None:
            data = load_multiple_aeronet(filenames)

        coords = CoordList()
        coords.append(Coord(data['longitude'], Metadata(name="Longitude",␣
↪shape=(len(data),),
                                                        units="degrees_east", range=(-
↪180, 180))))
        coords.append(Coord(data['latitude'], Metadata(name="Latitude",␣
↪shape=(len(data),),
                                                       units="degrees_north", range=(-
↪90, 90))))
        coords.append(Coord(data['altitude'], Metadata(name="Altitude",␣
↪shape=(len(data),), units="meters")))
        time_coord = Coord(data["datetime"], Metadata(name="DateTime", standard_name=
↪'time', shape=(len(data),),
                                                      units="DateTime Object"), "X")
        time_coord.convert_datetime_to_standard_time()
        coords.append(time_coord)

        return coords

    def create_coords(self, filenames, variable=None):
        return UngriddedCoordinates(self._create_coord_list(filenames))

    def create_data_object(self, filenames, variable):
        from cis.data_io.aeronet import load_multiple_aeronet
        from cis.exceptions import InvalidVariableError

        try:
            data_obj = load_multiple_aeronet(filenames, [variable])
        except ValueError:
            raise InvalidVariableError(variable + " does not exist in " +␣
↪str(filenames))

        coords = self._create_coord_list(filenames, data_obj)

        return UngriddedData(data_obj[variable],
                             Metadata(name=variable, long_name=variable,␣
↪shape=(len(data_obj),), missing_value=-999.0),
                             coords)
```

Advanced

This more advanced tutorial will cover some of the difficulties when reading in data which differs significantly from the structure CIS expects, and/or has little metadata in the associated files. We take the MODIS L2 plugin as our example, and discuss each method in turn.

There are a number of specific MODIS L2 products which we have tested using this plugin, each with their own file signature, and so in this plugin we take advantage of the fact that the regular expression returned by get_file_signature

can be a list. This way we create a simple regular expression for each MODIS L2 products that we're supporting - rather than trying to create one, more complicated, regular expression which matches just these products at the exclusion of all others:

```
def get_file_signature(self):
    product_names = ['MYD06_L2', 'MOD06_L2', 'MYD04_L2', 'MOD04_L2']
    regex_list = [r'.*' + product + '.*\.hdf' for product in product_names]
    return regex_list
```

We have implemented the optional get_variable_names method here because MODIS files sometimes contain variables which CIS is unable to handle due to their irregular shape. We only want to report the variable which CIS can read so we check each variable before adding it to the list of variables we return. We know that MODIS only contains SD variables so we can ignore any other types.

Note: HDF files can contain both Vdatas (VD) and Scientific Datasets (SD) data collections (among others). These are stored and accessed quite differently, which makes dealing with these files quite fiddly - we often have to treat each case separately. In this case we know MODIS files only have SD datasets which makes things a bit simpler.

```
def get_variable_names(self, filenames, data_type=None):
    import pyhdf.SD

    # Determine the valid shape for variables
    sd = pyhdf.SD.SD(filenames[0])
    datasets = sd.datasets()
    valid_shape = datasets['Latitude'][1]  # Assumes that latitude shape == longitude␣
↪shape (it should)

    variables = set([])
    for filename in filenames:
        sd = pyhdf.SD.SD(filename)
        for var_name, var_info in sd.datasets().iteritems():
            if var_info[1] == valid_shape:
                variables.add(var_name)

    return variables
```

MODIS data often has a scale factor built in, and stored against each variable, this method reads that scale factor for a particular variable and checks it against our built-in list of scale factors.

```
def __get_data_scale(self, filename, variable):
    from cis.exceptions import InvalidVariableError
    from pyhdf import SD

    try:
        meta = SD.SD(filename).datasets()[variable][0][0]
    except KeyError:
        raise InvalidVariableError("Variable "+variable+" not found")

    for scaling in self.modis_scaling:
        if scaling in meta:
            return scaling
    return None
```

In order to use data which has been scaled, we re-scale it on reading. This creates some overhead in the reading of the data, but saves considerable time when performing other operations on it later in the process. Routines like this can often be adapted from available Fortran or IDL routines (assuming no python routines are available) for your data.

```
def __field_interpolate(self,data,factor=5):
    '''
    Interpolates the given 2D field by the factor,
    edge pixels are defined by the ones in the centre,
    odd factors only!
    '''
    import numpy as np

    logging.debug("Performing interpolation...")

    output = np.zeros((factor*data.shape[0],factor*data.shape[1]))*np.nan
    output[int(factor/2)::factor,int(factor/2)::factor] = data
    for i in range(1,factor+1):
        output[(int(factor/2)+i):(-1*factor/2+1):factor,:] = i*((output[int(factor/
↪2)+factor::factor,:]-output[int(factor/2):(-1*factor):factor,:])
                                                                /
↪float(factor))+output[int(factor/2):(-1*factor):factor,:]
    for i in range(1,factor+1):
        output[:,(int(factor/2)+i):(-1*factor/2+1):factor] = i*((output[:,int(factor/
↪2)+factor::factor]-output[:,int(factor/2):(-1*factor):factor])
                                                                /
↪float(factor))+output[:,int(factor/2):(-1*factor):factor]
    return output
```

Next we read the coordinates from the file (using the same method of factoring out as we used in the Aeronet case).

```
def _create_coord_list(self, filenames, variable=None):
    import datetime as dt

    variables = ['Latitude', 'Longitude', 'Scan_Start_Time']
    logging.info("Listing coordinates: " + str(variables))
```

As usual we rely on the lower level IO reading routines to provide the raw data, in this case using the hdf.read routine.

```
sdata, vdata = hdf.read(filenames, variables)
```

Note: Notice we have to put the vdata data somewhere, even though we don't use it in this case.

We have to check whether we need to scale the coordinates to match the variable being read:

```
apply_interpolation = False
if variable is not None:
    scale = self.__get_data_scale(filenames[0], variable)
    apply_interpolation = True if scale is "1km" else False
```

Then we can read the coordinates, one at a time. We know the latitude information is stored in an SD dataset called Latitude, so we read that and interpolate it if needed.

```
lat = sdata['Latitude']
sd_lat = hdf.read_data(lat, "SD")
lat_data = self.__field_interpolate(sd_lat) if apply_interpolation else sd_lat
lat_metadata = hdf.read_metadata(lat, "SD")
lat_coord = Coord(lat_data, lat_metadata,'Y')
```

The same for Longitude:

```
lon = sdata['Longitude']
lon_data = self.__field_interpolate(hdf.read_data(lon,"SD")) if apply_interpolation␣
↪else hdf.read_data(lon,"SD")
lon_metadata = hdf.read_metadata(lon,"SD")
lon_coord = Coord(lon_data, lon_metadata,'X')
```

Next we read the time variable, remembering to convert it to our internal standard time. (We know that the MODIS' atomic clock time is referenced to the 1st January 1993.)

```
time = sdata['Scan_Start_Time']
time_metadata = hdf.read_metadata(time,"SD")
# Ensure the standard name is set
time_metadata.standard_name = 'time'
time_coord = Coord(time,time_metadata,"T")
time_coord.convert_TAI_time_to_std_time(dt.datetime(1993,1,1,0,0,0))

return CoordList([lat_coord,lon_coord,time_coord])
```

```
def create_coords(self, filenames, variable=None):
    return UngriddedCoordinates(self._create_coord_list(filenames))
```

For the create_data_object we are really just pulling the above methods together to read the specific variable the user has requested and combine it with the coordinates.

```
def create_data_object(self, filenames, variable):
    logging.debug("Creating data object for variable " + variable)

    # reading coordinates
    # the variable here is needed to work out whether to apply interpolation to the␣
↪lat/lon data or not
    coords = self._create_coord_list(filenames, variable)

    # reading of variables
    sdata, vdata = hdf.read(filenames, variable)

    # retrieve data + its metadata
    var = sdata[variable]
    metadata = hdf.read_metadata(var, "SD")

    return UngriddedData(var, metadata, coords)
```

We have also implemented the `AProduct.get_file_format()` method which allows some associated tools (for example the CEDA_DI tool) to use CIS to index files which they wouldn't otherwise be able to read. We just return a file format descriptor as a string.

```
def get_file_format(self, filenames):
    """
    Get the file format
    :param filenames: the filenames of the file
    :return: file format
    """

    return "HDF4/ModisL2"
```

The full MODIS L2 plugin is rather long to show but can be downloaded here.

Data plugin reference

This section provides a reference describing the expected behaviour of each of the functions a plugin can implement. The following methods are mandatory:

`AProduct.`**`get_file_signature`**`()`

This method should return a list of regular expressions, which CIS uses to decide which data product to use for a given file. If more than one regular expression is provided in the list then the file can match *any* of the expressions. The first product with a signature that matches the filename will be used. The order in which the products are searched is determined by the priority property, highest value first; internal products generally have a priority of 10.

For example, this would match all files with a name containing the string 'CODE' and with the 'nc' extension.:

```
return [r'.*CODE*.nc']
```

Note: If the signature has matched the framework will call `AProduct.get_file_type_error()`, this gives the product a chance to open the file and check the contents.

Returns A list of regex to match the product's file naming convention.

Return type list

`AProduct.`**`create_coords`**(*filenames*)

Reads the coordinates from one or more files. Note that this method may have to make certain assumptions about the file in order to return a single coordinate set. The user should be warned through the logger if this is the case.

Parameters **`filenames`** (*`list`*) – List of filenames to read coordinates from

Returns `CommonData` object

`AProduct.`**`create_data_object`**(*filenames*, *variable*)

Create and return an `CommonData` object for a given variable from one or more files.

Parameters

- **`filenames`** (*`list`*) – List of filenames of files to read
- **`variable`** (*`str`*) – Variable to read from the files

Returns An `CommonData` object representing the specified variable

Raises

- **`FileIOError`** – Unable to read a file
- **`InvalidVariableError`** – Variable not present in file

While these may be implemented optionally:

`AProduct.`**`get_variable_names`**(*filenames*, *data_type=None*)

Get a list of available variable names from the filenames list passed in. This general implementation can be overridden in specific products to include/exclude variables which may or may not be relevant. The data_type parameter can be used to specify extra information.

Parameters

- **`filenames`** (*`list`*) – List of string filenames of files to be read from

- **data_type** (*str*) – 'SD' or 'VD' to specify only return SD or VD variables from HDF files. This may take on other values in specific product implementations.

Returns A set of variable names as strings

Return type str

`AProduct.`**`get_file_type_error`**(*filename*)

Check a single file to see if it is of the correct type, and if not return a list of errors. If the return is None then there are no errors and this is the correct data product to use for this file.

This method gives a mechanism for a data product to identify itself as the correct product when a specific enough file signature cannot be provided. For example GASSP is a type of NetCDF file and so filenames end with .nc but so do other NetCDF files, so the data product opens the file and looks for the GASSP version attribute, and if it doesn't find it returns an error.

Parameters **filename** (*str*) – The filename for the file

Returns List of errors, or None

Return type list or None

`AProduct.`**`get_file_format`**(*filename*)

Returns a file format hierarchy separated by slashes, of the form `TopLevelFormat/SubFormat/SubFormat/Version`. E.g. `NetCDF/GASSP/1.0`, `ASCII/ASCIIHyperpoint` or `HDF4/CloudSat`. This is mainly used within the ceda_di indexing tool. If not set it will default to the products name.

A filename of an example file can be provided to enable the determination of, for example, a dataset version number.

Parameters **filename** (*str*) – Filename of file to be inspected

Returns File format, of the form `[parent/]format/specific instance/version`, or the class name

Return type str

Raises FileFormatError if there is an error

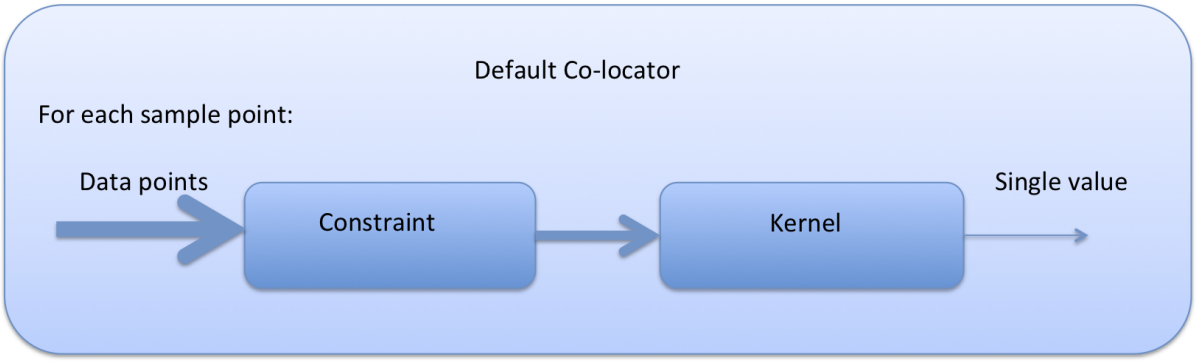
CHAPTER 17

Analysis plugin development

Users can write their own plugins for performing the collocation of two data sets. There are three different types of plugin available for collocation, first we will describe the overall design and how these different components interact, then each will be described in more detail.

Basic collocation design

The diagram below demonstrates the basic design of the collocation system, and the roles of each of the components. In the simple case of the default collocator (which returns only one value) the *Collocator* loops over each of the sample points, calls the relevant *Constraint* to reduce the number of data points, and then the *Kernel* which returns a single value, which the collocator stores.



Kernel

A kernel is used to convert the constrained points into values in the output. There are two sorts of kernel one which act on the final point location and a set of data points (these derive from `Kernel`) and the more specific kernels which act upon just an array of data (these derive from `AbstractDataOnlyKernel`, which in turn derives from `Kernel`). The data only kernels are less flexible but should execute faster. To create a

new kernel inherit from `Kernel` and implement the abstract method `Kernel.get_value()`. To make a data only kernel inherit from `AbstractDataOnlyKernel` and implement `AbstractDataOnlyKernel.get_value_for_data_only()` and optionally overload `AbstractDataOnlyKernel.get_value()`. These methods are outlined below.

`Kernel.`**`get_value`**(*point*, *data*)

This method should return a single value (if `Kernel.return_size` is 1) or a list of n values (if `Kernel.return_size` is n) based on some calculation on the data given a single point.

The data is deliberately left unspecified in the interface as it may be any type of data, however it is expected that each implementation will only work with a specific type of data (gridded, ungridded etc.) Note that this method will be called for every sample point and so could become a bottleneck for calculations, it is advisable to make it as quick as is practical. If this method is unable to provide a value (for example if no data points were given) a ValueError should be thrown.

Parameters

- **`point`** – A single HyperPoint
- **`data`** – A set of data points to reduce to a single value

Returns For return_size=1 a single value (number) otherwise a list of return values, which represents some operation on the points provided

Raises **`ValueError`** – When the method is unable to return a value

`AbstractDataOnlyKernel.`**`get_value_for_data_only`**(*values*)

This method should return a single value (if `Kernel.return_size` is 1) or a list of n values (if `Kernel.return_size` is n) based on some calculation on the the values (a numpy array).

Note that this method will be called for every sample point in which data can be placed and so could become a bottleneck for calculations, it is advisable to make it as quick as is practical. If this method is unable to provide a value (for example if no data points were given) a ValueError should be thrown. This method will not be called if there are no values to be used for calculations.

Parameters **`values`** – A numpy array of values (can not be none or empty)

Returns A single data item if return_size is 1 or a list of items containing `Kernel.return_size` items

Raises **`ValueError`** – If there are any problems creating a value

Constraint

The constraint limits the data points for a given sample point. The user can also add a new constraint mechanism by subclassing `Constraint` and providing an implementation for `Constraint.constrain_points()`. If more control is needed over the iteration sequence then the `Constraint.get_iterator()` method can also be overloaded. Note however that this may not be respected by all collocators, who may still iterate over all sample data points. It is possible to write your own collocator (or extend an existing one) to ensure the correct iterator is used - see the next section. Both these methods, and their signatures, are outlined below.

`Constraint.`**`constrain_points`**(*point*, *data*)

This method should return a subset of the data given a single reference point. It is expected that the data returned should be of the same type as that given - but this isn't mandatory. It is possible that this function will return zero points (no data), the collocation class is responsible for providing a fill_value.

Parameters

- **`point`** (*`HyperPoint`*) – A single HyperPoint

- **data** – A set of data points to be reduced

Returns A reduced set of data points

`Constraint.`**`get_iterator`**(*missing_data_for_missing_sample*, *coord_map*, *coords*, *data_points*, *shape*, *points*, *output_data*)

Iterator to iterate through the points needed to be calculated. The default iterator, iterates through all the sample points calling `Constraint.constrain_points()` for each one.

Parameters

- **missing_data_for_missing_sample** – If true anywhere there is missing data on the sample then final point is missing; otherwise just use the sample
- **coord_map** – Coordinate map - list of tuples of indexes of hyperpoint coord, data coords and output coords
- **coords** – The coordinates to map the data onto
- **data_points** – The (non-masked) data points
- **shape** – Shape of the final data values
- **points** – The original points object, these are the points to collocate
- **output_data** – Output data set

Returns Iterator which iterates through (sample indices, hyper point and constrained points) to be placed in these points

To enable a constraint to use a `AbstractDataOnlyKernel`, the method `get_iterator_for_data_only()` should be implemented (again though, this may be ignored by a collocator). An example of this is the `BinnedCubeCellOnlyConstraint.get_iterator_for_data_only()` implementation.

Collocator

Another plugin which is available is the collocation method itself. A new one can be created by subclassing `Collocator` and providing an implementation for `Collocator.collocate()`. This method takes a number of sample points and applies the given constraint and kernel methods on the data for each of those points. It is responsible for returning the new data object to be written to the output file. As such, the user could create a collocation routine capable of handling multiple return values from the kernel, and hence creating multiple data objects, by creating a new collocation method.

Note: The collocator is also responsible for dealing with any missing values in sample points. (Some sets of sample points may include values which may or may not be masked.) Sometimes the user may wish to mask the output for such points, the `missing_data_for_missing_sample` attribute is used to determine the expected behaviour.

The interface is detailed here:

`Collocator.`**`collocate`**(*points*, *data*, *constraint*, *kernel*)

The method is responsible for setting up and running the collocation. It should take a set of data and map that onto the given (sample) points using the constraint and kernel provided.

Parameters

- **points** – A set of sample points onto which we will collocate some other 'data'
- **data** – Some other data to be collocated onto the 'points'

- **constraint** – A `Constraint` instance which provides a `Constraint.constrain_points()` method, and optionally an `Constraint.get_iterator()` method
- **kernel** – A `Kernel` instance which provides a `Kernel.get_value()` method

Returns One or more `CommonData` (or subclasses of) objects whose coordinates lie on the points defined above.

Implementation

For all of these plugins any new variables, such as limits, constraint values or averaging parameters, are automatically set as attributes in the relevant object. For example, if the user wanted to write a new constraint method (`AreaConstraint`, say) which needed a variable called `area`, this can be accessed with `self.area` within the constraint object. This will be set to whatever the user specifies at the command line for that variable, e.g.:

```
$ ./cis.py col my_sample_file rain:"model_data_?.nc"::AreaConstraint,area=6000,fill_
↪value=0.0:nn_gridded
```

Example implementations of new collocation plugins are demonstrated below for each of the plugin types:

```
class MyCollocator(Collocator):

    def collocate(self, points, data, constraint, kernel):
        values = []
        for point in points:
            con_points = constraint.constrain_points(point, data)
            try:
                values.append(kernel.get_value(point, con_points))
            except ValueError:
                values.append(constraint.fill_value)
        new_data = LazyData(values, data.metadata)
        new_data.missing_value = constraint.fill_value
        return new_data


class MyConstraint(Constraint):

    def constrain_points(self, ref_point, data):
        con_points = []
        for point in data:
            if point.value > self.val_check:
                con_points.append(point)
        return con_points


class MyKernel(Kernel):

    def get_value(self, point, data):
        nearest_point = point.furthest_point_from()
        for data_point in data:
            if point.compdist(nearest_point, data_point):
                nearest_point = data_point
        return nearest_point.val
```

CHAPTER 18

Maintenance and Developer Guide

Source files

The cis source code is hosted at https://github.com/cedadev/jasmin_cis.git, while the conda recipes and other files are hosted here: https://github.com/cistools.

Test suites

The unit tests suite can be ran using Nose readily. Just go the root of the repository (i.e. cis) and type `nosetests cis/test/unit` and this will run the full suite of tests.

A comprehensive set of integration tests are also provided. There is a folder full of test data at: `/group_workspaces/jasmin/cis/cis_repo_test_files` which has been compressed and is available as a tar inside that folder.

To add files to the folder simply copy them in then delete the old tar file and create a new one with:

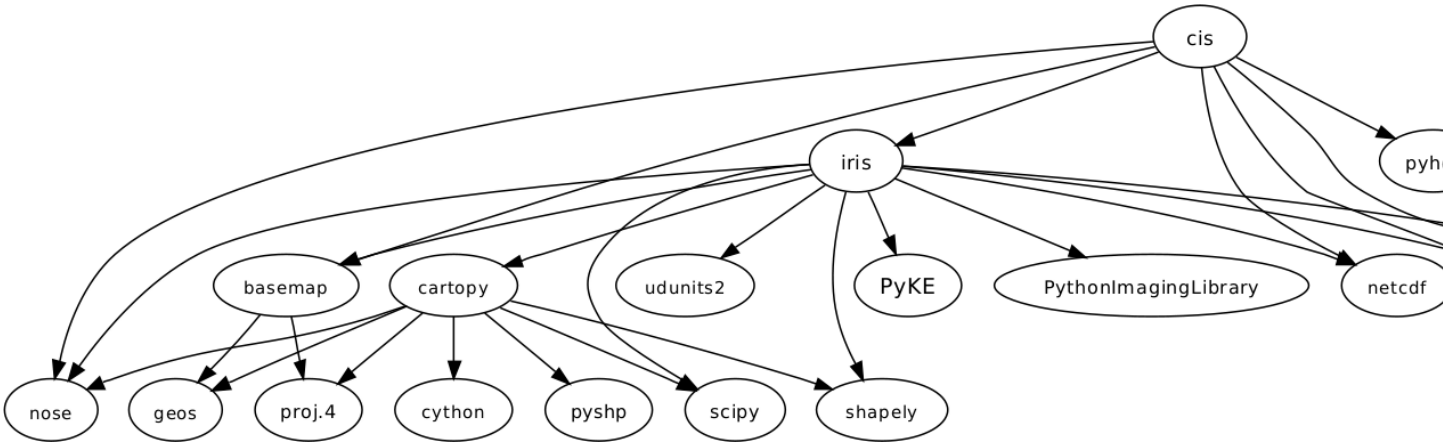
```
tar --dereference -zcvf cis_repo_test_files.tar.gz .
```

Ignore warning about file changing - it is because the tar file is in the directory. Having the tar file in the directory, however, means the archive can be easily unpacked, without creating an intermediate folder. To make the integration tests run this needs to be copied to the local machine and decompressed. Then set the environment variable `CIS_DATA_HOME` to the location of the data sets, and run `nosetests cis/test/integration`.

There are also a number of plot tests available under the `test/plot_tests` directory in the `test_plotting.py` script. These integration tests use matplotlib to perform a byte-wise comparision of the output against reference plots, using a pre-defined tolerance. Any tests which fail can be evaluated using the `idiff.py` tool in the same directory. Running this will present a graphical interface showing the reference plot, the test output, and the difference between them. You can either choose to accept the difference which will move the test output to the reference directory, or reject it.

Dependencies

A graph representing the dependency tree can be found at `doc/cis_dependency.dot` (use XDot to read it)



Creating a Release

To carry out intermediate releases follow this procedure:

1. Check the version number and status is updated in the CIS source code (cis/__init__.py)
2. Tag the new version on Github with new version number and release notes.
3. Create a tarball - use `python setup.py egg_info sdist` in the cis root dir.
4. Install this onto the release virtual environment: this is at `/group_workspaces/jasmin/cis/cis_dev_venv`. So activate the venv, upload the tarball somewhere on the GWS and then do `pip install <LOCATION_OF_TARBALL>`.
5. Create an anaconda build on each platform (OS X, Linux and Windows) - see below.
6. Request Phil Kershaw upload the tarball to PyPi. (Optional)

For a release onto JASMIN, complete the steps above and then ask Alan Iwi to produce an RPM, deploy it on a test VM, confirm functionality then rollout across full JAP and LOTUS nodes.

Anaconda Build

The Anaconda build recipes for CIS and the dependencies which can't be found either in the core channel, or in SciTools are stored in their own github repository here. To build a new CIS package clone the conda-recipes repository and then run the following command:

```
$ conda build -c cistools -c scitools cis
```

By default this will run the full unit-test suite before successful completion. You can also optionally run the integration test suite by specifying the CIS_DATA_HOME environment variable.

To upload the package to the cistools channel on Anaconda.org use:

```
$ binstar upload <package_location> -u cistools
```

Alternatively, when creating release candidates you may wish to upload the package to the 'beta' channel. This gives an opportunity to test the packaging and installation process on a number of machines. To do so, use:

```
$ binstar upload <package_location> -u cistools --channel beta
```

To install cis from the beta channel use:

```
$ conda install -c https://conda.binstar.org/cistools/channel/beta -c cistools -c␣
↪scitools cis
```

Documentation

The documentation and API reference are both generated using a mixture of markdown and autogenerated documentation using the Sphinx autodoc package. Build the documentation using:

```
python setup.py build_sphinx
```

This will output the documentation in html under the directory `doc/_build/html`.

Continuous Integration Server

JASMIN provide a Jenkins CI Server on which the CIS unit and integration tests are run whenever origin/master is updated. The integration tests take approximately 7 hours to run whilst the unit tests take about 5s. The Jenkins server is hosted on jasmin-sci1-dev at `/var/lib/jenkins` and is accessed at http://jasmin-sci1-dev.ceda.ac.uk:8080/

We also have a Travis cloud instance (https://travis-ci.org/cedadev/cis) which in principle allows us to build and test on both Linux and OS X. There are unit test builds currently working but because of a hard time limit on builds (120 minutes) the integration tests don't currently run.

Copying files to the CI server

The contents of the test folder will not be automatically copied across to the Jenkins directory, so if you add any files to the folder you'll need to manually copy them to the Jenkins directory or the integration tests will fail. The directory is `/var/lib/jenkins/workspace/CIS Integration Tests/cis/test/test_files/`. This is not entirely simple because:

- We don't have write permissions on the test folder
- Jenkins doesn't have read permissions for the CIS group_workspace

In order to copy files across we have done the following:

1. Copy the files we want to /tmp
2. Open up the CIS Integration Tests webpage and click 'Configure'
3. Scroll down to 'Build' where the shell script to be executed is found and insert a line to copy the file to the directory, e.g. `cp /tmp/file.nc /var/lib/jenkins/workspace/CIS Integration Tests/cis/test/test_files`
4. Run the CIS Integration Tests
5. Remove the line from the build script
6. Remove the files from /tmp

Problems with Jenkins

Sometimes the Jenkins server experiences problems which make it unusable. One particular issue we've encountered more than once is that Jenkins occasionally loses all its stylesheets and then becomes impossible to use. Asking CEDA support (or Phil Kershaw) to restart Jenkins should solve this.

CHAPTER 19

Indices and tables

- genindex
- modindex
- search

Index

G

R