EGADS Documentation

Release 0.8.4

EUFAR

Oct 04, 2017

CONTENTS

CHAPTER

ONE

INTRODUCTION

The EGADS (EUFAR General Airborne Data-processing Software) core is a Python-based library of processing and file I/O routines designed to help analyze a wide range of airborne atmospheric science data. EGADS purpose is to provide a benchmark for airborne data-processing through its community-provided algorithms, and to act as a reference by providing guidance to researchers with an open-source design and well-documented processing routines.

Python is used in development of EGADS due to its straightforward syntax and portability between systems. Users interact with data processing algorithms using the Python command-line, by creating Python scripts for more complex tasks, or by using the EGADS GUI for a simplified interaction. The core of EGADS is built upon a data structure that encapsulates data and metadata into a single object. This simplifies the housekeeping of data and metadata and allows these data to be easily passed between algorithms and data files. Algorithms in EGADS also contain metadata elements that allow data and their sources to be tracked through processing chains.

Note: Even if EGADS is easily accessible, a certain knowledge in Python is still required to use EGADS.

CHAPTER

TWO

INSTALLATION

The latest version of EGADS can be obtained from https://github.com/eufarn7sp/egads

2.1 Prerequisites

Use of EGADS requires the following packages:

- Python 2.7.10 or newer. Available at https://www.python.org/
- numpy 1.10.1 or newer. Available at http://numpy.scipy.org/
- scipy 0.15.0 or newer. Available at http://www.scipy.org/
- Python netCDF4 libraries 1.1.9 or newer. Available at https://pypi.python.org/pypi/netCDF4
- python_dateutil 2.4.2 or newer. Available at https://pypi.python.org/pypi/python-dateutil

2.2 Optional Packages

The following are useful when using or compiling EGADS:

- IPython - An optional package which simplifies Python command line usage (http://ipython.scipy.org). IPython is an enhanced interactive Python shell which supports tab-completion, debugging, command history, etc.
- setuptools - An optional package which allows easier installation of Python packages (http://pypi.python.org/pypi/setuptools). It gives access to the `easy_install` command which allows packages to be downloaded and installed in one step from the command line.

2.3 Installation

Since EGADS is a pure Python distribution, it does not need to be built. However, to use it, it must be installed to a location on the Python path. To install EGADS, first download and decompress the file. From the directory containing the file `setup.py`, type `python setup.py install` or `pip install egads` from the command line. To install to a user-specified location, type `python setup.py install --prefix=$MYDIR`. To avoid the installation of dependencies, use the option `--no-depts`. On Linux systems, the installation of EGADS in the user home directory is encouraged to ensure the proper operation of the EGADS logging system and of the new Graphical User Interface algorithm creation system.

2.4 Testing

To test EGADS after it is installed, run the run_tests.py Python script, or from Python, run the following commands:

```
>>> import egads
>>> egads.test()
```

2.5 Log

A logging system has been introduced in EGADS since the version 0.7.0. By default, the output file is available in the 'Python local site-packages/EGADS x.x.x/egads' directory and the logging level has been set to INFO. Both options for logging level and logging location have been set in a config file. Both options can be changed through EGADS using the `egads.set_log_options()` function, by passing a dictionary of option keys and values:

```
>>> import egads
>>> config_dict = {'level': 'INFO', 'path': '/path/to/log/directory/'}
>>> egads.set_log_options(config_dict)
>>> exit()
```

Actual options to control the logging system are for now:

- `level`: the logging level (`DEBUG`, `INFO`, `WARNING`, `CRITICAL`, `ERROR`).
- `path`: the path of the file containing all logs.

New logging options will be loaded at the next import of EGADS. Logging levels are the standard Python ones (`DEBUG`, `INFO`, `WARNING`, `CRITICAL`, `ERROR`). It is also possible to change dynamically the logging level in a script:

```
>>> egads.change_log_level('DEBUG')
```

That possibility is not permanent and will last until the script run is over.

CHAPTER

THREE

TUTORIAL

3.1 Exploring EGADS

The simplest way to start working with EGADS is to run it from the Python command line. To load EGADS into the Python name-space, simply import it:

```
>>> import egads
```

You may then begin working with any of the algorithms and functions contained in EGADS.

There are several useful methods to explore the routines contained in EGADS. The first is using the Python built-in `dir()` command:

```
>>> dir(egads)
```

returns all the classes and subpackages contained in EGADS. EGADS follows the naming conventions from the Python Style Guide (http://www.python.org/dev/peps/pep-0008), so classes are always `MixedCase`, functions and modules are generally `lowercase` or `lowercase_with_underscores`. As a further example,

```
>>> dir(egads.input)
```

would returns all the classes and subpackages of the `egads.input` module.

Another way to explore EGADS is by using tab completion, if supported by your Python installation. Typing

```
>>> egads.
```

then hitting `TAB` will return a list of all available options.

Python has built-in methods to display documentation on any function known as docstrings. The easiest way to access them is using the `help()` function:

```
>>> help(egads.input.NetCdf)
```

or

```
>>> egads.input.NetCdf?
```

will return all methods and their associated documentation for the *`NetCdf`* class.

3.1.1 Simple operations with EGADS

To have a list of file in a directory, use the following function:

```
>>> egads.input.get_file_list('path/to/all/netcdf/files/*.nc')
```

3.2 The EgadsData class

At the core of the EGADS package is a data class intended to handle data and associated metadata in a consistent way between files, algorithms and within the framework. This ensures that important metadata is not lost when combining data form various sources in EGADS.

Additionally, by subclassing the Quantities and Numpy packages, *EgadsData* incorporates unit comprehension to reduce unit-conversion errors during calculation, and supports broad array manipulation capabilities. This section describes how to employ the *EgadsData* class in the EGADS program scope.

3.2.1 Creating EgadsData instances

The *EgadsData* class takes four basic arguments:

- **value** Value to assign to *EgadsData* instance. Can be scalar, array, or other *EgadsData* instance.
- **units** Units to assign to *EgadsData* instance. Should be string representation of units, and can be a compound units type such as 'g/kg', 'm/s^2', 'feet/second', etc.
- **variable metadata** An instance of the *VariableMetadata* type or dictionary, containing keywords and values of any metadata to be associated with this *EgadsData* instance.
- **other attributes** Any other attributes added to the class are automatically stored in the *VariableMetadata* instance associated with the *EgadsData* instance.

The following are examples of creating *EgadsData* instances:

```
>>> x = egads.EgadsData([1,2,3], 'm')
>>> a = [1,2,3,4]
>>> b = egads.EgadsData(a, 'km', b_metadata)
>>> c = egads.EgadsData(28, 'degC', long_name="current temperature")
```

If, during the call to *EgadsData*, no units are provided, but a variable metadata instance is provided with a units property, this will then be used to define the *EgadsData* units:

```
>>> x_metadata = egads.core.metadata.VariableMetadata({'units':'m', 'long_name':
↪'Test Variable'})
>>> x = egads.EgadsData([1,2,3], x_metadata)
>>> print x.units
m
>>> print x.metadata
{'units': 'm', 'long_name': 'Test Variable'}
```

The *EgadsData* is a subclass of the Quantities and Numpy packages. Thus it allows different kind of operations like addition, substraction, slicing, and many more. For each of those operations, a new *EgadsData* class is created with all their attributes.

Note: With mathematical operands, as metadata define an *EgadsData* before an operation, and may not reflect the new *EgadsData*, it has been decided to not keep the metadata attribute. It's the responsability of the user to add a new *VariableMetadata* instance or a dictionary to the new *EgadsData* object. It is not true if a user wants to slice an *EgadsData*. In that case, metadata are automatically attributed to the new *EgadsData*.

3.2.2 Metadata

The metadata object used by *EgadsData* is an instance of *VariableMetadata*, a dictionary object containing methods to recognize, convert and validate known metadata types. It can reference parent metadata objects, such as those from an algorithm or data file, to enable users to track the source of a particular variable.

When reading in data from a supported file type (NetCDF, NASA Ames), or doing calculations with an EGADS algorithm, EGADS will automatically populate the associated metadata and assign it to the output variable. However, when creating an *EgadsData* instance manually, the metadata must be user-defined.

As mentioned, *VariableMetadata* is a dictionary object, thus all metadata are stored as keyword:value pairs. To create metadata manually, simply pass in a dictionary object containing the desired metadata:

```
>>> var_metadata_dict = {'long_name':'test metadata object',
                         '_FillValue':-9999}
>>> var_metadata = egads.core.metadata.VariableMetadata(var_metadata_dict)
```

To take advantage of its metadata recognition capabilities, a `conventions` keyword can be passed with the variable metadata to give a context to these metadata.

```
>>> var_metadata = egads.core.metadata.VariableMetadata(var_metadata_dict,␣
↪conventions='CF-1.0')
```

If a particular *VariableMetadata* object comes from a file or algorithm, the class attempts to assign the `conventions` automatically. While reading from a file, for example, the class attempts to discover the conventions used based on the "Conventions" keyword, if present.

3.2.3 Working with units

EgadsData subclasses Quantities, thus all of the latter's unit comprehension methods are available when using *EgadsData*. This section will outline the basics of unit comprehension. A more detailed tutorial of the unit comprehension capabilities can be found at http://packages.python.org/quantities/

In general, units are assigned to *EgadsData* instances when they are being created.

```
>>> a = egads.EgadsData([1,2,3], 'm')
>>> b = egads.EgadsData([4,5,6], 'meters/second')
```

Once a unit type has been assigned to an *EgadsData* instance, it will remain that class of unit and can only be converted between other types of that same unit. The `rescale` method can be used to convert between similar units, but will give an error if an attempt is made to convert to non-compatible units.

```
>>> a = egads.EgadsData([1,2,3], 'm')
>>> a_km = a.rescale('km')
>>> print a_km
['EgadsData', array([0.001, 0.002, 0.003]), 'km']
>>> a_grams = a.rescale('g')
ValueError: Unable to convert between units of "m" and "g"
```

Likewise, arithmetic operations between *EgadsData* instances are handled using the unit comprehension provided by Quantities, and behave . For example adding units of a similar type is permitted:

```
>>> a = egads.EgadsData(10, 'm')
>>> b = egads.EgadsData(5, 'km')
>>> a + b
['EgadsData', array(5010.0), 'm']
```

But, non-compatible types cannot be added. They can, however, be multiplied or divided:

```
>>> distance = egads.EgadsData(10, 'm')
>>> time = egads.EgadsData(1, 's')
>>> distance + time
ValueError: Unable to convert between units of "s" and "m"
>>> distance/time
['EgadsData', array(10), 'm/s']
```

3.3 Working with raw text files

EGADS provides the *egads.input.text_file_io.EgadsFile* class as a simple wrapper for interacting with generic text files. *EgadsFile* can read, write and display data from text files, but does not have any tools for automatically formatting input or output data.

3.3.1 Opening

To open a text file the *EgadsFile* class, use the `open(filename, permissions)()` method:

```
>>> import egads
>>> f = egads.input.EgadsFile()
>>> f.open('/pathname/filename.txt','r')
```

Valid values for permissions are:

- `r` – Read: opens file for reading only. Default value if nothing is provided.
- `w` – Write: opens file for writing, and overwrites data in file.
- `a` – Append: opens file for appending data.
- `r+` – Read and write: opens file for both reading and writing.

3.3.2 File Manipulation

The following methods are available to control the current position in the file and display more information about the file.

- `f.display_file()` – Prints contents of file out to standard output.
- `f.get_position()` – Returns current position in file as integer.
- `f.seek(location, from_where)` – Seeks to specified location in file. `location` is an integer specifying how far to seek. Valid options for `from_where` are `b` to seek from beginning of file, `c` to seek from current position in file and `e` to seek from the end of the file.
- `f.reset()` – Resets position to beginning of file.

3.3.3 Reading Data

Reading data is done using the `read(size)` method on a file that has been opened with `r` or `r+` permissions:

```
>>> import egads
>>> f = egads.input.EgadsFile()
>>> f.open('myfile.txt','r')
>>> single_char_value = f.read()
>>> multiple_chars = f.read(10)
```

If the `size` parameter is not specified, the `read()` function will input a single character from the open file. Providing an integer value *n* as the `size` parameter to `read(size)` will return *n* characters from the open file.

Data can be read line-by-line from text files using `read_line()`:

```
>>> line_in = f.read_line()
```

3.3.4 Writing Data

To write data to a file, use the `write(data)` method on a file that has been opened with `w`, `a` or `r+` permissions:

```
>>> import egads
>>> f = egads.input.EgadsFile()
>>> f.open('myfile.txt','a')
>>> data = 'Testing output data to a file.\n This text will appear on the 2nd line.
↪'
>>> f.write(data)
```

3.3.5 Closing

To close a file, simply call the `close()` method:

```
>>> f.close()
```

3.3.6 Tutorial

Here is a basic ASCII file, created by EGADS:

```
# The current file has been created with EGADS
# Institution: My Institution
# Author(s): John Doe
time    sea level    corr sea level
1.0    5.0    1.0
2.0    2.0    3.0
3.0    -2.0    -1.0
4.0    0.5    2.5
5.0    4.0    6.0
```

This file has been created with the following commands:

- import EGADS module:

```
>>> import egads
```

- create two main variables, following the official EGADS convention:

```
>>> data1 = egads.EgadsData(value=[5.0,2.0,-2.0,0.5,4.0], units='mm', name=
↪'sea level', scale_factor=1., add_offset=0., _FillValue=-9999)
>>> data2 = egads.EgadsData(value=[1.0,3.0,-1.0,2.5,6.0], units='mm', name=
↪'corr sea level', scale_factor=1., add_offset=0., _FillValue=-9999)
```

- create an independant variable, still by following the official EGADS convention:

```
>>> time = egads.EgadsData(value=[1.0,2.0,3.0,4.0,5.0], units='seconds since␣
↪19700101T00:00:00', name='time')
```

- create a new EgadsFile instance:

```
>>> f = egads.input.EgadsFile()
```

- use the following function to open a new file:

```
>>> f.open('main_raw_file.dat', 'w')
```

- prepare the headers if necessary:

```
>>> headers = '# The current file has been created with EGADS\n# Institution:␣
↪My Institution\n# Author(s): John Doe\n'
>>> headers += time.metadata["long_name"] + '    ' + data1.metadata["long_name
↪"] + '    ' + data2.metadata["long_name"] + '\n''My institution')
```

- prepare an object to receive all data:

```
>>> data = ''
>>> for i, _ in enumerate(time.value):
    ... data += str(time.value[i]) + '    ' + str(data1.value[i]) + '    ' +␣
↪str(data2.value[i]) + '\n'
```

- write the headers and data into the file

```
>>> f.write(headers)
>>> f.write(data)
```

- and do not forget to close the file:

```
>>> f.close()
```

3.4 Working with CSV files

egads.input.text_file_io.EgadsCsv is designed to easily input or output data in CSV format. Data input using *EgadsCsv* is separated into a list of arrays, which each column a separate array in the list.

3.4.1 Opening

To open a text file the *EgadsCsv* class, use the `open(pathname, permissions, delimiter, quotechar)` method:

```
>>> import egads
>>> f = egads.input.EgadsCsv()
>>> f.open('/pathname/filename.txt','r',',','"')
```

Valid values for permissions are:

- `r` – Read: opens file for reading only. Default value if nothing is provided.
- `w` – Write: opens file for writing, and overwrites data in file.
- `a` – Append: opens file for appending data.
- `r+` – Read and write: opens file for both reading and writing.

The `delimiter` argument is a one-character string specifying the character used to separate fields in the CSV file to be read; the default value is `,`. The `quotechar` argument is a one-character string specifying the character used to quote fields containing special characters in the CSV file to to be read; the default value is `"`.

3.4.2 File Manipulation

The following methods are available to control the current position in the file and display more information about the file.

- `f.display_file()` – Prints contents of file out to standard output.
- `f.get_position()` – Returns current position in file as integer.
- `f.seek(location, from_where)` – Seeks to specified location in file. `location` is an integer specifying how far to seek. Valid options for `from_where` are `b` to seek from beginning of file, `c` to seek from current position in file and `e` to seek from the end of the file.
- `f.reset()` – Resets position to beginning of file.

3.4.3 Reading Data

Reading data is done using the `read(lines, format)` method on a file that has been opened with `r` or `r+` permissions:

```
>>> import egads
>>> f = egads.input.EgadsCsv()
>>> f.open('mycsvfile.csv','r')
>>> single_line_as_list = f.read(1)
>>> all_lines_as_list = f.read()
```

`read(lines, format)` returns a list of the items read in from the CSV file. The arguments `lines` and `format` are optional. If `lines` is provided, `read(lines, format)` will read in the specified number of lines, otherwise it will read the whole file. `format` is an optional list of characters used to decompose the elements read in from the CSV files to their proper types. Options are:

- `i` – int

- `f` – float
- `l` – long
- `s` – string

Thus to read in the line:

`FGBTM,20050105T143523,1.5,21,25`

the command to input with proper formatting would look like this:

```
>>> data = f.read(1, ['s','s','f','f'])
```

3.4.4 Writing Data

To write data to a file, use the `write(data)` method on a file that has been opened with `w`, `a` or `r+` permissions:

```
>>> import egads
>>> f = egads.input.EgadsCsv()
>>> f.open('mycsvfile.csv','a')
>>> titles = ['Aircraft ID','Timestamp','Value1','Value2','Value3']
>>> f.write(titles)
```

where the `titles` parameter is a list of strings. This list will be output to the CSV, with each strings separated by the delimiter specified when the file was opened (default is `,`).

To write multiple lines out to a file, `writerows(data)` is used:

```
>>> data = [['FGBTM','20050105T143523',1.5,21,25],['FGBTM','20050105T143524',1.6,
↪20,25.6]]
>>> f.writerows(data)
```

3.4.5 Closing

To close a file, simply call the `close()` method:

```
>>> f.close()
```

3.4.6 Tutorial

Here is a basic CSV file, created by EGADS:

```
time,sea level,corrected sea level
1.0,5.0,1.0
2.0,2.0,3.0
3.0,-2.0,-1.0
4.0,0.5,2.5
5.0,4.0,6.0
```

This file has been created with the following commands:

- import EGADS module:

  ```
  >>> import egads
  ```

- create two main variables, following the official EGADS convention:

  ```
  >>> data1 = egads.EgadsData(value=[5.0,2.0,-2.0,0.5,4.0], units='mm', name=
  ↪'sea level', scale_factor=1., add_offset=0., _FillValue=-9999)
  >>> data2 = egads.EgadsData(value=[1.0,3.0,-1.0,2.5,6.0], units='mm', name=
  ↪'corr sea level', scale_factor=1., add_offset=0., _FillValue=-9999)
  ```

- create an independant variable, still by following the official EGADS convention:

```
>>> time = egads.EgadsData(value=[1.0,2.0,3.0,4.0,5.0], units='seconds since␣
↪19700101T00:00:00', name='time')
```

- create a new EgadsFile instance:

```
>>> f = egads.input.EgadsCsv()
```

- use the following function to open a new file:

```
>>> f.open('main_csv_file.csv','w',',','"')
```

- prepare the headers if necessary:

```
>>> headers = ['time', 'sea level', 'corrected sea level']
```

- prepare an object to receive all data:

```
>>> data = [time.value, data1.value, data2.value]
```

- write the headers and data into the file

```
>>> f.write(headers)
>>> f.write(data)
```

- and do not forget to close the file:

```
>>> f.close()
```

3.5 Working with NetCDF files

EGADS provides two classes to work with NetCDF files. The simplest, `egads.input.netcdf.NetCdf`, allows simple read/write operations to NetCDF files. The other, `egads.input.netcdf.EgadsNetCdf`, is designed to interface with NetCDF files conforming to the EUFAR Standards & Protocols data and metadata regulations. This class directly reads or writes NetCDF data using instances of the *EgadsData* class.

3.5.1 Opening

To open a NetCDF file, simply create a *NetCdf()* instance and then use the `open(pathname, permissions)` command:

```
>>> import egads
>>> f = egads.input.NetCdf()
>>> f.open('/pathname/filename.nc','r')
```

Valid values for permissions are:

- `r` – Read: opens file for reading only. Default value if nothing is provided.
- `w` – Write: opens file for writing, and overwrites data in file.
- `a` – Append: opens file for appending data.
- `r+` – Same as `a`.

3.5.2 Getting info

- `f.get_dimension_list()` – returns a list of all dimensions and their sizes
- `f.get_dimension_list(var_name)` – `var_name` is optional and if provided, the function returns a list of all dimensions and their sizes attached to `var_name`
- `f.get_attribute_list()` – returns a list of all top-level attributes
- `f.get_attribute_list(var_name)` – `var_name` is optional and if provided, the function returns a list of all attributes attached to `var_name`
- `f.get_variable_list()` – returns list of all variables
- `f.get_filename()` – returns filename for currently opened file
- `f.get_perms()` – returns the current permissions on the file that is open

3.5.3 Reading data

To read data from a file, use the `read_variable()` function:

```
>>> data = f.read_variable(var_name, input_range)
```

where `var_name` is the name of the variable to read in, and `input_range` (optional) is a list of min/max values.

If using the `egads.input.NetCdf()` class, an array of values contained in `var_name` will be returned. If using the `egads.input.EgadsNetCdf()` class, an instance of the *EgadsData* class will be returned containing the values and attributes of `var_name`.

3.5.4 Writing data

The following describe how to add dimensions or attributes to a file.

- `f.add_dim(dim_name, dim_size)` – add dimension to file
- `f.add_attribute(attr_name, attr_value)` – add attribute to file
- `f.add_attribute(attr_name, attr_value, var_name)` – `var_name` is optional and if provided, the function add attribute to `var_name`

Data can be output to variables using the `write_variable()` function as follows:

```
>>> f.write_variable(data, var_name, dims, type)
```

where `var_name` is a string for the variable name to output, `dims` is a tuple of dimension names (not needed if the variable already exists), and `type` is the data type of the variable. The default value is *double*, other valid options are *float*, *int*, *short*, *char* and *byte*.

If using *NetCdf*, values for `data` passed into `write_variable` must be scalar or array. Otherwise, if using *EgadsNetCdf*, an instance of *EgadsData* must be passed into `write_variable`. In this case, any attributes that are contained within the *EgadsData* instance are applied to the NetCDF variable as well.

3.5.5 Conversion from NetCDF to NASA/Ames file format

The conversion is only possible on opened NetCDF files. If modifications have been made and haven't been saved, the conversion won't take into account those modifications. Actually, the only File Format Index supported by the conversion in the NASA/Ames format is 1001. Consequently, if variables depend on multiple independant variables (e.g. `data` is function of `time`, `longitude` and `latitude`), the file won't be converted and the function will raise an exception. On the contrary, if multiple independant variables (or dimensions) exist, and if each variable depend on only one independant variable (e.g. `data` is only function of `time`), the file will be converted and the function will generate one file per independant variable. If the user needs to convert a complex file with variables depending on multiple independant variables, the conversion should be done manually by creating a NASA/Ames instance and a NASA/Ames dictionary, by populating the dictionary and by saving the file.

To convert a NetCDF file, simply use:

- `f.convert_to_nasa_ames()` – convert the currently opened NetCDF file to NASA/Ames file format
- `f.convert_to_nasa_ames(na_file, requested_ffi, float_format, delimiter, annotation, no_header)` – `na_file`, `requested_ffi`, `float_format`, `delimiter`, `annotation` and `no_header` are optional parameters ; `na_file` is the name of the output file once it has been converted, by default the name of the NetCDF file will be used with the extension .na ; `requested_ffi` is not used actually, but will be functional in a next version of EGADS ; `float_format` is the formatting string used for formatting floats when writing to output file, by default `%g` ; `delimiter` is a character or a sequence of character for use between data items in the data file, by default ' ' (four spaces) ; if `annotation` is set to `True`, write the output file with an additional left-hand column describing the contents of each header line, by default `False` ; if `no_header` is set to `True`, then only the data blocks are written to file, by default `False`

To convert a NetCDF file to NASA/Ames CSV format, simply use:

- `f.convert_to_csv()` – convert the currently opened NetCDF file to NASA/Ames CSV format
- `f.convert_to_csv(csv_file, float_format, annotation, no_header)` – `csv_file`, `float_format`, `annotation` and `no_header` are optional parameters ; `csv_file` is the name of the output file once it has been converted, by default the name of the NetCDF file will be used with the extension .csv ; `float_format` is the formatting string used for formatting floats when writing to output file, by default `%g` ; if `annotation` is set to `True`, write the output file with an additional left-hand column describing the contents of each header line, by default `False` ; if `no_header` is set to `True`, then only the data blocks are written to file, by default `False`

3.5.6 Other operations

- `f.get_attribute_value(attr_name)` – returns the value of a global attribute
- `f.get_attribute_value(attr_name, var_name)` – `var_name` is optional and if provided, the function returns the value of an attribute attached to `var_name`
- `f.change_variable_name(var_name, new_name)` – change the variable name in currently opened NetCDF file

3.5.7 Closing

To close a file, simply use the `close()` method:

```
>>> f.close()
```

Note: The EGADS *NetCdf* and *EgadsNetCdf* use the official NetCDF I/O routines, therefore, as described in the NetCDF documentation, it is not possible to remove a variable or more, and to modify the values of a variable. As attributes, global and those linked to a variable, are more dynamic, it is possible to remove, rename, or replace them.

3.5.8 Tutorial

Here is a NetCDF file, created by EGADS, and viewed by the command `ncdump -h ...`:

```
=> ncdump -h main_netcdf_file.nc
    netcdf main_netcdf_file {
    dimensions:
        time = 5 ;
    variables:
        double time(time) ;
            time:units = "seconds since 19700101T00:00:00" ;
            time:long_name = "time" ;
        double sea_level(time) ;
            sea_level:_FillValue = -9999. ;
            sea_level:category = "TEST" ;
            sea_level:scale_factor = 1. ;
            sea_level:add_offset = 0. ;
            sea_level:long_name = "sea level" ;
            sea_level:units = "mm" ;
        double corrected_sea_level(time) ;
            corrected_sea_level:_FillValue = -9999. ;
            corrected_sea_level:units = "mm" ;
            corrected_sea_level:add_offset = 0. ;
            corrected_sea_level:scale_factor = 1. ;
            corrected_sea_level:long_name = "corr sea level" ;

    // global attributes:
            :Conventions = "CF-1.0" ;
            :history = "the netcdf file has been created by EGADS" ;
            :comments = "no comments on the netcdf file" ;
            :institution = "My institution" ;
    }
```

This file has been created with the following commands:

- import EGADS module:

```
>>> import egads
```

- create two main variables, following the official EGADS convention:

```
>>> data1 = egads.EgadsData(value=[5.0,2.0,-2.0,0.5,4.0], units='mm', name=
↪'sea level', scale_factor=1., add_offset=0., _FillValue=-9999)
>>> data2 = egads.EgadsData(value=[1.0,3.0,-1.0,2.5,6.0], units='mm', name=
↪'corr sea level', scale_factor=1., add_offset=0., _FillValue=-9999)
```

- create an independant variable, still by following the official EGADS convention:

```
>>> time = egads.EgadsData(value=[1.0,2.0,3.0,4.0,5.0], units='seconds since␣
↪19700101T00:00:00', name='time')
```

- create a new EgadsNetCdf instance with a file name:

```
>>> f = egads.input.EgadsNetCdf('main_netcdf_file.nc', 'w')
```

- add the global attributes to the NetCDF file:

```
>>> f.add_attribute('Conventions', 'CF-1.0')
>>> f.add_attribute('history', 'the netcdf file has been created by EGADS')
>>> f.add_attribute('comments', 'no comments on the netcdf file')
>>> f.add_attribute('institution', 'My institution')
```

- add the dimension(s) of your variable(s), here it is `time`:

```
>>> f.add_dim('time', len(time))
```

- write the variable(s), it is a good practice to write at the first place the independant variable `time`:

```
>>> f.write_variable(time, 'time', ('time',), 'double')
>>> f.write_variable(data1, 'sea_level', ('time',), 'double')
>>> f.write_variable(data2, 'corrected_sea_level', ('time',), 'double')
```

- and do not forget to close the file:

```
>>> f.close()
```

3.6 Working with NASA Ames files

To work with NASA Ames files, EGADS incorporates the NAPpy library developed by Ag Stephens of BADC. Information about NAPpy can be found at http://proj.badc.rl.ac.uk/cows/wiki/CowsSupport/Nappy

In EGADS, the NAPpy API has been adapted to match the other EGADS file access methods. Thus, from EGADS, NASA Ames files can be accessed via the *egads.input.nasa_ames_io.NasaAmes* class. Actually, only the FFI 1001 has been interfaced with EGADS.

3.6.1 Opening

To open a NASA Ames file, simply create a *NasaAmes()* instance and then use the `open(pathname, permissions)` command:

```
>>> import egads
>>> f = egads.input.NasaAmes()
>>> f.open('/pathname/filename.na','r')
```

Valid values for permissions are:

- `r` – Read: opens file for reading only. Default value if nothing is provided.
- `w` – Write: opens file for writing, and overwrites data in file.
- `a` – Append: opens file for appending data.
- `r+` – Same as `a`.

Once a file has been opened, a dictionary of NASA/Ames format elements is loaded into memory. That dictionary will be used to overwrite the file or to save to a new file.

3.6.2 Getting info

- `f.get_attribute_list()` – returns a list of all top-level attributes
- `f.get_attribute_list(var_name, var_type, na_dict)` – `var_name` is optional and if provided, the function returns list of all attributes attached to `var_name` ; if `var_type` is provided the function will search in the variable type `var_type` by default ; `na_dict` is optional if provided, will return a list of all top-level attributes, or all `var_name` attributes, in the NASA/Ames dictionary `na_dict`
- `f.get_attribute_value(attr_name)` – returns the value of a global attribute named `attr_name`
- `f.get_attribute_value(attr_name, var_name, var_type, na_dict)` – `var_name`, `var_type` and `na_dict` are optional ; if `var_name` is provided, returns the value of an attribute named `attr_name` attached to a variable named `var_name` ; if `var_type` is provided, the function will search in the variable type `var_type` by default ; if `na_dict` is provided, returns the attribute value from the NASA/Ames dictionary `na_dict`
- `f.get_dimension_list()` – returns a list of all variable dimensions
- `f.get_dimension_list(na_dict, var_type)` – `var_type` is optional, if provided, the function returns a list of all variable dimensions based on the `var_type` by default ; `na_dict` is optional and will returns the dimension list from the NASA/Ames dictionary `na_dict` ;
- `f.get_variable_list()` – returns list of all variables ;
- `f.get_variable_list(na_dict)` – `na_dict` is optional and if provided, will return a list of all variables in the NASA/Ames dictionary `na_dict`
- `f.get_filename()` – returns filename for currently opened file

3.6.3 Reading data

To read data from a file, use the `read_variable()` function:

```
>>> data = f.read_variable(var_name)
```

where `var_name` is the name of the variable to read in. The data will be read in to an instance of the *EgadsData* class, containing the values and attributes of `var_name`.

3.6.4 Writing data

To write data to the current file or to a new file, the user must save a dictionary of NASA/Ames elements. Few functions are available to help him to prepare the dictionary:

- `f.create_na_dict` – create a new dictionary populated with standard NASA/Ames keys.
- `f.write_attribute_value(attr_name, attr_value)` – write or replace a specific attribute (from the official NASA/Ames attribute list) in the currently opened dictionary
- `f.write_attribute_value(attr_name, attr_value, var_name, var_type, na_dict)` – `var_name` and `var_type` are optional, if provided, write or replace a specific attribute linked to the variable `var_name` (`var_type` is by default equal to 'main') in the currently opened dictionary ; ccepted attributes for a variable are 'name', 'units', '_FillValue' and 'scale_factor', other attributes will be refused and should be passed as 'special comments' ; `na_dict` is optional and if provided the function will write the attribute in the NASA/Ames dictionary `na_dict`
- `f.write_variable(data, var_name)` – write or replace a variable ; the function will search if `data` is already in the dictionary by comparing `varname` with other variable names in the dictionary, if it is found, `data` will replace the old variable, if not `data` is considered as a new variable ; `data` can be an *EgadsData* or a vector/matrix.
- `f.write_variable(data, var_name, var_type, attr_dict, na_dict)` – `var_type`, `attr_dict` and `na_dict` are optional ; `attr_dict` (a dictionary of standard NASA/ames variable attributes: 'name', 'units', '_FillValue' and 'scale_factor') must be provided if `data` is not an *EgadsData* (in that case, variable attributes are retrieve from the *EgadsData*.metadata dictionary) ; if `na_dict` is provided, the function saves the variable in the NASA/Ames dictionary `na_dict`

3.6.5 Saving a file

Once a dictionary is ready, use the `save_na_file()` function to save the file:

```
>>> data = f.save_na_file(file_name, na_dict, float_format):
```

where `file_name` is the name of the new file or the name of the current file, `na_dict` the name of the dictionary to be saved (optional, if not provided, the current dictionary will be used), and `float_format` the format of the floating numbers in the file (by deffault, two decimal places).

3.6.6 Conversion from NASA/Ames file format to NetCDF

When a NASA/Ames file is opened, all metadata and data are read and stored in memory in a dedicated dictionary. The conversion will convert that dictionary to generate a NetCDF file. If modifications are made to the dictionary, the conversion will take into account those modifications. Actually, the only File Format Index supported by the conversion in the NASA/Ames format is 1001. Consequently, if variables depend on multiple independant variables (e.g. `data` is function of `time`, `longitude` and `latitude`), the file won't be converted and the function will raise an exception. If the user needs to convert a complex file with variables depending on multiple independant variables, the conversion should be done manually by creating a NetCDF instance and by populating the NetCDF files with NASA/Ames data and metadata.

To convert a NASA/Ames file, simply use:

- `f.convert_to_netcdf()` – convert the currently opened NASA/Ames file to NetCDF format.
- `f.convert_to_netcdf(nc_file)` – `nc_file` is an optional parameter ; `na_file` is the name of the output file once it has been converted, by default the name of the NASA/Ames file will be used with the extension .nc

3.6.7 Other operations

- `f.read_na_dict()` – returns a deep copy of the current opened file dictionary
- `f.na_format_information()` – returns a text explaining the structure of a NASA/Ames file to help the user to modify or to create his own dictionary

3.6.8 Closing

To close a file, simply use the `close()` method:

```
>>> f.close()
```

3.6.9 Tutorial

Here is a NASA/Ames file:

```
23    1001
John Doe
An institution
tide gauge
ATESTPROJECT
1    1
2017 1 30    2017 1 30
0.0
time (seconds since 19700101T00:00:00)
2
1    1
-9999    -9999
sea level (mm)
corr sea level (mm)
3
========SPECIAL COMMENTS===========
this file has been created with egads
=========END=========
4
========NORMAL COMMENTS===========
headers:
time    sea level   corrected sea level
=========END=========
1.00    5.00    1.00
2.00    2.00    3.00
3.00    -2.00    -1.00
4.00    0.50    2.50
5.00    4.00    6.00
```

This file has been created with the following commands:

- **import EGADS module:**

```
>>> import egads
```

- create two main variables, following the official EGADS convention:

```
>>> data1 = egads.EgadsData(value=[5.0,2.0,-2.0,0.5,4.0], units='mm', name=
↪'sea level', scale_factor=1, _FillValue=-9999)
>>> data2 = egads.EgadsData(value=[1.0,3.0,-1.0,2.5,6.0], units='mm', name=
↪'corr sea level', scale_factor=1, _FillValue=-9999)
```

- create an independant variable, still by following the official EGADS convention:

```
>>> time = egads.EgadsData(value=[1.0,2.0,3.0,4.0,5.0], units='seconds since␣
↪19700101T00:00:00', name='time')
```

- create a new NASA/Ames empty instance:

```
>>> f = egads.input.NasaAmes()
```

- initialize a new NASA/Ames dictionary:

```
>>> na_dict = f.create_na_dict()
```

- prepare the normal and special comments if needed, in a list, one cell for each line:

```
>>> scom = ['========SPECIAL COMMENTS==========','this file has been created␣
↪with egads','=========END=========']
>>> ncom = ['========NORMAL COMMENTS==========','headers:','time    sea level␣
↪  corrected sea level','=========END=========']
```

- populate the main NASA/Ames attributes:

```
>>> f.write_attribute_value('ONAME', 'John Doe', na_dict = na_dict) # ONAME is␣
↪the name of the author(s)
>>> f.write_attribute_value('ORG', 'An institution', na_dict = na_dict) # ORG␣
↪is tne name of the organization responsible for the data
>>> f.write_attribute_value('SNAME', 'tide gauge', na_dict = na_dict) # SNAME␣
↪is the source of data (instrument, observation, platform, ...)
>>> f.write_attribute_value('MNAME', 'ATESTPROJECT', na_dict = na_dict) #␣
↪MNAME is the name of the mission, campaign, programme, project dedicated to␣
↪data
>>> f.write_attribute_value('DATE', [2017, 1, 30], na_dict = na_dict) # DATE␣
↪is the date at which the data recorded in this file begin (YYYY MM DD)
>>> f.write_attribute_value('NIV', 1, na_dict = na_dict) # NIV is the number␣
↪of independent variables
>>> f.write_attribute_value('NSCOML', 3, na_dict = na_dict) # NSCOML is the␣
↪number of special comments lines or the number of elements in the SCOM list
>>> f.write_attribute_value('NNCOML', 4, na_dict = na_dict) # NNCOML is the␣
↪number of special comments lines or the number of elements in the NCOM list
>>> f.write_attribute_value('SCOM', scom, na_dict = na_dict) # SCOM is the␣
↪special comments attribute
>>> f.write_attribute_value('NCOM', ncom, na_dict = na_dict) # NCOM is the␣
↪normal comments attribute
```

- write each variable in the dictionary:

```
>>> f.write_variable(time, 'time', vartype="independant", na_dict = na_dict)
>>> f.write_variable(data1, 'sea level', vartype="main", na_dict = na_dict)
>>> f.write_variable(data2, 'corrected sea level', vartype="main", na_dict =␣
↪na_dict)
```

- and finally, save the dictionary to a NASA/Ames file:

```
>>> f.save_na_file('na_example_file.na', na_dict)
```

3.7 Converting between file formats

Since the first version of EGADS, the direct conversion was possible with the NAPpy library with the help of CDMS. As CDMS is not compatible with windows, that possibility has been dropped. However, two functions have been introduced to allow a conversion from NetCDF to NASA/Ames format, and from NASA/Ames format to NetCDF. Please read the section about NetCDF and NASA/Ames file handling to learn how to convert between those formats.

3.8 Working with algorithms

Algorithms in EGADS are stored in the `egads.algorithms` module, and separated into sub-modules by category (microphysics, thermodynamics, radiation, etc). Each algorithm follows a standard naming scheme, using the algorithm's purpose and source:

`{CalculatedParameter}{Detail}{Source}`

For example, an algorithm which calculates static temperature, which was provided by CNRM would be named:

`TempStaticCnrm`

3.8.1 Getting algorithm information

There are several methods to get information about each algorithm contained in EGADS. The EGADS Algorithm Handbook is available for easy reference outside of Python. In the handbook, each algorithm is described in detail, including a brief algorithm summary, descriptions of algorithm inputs and outputs, the formula used in the algorithm, algorithm source and links to additional references. The handbook also specifies the exact name of the algorithm as defined in EGADS. The handbook can be found on the EGADS website.

Within Python, usage information on each algorithm can be found using the `help()` command:

```
>>> help(egads.algorithms.thermodynamics.VelocityTasCnrm)

>>> Help on class VelocityTasCnrm in module egads.algorithms.thermodynamics.
    velocity_tas_cnrm:

class VelocityTasCnrm(egads.core.egads_core.EgadsAlgorithm)
 |  FILE        velocity_tas_cnrm.py
 |
 |  VERSION     $Revision: 104 $
 |
 |  CATEGORY    Thermodynamics
 |
 |  PURPOSE     Calculate true airspeed
 |
 |  DESCRIPTION Calculates true airspeed based on static temperature,
 |              static pressure and dynamic pressure using St Venant's
 |              formula.
 |
 |  INPUT       T_s         vector  K or C      static temperature
 |              P_s         vector  hPa         static pressure
 |              dP          vector  hPa         dynamic pressure
 |              cpa         coeff.  J K-1 kg-1  specific heat of air (dry
 |                                              air is 1004 J K-1 kg-1)
 |              Racpa       coeff.  ()          R_a/c_pa
 |
 |  OUTPUT      V_p         vector  m s-1       true airspeed
 |
 |  SOURCE      CNRM/GMEI/TRAMM
 |
 |  REFERENCES  "Mecanique des fluides", by S. Candel, Dunod.
 |
 |               Bulletin NCAR/RAF Nr 23, Feb 87, by D. Lenschow and
 |               P. Spyers-Duran
 |
...
```

3.8.2 Calling algorithms

Algorithms in EGADS generally accept and return arguments of *EgadsData* type, unless otherwise noted. This has the advantages of constant typing between algorithms, and allows metadata to be passed along the whole processing chain. Units on parameters being passed in are also checked for consistency, reducing errors in calculations, and rescaled if needed. However, algorithms will accept any normal data type, as well. They can also return non-*EgadsData* instances, if desired.

To call an algorithm, simply pass in the required arguments, in the order they are described in the algorithm help function. An algorithm call, using the `VelocityTasCnrm` in the previous section as an example, would therefore be the following:

```
>>> V_p = egads.algorithms.thermodynamics.VelocityTasCnrm().run(T_s, P_s, dP,
    cpa, Racpa)
```

where the arguments `T_s`, `P_s`, `dP`, etc are all assumed to be previously defined in the program scope. In this instance, the algorithm returns an *EgadsData* instance to `V_p`. To run the algorithm, but return a standard data type (scalar or array of doubles), set the `return_Egads` flag to `false`.

```
>>> V_p = egads.algorithms.thermodynamics.VelocityTasCnrm(return_Egads=false).
    run(T_s, P_s, dP, cpa, Racpa)
```

Note: When injecting a variable in an EgadsAlgorithm, the format of the variable should follow closely the documentation of the algorithm. If the variable is a scalar, and the algorithm needs a vector, the scalar should be surrounded by brackets: 52.123 -> [52.123].

3.9 Scripting

The recommended method for using EGADS is to create script files, which are extremely useful for common or repetitive tasks. This can be done using a text editor of your choice. The example script belows shows the calculation of density for all NetCDF files in a directory.

```
#!/usr/bin/env python

# import egads package
import egads
# import thermodynamic module and rename to simplify usage
import egads.algorithms.thermodynamics as thermo

# get list of all NetCDF files in 'data' directory
filenames = egads.input.get_file_list('data/*.nc')
f = egads.input.EgadsNetCdf()    # create EgadsNetCdf instance

for name in filenames:           # loop through files

    f.open(name, 'a')             # open NetCdf file with append permissions
    T_s = f.read_variable('T_t') # read in static temperature
    P_s = f.read_variable('P_s') # read in static pressure from file
    rho = thermo.DensityDryAirCnrm().run(P_s, T_s)   # calculate density
    f.write_variable(rho, 'rho', ('Time',))      # output variable

    f.close()                                    # close file
```

3.9.1 Scripting Hints

When scripting in Python, there are several important differences from other programming languages to keep in mind. This section outlines a few of these differences.

Importance of white space

Python differs from C++ and Fortran in how loops or nested statements are signified. Whereas C++ uses brackets ('{' and '}') and FORTRAN uses `end` statements to signify the end of a nesting, Python uses white space. Thus, for statements to nest properly, they must be set at the proper depth. As long as the document is consistent, the number of spaces used doesn't matter, however, most conventions call for 4 spaces to be used per level. See below for examples:

FORTRAN:

```
X = 0
DO I = 1,10
  X = X + I
  PRINT I
END DO
PRINT X
```

Python:

```
x = 0
for i in range(1,10):
    x = x + i
    print i
print x
```

3.10 Using the GUI

Since September 2016, a Graphical User Interface is available at https://github.com/eufarn7sp/egads-gui. It gives the user the possibility to explore data, apply/create algorithms, display and plot data. Still in beta state, the user will have the possibility in the future to work on a batch of file. For now, EGADS GUI comes as a simple python script and need to be launch from the terminal, if EGADS is installed, and once in the EGADS GUI directory:

```
>>> python egads_gui.py
```

It will be available soon as a stand alone (imbedding a version of EGADS CORE or using an already installed EGADS package).

CHAPTER

FOUR

ALGORITHM DEVELOPMENT

4.1 Introduction

The EGADS framework is designed to facilitate integration of third-party algorithms. This is accomplished through creation of Python modules containing the algorithm code, and corresponding LaTeX files which contain the algorithm methodology documentation. This section will explain the elements necessary to create these files, and how to incorporate them into the broader package.

4.2 Python module creation

To guide creation of Python modules containing algorithms in EGADS, an algorithm template has been included in the distribution. It can be found in ./egads/algorithms/file_templates/algorithm_template.py and is shown below:

```
__author__ = "mfreer, ohenry"
__date__ = "2016-12-14 15:04"
__version__ = "1.0"
__all__ = ['']

import egads.core.egads_core as egads_core
import egads.core.metadata as egads_metadata

# 1. Change class name to algorithm name (same as filename) but
#    following MixedCase conventions.

class AlgorithmTemplate(egads_core.EgadsAlgorithm):

# 2. Edit docstring to reflect algorithm description and input/output
#    parameters used

    """
    This file provides a template for creation of EGADS algorithms.

    FILE        algorithm_template.py

    VERSION     1.0

    CATEGORY    None

    PURPOSE     Template for EGADS algorithm files

    DESCRIPTION ...

    INPUT       inputs      var_type    units   description

    OUTPUT      outputs     var_type    units   description
```

```
    SOURCE      sources

    REFERENCES  references

    """

    def __init__(self, return_Egads=True):
        egads_core.EgadsAlgorithm.__init__(self, return_Egads)

        # 3. Complete output_metadata with metadata of the parameter(s) to be
        #    produced by this algorithm. In the case of multiple parameters,
        #    use the  following formula:
        #        self.output_metadata = []
        #        self.output_metadata.append(egads_metadata.VariableMetadata(...)
        #        self.output_metadata.append(egads_metadata.VariableMetadata(...)
        #        ...

        self.output_metadata = egads_metadata.VariableMetadata({
            'units':'%',
            'long_name':'template',
            'standard_name':'',
            'Category':['']
            })

        # 3 cont. Complete metadata with parameters specific to algorithm,
        #    including a list of inputs, a corresponding list of units, and
        #    the list of outputs. InputTypes are linked to the different
        #    var_type written in the docstring

        self.metadata = egads_metadata.AlgorithmMetadata({
            'Inputs':['input'],
            'InputUnits':['unit'],
            'InputTypes':['vector'],
            'InputDescription':['A description for an input'],
            'Outputs':['template'],
            'OutputDescription':['A description for an output'],
            'Purpose':'Template for EGADS algorithm files',
            'Description':'...',
            'Category':'None',
            'Source':'sources',
            'Reference':'references',
            'Processor':self.name,
            'ProcessorDate':__date__,
            'ProcessorVersion':__version__,
            'DateProcessed':self.now()
            }, self.output_metadata)

    # 4. Replace the 'inputs' parameter in the three instances below with the
    #    list of input parameters to be used in the algorithm.

    def run(self, inputs):

        return egads_core.EgadsAlgorithm.run(self, inputs)

    # 5. Implement algorithm in this section.

    def _algorithm(self, inputs):

        ## Do processing here:

        return result
```

The best practice before starting an algorithm is to copy this file and name it following the EGADS algo-

rithm file naming conventions, which is all lowercase with words separated by underscores. As an example, the file name for an algorithm calculating the wet bulb temperature contributed by DLR would be called `temperature_wet_bulb_dlr.py`.

Within the file itself, there are one rule to respect and several elements in this template that will need to be modified before this can be usable as an EGADS algorithm.:

1. **Format** An algorithm file is composed of different elements: metadata, class name, algorithm docstring, ... It is critical to respect the format of each element of an algorithm file, in particular the first metadata and the docstring, in term of beginning white spaces, line length, ... Even if it is not mandatory for EGADS itself, it will facilitate the integration of those algorithms in the new Graphical User Interface.
2. **Class name** The class name is currently 'AlgorithmTemplate', but this must be changed to the actual name of the algorithm. The conventions here are the same name as the filename (see above), but using MixedCase. So, following the example above, the class name would be TemperatureWetBulbDlr
3. **Algorithm docstring** The docstring is everything following the three quote marks just after the class definition. This section describes several essential aspects of the algorithm for easy reference directly from Python. This part is critical for the understanding of the algorithm by different users.
4. **Algorithm and output metadata** In the `__init__` method of the module, two important parameters are defined. The first is the 'output_metadata', which defines the metadata elements that will be assigned to the variable output by the algorithm. A few recommended elements are included, but a broader list of variable metadata parameters can be found in the NetCDF standards document on the EUFAR website (http://www.eufar.net/documents/6140, Annexe III). In the case that there are multiple parameters output by the algorithm, the output_metadata parameter can be defined as a list VariableMetadata instances.

   Next, the 'metadata' parameter defines metadata concerning the algorithm itself. These information include the names, types, descriptions and units of inputs; names and descriptions of outputs; name, description, purpose, category, source, reference, date and version of the algorithm; date processed; and a reference to the output parameters. Of these parameters, only the names, types, descriptions and units of the inputs, names and descriptions of the outputs and category, source, reference, description and purpose of the algorithm need to be altered. The other parameters (name, date and version of the processor, date processed) are populated automatically.

   self.output_metadata:

   - units: units of the output.
   - long_name: the name describing the output.
   - standard_name: a short name for the output.
   - Category: Name(s) of probe category - comma separated list (cf. EUFAR document http://www.eufar.net/documents/6140 for an example of possible categories).

   self.metadata:

   - Inputs: representation of each input in the documentation and in the code (ex: P_a for altitude pressure).
   - InputUnits: a list of all input units, one unit per input, '' for dimensionless input and 'None' for the input accepting every kind of units.
   - InputTypes: the type of the input (array, vector, coeff, ...) linked to the `var_type` string in the algorithm template ; the string `_optional` can be added to inform that the input is optional (used in the EGADS GUI).
   - InputDescription: short description of each input.
   - Outputs: representation of each output (ex: P_a for altitude pressure).
   - OutputDescription: short description of each output.
   - Purpose: the goal of the algorithm
   - Description: a description of the algorithm

- Category: the category of the algorithm (ex: Transforms, Thermodynamis, ...)
- Source : the source of the algorithm (ex: CNRM)
- Reference : the reference of the algorithm (ex: Doe et al, My wonderful algorithm, Journal of Algorithms, 11, pp 21-22, 2017).
- Processor: self.name
- ProcessorDate: `__date__`
- ProcessorVersion: `__version__`
- DateProcessed: self.now()

Note: For algorithms in which the output units depend on the input units (i.e. a purely mathematical transform, derivative, etc), there is a specific methodology to tell EGADS how to set the output units. To do this, set the appropriate `units` parameter of output_metadata to `inputn` where *n* is the number of the input parameter from which to get units (starting at 0). For algorithms in which the units of the input has no importance, the input units should set to `None`. For algorithms in which the input units are dimensionless (a factor, a quantity, a coefficient), the units on the input parameter should be set to `''`.

Note: EGADS accepts different kind of input type: coeff. for coefficient, vector, array, string, ... When writing the docstring of an algorithm and the metadata `InputTypes`, the user should write the type carefully as it is interpreted by EGADS. If a type depends on another variable or multiple variables, for example the time, or geographic coordinates, the variable name should be written between brackets (ex: array[lon,lat]). If a variable is optional, the user should add `, optional` to the type in the doctstring, and `_optional` to the type in the metadata `InputTypes`.

---

5. **Definition of parameters** In both the run and _algorithm methods, the local names intended for inputs need to be included. There are three locations where the same list must be added (marked in bold):
   - def run(self, **inputs**)
   - return egads_core.EgadsAlgorithm.run(self, **inputs**)
   - def _algorithm(self, **inputs**)
6. **Implementation of algorithm** The algorithm itself gets written in the _algorithm method and uses variables passed in by the user. The variables which arrive here are simply scalar or arrays, and if the source is an instance of EgadsData, the variables will be converted to the units you specified in the InputUnits of the algorithm metadata.
7. **Integration of the algorithm in EGADS** Once the algorithm file is ready, the user has to move it in the appropriate directory in the `./egads/algorithms/user` directory. Once it has been done, the `__init__.py` file has to be modified to declare the new algorithm. The following line can be added to the `__init__.py` file: `from the_name_of_the_file import \*`.

   If the algorithm requires a new directory, the user has to create it, move the file inside and create a `__init__.py` file to declare the new directory and the algoritm to EGADS. A template can be found in `./egads/algorithms/file_templates/init_template.py` and is shown below:

```
"""
EGADS new algorithms. See EGADS Algorithm Documentation for more info.
"""

__author__ = "ohenry"
__date__ = "$Date:: 2017-01-27 10:52#$"
__version__ = "$Revision:: 1      $"

import logging
try:
```

```
    from the_name_of_my_new_algorithm_file import *
    logging.info('egads [corrections] algorithms have been loaded')
except Exception:
    logging.error('an error occured during the loading of a [corrections]␣
↪algorithm')
```

4.3 Documentation creation

Within the EGADS structure, each algorithm has accompanying documentation in the EGADS Algorithm Handbook. These descriptions are contained in LaTeX files, organized in a structure similar to the toolbox itself, with one algorithm per file. These files can be found in the Documentation/EGADS Algorithm Handbook directory in the EGADS package downloaded from GitHub repository: https://github.com/eufarn7sp/egads.

A template is provided to guide creation of the documentation files. This can be found at Documentation/EGADS Algorithm Handbook/algorithms/algorithm_template.tex. The template is divided into 8 sections, enclosed in curly braces. These sections are explained below:

- **Algorithm name** Simply the name of the Python file where the algorithm can be found.
- **Algorithm summary** This is a short description of what the algorithm is designed to calculate, and should contain any usage caveats, constraints or limitations.
- **Category** The name of the algorithm category (e.g. Thermodynamics, Microphysics, Radiation, Turbulence, etc).
- **Inputs** At the minimum, this section should contain a table containing the symbol, data type (vector or coefficient), full name and units of the input parameters. An example of the expected table layout is given in the template.
- **Outputs** This section describes the parameters output from the algorithm, using the same fields as the input table (symbol, data type, full name and units). An example of the expected table layout is given in the template.
- **Formula** The mathematical formula for the algorithm is given in this section, if possible, along with a description of the techniques employed by the algorithm.
- **Author** Any information about the algorithm author (e.g. name, institution, etc) should be given here.
- **References** The references section should contain citations to publications which describe the algorithm.

In addition to these sections, the `index` and `algdesc` fields at the top of the file need to be filled in. The value of the `index` field should be the same as the algorithm name. The `algdesc` field should be the full English name of the algorithm.

Note: Any "_" character in plain text in LaTeX needs to be offset by a "\". Thus if the algorithm name is `temp_static_cnrm`, in LaTex, it should be input as `temp\_static\_cnrm`.

4.3.1 Example

An example algorithm is shown below with all fields completed.

```
%% $Date: 2012-02-17 18:01:08 +0100 (Fri, 17 Feb 2012) $
%% $Revision: 129 $
\index{temp\_static\_cnrm}
\algdesc{Static Temperature}
{ %%%%%% Algorithm name %%%%%%
temp\_static\_cnrm
}
{ %%%%%% Algorithm summary %%%%%%
```

```
Calculates static temperature of the air from total temperature.
This method applies to probe types such as the Rosemount.
}
{ %%%%%% Category %%%%%%
Thermodynamics
}
{ %%%%%% Inputs %%%%%%
$T_t$ &        Vector & Measured total temperature [K] \\
${\Delta}P$ & Vector & Dynamic pressure [hPa] \\
$P_s$ & Vector & Static pressure [hPa] \\
$r_f$ & Coeff. & Probe recovery coefficient \\
$R_a/c_{pa}$ & Coeff. & Gas constant of air divided by specific heat of air
at constant pressure
}
{ %%%%%% Outputs %%%%%%
$T_s$ & Vector & Static temperature [K]
}
{ %%%%%% Formula %%%%%%
\begin{displaymath}
 T_s = \frac{T_t}{1+r_f \left(\left(1+\frac{\Delta P}{P_s}\right)^{R_a/c_{pa}}
 -1\right)} \nonumber
\end{displaymath}
}
{ %%%%%% Author %%%%%%
CNRM/GMEI/TRAMM
}
{ %%%%%% References %%%%%%
}
```

CHAPTER

FIVE

EGADS API

5.1 Core Classes

class egads.core.egads_core.**EgadsData**(*value*, *units=''*, *variable_metadata=None*, *dtype='float64'*, ***attrs*)

Bases: quantities.quantity.Quantity

This class is designed using the EUFAR Standards & Protocols data and metadata recommendations. Its purpose is to store related data and metadata and allow them to be passed between functions and algorithms in a consistent manner.

Constructor Variables

Parameters

- **value** – Scalar or array of values to initialize EgadsData object.
- **units** (*string*) – Optional - String representation of units to be used for current EgadsData instance, e.g. 'm/s', 'kg', 'g/cm^3', etc.
- **variable_metadata** (VariableMetadata) – Optional - VariableMetadata dictionary object containing relevant metadata for the current EgadsData instance.
- ****attrs** – Optional - Keyword/value pairs of additional metadata which will be added into the existing variable_metadata object.

copy()
Generate and return a copy of the current EgadsData instance.

get_units()
Return units used in current EgadsData instance.

print_description()
Generate and return a description of current EgadsData instance.

print_shape()
Prints shape of current EgadsData instance

rescale(*units*)
Return a copy of the variable rescaled to the provided units.

Parameters **units** (*string*) – String representation of desired units.

class egads.core.egads_core.**EgadsAlgorithm**(*return_Egads=True*)

Bases: object

EGADS algorithm base class. All egads algorithms should inherit this class.

The EgadsAlgorithm class provides base methods for algorithms in EGADS and initializes algorithm attributes.

Initializes EgadsAlgorithm instance with None values for all standard attributes.

Constructor Variables

Parameters **return_Egads** (*bool*) – Optional - Flag used to configure which object type will be returned by the current EgadsAlgorithm. If `true` an :class: EgadsData instance with relevant metadata will be returned by the algorithm, otherwise an array or scalar will be returned.

get_info()
Print docstring of algorithm to standard output.

now()
Calculate and return current date/time in ISO 8601 format.

processor()
Indicate the algorithm used to produce the output variable

run(**args*)
Basic run method. This method should be called from EgadsAlgorithm children, passing along the correct inputs to the _call_algorithm method.

Parameters ***args** – Parameters to pass into algorithm in the order specified in algorithm metadata.

time_stamp()
Calculate and set date processed for all output variables.

5.2 Metadata Classes

class egads.core.metadata.**Metadata**(*metadata_dict={}*, *conventions=None*, *metadata_list=None*)
Bases: `dict`

This is a generic class designed to provide basic metadata storage and handling capabilities.

Initialize Metadata instance with given metadata in dict form.

Parameters **metadata_dict** (*dict*) – Dictionary object containing metadata names and values.

add_items(*metadata_dict*)
Method to add metadata items to current Metadata instance.

Parameters **metadata_dict** – Dictionary object containing metadata names and values.

compliance_check(*conventions=None*)
Checks for compliance with metadata conventions. If no specific conventions are provided, then compliance check will be based on metadata conventions listed in Conventions metadata field.

Parameters **conventions** (*string|list*) – Optional - Comma separated string or list of coventions to use for conventions check. Current conventions recognized are `CF`, `RAF`, `IWGADTS`, `EUFAR`, `NASA Ames`

set_conventions(*conventions*)
Sets conventions to be used in current Metadata instance

Parameters **conventions** (*list*) – List of conventions used in current metadata instance.

class egads.core.metadata.**FileMetadata**(*metadata_dict*, *filename*, *conventions_keyword='Conventions'*, *conventions=[]*)
Bases: *egads.core.metadata.Metadata*

This class is designed to provide basic storage and handling capabilities for file metadata.

Initialize Metadata instance with given metadata in dict form. Tries to determine which conventions are used by the metadata. The user can optionally supply which conventions the metadata uses.

Parameters

- **metadata_dict** (*dict*) – Dictionary object containing metadata names and values.
- **filename** (*string*) – Filename for origin of file metadata.
- **conventions_keyword** (*string*) – Optional - Keyword contained in metadata dictionary used to detect which metadata conventions are used.
- **conventions** (*list*) – Optional - List of metadata conventions used in provided metadata dictionary.

set_filename (*filename*)
Sets file object used for current FileMetadata instance.

Parameters **filename** (*string*) – Filename of provided metadata.

class egads.core.metadata.**VariableMetadata** (*metadata_dict*, *parent_metadata_obj=None*, *conventions=None*)
Bases: egads.core.metadata.Metadata

This class is designed to provide storage and handling capabilities for variable metadata.

Initialize VariableMetadata instance with given metadata in dict form. If VariableMetadata comes from a file, the file metadata object can be provided to auto-detect conventions. Otherwise, the user can specify which conventions are used in the variable metadata.

Parameters

- **metadata_dict** (*dict*) – Dictionary object contaning variable metadata names and values
- **parent_metadata_obj** (Metadata) – Metadata, optional Metadata object for the parent object of current variable (file, algorithm, etc). This field is optional.
- **conventions** (*list*) – Optional - List of metadata conventions used in provided metadata dictionary.

set_parent (*parent_metadata_obj*)
Sets parent object of VariableMetadata instance.

Parameters **parent_metadata_obj** (Metadata) – Optional - Metadata object for the parent object of the current variable (file, algorithm, etc)

class egads.core.metadata.**AlgorithmMetadata** (*metadata_dict*, *child_variable_metadata=None*)
Bases: egads.core.metadata.Metadata

This class is designed to provide storage and handling capabilities for EGADS algorithm metadata. Stores instances of VariableMetadata objects to use to populate algorithm variable outputs.

Initialize AlgorithmMetadata instance with given metadata in dict form and any child variable metadata.

Parameters

- **metadata_dict** (*dict*) – Dictionary object containing variable metadata names and values
- **child_varable_metadata** (*list*) – Optional - List containing VariableMetadata

assign_children (*child*)
Assigns children to current AlgorithmMetadata instance. Children are typically VariableMetadata instances. If VariableMetadata instance is used, this method also assigns current AlgorithmMetadata instance as parent in VariableMetadata child.

Parameters **child** (VariableMetadata) – Child metadata object to add to current instance children.

5.3 File Classes

class egads.input.input_core.**FileCore**(*filename=None*, *perms='r'*, ***kwargs*)
Bases: object

Abstract class which holds basic file access methods and attributes. Designed to be subclassed by NetCDF, NASA Ames and basic text file classes.

Constructor Variables

Parameters

- **filename** (*string*) – Optional - Name of file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are w for write (overwrites data in file), a and r+ for append, and r for read. r is the default value

Initializes file instance.

Parameters

- **filename** (*string*) – Optional - Name of file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are w for write (overwrites data in file), a and r+ for append, and r for read. r is the default value

close()
Close opened file.

get_filename()
If file is open, returns the filename.

get_perms()
Returns the current permissions on the file that is open. Returns None if no file is currently open. Options are w for write (overwrites data in file),''a'' and r+ for append, and r for read.

open(*filename*, *perms=None*)
Opens file given filename.

Parameters

- **filename** (*string*) – Name of file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are w for write (overwrites data in file), a and r+ for append, and r for read. r is the default value

egads.input.input_core.**get_file_list**(*path*)
Given path, returns a list of all files in that path. Wildcards are supported.

Example:

```
file_list = get_file_list('data/*.nc')
```

class egads.input.nasa_ames_io.**NasaAmes**(*filename=None*, *perms='r'*)
Bases: *egads.input.input_core.FileCore*

EGADS module for interfacing with NASA Ames files. This module adapts the NAPpy library to the file access methods used in EGADS. To keep compatibility with Windows, all functions calling CDMS or CDMS2 have been revoked. The user still can use egads.thirdparty.nappy functions to have access to CDMS possibilities under Linux and Unix.

Initializes NASA Ames instance.

Parameters

- **filename** (*string*) – Optional - Name of NetCDF file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are w for write (overwrites data), a and r+ for append, and r for read. r is the default value.

convert_to_netcdf(*nc_file=None*)
Convert a NASA/Ames dictionary to a NetCDF file.

Parameters **nc_file** (*string*) – Optional - String name of the netcdf file to be written. If no filename is passed, the function will used the name of the actually opened NASA/Ames file.

create_na_dict()
Create a typical NASA/Ames dictionary. It is intended to be saved in a new file. The user will have to populate the dictionary with other functions.

get_attribute_list(*varname=None*, *vartype='main'*, *na_dict=None*)
Returns a list of attributes found in current NASA Ames file either globally or attached to a given variable, depending on the type

Parameters

- **varname** (*string|int*) – Optional - Name or number of variable to get list of attributes from. If no variable name is provided, the function returns global attributes.
- **vartype** (*string|*) – Optional - type of variable to get list of attributes from. If no variable type is provided with the variable name, the function returns an attribute of the main variable .
- **na_dict** (*dict*) – Optional - The NASA/Ames dictionary in which to get the attribute list. By default, na_dict = None and the attribute list is retrieved from the currently opened NASA/Ames file . Only mandatory if creating a new file or creating a new dictionary.

get_attribute_value(*attrname*, *varname=None*, *vartype='main'*, *na_dict=None*)
Returns the value of an attribute found in current NASA Ames file either globally or attached to a given variable (only name, units, _FillValue and scale_factor), depending on the type

Parameters

- **attrname** (*string*) – String name of attribute to write in currently open file.
- **varname** (*string|int*) – Optional - Name or number of variable to get list of attributes from. If no variable name is provided, the function returns global attributes.
- **vartype** (*string*) – Optional - type of variable to get list of attributes from. If no variable type is provided with the variable name, the function returns an attribute of the main variable.
- **na_dict** (*dict*) – Optional - The NASA/Ames dictionary in which to get the attribute value. By default, na_dict = None and the attribute value is retrieved from the currently opened NASA/Ames file . Only mandatory if creating a new file or creating a new dictionary.

get_dimension_list(*vartype='main'*, *na_dict=None*)
Returns a dictionary of all dimensions linked to their variables in NASA Ames dictionary.

Parameters

- **vartype** (*string*) – Optional - the type of data to read Options are `independant` for independant variables, `main` for main variables and `auxiliary` for auxiliary variables.
- **na_dict** (*dict*) – Optional - The NASA/Ames dictionary in which to get the dimension list. By default, na_dict = None and the dimension list is retrieved from the currently opened NASA/Ames file . Only mandatory if creating a new file or creating a new dictionary.

get_variable_list(*na_dict=None*, *vartype='main'*)
Returns list of all variables in NASA Ames file.

Parameters

- **na_dict** (*dict*) – Optional - The NASA/Ames dictionary in which to get the variable list. By default, na_dict = None and the variable list is retrieved from the currently opened NASA/Ames file . Only mandatory if creating a new file or creating a new dictionary.
- **vartype** (*string*) – Optional - the type of data to read Options are `independant` for independant variables, `main` for main variables and `auxiliary` for auxiliary variables.

read_na_dict()
Read the dictionary from currently open NASA Ames file. Method accessible by the user to read the dictionary in a custom object.

read_variable(*varname*)
Read in variable from currently open NASA Ames file to :class: EgadsData object. Any additional variable metadata is additionally read in.

Parameters **varname** (*string|int*) – String name or sequential number of variable to read in from currently open file.

save_na_file(*filename=None*, *na_dict=None*, *float_format='%.g'*, *delimiter=None*, *annotation=False*, *no_header=False*)
Save a NASA/Ames dictionary to a file.

Parameters

- **filename** (*string*) – String name of the file to be written.
- **na_dict** (*dict*) – Optional - The NASA/Ames dictionary to be saved. If no dictionary is entered, the dictionary currently opened during the open file process will be saved.
- **float_format** (*string*) – Optional - The format of numbers to be saved. If no string is entered, values are round up to two decimal places.
- **delimiter** (*string*) – Optional - A character or multiple characters to separate data. By default ' ' (four spaces) is used
- **annotation** (*boolean*) – Optional - If annotation is True then add annotation column to left of file. Default - False.
- **no_header** (*boolean*) – Optional - If no_header is True then suppress writing the header and only write the data section. Default - False.

write_attribute_value(*attrname*, *attrvalue*, *na_dict=None*, *varname=None*, *vartype='main'*)
Write the value of an attribute in current NASA Ames file either globally or attached to a given variable (only name, units, _FillValue and scale_factor), depending on the type

Parameters

- **attrname** (*string|*) – String name of attribute to write in currently open file.
- **attrvalue** (*string|int|float*) – Value of attribute to write in currently open file.
- **na_dict** (*dict*) – Optional - dictionary in which the attribute will be added. By default, na_dict = None and the attribute value is added to the currently opened dictionary. Only mandatory if creating a new file or creating a new dictionary.
- **varname** (*string|int*) – Optional - Name or number of variable to get list of attributes from. If no variable name is provided, the function returns global attributes.
- **vartype** (*string|*) – Optional - type of variable to get list of attributes from. If no variable type is provided with the variable name, the function returns an attribute of the main variable .

write_variable(*data*, *varname=None*, *vartype='main'*, *attrdict=None*, *na_dict=None*)
Write or update a variable in the NASA/Ames dictionary.

Parameters

- **data** (*list|egadsData*) – Data to be written in the NASA/Ames dictionary. data can be a list of value or an EgadsData instance.
- **var_name** (*string|int*) – The name or the sequential number of the variable to be written in the dictionary.
- **vartype** (*string*) – The type of data to read, by default `main`. Options are `independant` for independant variables, `main` for main variables. `main` is the default value.
- **attrdict** (*dict*) – Optional - Dictionary of variable attribute linked to the variable to be written in the dictionary. Mandatory only if data is not an EgadsData instance and is not already present in the dictionary.
- **na_dict** (*dict*) – Optional - The NASA/Ames dictionary in which the variable will be added. By default, na_dict = None and the variable is added to the currently opened dictionary. Only mandatory if creating a new file or creating a new dictionary.

class egads.input.netcdf_io.**NetCdf**(*filename=None*, *perms='r'*, ***kwargs*)
Bases: *egads.input.input_core.FileCore*

EGADS class for reading and writing to generic NetCDF files.

This module is a sub-class of *FileCore* and adapts the Python NetCDF4 library to the EGADS file-access methods.

Initializes file instance.

Parameters

- **filename** (*string*) – Optional - Name of file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are `w` for write (overwrites data in file), `a` and `r+` for append, and `r` for read. `r` is the default value

add_attribute(*attrname*, *value*, *varname=None*)
Adds attribute to currently open file. If varname is included, attribute is added to specified variable, otherwise it is added to global file attributes.

Parameters

- **attrname** (*string*) – Attribute name.
- **value** (*string*) – Value to assign to attribute name.
- **varname** (*string*) – Optional - If varname is provided, attribute name and value are added to specified variable in the NetCDF file.

add_dim(*name*, *size*)
Adds dimension to currently open file.

Parameters

- **name** (*string*) – Name of dimension to add
- **size** (*integer*) – Integer size of dimension to add.

change_variable_name(*varname*, *newname*)
Change the variable name in currently opened NetCDF file.

Parameters

- **varname** (*string*) – Name of variable to rename.
- **oldname** (*string*) – the new name.

convert_to_csv(*csv_file=None*, *float_format='%g'*, *annotation=False*, *no_header=False*)
Converts currently open NetCDF file to CSV file using Nappy API.

Parameters

- **csv_file** (*string*) – Optional - Name of output CSV file. If none is provided, name of current NetCDF is used and suffix changed to .csv
- **float_format** (*string*) – Optional - The formatting string used for formatting floats when writing to output file. Default - %g
- **annotation** (*bool*) – Optional - If set to true, write the output file with an additional left-hand column describing the contents of each header line. Default - False.
- **no_header** (*bool*) – Optional - If set to true, then only the data blocks are written to file. Default - False.

convert_to_nasa_ames(*na_file=None*, *requested_ffi=1001*, *float_format='%g'*, *delimiter=None*, *annotation=False*, *no_header=False*)
Convert currently open NetCDF file to one or more NASA Ames files using Nappy. For now can only process NetCdf files to NASA/Ames FFI 1001 : only time as an independant variable.

Parameters

- **na_file** (*string*) – Optional - Name of output NASA Ames file. If none is provided, name of current NetCDF file is used and suffix changed to .na
- **requested_ffi** (*int*) – The NASA Ames File Format Index (FFI) you wish to write to. Options are limited depending on the data structures found.
- **delimiter** (*string*) – Optional - The delimiter desired for use between data items in the data file. Default - Tab.
- **float_format** (*string*) – Optional - The formatting string used for formatting floats when writing to output file. Default - %g
- **delimiter** – Optional - The delimiter desired for use between data items in the data file. Default - ' ' (four spaces).
- **annotation** (*bool*) – Optional - If set to true, write the output file with an additional left-hand column describing the contents of each header line. Default - False.
- **no_header** (*bool*) – Optional - If set to true, then only the data blocks are written to file. Default - False.

delete_attribute(*attrname*, *varname=None*)
Deletes attribute to currently open file. If varname is included, attribute is removed from specified variable, otherwise it is removed from global file attributes.

Parameters

- **attrname** (*string*) – Attribute name.
- **varname** (*string*) – Optional - If varname is provided, attribute removed from specified variable in the NetCDF file.

get_attribute_list(*varname=None*)
Returns a dictionary of attributes and values found in current NetCDF file either globally, or attached to a given variable.

Parameters **varname** (*string*) – Optional - Name of variable to get list of attributes from. If no variable name is provided, the function returns top-level NetCDF attributes.

get_attribute_value(*attrname*, *varname=None*)
Returns value of an attribute given its name. If a variable name is provided, the attribute is returned from the variable specified, otherwise the global attribute is examined.

Parameters

- **name** (*string*) – Name of attribute to examine

- **varname** (*string*) – Optional - Name of variable attribute is attached to. If none specified, global attributes are examined.

get_dimension_list(*varname=None*)
Returns a dictionary of dimensions and their sizes found in the current NetCDF file. If a variable name is provided, the dimension names and lengths associated with that variable are returned.

Parameters **varname** (*string*) – Optional - Name of variable to get list of associated dimensions for. If no variable name is provided, the function returns all dimensions in the NetCDF file.

get_perms()
Returns the current permissions on the file that is open. Returns None if no file is currently open. Options are `w` for write (overwrites data in file),''a'' and `r+` for append, and `r` for read.

get_variable_list()
Returns a list of variables found in the current NetCDF file.

open(*filename*, *perms=None*)
Opens NetCDF file given filename.

Parameters

- **filename** (*string*) – Name of NetCDF file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are `w` for write (overwrites data in file), `a` and `r+` for append, and `r` for read. `r` is the default value

read_variable(*varname*, *input_range=None*)
Reads a variable from currently opened NetCDF file.

Parameters

- **varname** (*string*) – Name of NetCDF variable to read in.
- **input_range** (*vector*) – Optional - Range of values in each dimension to input. TODO add example

write_variable(*value*, *varname*, *dims=None*, *ftype='double'*, *fillvalue=None*)
Writes/creates variable in currently opened NetCDF file.

Parameters

- **value** (*array*) – Array of values to output to NetCDF file.
- **varname** (*string*) – Name of variable to create/write to.
- **dims** (*tuple*) – Optional - Name(s) of dimensions to assign to variable. If variable already exists in NetCDF file, this parameter is optional. For scalar variables, pass an empty tuple.
- **type** (*string*) – Optional - Data type of variable to write. Defaults to `double`. If variable exists, data type remains unchanged. Options for type are `double`, `float`, `int`, `short`, `char`, and `byte`
- **fill_value** (*float*) – Optional - Overrides default NetCDF _FillValue, if provided.

class egads.input.netcdf_io.**EgadsNetCdf**(*filename=None*, *perms='r'*)
Bases: *egads.input.netcdf_io.NetCdf*

EGADS class for reading and writing to NetCDF files following EUFAR conventions. Inherits from the general EGADS NetCDF module.

Initializes NetCDF instance.

Parameters

- **filename** (*string*) – Optional - Name of NetCDF file to open.

- **perms** (*char*) – Optional - Permissions used to open file. Options are `w` for write (overwrites data), `a` and `r+` for append, and `r` for read. `r` is the default value.

convert_to_csv(*csv_file=None*, *float_format='%g'*, *annotation=False*, *no_header=False*)
Converts currently open NetCDF file to CSV file using Nappy API.

Parameters

- **csv_file** (*string*) – Optional - Name of output CSV file. If none is provided, name of current NetCDF is used and suffix changed to .csv
- **float_format** (*string*) – Optional - The formatting string used for formatting floats when writing to output file. Default - %g
- **annotation** (*bool*) – Optional - If set to true, write the output file with an additional left-hand column describing the contents of each header line. Default - False.
- **no_header** (*bool*) – Optional - If set to true, then only the data blocks are written to file. Default - False.

convert_to_nasa_ames(*na_file=None*, *requested_ffi=1001*, *float_format='%g'*, *delimiter=None*, *annotation=False*, *no_header=False*)
Convert currently open EGADS NetCDF file to one or more NASA Ames files using Nappy. For now can only process NetCdf files to NASA/Ames FFI 1001 : variables can only be dependant to one independant variable at a time.

Parameters

- **na_file** (*string*) – Optional - Name of output NASA Ames file. If none is provided, name of current NetCDF file is used and suffix changed to .na
- **requested_ffi** (*int*) – The NASA Ames File Format Index (FFI) you wish to write to. Options are limited depending on the data structures found.
- **float_format** (*string*) – Optional - The formatting string used for formatting floats when writing to output file. Default - %g
- **delimiter** (*string*) – Optional - The delimiter desired for use between data items in the data file. Default - ‘ ‘ (four spaces).
- **annotation** (*bool*) – Optional - If set to true, write the output file with an additional left-hand column describing the contents of each header line. Default - False.
- **no_header** (*bool*) – Optional - If set to true, then only the data blocks are written to file. Default - False.

read_variable(*varname*, *input_range=None*)
Reads in a variable from currently opened NetCDF file and maps the NetCDF attributies to an `EgadsData` instance.

Parameters

- **varname** (*string*) – Name of NetCDF variable to read in.
- **input_range** (*vector*) – Optional - Range of values in each dimension to input.

write_variable(*data*, *varname=None*, *dims=None*, *ftype='double'*)
Writes/creates variable in currently opened NetCDF file.

Parameters

- **data** (EgadsData) – Instance of EgadsData object to write out to file. All data and attributes will be written out to the file.
- **varname** (*string*) – Optional - Name of variable to create/write to. If no varname is provided, and if cdf_name attribute in EgadsData object is defined, then the variable will be written to cdf_name.

- **dims** (*tuple*) – Optional - Name(s) of dimensions to assign to variable. If variable already exists in NetCDF file, this parameter is optional. For scalar variables, pass an empty tuple.
- **type** (*string*) – Optional - Data type of variable to write. Defaults to `double`. If variable exists, data type remains unchanged. Options for type are `double`, `float`, `int`, `short`, `char`, and `byte`

class egads.input.text_file_io.**EgadsFile**(*filename=None*, *perms='r'*)
Bases: *egads.input.input_core.FileCore*

Generic class for interfacing with text files.

Initializes instance of EgadsFile object.

Parameters

- **filename** (*string*) – Optional - Name of file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are `w` for write (overwrites data), `a` for append `r+` for read and write, and `r` for read. `r` is the default value.

close()
Close opened file.

display_file()
Prints contents of file out to standard output.

get_position()
Returns current position in file.

read(*size=None*)
Reads data in from file.

Parameters **size** (*int*) – Optional - Number of bytes to read in from file. If left empty, entire file will be read in.

Returns String data from text file.

Return type string

read_line()
Reads single line of data from file.

reset()
Returns to beginning of file

seek(*location*, *from_where=None*)
Change current position in file.

Parameters

- **location** (*integer*) – Position in file to seek to.
- **from_where** (*char*) – Optional - Where to seek from. Valid options are `b` for beginning, `c` for current and `e` for end.

write(*data*)
Writes data to a file. Data must be in the form of a string, with line ends signified by `\n`.

Parameters **data** (*string*) – Data to output to current file at current file position. Data must be a string, with `\n` signifying line end.

class egads.input.text_file_io.**EgadsCsv**(*filename=None*, *perms='r'*, *delimiter=','*, *quotechar=""*)
Bases: *egads.input.text_file_io.EgadsFile*

Class for reading data from CSV files.

Initializes instance of EgadsFile object.

Parameters

- **filename** (*string*) – Optional - Name of file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are `w` for write (overwrites data), `a` for append `r+` for read and write, and `r` for read. `r` is the default value.
- **delimiter** (*string*) – Optional - One-character string used to separate fields. Default is ‘,’.
- **quotechar** (*string*) – Optional - One-character string used to quote fields containing special characters. The default is “”.

display_file()
Prints contents of file out to standard output.

open(*filename*, *perms*, *delimiter=None*, *quotechar=None*)
Opens file.

Parameters

- **filename** (*string*) – Name of file to open.
- **perms** (*char*) – Optional - Permissions used to open file. Options are `w` for write (overwrites data), `a` for append `r+` for read and write, and `r` for read. `r` is the default value.
- **delimiter** (*string*) – Optional - One-character string used to separate fields. Default is ‘,’.
- **quotechar** (*string*) – Optional - One-character string used to quote fields containing special characters. The default is “”.

read(*lines=None*, *out_format=None*)
Reads in and returns contents of csv file.

Parameters

- **lines** (*int*) – Optional - Number specifying the number of lines to read in. If left blank, the whole file will be read and returned.
- **format** (*list*) – Optional - List type composed of one character strings used to decompose elements read in to their proper types. Options are `i` for int, `f` for float, `l` for long and `s` for string.

Returns List of arrays of values read in from file. If a format string is provided, the arrays are returned with the proper data type.

Return type list of arrays

skip_line(*amount=1*)
Skips over line(s) in file.

Parameters **amount** (*int*) – Optional - Number of lines to skip over. Default value is 1.

write(*data*)
Writes single row out to file.

Parameters **data** (*list*) – Data to be output to file using specified delimiter.

writerows(*data*)
Writes data out to file.

Parameters **data** (*list*) – List of variables to output.

egads.input.text_file_io.**parse_string_array**(*data*, *data_format*)
Converts elements in string list using format list to their proper types.

Parameters

- **data** (*numpy.ndarray*) – Input string array.
- **data_format** (*list*) – List type composed of one character strings used to decompose elements read in to their proper types. Options are 'i' for int, 'f' for float, 'l' for long and 's' for string.

Returns Array parsed into its proper types.

Return type numpy.ndarray

PYTHON MODULE INDEX

e

INDEX