EGADS Lineage Documentation

Release 1.2.8

EUFAR, Olivier Henry

Dec 02, 2021

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.

As the EUFAR FP7 project ended the 31st of January 2018, the development of EGADS has been stopped for now. To continue to improve EGADS outside the scope of EUFAR, a new branch has been created: EGADS Lineage. EGADS Lineage is still EGADS, but compatible with Python 3 and maintained by Olivier Henry. All issues reported in EGADS and EGADS Lineage will be fixed in EGADS Lineage only. A merging of both project will probably happen in the next EUFAR project.

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 Lineage can be obtained from GitHub (https://github.com/EUFAR/egads/tree/Lineage) or from PyPi (https://pypi.org/project/egads-lineage/)

2.1 Prerequisites

Use of EGADS requires the following packages:

- Python 3.5.4 or newer. Available at https://www.python.org/
- numpy 1.14 or newer. Available at http://numpy.scipy.org/
- scipy 1.0 or newer. Available at http://www.scipy.org/
- Python netCDF4 libraries 1.3.0 or newer. Available at https://pypi.python.org/pypi/netCDF4
- h5py 2.10.0 or newer. Available at https://pypi.org/project/h5py
- python_dateutil 2.6.1 or newer. Available at https://pypi.python.org/pypi/python-dateutil
- quantities 0.12.1 or newer. Available at https://pypi.org/project/quantities
- requests 2.18.4 or newer. Optional, only for update checking. Available at https://pypi.org/project/requests/

2.2 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-lineage` 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`.

2.3 Testing

To test EGADS after it is installed, from Python terminal, run the following commands:

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

On Linux, if issues occure with NetCDF4 or H5py, please check the last section of this chapter for a possible solution.

2.4 Options

Since version 0.7.0, an .ini file has been added to EGADS to welcome few options: log level and path, automatic check for a new EGADS version on GitHub. If the file is not present in EGADS directory, when importing, EGADS will create it automatically with default options. It is possible to display the status of the configuration file:

```
>>> import egads
>>> egads.print_options()
EGADS options:
    - logging level: DEBUG
    - log path: PATH_TO_PYTHON\Python35\lib\site-packages\egads
    - update automatic check: False
```

Actually, the number of option is limited and will probably incrase in the future. Here is a list of all options:

- `level` in `LOG` section: one of the items in the following list `DEBUG`, `INFO`, `WARNING`, `CRITICAL`, `ERROR` ; it is used to set the logging level when EGADS is imported.
- `path` in `LOG` section: a string corresponding to an OS path ; it is used to set the directory path where the log file is saved.
- `check_update` in `OPTIONS` section: True or False ; it is used to let EGADS check for an update automatically when it is imported.

The file containing all options is now stored in the folder `.egads_lineage` in the user $HOME directory.

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 `.egads_lineage` 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
>>> egads.set_options(log_level='INFO', log_path='/path/to/log/directory/')
>>> egads.set_options(log_level='INFO')
>>> egads.set_options(log_path='/path/to/log/directory/')
>>> 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.

2.6 Update

Since version 0.8.6, EGADS can check for an update on GitHub. The check system is launched in a separate thread and can be used this way:

```
>>> import egads
>>> egads.check_update()
EGADS Lineage vx.x.x is available on GitHub. You can update EGADS Lineage by using␣
↪pip (``pip install egads-lineage --upgrade``)
or by using the following link: https://github.com/eufarn7sp/egads/releases/
↪download/x.x.x/egads-x.x.x.tar.gz
```

If the `check_update` option is set on True in the egads.ini file, EGADS will check automatically for an update each time it is imported. By default, the option is set on False. The user can modify the option this way:

```
>>> import egads
>>> egads.set_options(check_update=True)
>>> exit()
```

The module Requests is optional for EGADS but is mandatory to check for an update.

2.7 Uninstallation

Just run the following command from your terminal:

```
>>> pip uninstall egads-lineage
```

or remove manually all folders in your Python site-packages folder containing egads name.

In the $HOME directory, delete .egads_lineage directory if you don't want to keep options and logs of EGADS Lineage.

2.8 Issues with NetCDF4 and/or H5py on a Linux distribution

If NetCDF4 and H5py libraries are installed through Pypi, a crash can occure when trying to read/write a netcdf or an hdf file. Here are the different steps to fix that particular issue:

1. Uninstall entirely NetCDF4
2. Download NetCDF4 sources corresponding to the version installed with Pypi
3. Unzip the package, launch a terminal and build NetCDF4 module -> `python setup.py build`
4. Finally install NetCDF4 module -> `python setup.py install`
5. Check NetCDF4 integration into EGADS with EGADS test function

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, Hdf), 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 https://python-quantities.readthedocs.io/en/latest

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. 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, simply create a `EgadsFile` instance with the parameters *filename* and *perms*:

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

EgadsFile (*filename*[, *perms='r'*])
Open a text file.

Parameters

- **filename** (*string*) – path and filename of a text file
- **perms** (*string*) – permissions ; optional

Return type text file

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 the file out to a standard output.

`f.`**get_position**`()`
Returns the current position in the file as an integer.

`f.`**seek** (*location*[, *from_where='b'*])
Seeks to a specified location in the text file.

Parameters

- **location** (*int*) – it is an integer specifying how far to seek
- **from_where** (*string*) – it is an option to specify from where 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 ; optional

Return type position in the text file

`f.`**reset**`()`
Resets the position to the beginning of the 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', standard_
↪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', standard_
↪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', standard_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["standard_name"] + '    ' + data1.metadata[
↪"standard_name"] + '    ' + data2.metadata["standard_name"])
```

- 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 csv file, simply create a `EgadsCsv` instance with the parameters *filename*, *perms*, *delimiter* and *quotechar*:

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

EgadsCsv (*filename*[, *perms='r'*, *delimiter=','*, *quotechar='"'*])
Open a text file.

Parameters

- **filename** (*string*) – path and filename of a text file
- **perms** (*string*) – permissions ; optional
- **delimiter** (*string*) – a one-character string used to separate fields ; optional
- **quotechar** (*string*) – a one-character string used to quote fields containing special characters ; optional

Return type csv file

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.

3.4.2 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 the file out to a standard output.

`f.`**get_position**`()`
Returns the current position in the file as an integer.

`f.`**seek** (*location*[, *from_where='b'*])
Seeks to a specified location in the text file.

Parameters

- **location** (*int*) – it is an integer specifying how far to seek
- **from_where** (*string*) – it is an option to specify from where 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 ; optional

Return type position in the text file

`f.`**reset**`()`
Resets the position to the beginning of the 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()
```

`f.`**`read`**`(`[*lines=None*, *format=None*]`)`

Returns a list of items read in from the CSV file.

Parameters

- **`lines`** (*int*) – if it is provided, the function will read in the specified number of lines, otherwise it will read the whole file ; optional
- **`format`** (*string*) – it is an optional list of characters used to decompose the elements read in from the CSV files to their proper types, options are ; optional

Return type list of items read in from the CSV file

Valid options for *format*:

- `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', standard_
↪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', standard_
↪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', standard_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_io.NetCdf`, allows simple read/write operations to NetCDF files. The other, `egads.input.netcdf_io.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 `EgadsNetCdf` instance or a `NetCdf` instance with the parameters *filename* and *perms*:

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

EgadsNetCdf (*filename*[, *perms='r'*])

Open a NetCDF file conforming the the EUFAR Standards & Protocols data and metadata regulations.

Parameters

- **filename** (*string*) – path and filename of a NetCDF file
- **perms** (*string*) – permissions ; optional

Return type NetCDF file.

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** ([*varname=None*, *group_walk=False*, *details=False*])

Returns a dictionary of all dimensions with their sizes. *varname* is optional and can take three different forms: *varname* is a variable name, and in that case the function returns a dictionary of all dimensions and their sizes attached to *varname* at the root of the NetCDF file ; *varname* is a path to a group + a variable name, the function returns a dictionary of all dimensions and their sizes attached to *varname* in the specified group ; *varname* is a path to group + a group name, the function returns a dictionary of all dimensions and their sizes in the specified group. *group_walk* is optional and if True, the function will explore the entire file is *varname* is None, or from a specified group if *varname* is a group path. *details* is optional, and if True, the path of each dimension is included in the final dictionary.

Parameters

- **varname** (*string*) – Name of variable or group to get the list of associated dimensions from. If no variable name is provided, the function returns all dimensions at the root of the NetCDF file ; optional
- **group_walk** (*bool*) – if True, the function visits all groups (if at least one exists) to list all dimensions. False by default ; optional
- **details** (*bool*) – if True, dimension path is provided in the dictionary. False by default ; optional

Return type ordered dictionary of dimensions

```
>>> print(f.get_dimension_list('temperature'))
>>> print(f.get_dimension_list('test1/data/temperature', details=True))
>>> print(f.get_dimension_list('test1/data', group_walk=True))
```

f.**get_attribute_list**([*varname=None*])

Returns a list of all top-level attributes. *varname* is optional and can take three different forms: *varname* is a variable name, and in that case the function returns all attributes and their values attached to *varname* at the root of the NetCDF file ; *varname* is a path to a group + a variable name, the function returns all attributes and their values attached to *varname* in the specified group ; *varname* is a path to group + a group name, the function returns all attributes and their values attached to the specified group.

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

Return type dictionary of attributes

```
>>> print(f.get_attribute_list('temperature'))
>>> print(f.get_attribute_list('test1/data/temperature'))
>>> print(f.get_attribute_list('test1/data'))
```

f.**get_variable_list**([*groupname=None*, *group_walk=False*, *details=False*])

Returns a list of all variables at the root of the NetCDF file. If *groupname* is provided, the function will list all variables located at *groupname*. If group_walk is True, the function will list all variables in the NetCDF file from root, or from *groupname* if *groupname* is provided, to the last folder. If details is True, the function returns a list of dictionary containing the name of the variable and its path in the NetCDF file. By default, details is False and the function returns a simple list of variable names.

Parameters

- **groupname** (*string*) – the name of the group to get the variable list from ; optional
- **group_walk** (*bool*) – if True, the function lists all variables from the root of the file, or from *groupname* if provided, to the last group ; optional
- **details** (*bool*) – if True, the function returns a list of dictionary in which the key is the name of the variable, and the value is the path of the variable in the NetCDF file ; optional

Return type list of variables

```
>>> print(f.get_variable_list())
>>> print(f.get_variable_list('test1/data'))
>>> print(f.get_variable_list('test1/data', group_walk=True, details=True))
```

f.**get_group_list**([*groupname=None*, *details=False*])

Returns a list of groups found in the NetCDF file. If *groupname* is provided, the function returns all groups from *groupname* to the last group in *groupname*. The function returns a list of string if details is False. If details is True, it returns a list of dictionary in which the key is the name of the group and the value its path in the NetCDF file.

Parameters

- **groupname** (*string*) – name of a group where to get the group list ; optional
- **details** (*bool*) – if True, the function returns a list of dictionary in which the key is the name of the group and the value its path in the NetCDF file ; optional

Return type list of strings or list of dictionary

```
>>> print(f.get_group_list())
>>> print(f.get_group_list('test1', True))
```

f.**get_filename**()

Returns the filename for the currently opened file.

Return type filename

`f.get_perms()`
Returns the current permissions on the file that is open.

Return type permissions

3.5.3 Reading data

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

```
>>> data = f.read_variable(varname, input_range, read_as_float, replace_fill_value)
```

`f.read_variable(`*varname*[, *input_range=None*, *read_as_float=False*, *replace_fill_value=False*]`)`
If using the `NetCdf` class, an array of values contained in *varname* will be returned. If using the `EgadsNetCdf` class, an instance of the `EgadsData` class will be returned containing the values and attributes of *varname*. If a group path is present in *varname*, then the function reads the variable *varname* in that specified group.

Parameters

- **varname** (*string*) – name of a variable, with or without group path, in the NetCDF file
- **input_range** (*list*) – list of min/max values ; optional
- **read_as_float** (*bool*) – if True, EGADS reads the data and convert them to float numbers, if False, the data type is the type of data in file ; optional
- **replace_fill_value** (*bool*) – if True, EGADS reads the data and replace `_FillValue` or `missing_value` (if one of the attributes exists) in data by NaN (numpy.nan) ; optional

Return type data, `EgadsData` or array

```
>>> data = f.read_variable('temperature')
>>> data = f.read_variable('test1/data/temperature')
```

3.5.4 Writing data

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

`f.add_dim(`*name*, *size*`)`

Add a dimension to the NetCDF file. If a path to a group plus the dimension name is included in *name*, the dimension
added to the group. In that case, the group has to be created before.

Parameters

- **name** (*string*) – the name of the dimension
- **size** (*int*) – the size of the dimension

```
>>> f.add_dim('time', len(time))
>>> f.add_dim('time', len(time), 'test1/data')
>>> f.add_dim('time', len(time), ['test1', 'data'])
```

`f.add_attribute(`*attrname*, *value*[, *varname=None*]`)`
Add an attribute to the NetCDF file. If *varname* is None, the attribute is a global attribute, and if not, the attribute is a variable attribute attached to *varname*. If the path of a group is present in *varname*, the attribute is attached to the variable stored in the specified group. If *varname* is simply the path of a group, attribute is attached to the group.

Parameters

- **attrname** (*string*) – the name of the attribute
- **value** (*string | float | int*) – the value of the attribute
- **varname** (*string*) – the name of the variable | group to which to attach the attribute ; optional

```
>>> f.add_attribute('project', 'my project')
>>> f.add_attribute('project', 'my project', 'temperature')
>>> f.add_attribute('project', 'my project', 'test1/data/temperature')
>>> f.add_attribute('project', 'my project', 'test1/data')
```

If using `EgadsNetCdf`, data can be output to variables using the `write_variable()` function as follows:

```
>>> f.write_variable(data, varname, dims, ftype, fillvalue)
```

`f.`**write_variable**(*data*, *varname*[, *dims=None*, *ftype='double'*, *fillvalue=None*])

Write the values contained in *data* in the variable *varname* in a NetCDF file. If *varname* contains a path to a group, the variable will be created in the specified group, but in that case the group has to be created before. An instance of `EgadsData` must be passed into `write_variable`. Any attributes that are contained within the `EgadsData` instance are applied to the NetCDF variable as well. If an attribute with a name equal to `_FillValue` or `missing_value` is found, NaN in data will be automatically replaced by the missing value. If attributes with the name `scale_factor` and/or `add_offset` are found, those attributes are automatically applied to the data.

Parameters

- **data** (*EgadsData | array | vector | scalar*) – values to be stored in the NetCDF file
- **varname** (*string*) – the name of the variable, or the path of group + the name of the variable, in the NetCDF file
- **dims** (*tuple*) – a tuple of dimension names for data (not needed if the variable already exists) ; optional
- **ftype** (*string*) – the data type of the variable, the default value is *double*, other valid options are *float*, *int*, *short*, *char* and *byte* ; optional
- **fillvalue** (*float | int*) – if it is provided, it overrides the default NetCDF _FillValue ; optional, it doesn't exist if using `EgadsNetCdf`

```
>>> f.write_variable(data, 'particle_size', ('time', ))
>>> f.write_variable(data, 'test1/data/particle_size', ('time', ))
```

If using `NetCdf`, data can be output to variables using the `write_variable()` function as follows:

```
>>> f.write_variable(data, varname, dims, ftype, fillvalue, scale_factor, add_
↪offset)
```

`f.`**write_variable**(*data*, *varname*[, *dims=None*, *ftype='double'*, *fillvalue=None*, *scale_factor=None*, *add_offset=None*])

Write the values contained in *data* in the variable *varname* in a NetCDF file. If *varname* contains a path to a group, the variable will be created in the specified group, but in that case the group has to be created before. Values for *data* passed into `write_variable` must be scalar or array.

Parameters

- **data** (*EgadsData | array | vector | scalar*) – values to be stored in the NetCDF file
- **varname** (*string*) – the name of the variable, or the path of group + the name of the variable, in the NetCDF file

- **dims** (*tuple*) – a tuple of dimension names for data (not needed if the variable already exists) ; optional
- **ftype** (*string*) – the data type of the variable, the default value is *double*, other valid options are *float*, *int*, *short*, *char* and *byte* ; optional
- **fillvalue** (*float | int*) – if it is provided, it overrides the default NetCDF _FillValue ; optional, it doesn't exist if using `EgadsNetCdf`
- **scale_factor** (*float | int*) – if data must be scaled, use this parameter ; optional
- **add_offset** (*float | int*) – if an offset must be added to data, use this parameter ; optional

```
>>> f.write_variable(data, 'particle_size', ('time', ))
>>> f.write_variable(data, 'test1/data/particle_size', ('time', ), scale_
↪factor=1.256, add_offset=5.56)
```

3.5.5 Conversion from NetCDF to NASA Ames file format

The conversion is only possible on opened NetCDF files and with variables at the root of the NetCDF file. 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 is 1001. Consequently, if more than one independant variables are present in the NetCDF file, 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, and with the presence of groups, the conversion should be done manually by creating a NasaAmes instance and a NasaAmes dictionary, by populating the dictionary and by saving the file.

To convert a NetCDF file to NasaAmes file format, simply use:

`f.`**convert_to_nasa_ames**(*[na_file=None, float_format=None, delimiter=' ', no_header=False]*)

Convert the opened NetCDF file to NasaAmes file.

Parameters

- **na_file** (*string*) – it is the name of the output file once it has been converted, by default, *na_file* is None, and the name of the NetCDF file will be used with the extension .na ; optional
- **float_format** (*string*) – it is the formatting string used for formatting floats when writing to output file ; optional
- **delimiter** (*string*) – it is a character or a sequence of character to use between data items in the data file ; optional (by default ' ', 4 spaces)
- **no_header** (*bool*) – if it is set to `True`, then only the data blocks are written to file ; optional

```
>>> f.convert_to_nasa_ames(na_file='nc_converted_to_na.na', float_format='%.8f
↪', delimiter=';', no_header=False)
```

To convert a NetCDF file to NasaAmes CSV file format, simply use:

`f.`**convert_to_csv**(*[csv_file=None, float_format=None, no_header=False]*)

Convert the opened NetCDF file to NasaAmes CSV file.

Parameters

- **csv_file** (*string*) – it is the name of the output file once it has been converted, by default, *na_file* is None, and the name of the NetCDF file will be used with the extension .csv ; optional
- **float_format** (*string*) – it is the formatting string used for formatting floats when writing to output file ; optional

- **no_header** (*bool*) – if it is set to `True`, then only the data blocks are written to file ; optional

3.5.6 Conversion from NetCDF to Hdf5 file format

EGADS Lineage offers a direct possibility to convert a full NetCDF file to Hdf file format. In the case of complexe NetCdf files, a manual Hdf file creation and editing is still possible.

`f.`**convert_to_hdf**(*[filename=None]*)

Converts the opened NetCdf file to Hdf format following the EUFAR and EGADS convention. If groups exist, they are preserved in the new Hdf file.

Parameters **filename** (*string*) – if only a name is given, a Hdf file named `filename` is created in the NetCdf file folder ; if a path and a name are given, a Hdf file named `name` is created in the folder `path` ; optional

3.5.7 Other operations

`f.`**get_attribute_value**(*attrname[, varname=None]*)

Return the value of the global attribute *attrname*, or the value of the variable attribute *attrname* if *varname* is not None. If *varname* contains a path to a group + a variable name, the function returns the attribute value attached to the variable in the specified group. If *varname* is simple path of group, the functions returns the attribute value attached to the group.

Parameters

- **attrname** (*string*) – the name of the attribute
- **varname** (*string*) – the name of the variable | group to which the attribute is attached

Return type value of the attribute

```
>>> print(f.get_attribute_value('project'))
>>> print(f.get_attribute_value('long_name', 'temperature'))
>>> print(f.get_attribute_value('long_name', 'test1/data/temperature'))
>>> print(f.get_attribute_value('project', 'test1/data'))
```

`f.`**change_variable_name**(*varname*, *newname*)

Change a variable name in the currently opened NetCDF file. If *varname* contains a path to a group + a variable name, the variable name in the specified group is changed.

Parameters

- **attrname** (*string*) – the actual name of the variable
- **varname** (*string*) – the new name of the variable

```
>>> f.change_variable_name('particle_nbr', 'particle_number')
>>> f.change_variable_name('test1/data/particle_nbr', 'particle_number')
```

`f.`**add_group**(*groupname*)

Create a group in the NetCDF file. *groupname* can be a path + a group name or a sequence of group, in both cases, intermediary groups are created if needed.

Parameters **groupname** (*string | list*) – a group name or a list of group name

```
>>> f.add_group('MSL/north_atlantic/data')
>>> f.add_group(['MSL', 'north_atlantic', 'data'])
```

3.5.8 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.

3.5.9 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', standard_
↪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', standard_
↪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', standard_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 Hdf files

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

3.6.1 Opening

To open a Hdf file, simply create a `EgadsHdf` instance or a `Hdf` instance with the parameters *filename* and *perms*:

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

EgadsHdf (*filename*[, *perms='r'*])

Open a Hdf file conforming the the EUFAR Standards & Protocols data and metadata regulations.

Parameters

- **filename** (*string*) – path and filename of a Hdf file
- **perms** (*string*) – permissions ; optional

Return type Hdf file.

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`.

3.6.2 Getting info

f.**get_dimension_list** ([*varname=None*, *group_walk=False*, *details=False*])

Returns a dictionary of all dimensions with their sizes. *varname* is optional and can take three different forms: *varname* is a variable name, and in that case the function returns a dictionary of all dimensions and their sizes attached to *varname* at the root of the Hdf file ; *varname* is a path to a group + a variable name, the function returns a dictionary of all dimensions and their sizes attached to *varname* in the specified group ; *varname* is a path to a group + a group name, the function returns a dictionary of all dimensions and their sizes in the specified group. *group_walk* is optional and if True, the function will explore the entire file is *varname* is None, or from a specified group if *varname* is a group path. *details* is optional, and if True, the path of each dimension is included in the final dictionary.

Parameters

- **varname** (*string*) – Name of variable or group to get the list of associated dimensions from. If no variable name is provided, the function returns all dimensions at the root of the Hdf file ; optional
- **group_walk** (*bool*) – if True, the function visits all groups (if at least one exists) to list all dimensions. False by default ; optional
- **details** (*bool*) – if True, dimension path is provided in the dictionary. False by default ; optional

Return type ordered dictionary of dimensions

```
>>> print(f.get_dimension_list('temperature'))
>>> print(f.get_dimension_list('test1/data/temperature'))
>>> print(f.get_dimension_list('test1/data'))
```

f.**get_attribute_list**([*objectname=None*])
Returns a list of all top-level attributes. *varname* is optional and can take three different forms: *objectname* is a variable name, and in that case the function returns all attributes and their values attached to *objectname* at the root of the Hdf file ; *objectname* is a path to a group + a variable name, the function returns all attributes and their values attached to *objectname* in the specified group ; *objectname* is a path to a group + a group name, the function returns all attributes and their values attached to the specified group.

Parameters **objectname** (*string*) – name of a variable / group ; optional

Return type dictionary of attributes

```
>>> print(f.get_attribute_list('temperature'))
>>> print(f.get_attribute_list('test1/data/temperature'))
>>> print(f.get_attribute_list('test1/data'))
```

f.**get_variable_list**([*groupname=None*, *group_walk=False*, *details=False*])
Returns a list of all variables at the root of the Hdf file if *groupname* is None, otherwise a list of all variables in the group *groupname*. If *group_walk* is True, the the function will explore all the file or from *groupname* if *groupname* is provided. If *details* is True, the function returns a list of dictionary containing the name of the variable and its path in the Hdf file. By default, details is False and the function returns a simple list of variable names.

Parameters

- **groupname** (*string*) – the name of the group to get the list from ; optional
- **group_walk** (*bool*) – if True, the function visits all groups (if at least one exists) to list all variables. False by default ; optional
- **details** (*bool*) – if True, the function returns a list of dictionary in which the key is the name of the variable, and the value is the path of the variable in the Hdf file ; optional

Return type list of variables

```
>>> print(f.get_variable_list())
>>> print(f.get_variable_list(details=True))
```

f.**get_file_structure**()
Returns a view of the file structure, groups and datasets.

Return type list of strings and Hdf objects

```
>>> print(f.get_file_structure())
```

f.**get_group_list**([*groupname=None*, *details=False*])
Returns a list of groups found in the Hdf file. If *groupname* is provided, the function returns all groups from *groupname* to the last group in *groupname*. The function returns a list of string if details is False. If details is True, it returns a list of dictionary in which the key is the name of the group and the value its path in the Hdf file.

Parameters

- **groupname** (*string*) – name of a group where to get the group list ; optional
- **details** (*bool*) – if True, the function returns a list of dictionary in which the key is the name of the group and the value its path in the Hdf file ; optional

Return type list of strings or list of dictionary

```
>>> print(f.get_group_list())
>>> print(f.get_group_list('test1', True))
```

f.**get_filename**()
Returns the filename for the currently opened file.

Return type filename

f.**get_perms**()
Returns the current permissions on the file that is open.

Return type permissions

3.6.3 Reading data

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

```
>>> data = f.read_variable(varname, input_range, read_as_float, replace_fill_value)
```

f.**read_variable**(*varname*[, *input_range=None*, *read_as_float=False*, *replace_fill_value=False*])
If using the `Hdf` class, an array of values contained in *varname* will be returned. If using the `EgadsHdf` class, an instance of the `EgadsData` class will be returned containing the values and attributes of *varname*. If a group path is present in *varname*, then the function reads the variable *varname* in that specified group.

Parameters

- **varname** (*string*) – name of a variable, with or without group path, in the Hdf file
- **input_range** (*list*) – list of min/max values ; optional
- **read_as_float** (*bool*) – if True, EGADS reads the data and convert them to float numbers, if False, the data type is the type of data in file ; optional
- **replace_fill_value** (*bool*) – if True, EGADS reads the data and replace `_FillValue` or `missing_value` (if one of the attributes exists) in data by NaN (numpy.nan) ; optional

Return type data, `EgadsData` or array

```
>>> data = f.read_variable('temperature')
>>> data = f.read_variable('test1/data/temperature')
```

3.6.4 Writing data

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

f.**add_dim**(*name*, *data*[, *ftype='double'*])
Add a dimension to the Hdf file. The name of the dimension can include a path to a group where to store the dimension. In that case, the group has to be created before. If using `Hdf`, values for *data* passed into `add_dim` must be scalar or array. Otherwise, if using `EgadsHdf`, an instance of `EgadsData` must be passed into `add_dim`. In this case, any attributes that are contained within the `EgadsData` instance are applied to the Hdf variable as well.

Parameters

- **name** (*string*) – the name of the dimension, or the path to a group + the name of the dimension
- **data** (*EgadsData|array|vector|scalar*) – values to be stored in the Hdf file
- **ftype** (*string*) – the data type of the variable, the default value is *double*, other valid options are *float*, *int*, *short*, *char* and *byte* ; optional

```
>>> f.add_dim('time', time_data, len(time))
>>> f.add_dim('test1/data/time', time_data, len(time))
```

`f.`**`add_attribute`**(*attrname*, *value*[, *objname=None*])

Add an attribute to the Hdf file. If *objname* is None, the attribute is a global attribute, and if not, the attribute is attached to *objname*. *objname* can be a group (with or without a path) or variable (with or without a path).

Parameters

- **attrname** (*string*) – the name of the attribute
- **value** (*string | float | int*) – the value of the attribute
- **objname** (*string*) – the name of the variable | group to which to attach the attribute ; optional

```
>>> f.add_attribute('project', 'my project')
>>> f.add_attribute('project', 'my project', 'temperature')
>>> f.add_attribute('project', 'my project', 'test1/data/temperature')
>>> f.add_attribute('project', 'my project', 'test1/data')
```

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

```
>>> f.write_variable(data, varname, dims, ftype, fillvalue)
```

`f.`**`write_variable`**(*data*, *varname*[, *dims=None*, *ftype='double'*])

Write the values contained in *data* in the variable *varname* in a Hdf file. If *varname* contains a path to a group, the variable will be created in the specified group, but in that case the group has to be created before. If using `Hdf`, values for *data* passed into `write_variable` must be scalar or array. Otherwise, if using `EgadsHdf`, 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 Hdf variable as well.

Parameters

- **data** (*EgadsData | array | vector | scalar*) – values to be stored in the Hdf file
- **varname** (*string*) – the name of the variable, or the path of group + the name of the variable, in the Hdf file
- **dims** (*tuple*) – a tuple of dimension names for data (not needed if the variable already exists) ; optional
- **ftype** (*string*) – the data type of the variable, the default value is *double*, other valid options are *float*, *int*, *short*, *char* and *byte* ; optional

```
>>> f.write_variable(data, 'particle_size', ('time', ))
>>> f.write_variable(data, 'test1/data/particle_size', ('time', ))
```

3.6.5 Conversion from Hdf5 to NASA Ames file format

The conversion is only possible on opened Hdf files and with variables at the root of the Hdf file. 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 is 1001. Consequently, if more than one independant variables are present in the Hdf file, 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, and with the presence of groups, the conversion should be done manually by creating a NasaAmes instance and a NasaAmes dictionary, by populating the dictionary and by saving the file.

To convert a Hdf file to NasaAmes file format, simply use:

`f.`**`convert_to_nasa_ames`**([*na_file=None*, *float_format=None*, *delimiter=' '*, *no_header=False*])

Convert the opened NetCDF file to NasaAmes file.

Parameters

- **na_file** (*string*) – it is the name of the output file once it has been converted, by default, *na_file* is None, and the name of the Hdf file will be used with the extension .na ; optional
- **float_format** (*string*) – it is the formatting string used for formatting floats when writing to output file ; optional
- **delimiter** (*string*) – it is a character or a sequence of character to use between data items in the data file ; optional (by default ' ', 4 spaces)
- **no_header** (*bool*) – if it is set to `True`, then only the data blocks are written to file ; optional

```
>>> f.convert_to_nasa_ames(na_file='nc_converted_to_na.na', float_format='%.8f
↪', delimiter=';', no_header=False)
```

To convert a Hdf file to NasaAmes CSV file format, simply use:

`f.`**convert_to_csv**(*[csv_file=None, float_format=None, no_header=False]*)

Convert the opened Hdf file to NasaAmes CSV file.

Parameters

- **csv_file** (*string*) – it is the name of the output file once it has been converted, by default, *na_file* is None, and the name of the Hdf file will be used with the extension .csv ; optional
- **float_format** (*string*) – it is the formatting string used for formatting floats when writing to output file ; optional
- **no_header** (*bool*) – if it is set to `True`, then only the data blocks are written to file ; optional

3.6.6 Conversion from Hdf5 to NetCDF file format

EGADS Lineage offers a direct possibility to convert a full Hdf file to NetCDF file format. In the case of complexe Hdf5 files, a manual NetCDF file creation and editing is still possible.

`f.`**convert_to_netcdf**(*[filename=None]*)

Converts the opened Hdf file to NetCdf format following the EUFAR and EGADS convention. If groups exist, they are preserved in the new NetCDF file.

Parameters **filename** (*string*) – if only a name is given, a NetCDF file named `filename` is created in the HDF file folder ; if a path and a name are given, a NetCDF file named `name` is created in the folder `path` ; optional

3.6.7 Other operations

`f.`**get_attribute_value**(*attrname[, objectname=None]*)

Return the value of the global attribute *attrname*, or the value of the variable attribute *attrname* if *objectname* is not None. If *objectname* contains a path to a group + a variable name, the function returns the attribute value attached to the variable in the specified group. If *objectname* is simple path of group, the functions returns the attribute value attached to the group.

Parameters

- **attrname** (*string*) – the name of the attribute
- **objectname** (*string*) – the name of the variable | group to which the attribute is attached

Return type value of the attribute

```
>>> print(f.get_attribute_value('project'))
>>> print(f.get_attribute_value('long_name', 'temperature'))
>>> print(f.get_attribute_value('long_name', 'test1/data/temperature'))
>>> print(f.get_attribute_value('project', 'test1/data'))
```

f.**add_group**(*groupname*)

Create a group in the Hdf file. *groupname* can be a path + a group name or a sequence of group, in both cases, intermediary groups are created if needed.

Parameters **groupname** (*string | list*) – a group name or a list of group name

```
>>> f.add_group('MSL/north_atlantic/data')
>>> f.add_group(['MSL', 'north_atlantic', 'data'])
```

f.**delete_attribute**(*attrname*[, *objectname=None*])

Delete the attribute *attrname* at the root of the Hdf file if *objectname* is None, or attached to *objectname*. *objectname* can be the name of a variable or a group, or a path to a group plus the name of a variable or a group.

Parameters

- **attrname** (*string*) – the name of the attribute
- **objectname** (*string*) – the name of the variable | group to which the attribute is attached

```
>>> f.delete_attribute('long_name')
>>> f.delete_attribute('long_name', 'temperature')
>>> f.delete_attribute('long_name', 'test1/data/temperature')
>>> f.delete_attribute('project', 'test1/data')
```

f.**delete_group**(*groupname*)

Delete the group *groupname* in the Hdf file. *groupname* can be a name of a group at the root of the Hdf file, or a path to a group plus the name of a a group.

Parameters **attrname** (*string*) – the name of the group

```
>>> f.delete_group('data')
>>> f.delete_group('test1/data')
```

f.**delete_variable**(*varname*)

Delete the variable *varname* in the Hdf file. *varname* can be the name of a variable or a path to a group plus the name of a variable.

Parameters **varname** (*string*) – the name of the variable

```
>>> f.delete_variable('temperature')
>>> f.delete_variable('test1/data/temperature')
```

3.6.8 Closing

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

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

3.6.9 Dataset with compound data

Dataset with compound data are not specifically handled by EGADS. When a dataset is read and contains compound data, it is possible to access the different fields in this way:

```
>>> temperature = f.read_variable('temperature')
>>> date_field = temperature['date']
```

Obviously, if the user declared an EgadsHdf instance to read the file, the compound data is still an EgadsData instance, with the same metadata and units. If the user declared an Hdf instance to read the file, the compound data is a Numpy ndarray instance. In EGADS, units are handled automatically. Thus, with compound data and multiple data in the same dataset, it is highly likely that units won't be handled properly, and will be set to dimensionless most of the time.

Note: With the instance EgadsHdf, an attribute has been added to the EgadsData instance, `compound_data`, which informs the user about the dataset type. If the dataset contains compound data, the attribute `compound_data` is set to `True` ; if not it is set to `False`. Then it is the responsability of the user to explore the different fields in the dataset.

3.6.10 Tutorial

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

```
=> ncdump -h main_hdf_file.hdf5
   netcdf main_hdf_file {
   dimensions:
           time = 5 ;
   variables:
           double corrected_sea_level(time) ;
                   string corrected_sea_level:name = "corr sea level" ;
                   corrected_sea_level:scale_factor = 1. ;
                   corrected_sea_level:_FillValue = -9999 ;
                   string corrected_sea_level:units = "mm" ;
           double sea_level(time) ;
                   string sea_level:name = "sea level" ;
                   sea_level:scale_factor = 1. ;
                   sea_level:_FillValue = -9999 ;
                   string sea_level:units = "mm" ;
           double time(time) ;
                   string time:name = "time" ;
                   string time:units = "seconds since 19700101T00:00:00" ;

   // global attributes:
                   string :Conventions = "CF-1.0" ;
                   string :history = "the hdf file has been created by EGADS" ;
                   string :comments = "no comments on the hdf file" ;
                   string :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 EgadsHdf instance with a file name:

```
>>> f = egads.input.EgadsHdf('main_hdf_file.hdf5', 'w')
```

- add the global attributes to the Hdf file:

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

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

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

- write the variable(s), and no need to write the variable `time`, it has already been added by the command `add_dim()`:

```
>>> 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()
```

3.7 Working with NASA Ames files

EGADS provides two classes to work with NASA Ames files. The simplest, `egads.input.nasa_ames_io.NasaAmes`, allows simple read/write operations. The other, `egads.input.nasa_ames_io.EgadsNasaAmes`, is designed to interface with NASA Ames files conforming to the EUFAR Standards & Protocols data and metadata regulations. This class directly reads or writes NASA Ames file using instances of the `EgadsData` class. Actually, only the FFI 1001 has been interfaced with EGADS.

3.7.1 Opening

To open a NASA Ames file, simply create a `EgadsNasaAmes` instance with the parameters *pathname* and *permissions*:

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

EgadsNasaAmes (*pathname*[, *permissions='r'*])
Open a NASA Ames file conforming the the EUFAR Standards & Protocols data and metadata regulations.

Parameters

- **filename** (*string*) – path and filename of a NASA Ames file
- **perms** (*string*) – permissions ; optional

Return type NasaAmes file.

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 a new file.

3.7.2 Getting info

`f.`**get_dimension_list** ([*na_dict=None*])
Returns a list of all variable dimensions.

Parameters **na_dict** (*dict*) – if provided, the function get dimensions from the NasaAmes dictionary *na_dict*, if not dimensions are from the opened file ; optional

Return type dictionary of dimensions

`f.`**get_attribute_list** ([*varname=None*, *vartype='main'*, *na_dict=None*])
Returns a dictionary of all top-level attributes.

Parameters

- **varname** (*string*) – name of a variable, if provided, the function returns a dictionary of all attributes attached to *varname* ; optional
- **vartype** (*string*) – if provided and *varname* is not `None`, the function will search in the variable type *vartype* by default ; optional
- **na_dict** (*dict*) – if provided, it will return a list of all top-level attributes, or all *varname* attributes, from the NasaAmes dictionary *na_dict* ; optional

Return type dictionary of attributes

f.**get_attribute_value**(*attrname*[, *varname=None*, *vartype='main'*, *na_dict=None*])
Returns the value of a top-level attribute named *attrname*.

Parameters

- **attrname** (*string*) – the name of the attribute
- **varname** (*string*) – name of a variable, if provided, the function returns the value of the attribute attached to *varname* ; optional
- **vartype** (*string*) – if provided and *varname* is not None, the function will search in the variable type *vartype* by default ; optional
- **na_dict** (*dict*) – if provided, it will return the value of an attribute from the NasaAmes dictionary *na_dict* ; optional

Return type value of attribute

f.**get_variable_list**([*na_dict=None*])
Returns a list of all variables.

Parameters **na_dict** (*dict*) – if provided, it will return the list of all variables from the NasaAmes dictionary *na_dict* ; optional

Return type list of variables

f.**get_filename**()
Returns the filename for the currently opened file.

Return type filename

f.**get_perms**()
Returns the current permissions on the file that is open.

Return type permissions

3.7.3 Reading data

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

```
>>> data = f.read_variable(varname, na_dict, read_as_float, replace_fill_value)
```

f.**read_variable**(*varname*[, *na_dict=None*, *read_as_float=False*, *replace_fill_value=False*])
If using the NasaAmes class, an array of values contained in *varname* will be returned. If using the EgadsNasaAmes class, an instance of the EgadsData class will be returned containing the values and attributes of *varname*.

Parameters

- **varname** (*string*) – name of a variable in the NasaAmes file
- **na_dict** (*dict*) – it will tell to EGADS in which Nasa Ames dictionary to read data, if na_dict is None, data are read in the opened file ; optional
- **read_as_float** (*bool*) – if True, EGADS reads the data and convert them to float numbers, if False, the data type is the type of data in file ; optional
- **replace_fill_value** (*bool*) – if True, EGADS reads the data and replace _FillValue or missing_value (if one of the attributes exists) in data by NaN (numpy.nan) ; optional

Return type data, EgadsData or array

3.7.4 Writing data

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

`f.`**`create_na_dict`**`()`
Create a new dictionary populated with standard NasaAmes keys

`f.`**`write_attribute_value`**`(`*attrname*, *attrvalue*[, *na_dict=None*, *varname=None*, *vartype='main'*]`)`
Write or replace a specific attribute (from the official NasaAmes attribute list) in the currently opened dictionary.

Parameters

- **attrname** (*string*) – name of the attribute in the NasaAmes dictionary
- **attrvalue** (*string|float|integer|list|array*) – value of the attribute
- **na_dict** (*dict*) – if provided the function will write the attribute in the NasaAmes dictionary *na_dict* ; optional
- **varname** (*string*) – if provided, write or replace a specific attribute linked to the variable *var_name* in the currently opened dictionary ; accepted attributes for a variable are 'name', 'units', '_FillValue' and 'scale_factor', other attributes will be refused and should be passed as 'special comments' ; optional
- **vartype** (*string*) – if provided and *varname* is not `None`, the function will search in the variable type *vartype* by default ; optional

`f.`**`write_variable`**`(`*data*[, *varname=None*, *vartype='main'*, *attrdict=None*, *na_dict=None*]`)`
Write or replace a variable in the currently opened dictionary. If using the `NasaAmes` class, an array of values for *data* is asked. If using the `EgadsNasaAmes` class, an instance of the `EgadsData` class must be injected for *data*. If a `EgadsData` is passed into the `write_variable` function, any attributes that are contained within the `EgadsData` instance are automatically populated in the NASA Ames dictionary as well, those which are not mandatory are stored in the 'SCOM' attribute. If an attribute with a name equal to `_FillValue` or `missing_value` is found, NaN in data will be automatically replaced by the missing value.

Parameters

- **data** (*EgadsData|array|vector|scalar*) – values to be stored in the NasaAmes file
- **varname** (*string*) – the name of the variable ; if data is an `EgadsData`, mandatory if 'standard_name' or 'long_name' is not an attribute of *data* ; absolutely mandatory if *data* is not an `EgadsData` ; optional
- **vartype** (*string*) – the type of *data*, 'independant' or 'main', only mandatory if *data* must be stored as an independant variable (dimension) ; optional
- **attrdict** (*dict*) – a dictionary containing mandatory attributes ('name', 'units', '_FillValue' and 'scale_factor'), only mandatory if *data* is not an `EgadsData` ; optional
- **na_dict** (*dict*) – if provided, the function stores the variable in the NasaAmes dictionary *na_dict*

3.7.5 Saving a file

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

```
>>> data = f.save_na_file(filename, na_dict, float_format, delimiter, no_header):
```

f.**save_na_file**([*filename=None*, *na_dict=None*, *float_format=None*, *delimiter=' '*, *no_header=False*])
Save the opened NasaAmes dictionary and file.

Parameters

- **filename** (*string*) – is the name of the new file, if not provided, the name of the opened NasaAmes file is used ; optional
- **na_dict** (*dict*) – the name of the NasaAmes dictionary to be saved, if not provided, the opened dictionary will be used ; optional
- **float_format** (*string*) – the format of the floating numbers in the file (by default, no round up) ; optional
- **delimiter** (*string*) – it is a character or a sequence of character to use between data items in the data file ; optional (by default ' ', 4 spaces)
- **no_header** (*bool*) – if it is set to `True`, then only the data blocks are written to file ; optional

3.7.6 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**([*nc_file=None*, *na_dict=None*])
Convert the opened NasaAmes file to NetCDF file format.

Parameters

- **nc_file** (*string*) – if provided, the function will use *nc_file* for the path and name of the new_file, if not, the function will take the name and path of the opened NasaAmes file and replace the extension by '.nc' ; optional
- **na_dict** (*dict*) – the name of the NasaAmes dictionary to be converted, if not provided, the opened dictionary will be used ; optional

3.7.7 Conversion from NASA/Ames file format to Hdf5

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 Hdf 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 Hdf instance and by populating the Hdf files with NASA/Ames data and metadata.

To convert a NASA/Ames file, simply use:

f.**convert_to_hdf**([*hdf_file=None*, *na_dict=None*])
Convert the opened NasaAmes file to Hdf file format.

Parameters

- **hdf_file** (*string*) – if provided, the function will use *hdf_file* for the path and name of the new_file, if not, the function will take the name and path of the opened NasaAmes file and replace the extension by '.h5' ; optional
- **na_dict** (*dict*) – the name of the NasaAmes dictionary to be converted, if not provided, the opened dictionary will be used ; optional

3.7.8 Other operations

`f.`**`read_na_dict`**`()`
Returns a deep copy of the current opened file dictionary

Return type deep copy of a dictionary

`egads.input.nasa_ames_io.`**`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

Return type string

3.7.9 Closing

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

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

3.7.10 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, or only one string with lines separated by \n:

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

3.8 Working with algorithms

Algorithms in EGADS are stored in the `egads.algorithms` module for embedded algorithms and in `egads.user_algorithms` module for user-defined algorithms. They are 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: 1.02
 |
 |  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)
```

If an algorithm has been created by a user and is not embedded by default in EGADS, it should be called like this:

```
>>> V_p = egads.user_algorithms.thermodynamics.VelocityTasCnrm().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:

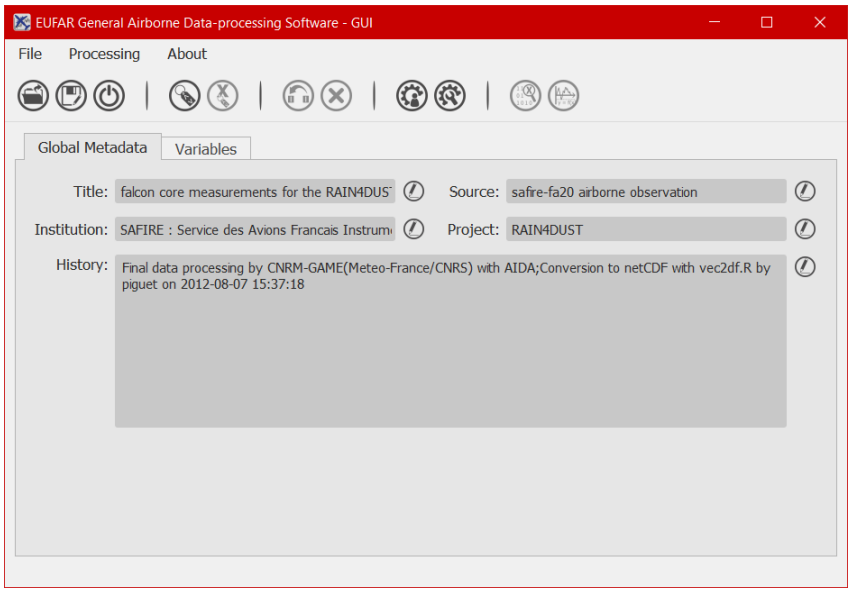
```
X = 0
DO I = 1,10
  X = X + I
  PRINT I
END DO
PRINT X
```

Python:

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



EGADS GUI can be launched as a simple python script from the terminal, if EGADS is installed, and once in the EGADS GUI directory:

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

Since version 1.0.0, a stand-alone package is available for those who wants to use the GUI without a Python installation. In that case, look for `EGADS Lineage GUI STA` in the release part of the repository. For Windows (from Windows 7 32), donwload the .msi package and launch the installation, it should be installed outside ProgramFiles to avoid issues with admin rights, then the GUI can be run by double clicking on egads_gui.exe or from the shortcut in the Startup menu. A .zip package is also available for those who don't want to install it. For Linux (from Linux 4.15), download the tar.gz package somewhere on your hard drive (preferably in your home directory), extract it and run egads_gui. The stand-alone versions for Linux and Windows have been created with PyInstaller, Windows 7 and Ubuntu 18.04.

Note: As for EGADS, the Graphical User Interface is available from two branches: master and Lineage (https://github.com/EUFAR/egads-gui/tree/Lineage). The Lineage one is only compatible with Python 3 and the earlier versions of EGADS Lineage.

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 doc/source/example_files/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
```

(continues on next page)

(continued from previous page)

```
    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'],
                       'OutputUnits':[%],
            'OutputTypes':['vector'],
            '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:
```

(continues on next page)

(continued from previous page)

```
        return result
```

The best practice before starting an algorithm is to copy this file and name it following the EGADS algorithm 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, units, types 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).
- OutputUnits: units of each output (cf. self.output_metadata['units']).
- OutputTypes: type of each output (ex: vector).
- 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 `$HOME/.egads_lineage/user_algorithms` 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 in the `user` directory, 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 `doc/source/example_files/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/EUFAR/egads/tree/Lineage.

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

5.2 Metadata Classes

5.3 File Classes

INDEX