pyaerocom

pyaerocom developers

Apr 25, 2024

CONTENTS:

Official website of pyaerocom, a Python package containing reading, post analysis and visualisation tools for the AeroCom project.

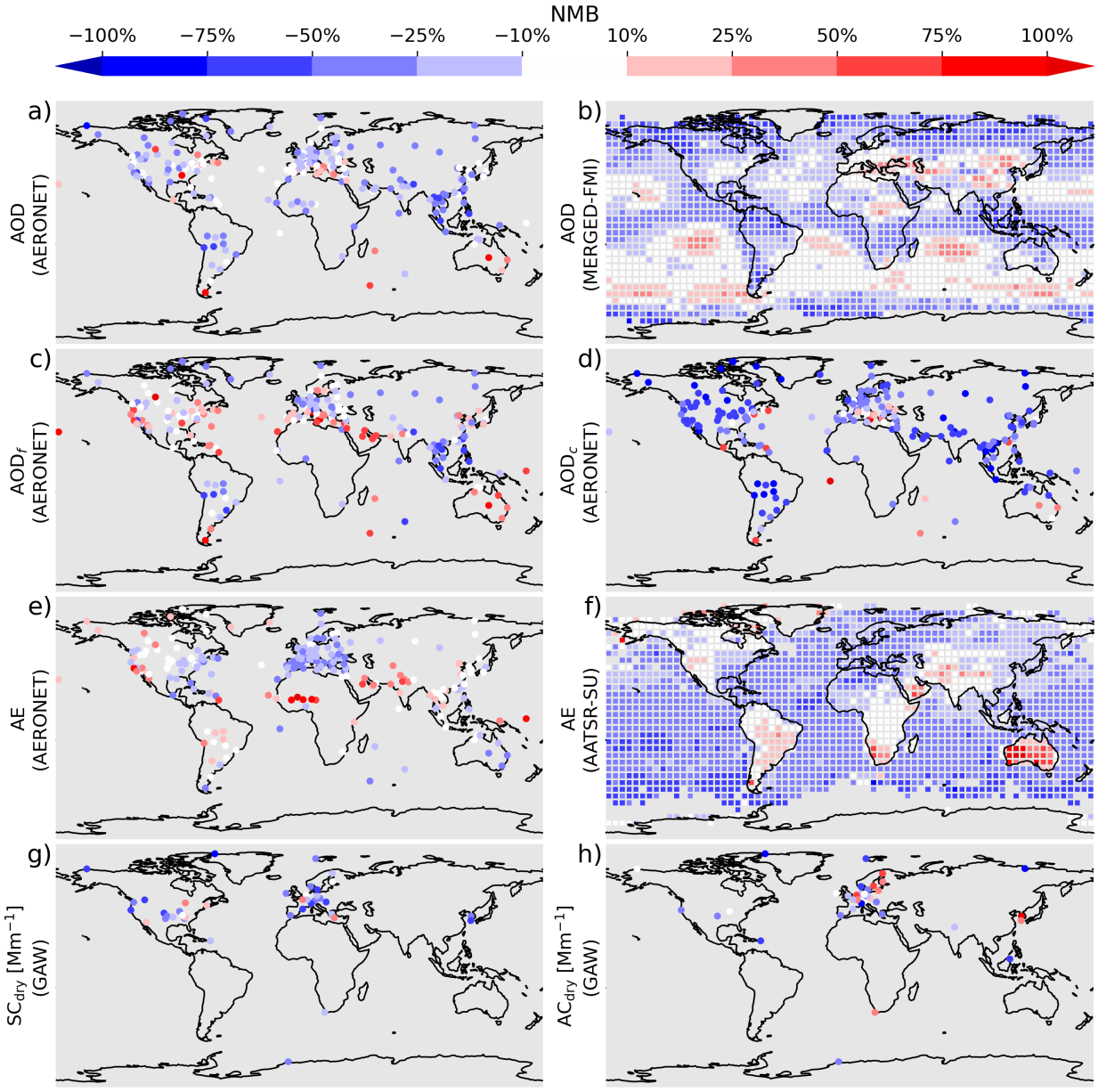


Fig. 1: Bias maps of the AeroCom ensemble median compared to several observation records (Figure 5 from Gliß et al., 2021, processed with pyaerocom)

CHAPTER

ONE

ABOUT

pyaerocom is written and tested for python >= 3.6 and for unix based systems. It provides tools for processing and plotting of data related to the AeroCom project.

This includes support for reading and processing of modeldata (e.g. AeroCom, EMEP), satellite data (e.g. MODIS, AATSR) and ground based observation datasets (e.g. AERONET, EBAS, EARLINET). In addition, pyaerocom provides tools for colocation and cross evaluation of different datasets using commonly used statistical metrics such as several biases, gross-errors, or correlation coefficients.

1.1 AeroCom

The AeroCom-project (http://aerocom.met.no/) is an open international initiative of scientists interested in the advancement of the understanding of the global aerosol and its impact on climate. A large number of observations (including MODIS, POLDER, MISR, AVHRR, SEAWIFS, TOMS, AATSR, AERONET and surface concentrations) and results from more than 14 global models have been assembled to document and compare state of the art modeling of the global aerosol. A common protocol has been established and models are asked to make use of the AeroCom emission inventories for the year 2000 and preindustrial times. Results are documented via interactive websites which give access to 2D fields and standard comparisons to observations. Regular workshops are held to discuss findings and future directions.

This repository contains the AeroCom python tools which are / will be used to produce the standard AeroCom analyses shown at the AeroCom phase 2 interface (http://aerocom.met.no/cgi-bin/AeroCom/aerocom/surfobs_annualrs.pl)

At this point the tools are co-operational together with the IDL based aerocom-tools that cannot be made public because they use 3rd party libraries with a non GPL compatible license.

1.2 Main features

- Reading routines for many ground based observation databases, such as:
  - AERONET Sun, SDA and Inversion products.
  - EBAS database.
  - EEA Air Quality e-Reporting (AQ e-Reporting).
  - AirNow.
  - **`Ministry of Environment Protection (MEP) in China`__.**
  - *GHOST* (Globally Harmonised Observational Surface Treatment) (see e.g., Petetin et al., 2020 for more information).
- Reading routines for level 3 gridded satellite observations, such as:

  - MODIS Aerosol Product.
  - CALIPSO CALIOP Lidar observations.
  - ENVISAT AATSR.
- Data harmonization tools following the CF conventions.
- Intuitive data objects for analysis of gridded data and ungridded (point-cloud) observations.
- Sophisticated and flexible colocation routines for model evaluation and intercomparison of observations.
- Interfaces for conversion of data to data types of related data analysis libraries such as pandas, numpy, xarray or iris.
- Data visualization tools and interfaces to common plotting libraries such as matplotlib or cartopy.
- Tools for statistical analysis of model performance.
- Toolbox for analysis of trends in time-series.
- Tools to compute ensemble averages from multiple model outputs.
- High-level tools for automated analyses of multi-model and multi-obs inter-comparison studies.

1.3 Usage examples

- Processing of data for the new AeroCom Model Evaluation interface.
- Processing and harmonization of observations for Aerosol Trends interface.
- pyaerocom was used for the model evaluation study by Gliß et al., 2020.
- pyaerocom was used for the trends analysis by Mortier et al., 2020.

1.4 Access to AeroCom users database

The AeroCom users database contains model diagnostics from all AeroCom phases, ready for analysis.

If you wish to get access to the database, please follow the instructions provided in the following link:

https://wiki.met.no/aerocom/data_retrieval

NOTE: the users database does not contain any ground based observational data (such as EBAS, AERONET, etc.) but only the AeroCom model data available in the database as well as some gridded level 3 satellite data which may be used for model evaluation. Once you have access to the user database you may mount the file-system locally (e.g. via *sshfs* and register the data-paths you need in pyaerocom, for details see tutorials, more info below).

1.5 Remark for Windows users

pyaerocom is not tested on Windows systems and may only work in parts and thus some features may not work on Windows machines at the moment. In particular, features that rely on and are built upon access to the AeroCom database servers and automatic database path navigation. This includes the automised reading of gridded and ungridded data using either of the pre-defined path infrastuctures (e.g. check out paths.ini or paths_user_server.ini). However, you may still define file locations in your Python scripts yourself and use the more low-level features for reading the data. Windows support will be provided soon. Please let us know if you intend to use pyaerocom on a Windows machine so that we can consider adjusting our priorities, or also if you have any questions related to the usage.

CHAPTER

TWO

INSTALLATION

You can install PyAerocom via pip

2.1 Install from source into a new virtual environment

Installation into a new virtual environment (recommended for machines with newer python version and binary libraries) named *.venv* via:

```
# create and activate new virtual environment
python3 -m venv --prompt pya .venv
source .venv/bin/activate

# update pip
python3 -m pip install -U pip

# install pyaerocom on machines with Proj8 or newer
# e.g. Ubuntu 22.04 LTS (Jammy Jellyfish)
pip install pyaerocom
```

2.2 Install from source into a conda environment

If you use the *conda* package manager, please make sure to activate the environment you want to install pyaerocom into. For more information about conda environments, see here.

Please make sure to install all requirements (see below) before installing pyaerocom from source. You can do that with the provided file pyaerocom_env.yml.

To install pyaerocom from source, please download and extract the latest release (or clone the repo) and install from the top-level directory (that contains a file *pyproject.toml*) using:

```
pip install --no-deps .
```

The *–no-deps* option will ensure that only the pyearocom package is installed, preserving the conda environment.

Alternatively, if you plan to apply local changes to the pyaerocom source code, you may install in editable mode (i.e. setuptools "develop mode") including the test dependencies:

```
pip install --no-deps -e .
```

You may also download and extract (or clone) the GitHub repo to install the very latest (not yet released) version of pyaerocom. Note, if you install in develop mode, make sure you do not have pyaerocom installed already in the site packages directory, check e.g. **`conda list pyaerocom`__** .

2.3 Install from source into a default environment

If you want PyAerocom in your default installation of python, then you install the latest released version of pyaerocom and its depencencies:

```
# install pyaerocom on machines with Proj8 or newer
# e.g. Ubuntu 22.04 LTS (Jammy Jellyfish)
python3 -m pip install pyaerocom
```

This type of installation is no longer allowed on newer OS-installations, i.e. Ubuntu 24.04. Use the installation into a new virtual environment instead.

CHAPTER

THREE

TUTORIALS

pyaerocom tutorials and examples. All content displayed here is based on the official pyaerocom tutorials which are not part of the pyaerocom GitHub repository but are available through the pyaerocom-tutorials repo.

All tutorials are jupyter notebooks.

Note: The tutorials are currently undergoing revision, as the initial set of tutorial notebooks cannot be interactively executed from the outside world, since they relied on access to internal data servers of the Norwegian Meteorological Institute.

These *old* tutorials can be found below in Section "Outdated tutorials". They are still mostly valid and provide a comprehensive introduction into pyaerocom and its main features.

All other tutorials, that are shown in the following rely on publicly available example data, and can thus, be run interactively.

3.1 Getting started

3.1.1 Checking pyaerocom installation and access to data

Import pyaerocom

It all begins with an import:

[1]:
```
import pyaerocom as pya
pya.__version__
```

```
/home/jonasg/miniconda3/envs/pyadev/lib/python3.9/site-packages/geonum/__init__.py:26:␣
↪UserWarning: Plotting of maps etc. is deactivated, please install Basemap
  warn('Plotting of maps etc. is deactivated, please install Basemap')
```

[1]:
```
'0.12.0.dev1'
```

When imported, pyaerocom will automatically check access to different default data locations (e.g. mount to PPI at MET Norway, or ~/MyPyaerocom/data), and in case a data source location is detected, associated data directories for accessing these data are instantiated (details below).

Check available datasets and directories

Accessible data and default paths for certain datasets are available (and can be updated via the `const` module).

[2]:
```
pya.const
```

[2]:
```
<pyaerocom.config.Config at 0x7f3a0f5932b0>
```

Data search directories

[3]:
```
pya.const.DATA_SEARCH_DIRS
```

[3]:
```
['/home/jonasg/MyPyaerocom/data/modeldata/',
 '/home/jonasg/MyPyaerocom/data/obsdata/']
```

This list contains all directories where pyaerocom will search for model and observation data. pyaerocom will search for both model and observation data in all directories that are specified here. Searching can be done using and if nothing can be found for a certain query, an Exception is raised. Let's try to find some data from the TM5 chemistry-transport-model:

[4]:
```
try:
    pya.browse_database('*TM5*')
except pya.exceptions.DataSearchError as e:
    print(e)
```

```
No matches could be found for search pattern *TM5*
```

Well, that's expected as no data search directories are specified (and most likely, no TM5 data is available on whatever which machine this notebook is executed). Let's make this a little more interesting. We need some data!

Downloading the pyaerocom testdata-minimal dataset

The testdata-minimal dataset was developed for automatic testing of pyaerocom and is well suited to illustrate the main features of pyaerocom, without too requiring too heavy computing resources or data storage. It is very easy to get these data:

[5]:
```
from pyaerocom.testdata_access import TestDataAccess
TestDataAccess().download()
```

```
Downloading pyaerocom testdata into /home/jonasg/MyPyaerocom
```

[5]:
```
True
```

Now we have a path, where there is supposed to be some data. Awesomeness!

[6]:
```
dataloc = f'{pya.const.HOMEDIR}MyPyaerocom/testdata-minimal/'
dataloc
```

[6]:
```
'/home/jonasg/MyPyaerocom/testdata-minimal/'
```

Side comment: If this way of formatting python strings looks weird to you, don't worry, this is because it is a rather new feature (as of Nov 2020).

[7]:
```
import os
os.listdir(dataloc)
```

```
[7]: ['README.md', 'scripts', 'coldata', 'obsdata', 'modeldata', 'config']
```

Let's look into the modeldata directory (obsdata follows later).

```
[8]: os.listdir(dataloc + 'modeldata')
[8]: ['EMEP_2017', 'TM5-met2010_CTRL-TEST']
```

Adding data search directories

Great, found something. Let's tell pyaerocom about it.

```
[9]: pya.const.add_data_search_dir(dataloc + 'modeldata')
```

Now, let's repeat what we did before.

```
[10]: pya.const.DATA_SEARCH_DIRS
[10]: ['/home/jonasg/MyPyaerocom/data/modeldata/',
       '/home/jonasg/MyPyaerocom/data/obsdata/',
       '/home/jonasg/MyPyaerocom/testdata-minimal/modeldata']
```

```
[11]: pya.browse_database('*TM5*')

Pyaerocom ReadGridded
---------------------
Data ID: TM5-met2010_CTRL-TEST
Data directory: /home/jonasg/MyPyaerocom/testdata-minimal/modeldata/TM5-met2010_CTRL-
↪TEST/renamed
Available experiments: ['AP3']
Available years: [2010, 9999]
Available frequencies ['daily' 'monthly']
Available variables: ['abs550aer', 'od550aer']
[11]: ['TM5-met2010_CTRL-TEST']
```

Nice! This worked, and there is even a lot of additional information, that comes in handy. The latter is because the underlying NetCDF files in the data directory are stored using AeroCom file naming conventions. Each dataset has it's own ID (usually the directory name and can be accessed via this ID). For this example TM5 dataset the ID is *TM5-met2010_CTRL-TEST* as can be seen in the output from the browsing method.

pyaerocom makes extensive use og these conventions, which makes it easy to streamline analyses of many different models and observation records. However, as we shall see below, the latter are often formatted in many different ways, as observations from many different databases are used.

```
[12]: reader = pya.io.ReadGridded('TM5-met2010_CTRL-TEST')
      reader
[12]:
Pyaerocom ReadGridded
---------------------
Data ID: TM5-met2010_CTRL-TEST
Data directory: /home/jonasg/MyPyaerocom/testdata-minimal/modeldata/TM5-met2010_CTRL-
↪TEST/renamed
```

(continues on next page)

(continued from previous page)

```
Available experiments: ['AP3']
Available years: [2010, 9999]
Available frequencies ['daily' 'monthly']
Available variables: ['abs550aer', 'od550aer']
```

Tiny detour: AeroCom file naming conventions

Let's have a brief look at such a filename (taking the first file in the data directory):

[13]:
```
first_file = reader.files[0]
os.path.basename(first_file)
```

[13]:
```
'aerocom3_TM5-met2010_AP3-CTRL2019_abs550aer_Column_2010_daily.nc'
```

The template is:

aerocom3_<ModelName>-<MeteoConfigSpecifier>_<ExperimentName>-<PerturbationName>_<VariableName>_<Vertical
nc

So the above filename uses **TM5** model, 2010 meteorology (**met2010**), and this version is for AeroCom Phase III (**AP3**) experiment, particularly for the 2019 Control (**CTRL2019**) perturbation. Variable is **abs550aer** (which is the aerosol absorption optical depth, or AAOD), which is representative for a whole atmospheric **Column**, the simulated year is **2010** (here it is the same as meteorology, but this must not always be the case) and the temporal resolution is **daily**.

If you want to learn more about AeroCom conventions and ongoing experiments, see here.

The metadata that is extracted from the filenames is accessible via:

[14]:
```
reader.file_info
```

[14]:
```
    var_name  year  ts_type vert_code                data_id name    meteo  \
0  abs550aer  2010    daily    Column  TM5-met2010_CTRL-TEST  TM5  met2010
4  abs550aer  2010  monthly    Column  TM5-met2010_CTRL-TEST  TM5  met2010
1  abs550aer  9999    daily    Column  TM5-met2010_CTRL-TEST  TM5  met2010
2   od550aer  2010    daily    Column  TM5-met2010_CTRL-TEST  TM5  met2010
3   od550aer  2010  monthly    Column  TM5-met2010_CTRL-TEST  TM5

  experiment perturbation  is_at_stations     3D  \
0        AP3     CTRL2019           False  False
4        AP3     CTRL2019           False  False
1        AP3     CTRL2019           False  False
2        AP3     CTRL2019           False  False
3        AP3     CTRL2016           False  False

                                       filename
0  aerocom3_TM5-met2010_AP3-CTRL2019_abs550aer_Co...
4  aerocom3_TM5-met2010_AP3-CTRL2019_abs550aer_Co...
1  aerocom3_TM5-met2010_AP3-CTRL2019_abs550aer_Co...
2  aerocom3_TM5-met2010_AP3-CTRL2019_od550aer_Col...
3  aerocom3_TM5_AP3-CTRL2016_od550aer_Column_2010...
```

How do I know what the variable names mean?

You can check all variables via `pyaerocom.const.VARS`, which is a dictionary-like object that allows access to variables and in most cases, provides relevant additional information such as the CF standard_name. For instance, for the above `abs550aer`:

[15]:
```
var = pya.const.VARS['abs550aer']
var
```

[15]:
```
abs550aer
standard_name: atmosphere_absorption_optical_thickness_due_to_ambient_aerosol_particles;␣
↪Unit: 1
```

[16]:
```
var.long_name
```

[16]:
```
'Absorption aerosol optical depth (AAOD) at 550nm'
```

Or the extinction (scattering + absorption) aerosol optical depth (AOD), called `od550aer`:

[17]:
```
var = pya.const.VARS['od550aer']
var
```

[17]:
```
od550aer
standard_name: atmosphere_optical_thickness_due_to_ambient_aerosol_particles; Unit: 1
```

Reading of model data using `ReadGridded` class

The above instantiated `ReadGridded` interface relies on and makes use of these conventions. This class is also the standard interface to read the modeldata into instances of the `pyaerocom.GriddedData`

[18]:
```
aaod_tm5 = reader.read_var('abs550aer', start=2010, ts_type='monthly')
aaod_tm5
```

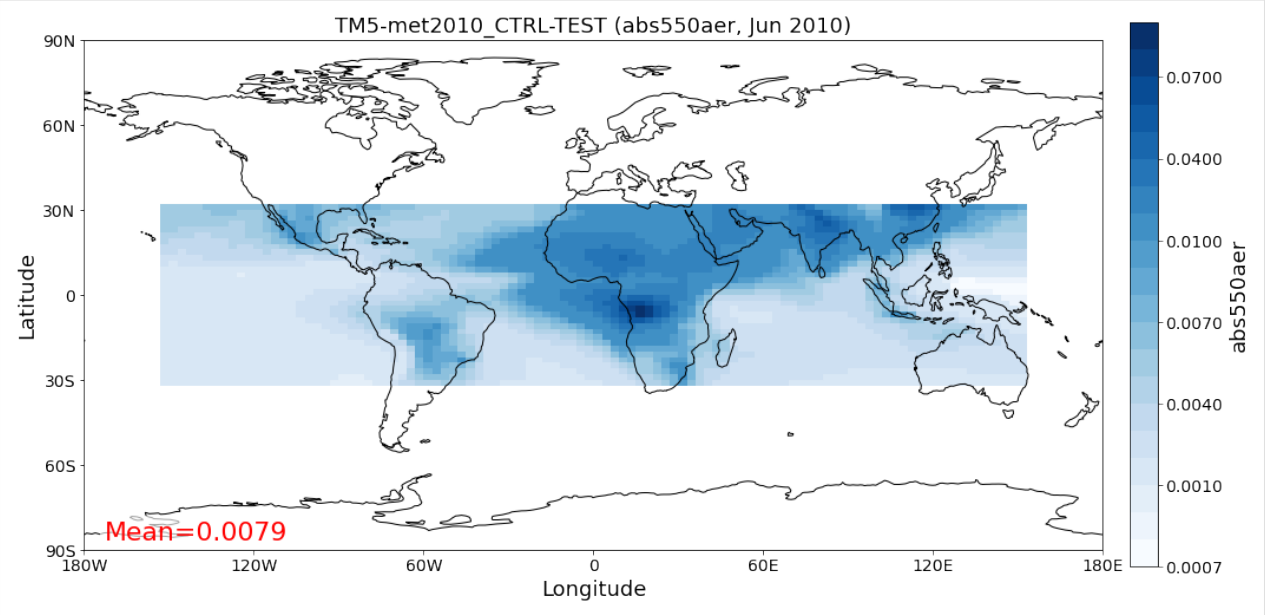
[18]:
```
pyaerocom.GriddedData: (abs550aer, TM5-met2010_CTRL-TEST)
<iris 'Cube' of atmosphere_absorption_optical_thickness_due_to_ambient_aerosol / (1)␣
↪(time: 12; latitude: 90; longitude: 120)>
```

Under the hood, the `GriddedData` object is an iris.Cube, and it is **single variable**, that is, it does not support reading of multiple variable fields (e.g. AOD and AAOD sharing the same lat, lon and time dimensions).

The `GriddedData` object is introduced in more detail in other tutorials, but what is a tutorial without a nice, self-explanatory plot anyways?

[19]:
```
aaod_tm5.sel(latitude=(-30, 30), longitude=(-150, 150)).quickplot_map('06/2010');
```

```
/home/jonasg/github/pya/pyaerocom/pyaerocom/plot/mapping.py:438:␣
↪MatplotlibDeprecationWarning: The 'cmap' parameter to Colorbar has no effect because␣
↪it is overridden by the mappable; it is deprecated since 3.3 and will be removed two␣
↪minor releases later.
  cbar = fig.colorbar(disp, cmap=cmap, norm=norm, #boundaries=bounds,
/home/jonasg/github/pya/pyaerocom/pyaerocom/plot/mapping.py:438:␣
↪MatplotlibDeprecationWarning: The 'norm' parameter to Colorbar has no effect because␣
↪it is overridden by the mappable; it is deprecated since 3.3 and will be removed two␣
↪minor releases later.
  cbar = fig.colorbar(disp, cmap=cmap, norm=norm, #boundaries=bounds,
```



Registering and reading of *ungridded* observational data

. . . COMING VERY SOON!!

Until then, checkout the section on ungridded observations in the following tutorial getting_started_analysis.

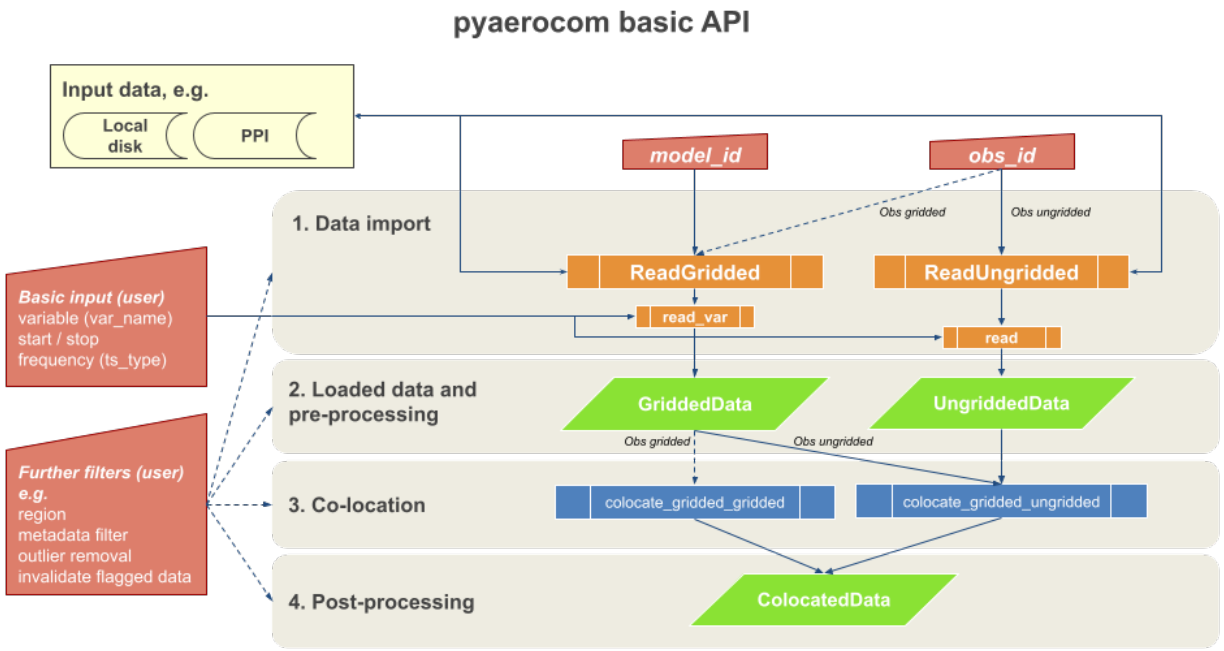
3.1.2 Diving deeper into the analysis and API

This notebook is meant to give a quick introduction into the main features and workflows of pyaerocom.

This includes brief introductions into the following features:

What?	Relevant classes and/or methods
Finding model and observation data	`pya.browse_database <https://pyaerocom.met.no/api.html#pyaerocom.io.utils.browse_database>`__
Reading of **gridded** model data	`pya.io.ReadGridded <https://pyaerocom.met.no/api.html#pyaerocom.io.readgridded.ReadGridded>`__
Working with **gridded** data	`pya.GriddedData <https://pyaerocom.met.no/api.html#pyaerocom.griddeddata.GriddedData>`__
Reading of **ungridded** observation data	`pya.io.ReadUngridded <https://pyaerocom.met.no/api.html#reading-of-ungridded-data>`__
Working with **ungridded** data	`pya.UngriddedData <https://pyaerocom.met.no/api.html#pyaerocom.ungriddeddata.UngriddedData>`__
Working with data from individual site locations	`pya.StationData <https://pyaerocom.met.no/api.html#pyaerocom.stationdata.StationData>`__
Colocation of model and observational data	High-level: `pya.colocation_auto <https://pyaerocom.met.no/api.html#module-pyaerocom.colocation_auto>`__
	Low-level: `pya.colocation <https://pyaerocom.met.no/api.html#module-pyaerocom.colocation>`__
Working with **colocated** data	`pya.ColocatedData <https://pyaerocom.met.no/api.html#pyaerocom.colocateddata.ColocatedData>`__

In a graphical way it introduces the main data object and processing routines for model and observation comparisons with pyaerocom, illustrated in the following flowchart:



Only that in this example "Data server" is the local computer that has the minimal testdataset as an example dataset.

[1]:
```
import pyaerocom as pya
pya.__version__
```

```
/home/jonasg/miniconda3/envs/pyadev/lib/python3.9/site-packages/geonum/__init__.py:26:␣
↪UserWarning: Plotting of maps etc. is deactivated, please install Basemap
```

(continues on next page)

(continued from previous page)

```
  warn('Plotting of maps etc. is deactivated, please install Basemap')
```

[1]:
```
'0.12.0.dev1'
```

Should be at least 0.10.X

Check access to testdata

NOTE: details regarding testdata access and intialization are covered in tutorial notebook *getting_started_setup.ipynb*.

[2]:
```
from pyaerocom.testdata_access import initialise
initialise()
```

```
Adding data search directory /home/jonasg/MyPyaerocom/testdata-minimal/modeldata.
Adding data search directory /home/jonasg/MyPyaerocom/testdata-minimal/obsdata.
Adding data search directory /home/jonasg/MyPyaerocom/testdata-minimal/config.
Adding ungridded dataset AeronetSunV3L2Subset.daily located at /home/jonasg/MyPyaerocom/
↪testdata-minimal/obsdata/AeronetSunV3Lev2.daily/renamed.Reader: <class 'pyaerocom.io.
↪read_aeronet_sunv3.ReadAeronetSunV3'>
Adding ungridded dataset AeronetSDAV3L2Subset.daily located at /home/jonasg/MyPyaerocom/
↪testdata-minimal/obsdata/AeronetSDAV3Lev2.daily/renamed.Reader: <class 'pyaerocom.io.
↪read_aeronet_sdav3.ReadAeronetSdaV3'>
Adding ungridded dataset AeronetInvV3L2Subset.daily located at /home/jonasg/MyPyaerocom/
↪testdata-minimal/obsdata/AeronetInvV3Lev2.daily/renamed.Reader: <class 'pyaerocom.io.
↪read_aeronet_invv3.ReadAeronetInvV3'>
Adding ungridded dataset EBASSubset located at /home/jonasg/MyPyaerocom/testdata-minimal/
↪obsdata/EBASMultiColumn.Reader: <class 'pyaerocom.io.read_ebas.ReadEbas'>
Adding ungridded dataset AirNowSubset located at /home/jonasg/MyPyaerocom/testdata-
↪minimal/obsdata/AirNowSubset.Reader: <class 'pyaerocom.io.read_airnow.ReadAirNow'>
Adding ungridded dataset G.EEA.daily.Subset located at /home/jonasg/MyPyaerocom/testdata-
↪minimal/obsdata/GHOST/data/EEA_AQ_eReporting/daily.Reader: <class 'pyaerocom.io.read_
↪ghost.ReadGhost'>
Adding ungridded dataset G.EEA.hourly.Subset located at /home/jonasg/MyPyaerocom/
↪testdata-minimal/obsdata/GHOST/data/EEA_AQ_eReporting/hourly.Reader: <class 'pyaerocom.
↪io.read_ghost.ReadGhost'>
Adding ungridded dataset G.EBAS.daily.Subset located at /home/jonasg/MyPyaerocom/
↪testdata-minimal/obsdata/GHOST/data/EBAS/daily.Reader: <class 'pyaerocom.io.read_ghost.
↪ReadGhost'>
Adding ungridded dataset G.EBAS.hourly.Subset located at /home/jonasg/MyPyaerocom/
↪testdata-minimal/obsdata/GHOST/data/EBAS/hourly.Reader: <class 'pyaerocom.io.read_
↪ghost.ReadGhost'>
pyaerocom-testdata is ready to be used. The data is available at /home/jonasg/
↪MyPyaerocom/testdata-minimal
```

Model data: Reading of and working with *gridded* data

This section provides an introduction into the following pyaerocom classes and architectures:

- `pyaerocom.io.ReadGridded <https://pyaerocom.met.no/api.html#module-pyaerocom.io.readgridded>`__
- `pyaerocom.GriddedData <https://pyaerocom.met.no/api.html#module-pyaerocom.griddeddata>`__

*you may click the links to see the online documentation of these classes.

Pre-remark on the `ReadGridded` class

As you could see in tutorial getting_started_setup.ipynb the `ReadGridded` class makes extensive use of the AeroCom file naming conventions. So if you have model data that is stored using different conventions (e.g. CMIP6), this class will not be of much help (yet) for filtering the correct files to read. In that case you may locate a model NetCDF file yourself and pass it directly into a GriddedData object on initialisation.

The testdataset contains data from the TM5 model, which is used in the following. You can use the `browse_database` function of pyaerocom to find model ID's (which can be quite cryptic sometimes) using wildcard pattern search.

[3]:
```
pya.browse_database('*TM5*')
```

```
Pyaerocom ReadGridded
---------------------
Data ID: TM5JRCCY2IPCCV1_SR6SA
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/HTAP-PHASE-I/
↪TM5JRCCY2IPCCV1_SR6SA/renamed
Available experiments: ['SR6SA']
Available years: [2001]
Available frequencies ['monthly']
Available variables: ['MMR_BCSR6SA', 'MMR_NO3SR6SA', 'MMR_POMSR6SA', 'MMR_SO4SR6SA']

Pyaerocom ReadGridded
---------------------
Data ID: TM5JRCCY2IPCCV1_SR6NA
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/HTAP-PHASE-I/
↪TM5JRCCY2IPCCV1_SR6NA/renamed
Available experiments: ['SR6NA']
Available years: [2001]
Available frequencies ['monthly']
Available variables: ['MMR_BCSR6NA', 'MMR_NO3SR6NA', 'MMR_POMSR6NA', 'MMR_SO4SR6NA']

Pyaerocom ReadGridded
---------------------
Data ID: TM5-JRC-cy2-ipcc-v1_SR1
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/HTAP-PHASE-I/TM5-
↪JRC-cy2-ipcc-v1_SR1/renamed
Available experiments: ['SR1']
Available years: [2001]
Available frequencies ['monthly']
Available variables: ['vmro3']

Pyaerocom ReadGridded
---------------------
```

(continues on next page)

(continued from previous page)

```
Data ID: TM5JRCCY2IPCCV1_SR6EU
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/HTAP-PHASE-I/
↪TM5JRCCY2IPCCV1_SR6EU/renamed
Available experiments: ['SR6EU']
Available years: [2001]
Available frequencies ['monthly']
Available variables: ['MMR_BCSR6EU', 'MMR_NO3SR6EU', 'MMR_POMSR6EU', 'MMR_SO4SR6EU']

Pyaerocom ReadGridded
---------------------
Data ID: TM5JRCCY2IPCCV1_SR6EA
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/HTAP-PHASE-I/
↪TM5JRCCY2IPCCV1_SR6EA/renamed
Available experiments: ['SR6EA']
Available years: [2001]
Available frequencies ['monthly']
Available variables: ['MMR_BCSR6EA', 'MMR_NO3SR6EA', 'MMR_POMSR6EA', 'MMR_SO4SR6EA']

Pyaerocom ReadGridded
---------------------
Data ID: TM5JRCCY2IPCCV1_SR1
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/HTAP-PHASE-I/
↪TM5JRCCY2IPCCV1_SR1/renamed
Available experiments: ['SR1']
Available years: [2001]
Available frequencies ['monthly']
Available variables: ['SCONCBC', 'SCONCNO3', 'SCONCPM25', 'SCONCPOM', 'SCONCSO4']
```

```
FileNotFoundError('None of the available files in /lustre/storeA/project/aerocom/aerocom-
↪users-database/AEROCOM-PHASE-I/TM5_B/renamed matches a registered pyaerocom file␣
↪convention')
FileNotFoundError('None of the available files in /lustre/storeA/project/aerocom/aerocom-
↪users-database/AEROCOM-PHASE-I/TM5_B/renamed matches a registered pyaerocom file␣
↪convention')
```

```
Reading failed for TM5_B. Error: AttributeError("'NoneType' object has no attribute
↪'experiment'")

Pyaerocom ReadGridded
---------------------
Data ID: TM5-V3.A2.HCA-0
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-II/
↪TM5-V3.A2.HCA-0/renamed
Available experiments: ['']
Available years: [2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009]
Available frequencies ['daily' 'monthly']
Available variables: ['abs550aer', 'abs550dryaer', 'airmass', 'asyaer', 'drydms',
↪'drydust', 'dryso2', 'dryso4', 'dryss', 'ec550aer', 'ec550dryaer', 'emibc', 'emidms',
↪'emidust', 'emioa', 'emiso2', 'emiso4', 'emiss', 'hus', 'loadbc', 'loaddust', 'loadno3
↪', 'loadoa', 'loadso4', 'loadss', 'od440aer', 'od550aer', 'od550aerh2o', 'od550bc',
↪'od550dust', 'od550lt1aer', 'od550lt1dust', 'od550no3', 'od550oa', 'od550so4', 'od550ss
↪', 'od870aer', 'pmid3d', 'precip', 'pressure', 'ps', 'scatc550dryaer', 'sconcbc',
↪'sconcdust', 'sconcno3', 'sconcoa', 'sconcso4', 'sconcss', 'temp', 'wetbc', 'wetdms',
```

(continues on next page)

(continued from previous page)

```
↪'wetdust', 'wetoa', 'wetso2', 'wetso4', 'wetss', 'ang4487aer', 'od550gt1aer',
↪'fmf550aer', 'pmid']

Pyaerocom ReadGridded
---------------------
Data ID: TM5-V3.A2.HCA-IPCC
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-II/
↪TM5-V3.A2.HCA-IPCC/renamed
Available experiments: ['']
Available years: [2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009]
Available frequencies ['daily' 'monthly' 'hourly']
Available variables: ['abs550aer', 'abs550dry1Daer', 'abs550dryaer', 'airmass', 'asyaer',
↪ 'clt', 'conccn1Dmode01', 'conccn1Dmode02', 'conccn1Dmode03', 'conccn1Dmode04',
↪'conccn1Dmode05', 'conccn1Dmode06', 'conccn1Dmode07', 'conccnmode01', 'conccnmode02',
↪'conccnmode03', 'conccnmode04', 'conccnmode05', 'conccnmode06', 'conccnmode07', 'drybc
↪', 'drydust', 'dryhno3', 'drynh3', 'dryno2', 'drynoy', 'dryoa', 'dryso2', 'dryso4',
↪'dryss', 'ec550aer', 'ec550dry1Daer', 'ec550dryaer', 'emibc', 'emidms', 'emidust',
↪'eminh3', 'eminox', 'emioa', 'emiso2', 'emiso4', 'emiss', 'hus', 'loadbc', 'loaddust',
↪'loadno3', 'loadoa', 'loadso4', 'loadss', 'mmr1Daerh2o', 'mmr1Dtr01', 'mmr1Dtr02',
↪'mmr1Dtr03', 'mmr1Dtr04', 'mmr1Dtr05', 'mmr1Dtr06', 'mmr1Dtr07', 'mmr1Dtr08',
↪'mmr1Dtr09', 'mmr1Dtr10', 'mmr1Dtr11', 'mmr1Dtr12', 'mmr1Dtr13', 'mmr1Dtr14',
↪'mmr1Dtr15', 'mmr1Dtr16', 'mmr1Dtr17', 'mmr1Dtr18', 'mmr1Dtr19', 'mmraerh2o', 'mmrbc',
↪'mmrdu', 'mmrno3', 'mmroa', 'mmrso4', 'mmrss', 'mmrtr01', 'mmrtr02', 'mmrtr03',
↪'mmrtr04', 'mmrtr05', 'mmrtr06', 'mmrtr07', 'mmrtr08', 'mmrtr09', 'mmrtr10', 'mmrtr11',
↪ 'mmrtr12', 'mmrtr13', 'mmrtr14', 'mmrtr15', 'mmrtr16', 'mmrtr17', 'mmrtr18', 'mmrtr19
↪', 'od440aer', 'od550aer', 'od550aerh2o', 'od550bc', 'od550dust', 'od550lt1aer',
↪'od550lt1dust', 'od550no3', 'od550oa', 'od550so4', 'od550ss', 'od870aer', 'pmid3d',
↪'precip', 'pressure', 'ps', 'rsds', 'rsdscs', 'rsdscsdif', 'rsdscsvis', 'rsdt', 'rsus',
↪ 'rsut', 'rsutcs', 'sconcbc', 'sconcdust', 'sconcmsa', 'sconcno3', 'sconcoa', 'sconcso4
↪', 'sconcss', 'temp', 'vmrdms', 'vmrhno3', 'vmrno', 'vmrno2', 'vmrpan', 'vmrso2',
↪'wet3Dbc', 'wet3Ddu', 'wet3Dhno3', 'wet3Dnh4', 'wet3Dnoy', 'wet3Doa', 'wet3Dso2',
↪'wet3Dso4', 'wet3Dss', 'wetbc', 'wetdust', 'wethno3', 'wetnh4', 'wetnoy', 'wetoa',
↪'wetso2', 'wetso4', 'wetss', 'ang4487aer', 'od550gt1aer', 'fmf550aer', 'pmid', 'wetdu']

Pyaerocom ReadGridded
---------------------
Data ID: TM5-V3.A2.CTRL
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-II/
↪TM5-V3.A2.CTRL/renamed
Available experiments: ['']
Available years: [2006]
Available frequencies ['daily' 'monthly' 'hourly']
Available variables: ['abs550aer', 'abs550dry1Daer', 'abs550dryaer', 'airmass',
↪'ang4487aer', 'asyaer', 'conccn1Dmode01', 'conccn1Dmode02', 'conccn1Dmode03',
↪'conccn1Dmode04', 'conccn1Dmode05', 'conccn1Dmode06', 'conccn1Dmode07', 'conccnmode01',
↪ 'conccnmode02', 'conccnmode03', 'conccnmode04', 'conccnmode05', 'conccnmode06',
↪'conccnmode07', 'drybc', 'drydust', 'dryhno3', 'drynh3', 'dryno2', 'drynoy', 'dryoa',
↪'dryso2', 'dryso4', 'dryss', 'ec550aer', 'ec550dry1Daer', 'ec550dryaer', 'emibc',
↪'emidms', 'emidust', 'eminh3', 'eminox', 'emioa', 'emiso2', 'emiso4', 'emiss', 'hus',
↪'loadbc', 'loaddust', 'loadno3', 'loadoa', 'loadso4', 'loadss', 'mmr1Daerh2o',
↪'mmr1Dtr01', 'mmr1Dtr02', 'mmr1Dtr03', 'mmr1Dtr04', 'mmr1Dtr05', 'mmr1Dtr06',
↪'mmr1Dtr07', 'mmr1Dtr08', 'mmr1Dtr09', 'mmr1Dtr10', 'mmr1Dtr11', 'mmr1Dtr12',
```

(continues on next page)

(continued from previous page)

```
↪'mmr1Dtr13', 'mmr1Dtr14', 'mmr1Dtr15', 'mmr1Dtr16', 'mmr1Dtr17', 'mmr1Dtr18',
↪'mmr1Dtr19', 'mmraerh2o', 'mmrbc', 'mmrdu', 'mmrno3', 'mmroa', 'mmrso4', 'mmrss',
↪'mmrtr01', 'mmrtr02', 'mmrtr03', 'mmrtr04', 'mmrtr05', 'mmrtr06', 'mmrtr07', 'mmrtr08',
↪ 'mmrtr09', 'mmrtr10', 'mmrtr11', 'mmrtr12', 'mmrtr13', 'mmrtr14', 'mmrtr15', 'mmrtr16
↪', 'mmrtr17', 'mmrtr18', 'mmrtr19', 'od440aer', 'od550aer', 'od550aerh2o', 'od550bc',
↪'od550dust', 'od550lt1aer', 'od550lt1dust', 'od550no3', 'od550oa', 'od550so4', 'od550ss
↪', 'od870aer', 'pmid3d', 'precip', 'pressure', 'ps', 'sconcbc', 'sconcdust', 'sconcmsa
↪', 'sconcno3', 'sconcoa', 'sconcso4', 'sconcss', 'temp', 'vmrdms', 'vmrhno3', 'vmrno',
↪'vmrno2', 'vmrpan', 'vmrso2', 'wet3Dbc', 'wet3Ddu', 'wet3Dhno3', 'wet3Dnh4', 'wet3Dnoy
↪', 'wet3Doa', 'wet3Dso2', 'wet3Dso4', 'wet3Dss', 'wetbc', 'wetdust', 'wethno3', 'wetnh4
↪', 'wetnoy', 'wetoa', 'wetso2', 'wetso4', 'wetss', 'od550gt1aer', 'fmf550aer', 'pmid',
↪'wetdu']

Pyaerocom ReadGridded
---------------------
Data ID: TM5-V3.A2.PRE
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-II/
↪TM5-V3.A2.PRE/renamed
Available experiments: ['']
Available years: [1850]
Available frequencies ['daily' 'monthly']
Available variables: ['abs550aer', 'abs550dryaer', 'airmass', 'asyaer', 'clt', 'drybc',
↪'drydms', 'drydust', 'dryhno3', 'drynh3', 'dryno2', 'drynoy', 'dryoa', 'dryso2',
↪'dryso4', 'dryss', 'ec550aer', 'ec550dryaer', 'emibc', 'emidms', 'emidust', 'eminh3',
↪'eminox', 'emioa', 'emiso2', 'emiso4', 'emiss', 'hus', 'loadbc', 'loaddust', 'loadno3',
↪ 'loadoa', 'loadso4', 'loadss', 'od440aer', 'od550aer', 'od550aerh2o', 'od550bc',
↪'od550dust', 'od550lt1aer', 'od550lt1dust', 'od550no3', 'od550oa', 'od550so4', 'od550ss
↪', 'od870aer', 'precip', 'pressure', 'ps', 'rsds', 'rsdscs', 'rsdscsdif', 'rsdscsvis',
↪'rsdt', 'rsus', 'rsut', 'rsutcs', 'sconcbc', 'sconcdust', 'sconcmsa', 'sconcno3',
↪'sconcoa', 'sconcso4', 'sconcss', 'temp', 'vmrdms', 'vmrhno3', 'vmrno', 'vmrno2',
↪'vmrpan', 'vmrso2', 'wet3Dbc', 'wet3Ddu', 'wet3Dhno3', 'wet3Dnh4', 'wet3Dnoy', 'wet3Doa
↪', 'wet3Dso2', 'wet3Dso4', 'wet3Dss', 'wetbc', 'wetdms', 'wetdust', 'wethno3', 'wetnh4
↪', 'wetnoy', 'wetoa', 'wetso2', 'wetso4', 'wetss', 'ang4487aer', 'od550gt1aer',
↪'fmf550aer', 'wetdu']

Pyaerocom ReadGridded
---------------------
Data ID: TM5_AP3-INSITU
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-III/
↪TM5_AP3-INSITU/renamed
Available experiments: ['AP3']
Available years: [2010]
Available frequencies ['monthly' 'daily' 'hourly']
Available variables: ['abs350aer', 'abs440aer', 'abs440dryaer', 'abs550aer',
↪'abs550dryaer', 'abs550drylt1aer', 'abs870aer', 'abs870dryaer', 'airmass', 'asyaer',
↪'asydryaer', 'depbc', 'depdms', 'depdust', 'dephno3', 'depmsa', 'depn', 'depnh3',
↪'depnh4', 'depnhx', 'depno2', 'depno3', 'depnoy', 'depo3', 'depoa', 'deps', 'depso2',
↪'depso4', 'depss', 'dh', 'drybc', 'drydms', 'drydust', 'dryhno3', 'drynh3', 'dryno2',
↪'dryno3', 'drynoy', 'dryo3', 'dryoa', 'dryso2', 'dryso4', 'dryss', 'ec440dryaer',
↪'ec550aer', 'ec550dryaer', 'ec550drylt1aer', 'ec870dryaer', 'emibc', 'emico', 'emidms',
↪ 'emidust', 'emiisop', 'emin', 'eminh3', 'eminox', 'emioa', 'emis', 'emiso2', 'emiso4',
↪ 'emiss', 'emiterp', 'hus', 'loadbc', 'loaddust', 'loadno3', 'loadoa', 'loadso4',
```

(continues on next page)

(continued from previous page)

```
↪'loadss', 'mmrbc', 'mmrdust', 'mmrmsa', 'mmrnh4', 'mmrno3', 'mmroa', 'mmrso4', 'mmrss',
↪ 'od350aer', 'od440aer', 'od550aer', 'od550aerh2o', 'od550bc', 'od550dust',
↪'od550lt1aer', 'od550lt1dust', 'od550lt1ss', 'od550no3', 'od550oa', 'od550so4',
↪'od550ss', 'od870aer', 'plev', 'pr', 'precip', 'sconcbc', 'sconcdust', 'sconcmsa',
↪'sconcno3', 'sconcoa', 'sconcso4', 'sconcss', 'ta', 'temp', 'vmrch4', 'vmrco', 'vmrno',
↪ 'vmrno2', 'vmro3', 'vmroh', 'wetbc', 'wetdms', 'wetdust', 'wethno3', 'wetmsa', 'wetnh3
↪', 'wetnh4', 'wetno3', 'wetnoy', 'wetoa', 'wetso2', 'wetso4', 'wetss', 'ang4487aer',
↪'angabs4487aer', 'od550gt1aer', 'vmrox', 'fmf550aer']

Pyaerocom ReadGridded
---------------------
Data ID: TM5_AP3-CTRL2016
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-III/
↪TM5_AP3-CTRL2016/renamed
Available experiments: ['AP3']
Available years: [2006, 2008, 2010]
Available frequencies ['monthly' '3hourly']
Available variables: ['abs350aer', 'abs440aer', 'abs440dryaer', 'abs550aer',
↪'abs550dryaer', 'abs550drylt1aer', 'abs870aer', 'abs870dryaer', 'airmass', 'asyaer',
↪'asydryaer', 'deltaz3d', 'depbc', 'depdms', 'depdust', 'dephno3', 'depmsa', 'depn',
↪'depnh3', 'depnh4', 'depnhx', 'depno2', 'depno3', 'depnoy', 'depo3', 'depoa', 'deps',
↪'depso2', 'depso4', 'depss', 'dh', 'drybc', 'drydms', 'drydust', 'dryhno3', 'drynh3',
↪'dryno2', 'dryno3', 'drynoy', 'dryo3', 'dryoa', 'dryso2', 'dryso4', 'dryss',
↪'ec440dryaer', 'ec550aer', 'ec550dryaer', 'ec550drylt1aer', 'ec870dryaer', 'emibc',
↪'emico', 'emidms', 'emidust', 'emiisop', 'emin', 'eminh3', 'eminox', 'emioa', 'emis',
↪'emiso2', 'emiso4', 'emiss', 'emiterp', 'humidity3d', 'hus', 'loadbc', 'loaddust',
↪'loadno3', 'loadoa', 'loadso4', 'loadss', 'od350aer', 'od440aer', 'od550aer',
↪'od550aer3d', 'od550aerh2o', 'od550bc', 'od550dryaer', 'od550dust', 'od550lt1aer',
↪'od550lt1dust', 'od550lt1ss', 'od550no3', 'od550oa', 'od550so4', 'od550ss', 'od870aer',
↪ 'pr', 'sconcbc', 'sconcdust', 'sconcmsa', 'sconcnh4', 'sconcno3', 'sconcoa', 'sconcso4
↪', 'sconcss', 'ta', 'temp', 'vmrch4', 'vmrco', 'vmrno', 'vmrno2', 'vmro3', 'vmroh',
↪'wetbc', 'wetdms', 'wetdust', 'wethno3', 'wetmsa', 'wetnh3', 'wetnh4', 'wetno3',
↪'wetnoy', 'wetoa', 'wetso2', 'wetso4', 'wetss', 'ang4487aer', 'angabs4487aer',
↪'od550gt1aer', 'vmrox', 'fmf550aer', 'deltaz', 'humidity']

Pyaerocom ReadGridded
---------------------
Data ID: TM5_AP3-CTRL2015
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-III/
↪TM5_AP3-CTRL2015/renamed
Available experiments: ['AP3']
Available years: [2010]
Available frequencies ['monthly']
Available variables: ['depbc', 'depdust', 'depno3', 'depoa', 'depso4', 'depss', 'drybc',
↪'drydust', 'dryno3', 'dryoa', 'dryso4', 'dryss', 'emibc', 'emidms', 'emidust', 'eminox
↪', 'emioa', 'emiso2', 'emiso4', 'emiss', 'loadbc', 'loaddust', 'loadno3', 'loadoa',
↪'loadso4', 'loadss', 'od550aer', 'od550bc', 'od550dust', 'od550no3', 'od550oa',
↪'od550so4', 'od550ss', 'sconcbc', 'sconcdust', 'sconcno3', 'sconcoa', 'sconcso4',
↪'sconcss', 'wetbc', 'wetdust', 'wetno3', 'wetoa', 'wetso4', 'wetss']

Pyaerocom ReadGridded
---------------------
```

(continues on next page)

(continued from previous page)

```
Data ID: TM5_AP3-INSITU-TIER3
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-III/
↪TM5_AP3-INSITU-TIER3/renamed
Available experiments: ['AP3']
Available years: [2010]
Available frequencies ['hourly']
Available variables: ['abs440dryaer', 'abs550aer', 'abs550dryaer', 'abs550drylt1aer',
↪'abs550rh40aer', 'abs550rh55aer', 'abs550rh65aer', 'abs550rh75aer', 'abs550rh85aer',
↪'abs870dryaer', 'airmass', 'asydryaer', 'dh', 'ec440dryaer', 'ec550aer', 'ec550aerh2o',
↪ 'ec550bc', 'ec550dryaer', 'ec550drylt1aer', 'ec550dust', 'ec550no3', 'ec550oa',
↪'ec550rh40aer', 'ec550rh55aer', 'ec550rh65aer', 'ec550rh75aer', 'ec550rh85aer',
↪'ec550so4', 'ec550ss', 'ec870dryaer', 'hus', 'mmrbc', 'mmrdust', 'mmrmsa', 'mmrnh4',
↪'mmrno3', 'mmroa', 'mmrso4', 'mmrss', 'od550aer', 'od550aerh2o', 'od550bc', 'od550dust
↪', 'od550no3', 'od550oa', 'od550so4', 'od550ss', 'plev', 'pr', 'ta']

Pyaerocom ReadGridded
---------------------
Data ID: TM5-met2010_AP3-CTRL2019
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/AEROCOM-PHASE-III-
↪2019/TM5-met2010_AP3-CTRL2019/renamed
Available experiments: ['AP3']
Available years: [1850, 2010]
Available frequencies ['monthly' 'daily']
Available variables: ['abs350aer', 'abs440aer', 'abs440dryaer', 'abs550aer',
↪'abs550dryaer', 'abs550drylt1aer', 'abs870aer', 'abs870dryaer', 'airmass', 'asyaer',
↪'asydryaer', 'depbc', 'depdms', 'depdust', 'dephno3', 'depmsa', 'depn', 'depnh3',
↪'depnh4', 'depnhx', 'depno2', 'depno3', 'depnoy', 'depo3', 'depoa', 'deps', 'depso2',
↪'depso4', 'depsoa', 'depss', 'dh', 'drybc', 'drydms', 'drydust', 'dryhno3', 'drynh3',
↪'dryno2', 'dryno3', 'drynoy', 'dryo3', 'dryoa', 'dryso2', 'dryso4', 'drysoa', 'dryss',
↪'ec440dryaer', 'ec550aer', 'ec550dryaer', 'ec550drylt1aer', 'ec870dryaer', 'emibc',
↪'emico', 'emidms', 'emidust', 'emiisop', 'emin', 'eminh3', 'eminox', 'emis', 'emiso2',
↪'emiso4', 'emiss', 'emiterp', 'emivoc', 'hus', 'loadbc', 'loaddust', 'loadno3', 'loadoa
↪', 'loadso4', 'loadsoa', 'loadss', 'od350aer', 'od440aer', 'od550aer', 'od550aerh2o',
↪'od550bc', 'od550dust', 'od550lt1aer', 'od550lt1dust', 'od550lt1ss', 'od550no3',
↪'od550oa', 'od550so4', 'od550soa', 'od550ss', 'od870aer', 'pr', 'sconcbc', 'sconcdust',
↪ 'sconcmsa', 'sconcnh4', 'sconcno3', 'sconcoa', 'sconcso4', 'sconcsoa', 'sconcss', 'ta
↪', 'temp', 'vmrch4', 'vmrco', 'vmrno', 'vmrno2', 'vmro3', 'vmroh', 'wetbc', 'wetdms',
↪'wetdust', 'wethno3', 'wetmsa', 'wetnh3', 'wetnh4', 'wetno3', 'wetnoy', 'wetoa',
↪'wetso2', 'wetso4', 'wetsoa', 'wetss', 'ang4487aer', 'angabs4487aer', 'od550gt1aer',
↪'vmrox', 'fmf550aer']

Pyaerocom ReadGridded
---------------------
Data ID: TM5-met2010_CTRL-TEST
Data directory: /home/jonasg/MyPyaerocom/testdata-minimal/modeldata/TM5-met2010_CTRL-
↪TEST/renamed
Available experiments: ['AP3']
Available years: [2010, 9999]
Available frequencies ['daily' 'monthly']
Available variables: ['abs550aer', 'od550aer']
```

[3]:
```
['TM5JRCCY2IPCCV1_SR6SA',
```

(continues on next page)

(continued from previous page)

```
 'TM5JRCCY2IPCCV1_SR6NA',
 'TM5-JRC-cy2-ipcc-v1_SR1',
 'TM5JRCCY2IPCCV1_SR6EU',
 'TM5JRCCY2IPCCV1_SR6EA',
 'TM5JRCCY2IPCCV1_SR1',
 'TM5_B',
 'TM5-V3.A2.HCA-0',
 'TM5-V3.A2.HCA-IPCC',
 'TM5-V3.A2.CTRL',
 'TM5-V3.A2.PRE',
 'TM5_AP3-INSITU',
 'TM5_AP3-CTRL2016',
 'TM5_AP3-CTRL2015',
 'TM5_AP3-INSITU-TIER3',
 'TM5-met2010_AP3-CTRL2019',
 'TM5-met2010_CTRL-TEST']
```

[4]:
```
model_id = 'TM5-met2010_CTRL-TEST'
reader = pya.io.ReadGridded(model_id)
```

You can have a look at the individual files and corresponding metadata using the `file_info` attribute:

[5]:
```
reader.file_info
```

[5]:
```
    var_name  year  ts_type vert_code                data_id name    meteo  \
0  abs550aer  2010    daily    Column  TM5-met2010_CTRL-TEST  TM5  met2010
4  abs550aer  2010  monthly    Column  TM5-met2010_CTRL-TEST  TM5  met2010
1  abs550aer  9999    daily    Column  TM5-met2010_CTRL-TEST  TM5  met2010
2   od550aer  2010    daily    Column  TM5-met2010_CTRL-TEST  TM5  met2010
3   od550aer  2010  monthly    Column  TM5-met2010_CTRL-TEST  TM5

  experiment perturbation  is_at_stations     3D  \
0        AP3     CTRL2019           False  False
4        AP3     CTRL2019           False  False
1        AP3     CTRL2019           False  False
2        AP3     CTRL2019           False  False
3        AP3     CTRL2016           False  False

                                       filename
0  aerocom3_TM5-met2010_AP3-CTRL2019_abs550aer_Co...
4  aerocom3_TM5-met2010_AP3-CTRL2019_abs550aer_Co...
1  aerocom3_TM5-met2010_AP3-CTRL2019_abs550aer_Co...
2  aerocom3_TM5-met2010_AP3-CTRL2019_od550aer_Col...
3  aerocom3_TM5_AP3-CTRL2016_od550aer_Column_2010...
```

You can also filter this attribute based on what you are interested in. E.g.:

[6]:
```
files = reader.filter_files(var_name='od550aer')
files
```

[6]:
```
   var_name  year  ts_type vert_code                data_id name    meteo  \
2  od550aer  2010    daily    Column  TM5-met2010_CTRL-TEST  TM5  met2010
3  od550aer  2010  monthly    Column  TM5-met2010_CTRL-TEST  TM5
```

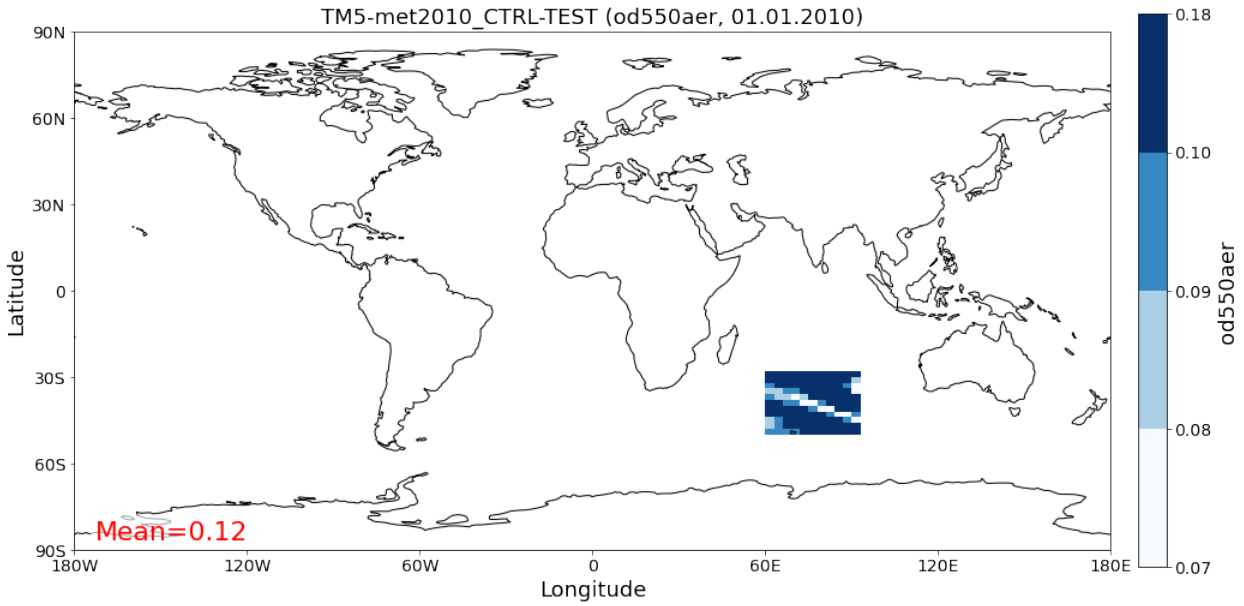
(continues on next page)

(continued from previous page)

```
  experiment perturbation  is_at_stations     3D  \
2        AP3     CTRL2019           False  False
3        AP3     CTRL2016           False  False

                                              filename
2  aerocom3_TM5-met2010_AP3-CTRL2019_od550aer_Col...
3  aerocom3_TM5_AP3-CTRL2016_od550aer_Column_2010...
```

[7]:
```
od550aer = reader.read_var('od550aer')
```

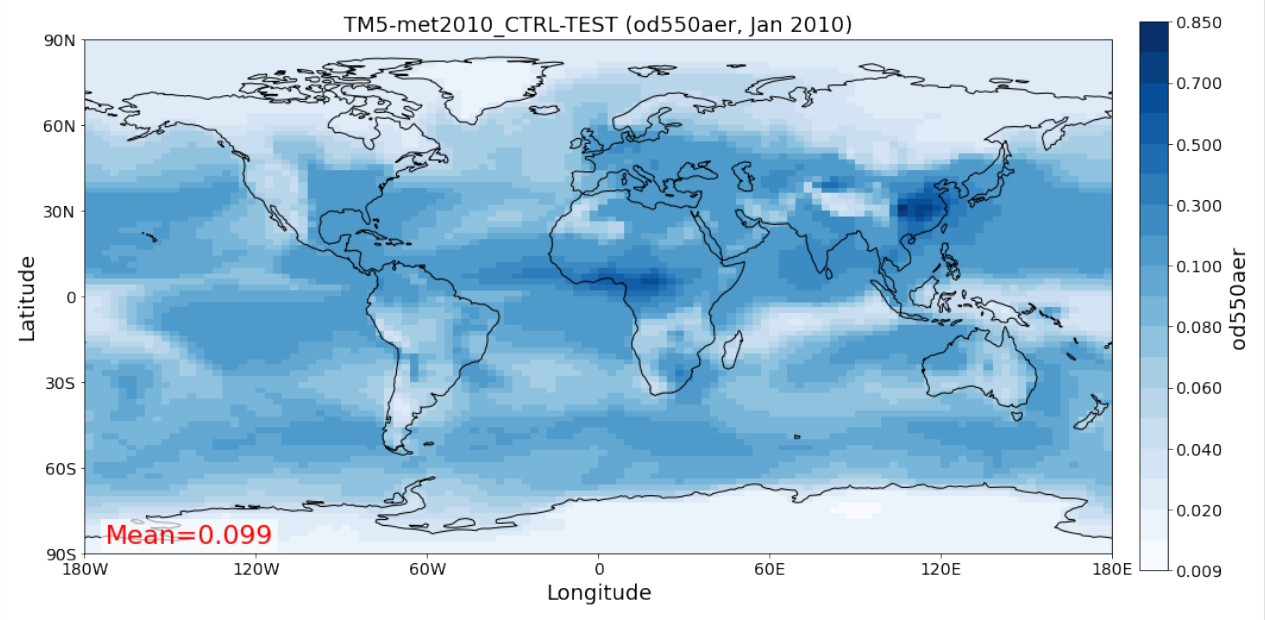
[8]:
```
od550aer.quickplot_map();
```



Ups, this looks rather incomplete. The reason is that pyaerocom picked the available *daily* dataset, which is cropped in the *minimal* testdataset for storage purpose. Let's try monthly.

[9]:
```
od550aer_tm5 = reader.read_var('od550aer', ts_type='monthly')
od550aer_tm5.quickplot_map();
```

```
Rearranging longitude dimension from 0 -> 360 definition to -180 -> 180 definition
```



Looking better. You may wonder why only January is displayed here. This is because `quickplot_map` picks the first available timestamp in the dataset, you may specify that explicitly.

Under the hood pyaerocom.GriddedData is based on the iris.Cube object class (iris library) and features very similar functionality (and more).

The loaded `Cube` instance can be accessed via:

[10]:
```
od550aer_tm5.cube
```

[10]:
```
<iris 'Cube' of atmosphere_optical_thickness_due_to_ambient_aerosol / (1) (time: 12;␣
↪latitude: 90; longitude: 120)>
```

If you have not heard of xarray, you should check it out. If you have heard of it (or maybe even used it already) you may convert a `GriddedData` object to an `xarray.DataArray` via:

[11]:
```
xarr = od550aer_tm5.to_xarray()
xarr
```

[11]:
```
<xarray.DataArray 'od550aer' (time: 12, lat: 90, lon: 120)>
dask.array<filled, shape=(12, 90, 120), dtype=float32, chunksize=(12, 90, 61),␣
↪chunktype=numpy.ndarray>
Coordinates:
  * time     (time) object 2010-01-15 12:00:00 ... 2010-12-15 12:00:00
  * lat      (lat) float64 -89.0 -87.0 -85.0 -83.0 -81.0 ... 83.0 85.0 87.0 89.0
  * lon      (lon) float64 -181.5 -178.5 -175.5 -172.5 ... 169.5 172.5 175.5
Attributes: (12/25)
    standard_name:     atmosphere_optical_thickness_due_to_ambient_aerosol
    long_name:         Ambient Aerosol Optical Thickness at 550 nm
    institution:       Royal Netherlands Meteorological Institute, De Bilt, T...
    institute_id:      KNMI
    source:            TM5-mp: CTM ERA-Interim 3x2 34L
    model_id:          TM5
    ...                ...
    computed:          False
```

(continues on next page)

(continued from previous page)

```
    concatenated:       False
    meteo:
    experiment:         AP3
    perturbation:       CTRL2016
    cell_methods:       longitude: latitude: point time: mean
```

Simply print the object.

[12]:
```
print(od550aer)
```

```
pyaerocom.GriddedData: (od550aer, TM5-met2010_CTRL-TEST)
atmosphere_optical_thickness_due_to_ambient_aerosol / (1) (time: 365; latitude: 11;␣
↪longitude: 11)
     Dimension coordinates:
          time                                                  x              -          ␣
↪     -
          latitude                                              -              x          ␣
↪     -
          longitude                                             -              -          ␣
↪     x
     Attributes:
          Conventions: CF-1.6
          NCO: 4.7.2
          computed: False
          concatenated: False
          contact: Twan van Noije (noije@knmi.nl)
          data_id: TM5-met2010_CTRL-TEST
          experiment: AP3
          experiment_id: AP3-CTRL2019
          from_files: ['/home/jonasg/MyPyaerocom/testdata-minimal/modeldata/TM5-met2010_
↪CTRL...
          history: Wed Jul  8 10:31:53 2020: ncks -d lat,20,30 -d lon,20,30 raw/aerocom3_
↪TM5-met2010_AP3-CTRL2019_od550aer_Column_2010_daily.nc...
          institute_id: KNMI
          institution: Royal Netherlands Meteorological Institute, De Bilt, The␣
↪Netherlands
          meteo: met2010
          model_id: TM5
          outliers_removed: False
          perturbation: CTRL2019
          project_id: AeroCom Phase 3
          reader: None
          references: Van Noije, T.P.C., et al. (Geosci. Model Dev., 7, 2435-2475, 2014);
↪ Bergman,...
          region: None
          regridded: False
          source: TM5-mp, r1058: CTM ERA-Interim 3x2 34L
          timedim-corrected: True
          title: TM5 model output prepared for AeroCom Phase 3
          ts_type: daily
          var_name_read: n/d
          vert_code: Column
     Cell methods:
```

(continues on next page)

(continued from previous page)

```
        point: longitude, latitude
        mean: time
```

Dimension coordinates can be simply accessed either using `[]` or `.` operator, e.g.

[13]:
```
od550aer['latitude']
```

[13]:
```
DimCoord(array([-49., -47., -45., -43., -41., -39., -37., -35., -33., -31., -29.]),␣
↪bounds=array([[-50., -48.],
       [-48., -46.],
       [-46., -44.],
       [-44., -42.],
       [-42., -40.],
       [-40., -38.],
       [-38., -36.],
       [-36., -34.],
       [-34., -32.],
       [-32., -30.],
       [-30., -28.]]), standard_name='latitude', units=Unit('degrees'), long_name=
↪'Center coordinates for latitudes', var_name='lat')
```

[14]:
```
od550aer.longitude
```

[14]:
```
DimCoord(array([61.5, 64.5, 67.5, 70.5, 73.5, 76.5, 79.5, 82.5, 85.5, 88.5, 91.5]),␣
↪bounds=array([[60., 63.],
       [63., 66.],
       [66., 69.],
       [69., 72.],
       [72., 75.],
       [75., 78.],
       [78., 81.],
       [81., 84.],
       [84., 87.],
       [87., 90.],
       [90., 93.]]), standard_name='longitude', units=Unit('degrees'), long_name='Center␣
↪coordinates for longitudes', var_name='lon')
```

They are instances of `iris.coords.DimCoords`, as defined in the underlying `Cube` instance used in the `GriddedData` object.

Time stamps

Time stamps are represented as numerical values with respect to a reference date and frequency, according to the CF conventions. They can be accessed via the `time` attribute of the data class (if the data contains a time dimension).

[15]:
```
od550aer_tm5.time
```

[15]:
```
DimCoord(array([58454.5, 58484. , 58513.5, 58544. , 58574.5, 58605. , 58635.5,
       58666.5, 58697. , 58727.5, 58758. , 58788.5]), standard_name='time', units=Unit(
↪'days since 1850-01-01 00:00', calendar='julian'), long_name='Time', var_name='time')
```

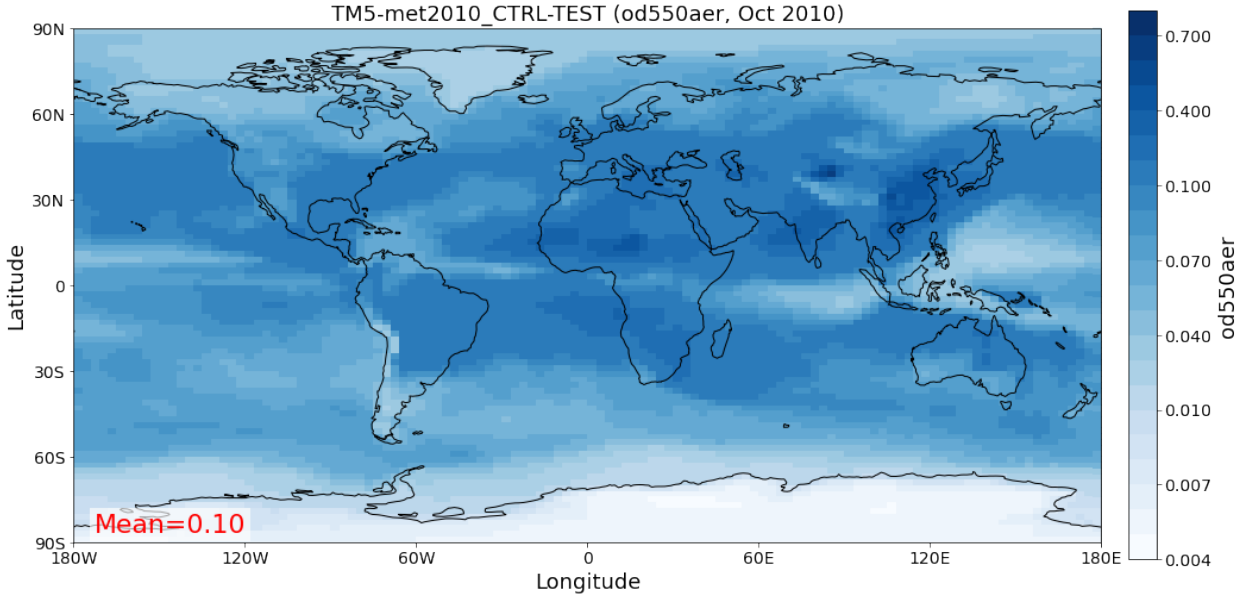
You may also want the time-stamps in the form of actual datetime-like objects. These can be computed using the `time_stamps()` method:

[16]:
```
od550aer.time_stamps()[0:3]
```

[16]:
```
array(['2010-01-01T00:00:00.000000', '2010-01-02T00:00:00.000000',
       '2010-01-03T00:00:00.000000'], dtype='datetime64[us]')
```

As introduced above, maps of individual time stamps can be plotted using the quickplot_map method. Above we used the default call, which chooses the first available timestamp. You may also specify which date you are interested in:

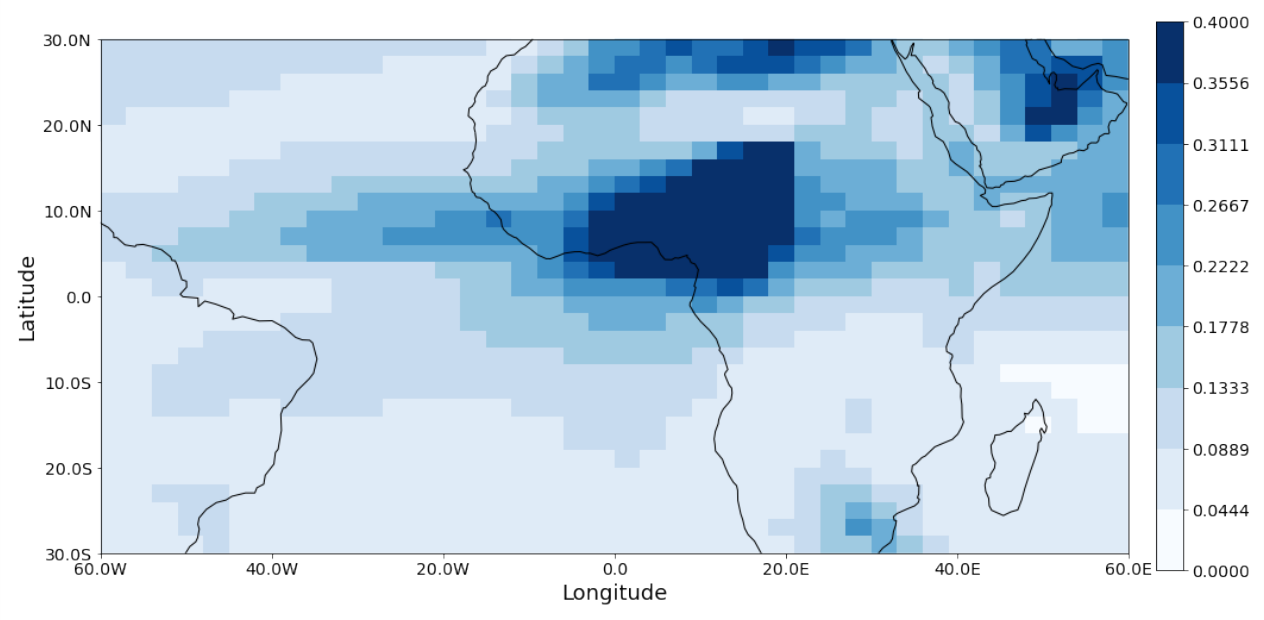
[17]:
```
od550aer_tm5.quickplot_map('2010-10');
```



If you want more control on the input parameters of the map plotting function (e.g. color-binning, lower, upper limit, colorbar, etc.), you may use the underlying plot method (that is also used in `GriddedData.quickplot_map`, which is available at `pya.plot.mapping.plot_griddeddata_on_map <https://pyaerocom.met.no/api.html#pyaerocom.plot.mapping.plot_griddeddata_on_map>`__, e.g.:

[18]:
```
pya.plot.mapping.plot_griddeddata_on_map(od550aer_tm5[1], xlim=(-60, 60), ylim=(-30, 30),
↪ vmin=0, vmax=0.4, log_scale=False);
```

```
/home/jonasg/github/pya/pyaerocom/pyaerocom/mathutils.py:396: RuntimeWarning: divide by␣
↪zero encountered in log10
  return np.floor(np.log10(abs(np.asarray(num)))).astype(int)
```



Regional filtering can be performed using the Filter class.

Rectangular regions

An overview of rectangular AeroCom default regions can be accessed via:

[19]:
```
print(pya.const.OLD_AEROCOM_REGIONS)
```

```
['WORLD', 'ASIA', 'AUSTRALIA', 'CHINA', 'EUROPE', 'INDIA', 'NAFRICA', 'SAFRICA',
↪'SAMERICA', 'NAMERICA']
```

Let's choose north Africa as an example. Create instance of Filter class:

[20]:
```
f = pya.Filter('NAFRICA')
```

You can print its `region` attribute to see the edges:

[21]:
```
print(f.region)
```

```
pyaeorocom Region
Name: NAFRICA
Longitude range: [-17, 50]
Latitude range: [0, 40]
Longitude range (plots): [-17, 50]
Latitude range (plots): [0, 40]
```

Now apply to the model data object:

[22]:
```
od550aer_nafrica = f(od550aer_tm5)
```

Compare shapes:

[23]:
```
od550aer_nafrica
```

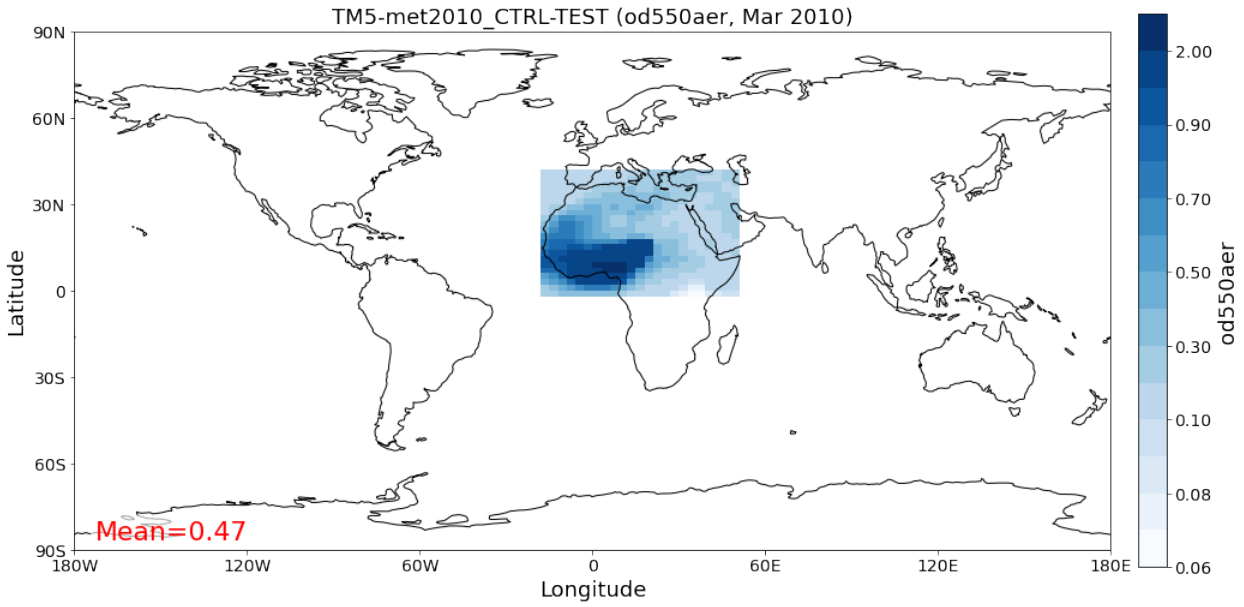
[23]:
```
pyaerocom.GriddedData: (od550aer, TM5-met2010_CTRL-TEST)
<iris 'Cube' of atmosphere_optical_thickness_due_to_ambient_aerosol / (1) (time: 12;␣
↪latitude: 22; longitude: 23)>
```

[24]:
```
od550aer_tm5
```

[24]:
```
pyaerocom.GriddedData: (od550aer, TM5-met2010_CTRL-TEST)
<iris 'Cube' of atmosphere_optical_thickness_due_to_ambient_aerosol / (1) (time: 12;␣
↪latitude: 90; longitude: 120)>
```

As you can see, the filtered object is reduced in the longitude and latitude dimension. Let's have a look:

[25]:
```
od550aer_nafrica.quickplot_map('March 2010');
```



Binary region masks

Available HTAP binary filter masks can be accessed via:

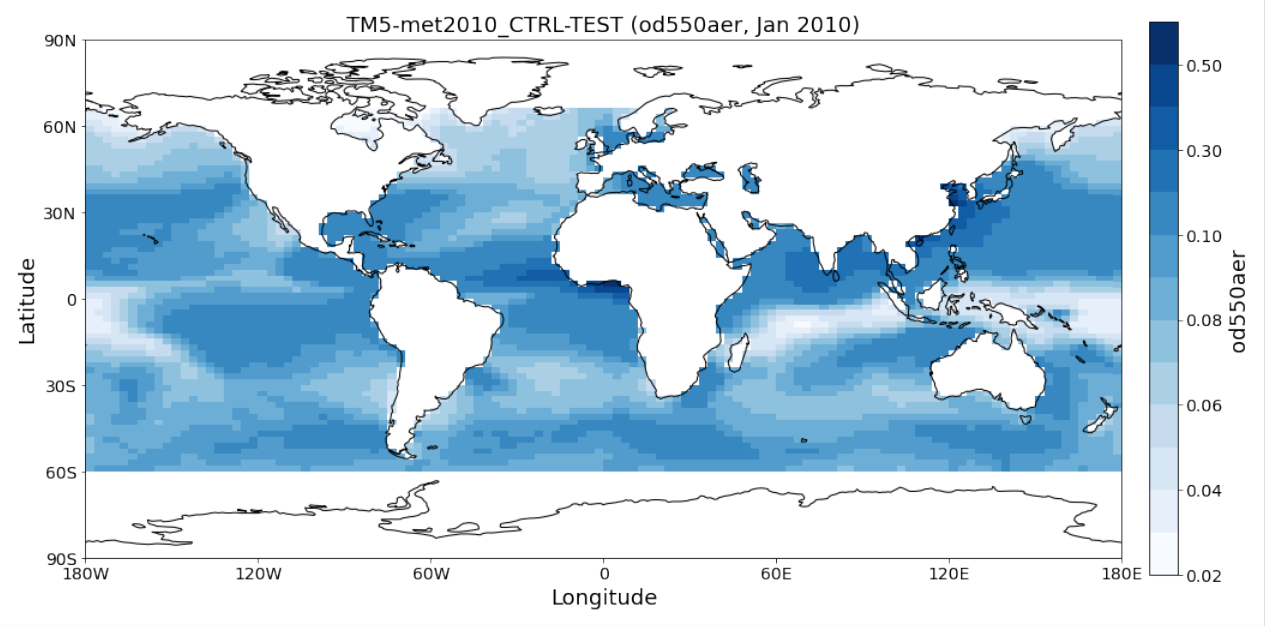
[26]:
```
print(pya.const.HTAP_REGIONS)
```

```
['PAN', 'EAS', 'NAF', 'MDE', 'LAND', 'SAS', 'SPO', 'OCN', 'SEA', 'RBU', 'EEUROPE', 'NAM',
↪ 'WEUROPE', 'SAF', 'USA', 'SAM', 'EUR', 'NPO', 'MCA']
```

And they are handled in the same way as the rectangular regions:

[27]:
```
pya.Filter('OCN')(od550aer_tm5).quickplot_map();
```

```
Failed to compute / add area weighted mean. Reason: ValueError('Format specifier missing␣
↪precision')
```



As you can see the provided HTAP region masks are only valid within 60°S to 60°N.

Filtering of time

Filtering of time is not included in the Filter class (which only allows for regional filtering) but can be easily performed from the `GriddedData` object directly. If you know the indices of the time stamps you want to crop, you can simply use numpy indexing syntax (remember that we have a 3D array containing time, latitude and lonfgitude).

Let's say we are interested in the (northern hemispheric) summer months of June to September.

Since the time dimension corresponds the first index in the 3D data (time, lat, lon), and since we know, that we have monthly 2010 data (see above), we may use:

[28]:
```
od550aer_summer = od550aer_tm5[5:8]
od550aer_summer.time_stamps()
```

[28]:
```
array(['2010-06-15T00:00:00.000000', '2010-07-15T12:00:00.000000',
       '2010-08-15T12:00:00.000000'], dtype='datetime64[us]')
```

However, this methodology might not always be handy (imagine you have a 10 year dataset of `3hourly` sampled data and want to extract three months in the 6th year ...). In that case, you can perform the cropping using the actual timestamps:

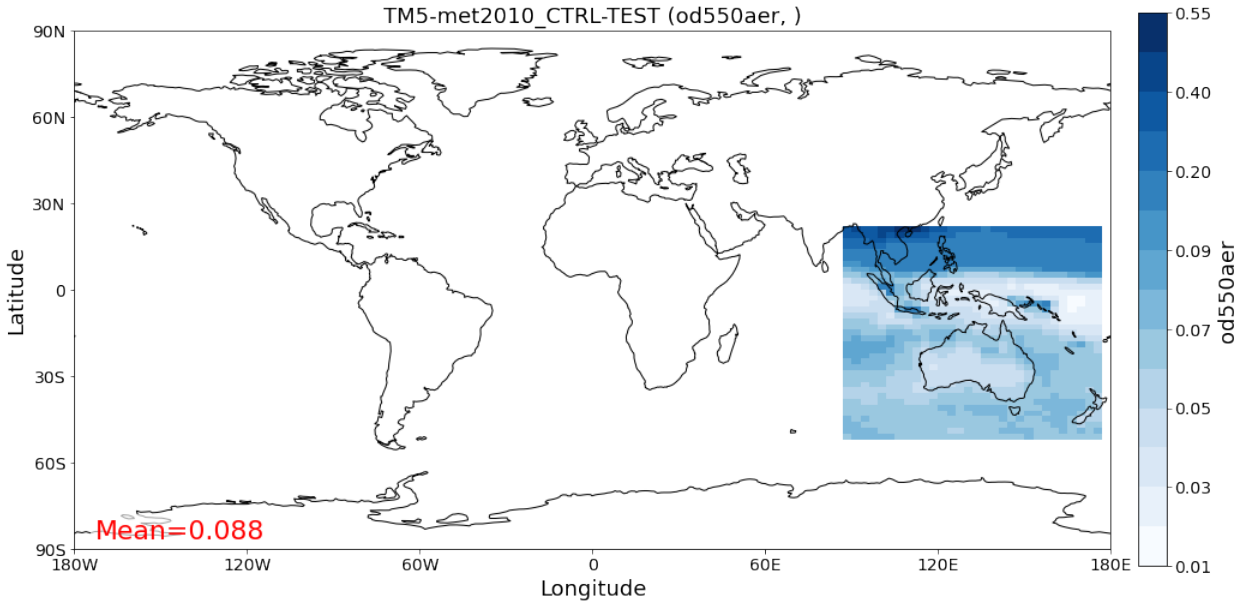
[29]:
```
od550aer_tm5.crop(time_range=('6-2010', '9-2010')).time_stamps()
```

[29]:
```
array(['2010-06-15T00:00:00.000000', '2010-07-15T12:00:00.000000',
       '2010-08-15T12:00:00.000000'], dtype='datetime64[us]')
```

Data selection over multiple dimensions

Inspired by the xarray.DataArray.sel method, a similar method was implemented in `GriddedData`:

[30]:
```
od550aer_tm5.sel(time='April 2010', longitude=(90, 179), latitude=(-50, 20)).quickplot_
↪map();
```



NOTE: Before release of version 0.10.0, there was a bug that led to a crash if a time range (i.e. `time=(start, stop)`) was passed into the `sel` method.

You may regrid `GriddedData` using the `regrid` method (for regional regridding) or the `resample_time` method (for temporal resmpling). Like already done above, the calls may be combined, e.g.:

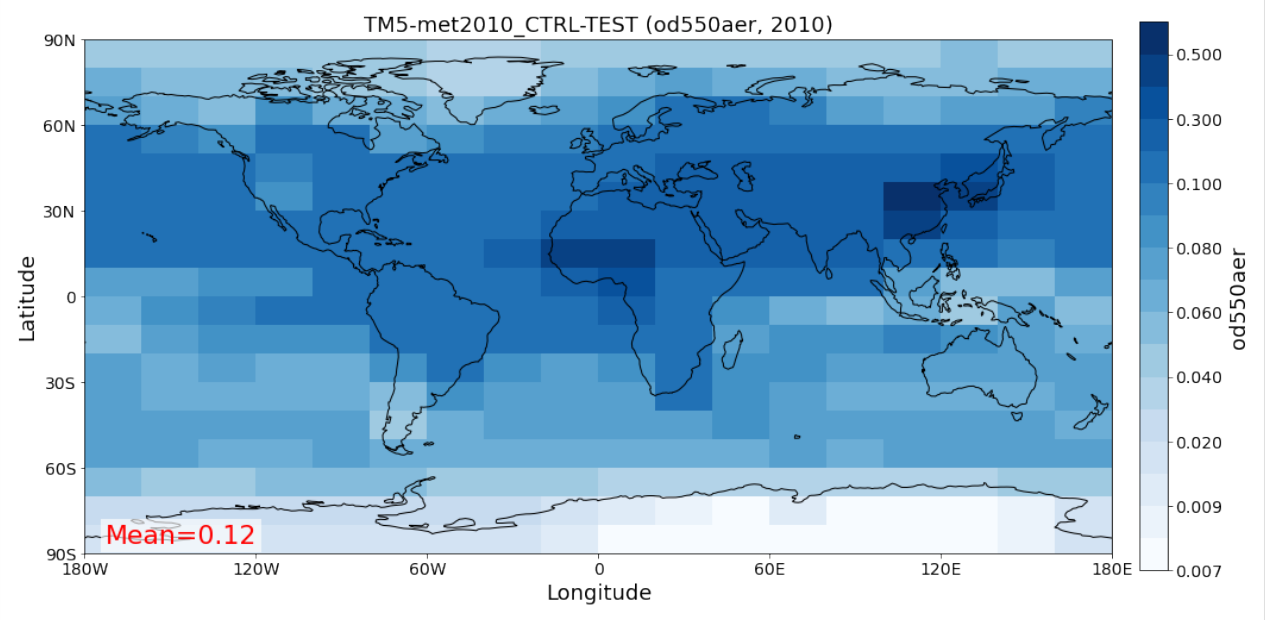
[31]:
```
lowres = od550aer_tm5.regrid(lat_res_deg=10, lon_res_deg=20).resample_time('yearly')
lowres
```

[31]:
```
pyaerocom.GriddedData: (od550aer, TM5-met2010_CTRL-TEST)
<iris 'Cube' of od550aer / (1) (time: 1; latitude: 18; longitude: 18)>
```

As you can see, the time dimension only has one entry, as expected, as the data only contains 2010 timestamps and we computed a yearly average, lat and lon dimensions are also reduced, accordingly.

[32]:
```
lowres.quickplot_map();
```

```
/home/jonasg/miniconda3/envs/pyadev/lib/python3.9/site-packages/iris/coords.py:1784:␣
↪UserWarning: Coordinate 'longitude' is not bounded, guessing contiguous bounds.
  warnings.warn(
/home/jonasg/miniconda3/envs/pyadev/lib/python3.9/site-packages/iris/coords.py:1784:␣
↪UserWarning: Coordinate 'latitude' is not bounded, guessing contiguous bounds.
  warnings.warn(
```



Regional averaging

The actual cell sizes of latitude and longitude coordinates vary, dependent on where you are, that is, they are largest close to the equator, and smallest near the poles. When computing a regional average, this needs to be considered (i.e. values need to be weighted by their actual cell size). This is *area weighted regridding* is implemented in the `iris` library and is used by default in `GriddedData`, for instance, when calling:

[33]:
```
od550aer_tm5.mean()
```

[33]:
```
0.11864813532841474
```

You may specify if you do not want to use area weighting:

[34]:
```
od550aer_tm5.mean(areaweighted=False)
```

[34]:
```
0.09825691
```

Makes quite a difference, doesn't it?

Time-series at individual coordinates can be extracted from a `GriddedData` object via:

[35]:
```
ts_data = od550aer_tm5.to_time_series(latitude=60, longitude=11)
ts_data
```

[35]:
```
[StationData: {'dtime': [], 'var_info': BrowseDict: {'od550aer': {'units': Unit('1')}},
↪'station_coords': {'latitude': None, 'longitude': None, 'altitude': None}, 'data_err':␣
↪BrowseDict: {}, 'overlap': BrowseDict: {}, 'numobs': BrowseDict: {}, 'data_flagged':␣
↪BrowseDict: {}, 'filename': None, 'station_id': None, 'station_name': None,
↪'instrument_name': None, 'PI': None, 'country': None, 'country_code': None, 'ts_type':
↪'monthly', 'latitude': 61.0, 'longitude': 10.5, 'altitude': nan, 'data_id': 'TM5-
↪met2010_CTRL-TEST', 'dataset_name': None, 'data_product': None, 'data_version': None,
↪'data_level': None, 'framework': None, 'instr_vert_loc': None, 'revision_date': None,
↪'website': None, 'ts_type_src': None, 'stat_merge_pref_attr': None, 'od550aer': 2010-
↪01-15 12:00:00    0.049607
```

(continues on next page)

(continued from previous page)

```
2010-02-14 00:00:00    0.061162
2010-03-15 12:00:00    0.069986
2010-04-15 00:00:00    0.097556
2010-05-15 12:00:00    0.103770
2010-06-15 00:00:00    0.107482
2010-07-15 12:00:00    0.146354
2010-08-15 12:00:00    0.145518
2010-09-15 00:00:00    0.078066
2010-10-15 12:00:00    0.077722
2010-11-15 00:00:00    0.037447
2010-12-15 12:00:00    0.039024
dtype: float32}]
```

As you can see from the output, the return value of this method is a list, that contains *one* `pyaerocom.StationData` object. The reason why this method returns a list is because it is usually called with many input coordinates (e.g. all site locations of an observation network), and thus, returns a list of `StationData` objects, one for each input coordinate.

The `StationData` object is basically a dictionary-like object with some extra functionality.

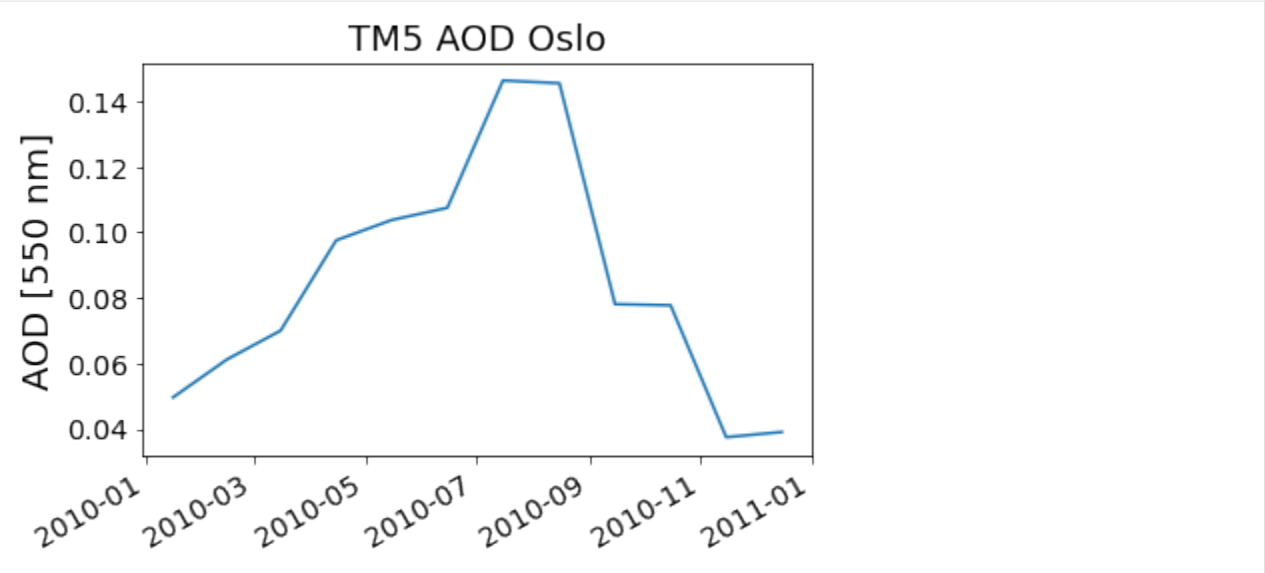
[36]:
```
station = ts_data[0]
```

The actual time-series is a `pandas.Series` object and can be accessed through the variable name (remember, `GriddedData` instances are single variable).

[37]:
```
ts = station['od550aer']
ts
```

[37]:
```
2010-01-15 12:00:00    0.049607
2010-02-14 00:00:00    0.061162
2010-03-15 12:00:00    0.069986
2010-04-15 00:00:00    0.097556
2010-05-15 12:00:00    0.103770
2010-06-15 00:00:00    0.107482
2010-07-15 12:00:00    0.146354
2010-08-15 12:00:00    0.145518
2010-09-15 00:00:00    0.078066
2010-10-15 12:00:00    0.077722
2010-11-15 00:00:00    0.037447
2010-12-15 12:00:00    0.039024
dtype: float32
```

[38]:
```
ax = ts.plot()
ax.set_title('TM5 AOD Oslo')
ax.set_ylabel('AOD [550 nm]');
```



Let's have a closer look at the observations. After all, the main purpose of the AeroCom initiative is to compare models with observations. As we shall see below, the just introduced `StationData` object will play a key role when bringing *gridded* model data (`GriddedData`) together with *ungridded* observational data, such as measurements of a certain variable at a given site location.

In the following section the reading of ungridded data is illustrated based on the example of AERONET version 3 (level 2) data.

Observational data: Reading of and working with ungridded data

This section provides brief introductions into the following pyaerocom classes and architectures:

- `` `pya.io.ReadUngridded <https://pyaerocom.met.no/api.html#pyaerocom.io.readungridded.ReadUngridded>`__ ``
- `` `pya.UngriddedData <https://pyaerocom.met.no/api.html#pyaerocom.ungriddeddata.UngriddedData>`__ ``
- `` `pya.StationData <https://pyaerocom.met.no/api.html#pyaerocom.stationdata.StationData>`__ ``

Primer on observational data

Other than model data, which can be provided as a gridded object over a certain domain (e.g. latitude, longitude, time) and in that, can be considered **fully sampled**, observational data is usually **sparsely** sampled in space and time.

That is, consider a network of observations of a certain variable (e.g. od550aer, or AOD), with many different site locations around the globe. Each of these sites is measuring the variable at that exact location, and the whole network of sites makes a point cloud of site locations in the latitude, longitude domain. In addition, since these are real world measurements, the temporal sampling itself between the different sites is not synchronised, that is, each site is measuring independently of any other site.

For instance, the AERONET network is a global network of sun photometer measurements, that can measure the AOD at several wavelengths based on measurements of the solar irradiance. Thus, at the least, these measurements require 2 things:

1. Daylight
2. A clear sky

Thus, it is needless to say, that a site in Antarctica cannot measure at the same time as a site in Ny-Ålesund (actually, that is also not strictly true, as AERONET now also provides AOD measurements based on the lunar irradiance, but I hope you got the point anyways).

This should illustrate, that it is more difficult to define a *harmonised* and yet, flexible data format for such observational databases. In pyaerocom, the `UngriddedData` object is designed for such point cloud data and typically holds the data belonging to a whole observation network, that is:

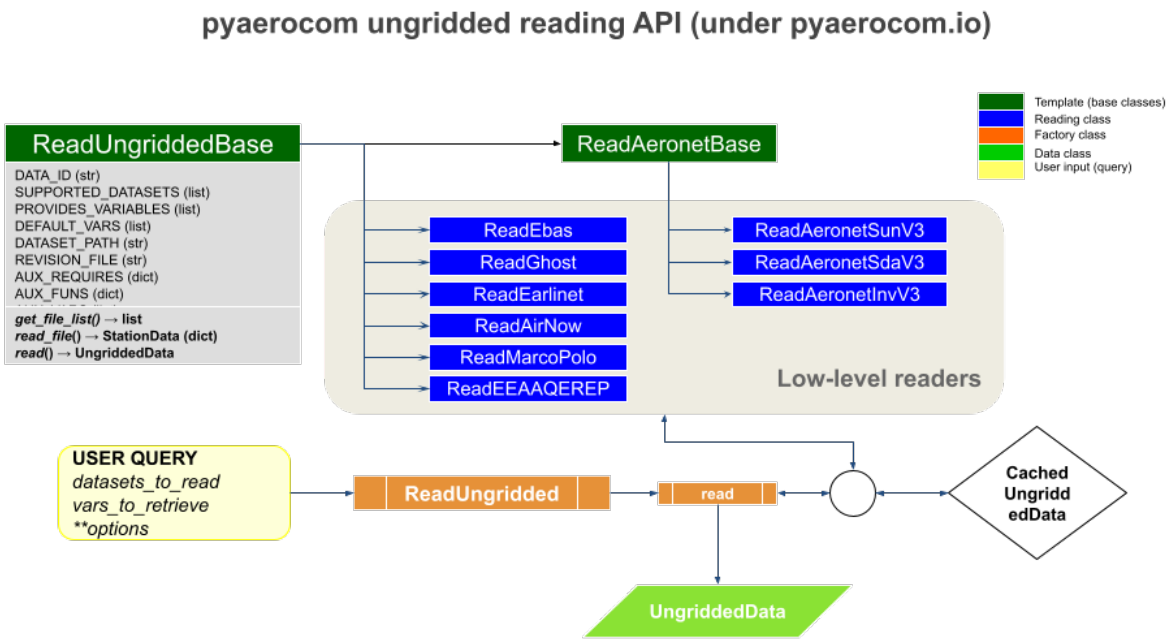
The ``UngriddedData`` object can be considered a point-cloud-like dataobject that holds individual time-series from many locations around the globe and the associated metadata for each site and measurement

Moreover, since observational data typically comes from many different observation networks, the formats in which these data are stored typically vary from network to network, which makes it harder to read the data, compared to model data which typically comes as NetCDF file and these days, most often follow some metadata conventions such as the CF conventions.

Data from the AERONET network (that is introduced in the following), for instance, is provided in the form of column seperated text files per measurement station, where columns correspond to different variables and data rows to individual time stamps.

As a result, custom reading routines for individual observation networks need to be implemented, and pyaerocom provides reading support for many commonly used observational databases such as AERONET, or the EBAS or EARLINET data.

The basic workflow for reading of ungridded data, such as Aeronet data, is very similar to the reading of gridded data (comprising a reading class that handles a query and returns a data class, here UngriddedData. However, under the hood, the implementation is a little more complicated, as there are reading classes for each supported network, as illustrated in the following flowchart:



The actual classes handling the reading of data (for a given dataset) are indicated in blue. The orange `ReadUngridded` class is a factory class, that knows about the blue reading classes via a unique ID (similar to the gridded reading). Thus, as indicated, as a user, you do not need to know which exact reading class you need, you just need the ID and `ReadUngridded` will know which (blue) reader to use. To summarise, what you need for reading an ungridded dataset is:

1. A path where the actual datafiles are located
2. An unique ID, that links that path with a name
3. A reader that can read the class

The first 2 points are available via:

[39]:
```
pya.const.OBSLOCS_UNGRIDDED
```

[39]:
```
OrderedDict([('AeronetSunV2Lev1.5.daily', '/lustre/storeA/project/aerocom/'),
             ('AeronetSun_2.0_NRT',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunNRT'),
             ('AeronetSunV2Lev2.daily',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetRaw2.0/
↪renamed'),
             ('AeronetSunV2Lev2.AP',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSun2.
↪0AllPoints/renamed'),
             ('AeronetSDAV2Lev2.daily',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSun2.0.SDA.
↪daily/renamed'),
             ('AeronetSDAV2Lev2.AP',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSun2.0.SDA.
↪AP/renamed'),
             ('AeronetInvV2Lev1.5.daily',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Aeronet.Inv.V2L1.
↪5.daily/renamed'),
             ('AeronetInvV2Lev1.5.AP',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/'),
             ('AeronetInvV2Lev2.daily',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Aeronet.Inv.V2L2.
↪0.daily/renamed'),
             ('AeronetInvV2Lev2.AP',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/'),
             ('AeronetSunV3Lev1.5.daily',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev1.
↪5.daily/renamed'),
             ('AeronetSunV3Lev1.5.AP',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev1.
↪5.AP/renamed'),
             ('AeronetSunV3Lev2.daily',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev2.
↪0.daily/renamed'),
             ('AeronetSunV3Lev2.AP',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev2.
↪0.AP/renamed'),
             ('AeronetSDAV3Lev1.5.daily',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Aeronet.SDA.V3L1.
↪5.daily/renamed'),
             ('AeronetSDAV3Lev1.5.AP',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/'),
             ('AeronetSDAV3Lev2.daily',
              '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Aeronet.SDA.V3L2.
↪0.daily/renamed'),
```

(continues on next page)

(continued from previous page)

```
            ('AeronetSDAV3Lev2.AP',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/'),
            ('AeronetInvV3Lev1.5.daily',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Aeronet.Inv.V3L1.
↪5.daily/renamed'),
            ('AeronetInvV3Lev2.daily',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Aeronet.Inv.V3L2.
↪0.daily/renamed'),
            ('EBASMC',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/EBASMultiColumn/
↪data'),
            ('EEAAQeRep',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/EEA_AQeRep/renamed
↪'),
            ('EARLINET',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Export/Earlinet/
↪CAMS/data'),
            ('GAWTADsubsetAasEtAl',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/PYAEROCOM/
↪GAWTADSulphurSubset/data'),
            ('DMS_AMS_CVO',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/PYAEROCOM/DMS_AMS_
↪CVO/data'),
            ('GHOST.EEA.daily',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/GHOST/data/EEA_AQ_
↪eReporting/daily'),
            ('GHOST.EEA.hourly',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/GHOST/data/EEA_AQ_
↪eReporting/hourly'),
            ('GHOST.EEA.monthly',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/GHOST/data/EEA_AQ_
↪eReporting/monthly'),
            ('GHOST.EBAS.daily',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/GHOST/data/EBAS/
↪daily'),
            ('GHOST.EBAS.hourly',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/GHOST/data/EBAS/
↪hourly'),
            ('GHOST.EBAS.monthly',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/GHOST/data/EBAS/
↪monthly'),
            ('EEAAQeRep.NRT',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/EEA_AQeRep.NRT/
↪renamed/'),
            ('EEAAQeRep.v2',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/EEA_AQeRep.v2/
↪renamed/'),
            ('AirNow',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/MACC_INSITU_AirNow
↪'),
            ('MarcoPolo',
             '/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/CHINA_MP_NRT'),
```

(continues on next page)

(continued from previous page)

```
             ('AeronetSunV3L2Subset.daily',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/AeronetSunV3Lev2.daily/
↪renamed'),
             ('AeronetSDAV3L2Subset.daily',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/AeronetSDAV3Lev2.daily/
↪renamed'),
             ('AeronetInvV3L2Subset.daily',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/AeronetInvV3Lev2.daily/
↪renamed'),
             ('EBASSubset',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/EBASMultiColumn'),
             ('AirNowSubset',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/AirNowSubset'),
             ('G.EEA.daily.Subset',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/GHOST/data/EEA_AQ_
↪eReporting/daily'),
             ('G.EEA.hourly.Subset',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/GHOST/data/EEA_AQ_
↪eReporting/hourly'),
             ('G.EBAS.daily.Subset',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/GHOST/data/EBAS/daily'),
             ('G.EBAS.hourly.Subset',
              '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/GHOST/data/EBAS/hourly
↪')])
```

And the reader classes that are supposed to be used for each of these IDs is provided in the `ReadUngridded` class header:

[40]:
```
pya.io.ReadUngridded.SUPPORTED_READERS
```

[40]:
```
[pyaerocom.io.read_aeronet_invv3.ReadAeronetInvV3,
 pyaerocom.io.read_aeronet_invv2.ReadAeronetInvV2,
 pyaerocom.io.read_aeronet_sdav2.ReadAeronetSdaV2,
 pyaerocom.io.read_aeronet_sdav3.ReadAeronetSdaV3,
 pyaerocom.io.read_aeronet_sunv2.ReadAeronetSunV2,
 pyaerocom.io.read_aeronet_sunv3.ReadAeronetSunV3,
 pyaerocom.io.read_earlinet.ReadEarlinet,
 pyaerocom.io.read_ebas.ReadEbas,
 pyaerocom.io.read_gaw.ReadGAW,
 pyaerocom.io.read_aasetal.ReadAasEtal,
 pyaerocom.io.read_ghost.ReadGhost,
 pyaerocom.io.read_airnow.ReadAirNow,
 pyaerocom.io.read_marcopolo.ReadMarcoPolo,
 pyaerocom.io.read_eea_aqerep.ReadEEAAQEREP,
 pyaerocom.io.read_eea_aqerep_v2.ReadEEAAQEREP_V2]
```

The link between ID (keys of `const.OBSLOCS_UNGRIDDED`) and reader is available in the actual readers themselves, e.g.:

[41]:
```
pya.io.read_aeronet_sunv3.ReadAeronetSunV3.SUPPORTED_DATASETS
```

[41]:
```
['AeronetSunV3Lev1.5.daily',
 'AeronetSunV3Lev1.5.AP',
```

(continues on next page)

(continued from previous page)

```
 'AeronetSunV3Lev2.daily',
 'AeronetSunV3Lev2.AP',
 'AeronetSunV3L2Subset.daily']
```

But these are details that you usually do not need to worry about. If you want to register a new observation dataset, you need the 3 points specified above and can add it via:

[42]:
```
aeronet_sun_datadir = f'{pya.const.OUTPUTDIR}/testdata-minimal/obsdata/AeronetSunV3Lev2.
↪daily/renamed'
pya.const.add_ungridded_obs(obs_id='Bla',
                            data_dir=aeronet_sun_datadir,
                            reader=pya.io.read_aeronet_sunv3.ReadAeronetSunV3)
```

Now, we basically have 2 names for the same dataset:

[43]:
```
pya.io.read_aeronet_sunv3.ReadAeronetSunV3.SUPPORTED_DATASETS
```

[43]:
```
['AeronetSunV3Lev1.5.daily',
 'AeronetSunV3Lev1.5.AP',
 'AeronetSunV3Lev2.daily',
 'AeronetSunV3Lev2.AP',
 'AeronetSunV3L2Subset.daily',
 'Bla']
```

That is, the data under the above directory is now accessible via 2 IDs: `Bla` and `AeronetSunV3L2Subset.daily`.

Before continuing with the reading of observational data, some things need to be said related to the caching of `UngriddedData` objects.

Caching of UngriddedData

Reading of ungridded data is often rather time-consuming. Therefore, pyaerocom uses a caching strategy that stores loaded instances of the `UngriddedData` class as pickle files in a cache directory (illustrated in the flowchart shown above). The loaction of the cache directory can be accessed via:

[44]:
```
pya.const.CACHEDIR
```

[44]:
```
'/home/jonasg/MyPyaerocom/_cache/jonasg'
```

You may change this directory if required.

[45]:
```
f'Caching is active? {pya.const.CACHING}'
```

[45]:
```
'Caching is active? True'
```

Deactivate / Activate caching

[46]:
```
pya.const.CACHING = False
```

[47]:
```
pya.const.CACHING = True
```

Note: if caching is active, make sure you have enough disk quota or change location where the cache files are stored.

Read Aeronet Sun v3 level 2 data

As illustrated in the flowchart above, ungridded observation data can be imported using the `ReadUngridded` class. Like for the model data, observation datasets can be searched as follows:

[48]:
```
pya.browse_database('Aeronet*');
```

```
Found more than 20 matches for input pattern Aeronet*:

Matches: ['AeronetSunV2Lev1.5.daily', 'AeronetSun_2.0_NRT', 'AeronetSunV2Lev2.daily',
↪'AeronetSunV2Lev2.AP', 'AeronetSDAV2Lev2.daily', 'AeronetInvV2Lev1.5.daily',
↪'AeronetInvV2Lev1.5.AP', 'AeronetInvV2Lev2.daily', 'AeronetInvV2Lev2.AP',
↪'AeronetSunV3Lev1.5.daily', 'AeronetSunV3Lev1.5.AP', 'AeronetSunV3Lev2.daily',
↪'AeronetSunV3Lev2.AP', 'AeronetSDAV3Lev1.5.daily', 'AeronetSDAV3Lev1.5.AP',
↪'AeronetSDAV3Lev2.daily', 'AeronetSDAV3Lev2.AP', 'AeronetInvV3Lev1.5.daily',
↪'AeronetInvV3Lev2.daily', 'AeronetSunV3L2Subset.daily', 'AeronetSDAV3L2Subset.daily',
↪'AeronetInvV3L2Subset.daily', 'AERONET_TESTBED-SK']

To receive more detailed information, please specify search ID more accurately
```

The search routine found 3 matches for the 3 different AERONET data products: Sun, SDA, and Inv (inversion). You may read more about the different products at the AERONET website.

Let's continue with the "Sun" product (AERONET Direct Sun algorithm). As you can see from the output above, this dataset contains daily averages, which is convenient to use for model evaluation.

[49]:
```
obs_id = 'AeronetSunV3L2Subset.daily'
```

[50]:
```
obs_reader = pya.io.ReadUngridded(obs_id)
print(obs_reader)
```

```
Dataset name: AeronetSunV3L2Subset.daily
Data directory: /home/jonasg/MyPyaerocom/testdata-minimal/obsdata/AeronetSunV3Lev2.daily/
↪renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang44&87aer', 'od550aer']
Last revision: n/d
```

Let's read the data (you can read a single or multiple variables at the same time). For now, we only read the AOD at 550 nm:

[51]:
```
od550aer_aeronet = obs_reader.read(vars_to_retrieve='od550aer')
od550aer_aeronet
```

[51]:
```
UngriddedData <networks: ['AeronetSunV3L2Subset.daily']; vars: ['od550aer']; instruments:
↪ ['sun_photometer'];No. of metadata units: 22
```

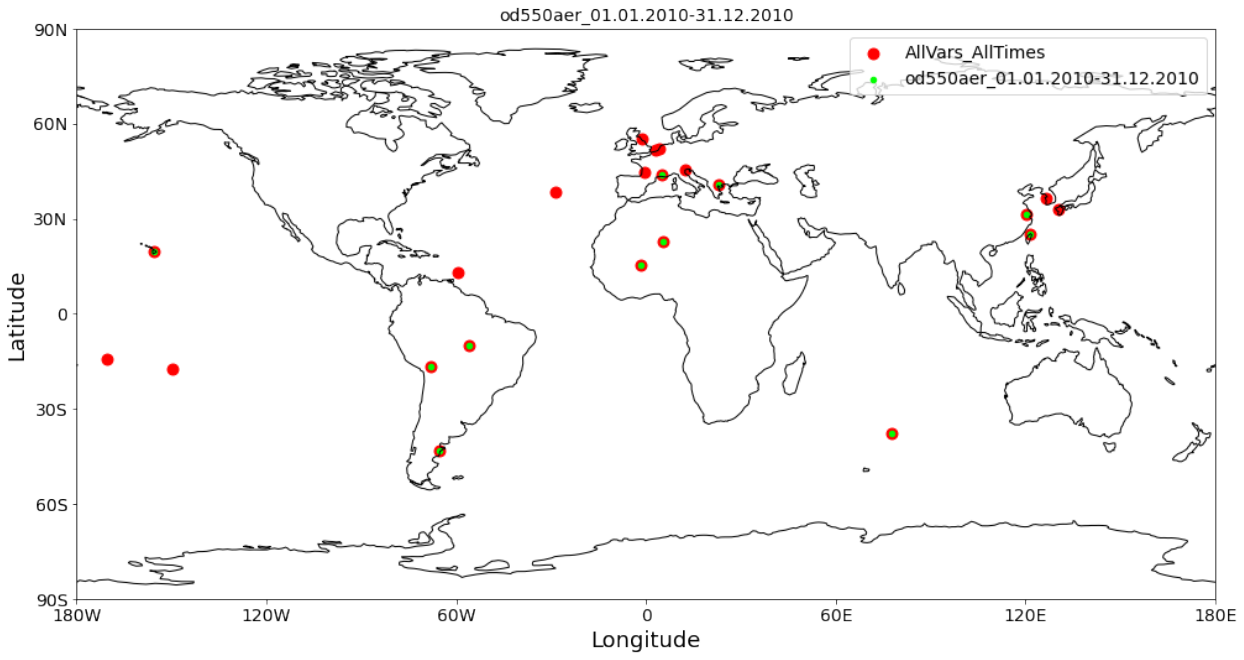
As you can see, the data object is of type `UngriddedData`. Other than `GriddedData`, `UngriddedData` can hold an arbitrary number of variables, and even networks. The number of metadata units indicates the number of data files that have been read.

Plot all station coordinates

To get an overview, you can plot all site coordinates contained in the dataset. You can also plot multiple times into the same map with different input criteria. For instance, below we first plot all site locations available in the data (in red), and then, on top of it, in green, we plot sites that contain data in 2010.

[52]:
```
ax = od550aer_aeronet.plot_station_coordinates(markersize=80)
od550aer_aeronet.plot_station_coordinates(color='lime', var_name='od550aer', start=2010,␣
↪stop=2011, markersize=20, ax=ax)
```

[52]:
```
<GeoAxes:title={'center':'od550aer_01.01.2010-31.12.2010'}, xlabel='Longitude', ylabel=
↪'Latitude'>
```



Access of individual stations

For intercomparison with model data, we are interested in time-series from individual sites. You can check out all existing site-location names via:

[53]:
```
od550aer_aeronet.unique_station_names
```

[53]:
```
['AAOT',
 'ARIAKE_TOWER',
 'Agoufou',
 'Alta_Floresta',
```

(continues on next page)

(continued from previous page)

```
 'American_Samoa',
 'Amsterdam_Island',
 'Anmyon',
 'Avignon',
 'Azores',
 'BORDEAUX',
 'Barbados',
 'Blyth_NOAH',
 'La_Paz',
 'Mauna_Loa',
 'Tahiti',
 'Taihu',
 'Taipei_CWB',
 'Tamanrasset_INM',
 'The_Hague',
 'Thessaloniki',
 'Thornton_C-power',
 'Trelew']
```

To access individual site location data as `StationData` you can simply do:

[54]:
```
station_data = od550aer_aeronet['La_Paz'] # this is fully equivalent with aeronet_data.
↪to_station_data('Leipzig')
station_data
```

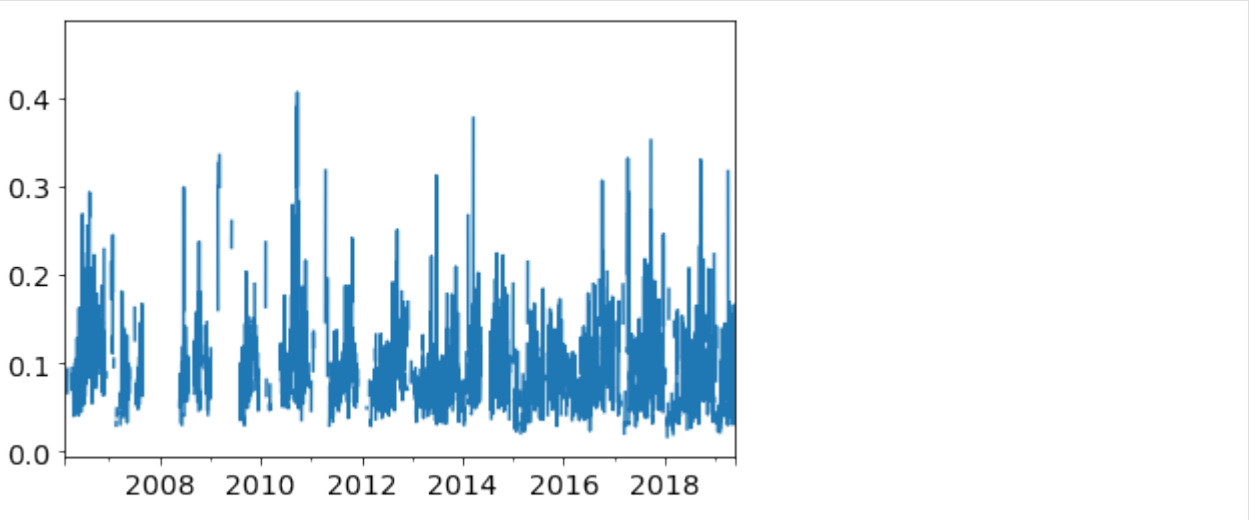
[54]:
```
StationData: {'dtime': array(['2006-02-18T00:00:00.000000000', '2006-02-19T00:00:00.
↪000000000',
       '2006-02-20T00:00:00.000000000', ...,
       '2019-05-24T00:00:00.000000000', '2019-05-25T00:00:00.000000000',
       '2019-05-26T00:00:00.000000000'], dtype='datetime64[ns]'), 'var_info': BrowseDict:
↪ {'od550aer': OrderedDict([('units', '1'), ('overlap', False), ('ts_type', 'daily'), (
↪'min_num_obs', None), ('how', 'mean')])}, 'station_coords': {'latitude': -16.
↪538999999999998, 'longitude': -68.066467, 'altitude': 3439.0}, 'data_err': BrowseDict:
↪{}, 'overlap': BrowseDict: {}, 'numobs': BrowseDict: {}, 'data_flagged': BrowseDict: {}
↪, 'filename': '/home/jonasg/MyPyaerocom/testdata-minimal/obsdata/AeronetSunV3Lev2.
↪daily/renamed/La_Paz.lev30', 'station_id': None, 'station_name': 'La_Paz', 'instrument_
↪name': 'sun_photometer', 'PI': 'Brent_Holben', 'country': None, 'country_code': None,
↪'ts_type': 'daily', 'latitude': -16.538999999999998, 'longitude': -68.066467, 'altitude
↪': 3439.0, 'data_id': 'AeronetSunV3L2Subset.daily', 'dataset_name': None, 'data_product
↪': None, 'data_version': None, 'data_level': None, 'framework': None, 'instr_vert_loc':
↪ None, 'revision_date': None, 'website': None, 'ts_type_src': 'daily', 'stat_merge_
↪pref_attr': None, 'data_revision': 'n/d', 'od550aer': 2006-02-18    0.087568
2006-02-19         NaN
2006-02-20         NaN
2006-02-21         NaN
2006-02-22    0.101807
               ...
2019-05-22         NaN
2019-05-23    0.059489
2019-05-24    0.049292
2019-05-25    0.074172
2019-05-26    0.060137
Freq: D, Length: 4846, dtype: float64}
```

As you can see, the returned object is of type `StationData` which has been introduced above (remember, we extracted a time series from the TM5 model for the location of Oslo).

As mentioned above, it can be used like a dictionary, and the variable time-series can be accessed via:

[55]:
```
station_data['od550aer'].plot()
```

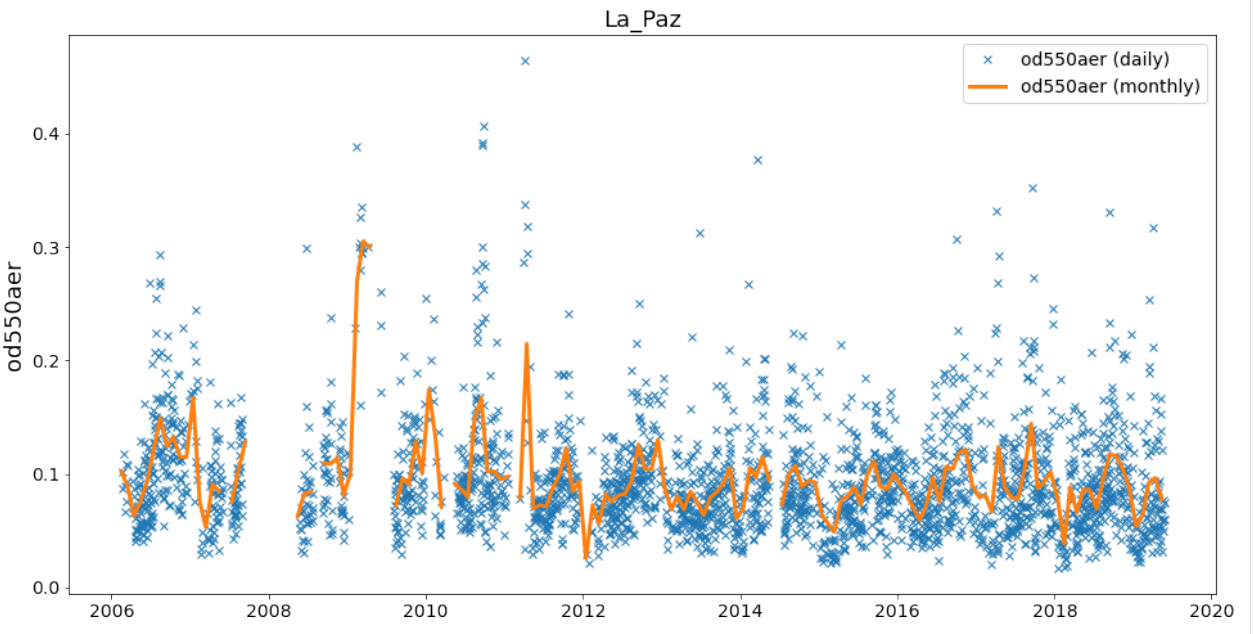
[55]:
```
<AxesSubplot:>
```



You may also plot directly from the `StationData` object (and do some more other hopefully self-explanatory things):

[56]:
```
ax = station_data.plot_timeseries('od550aer', marker='x', ls='none')
station_data.resample_time(var_name='od550aer', ts_type='monthly').plot_timeseries(
↪'od550aer', marker=' ', ls='-', lw=3, ax=ax)
```

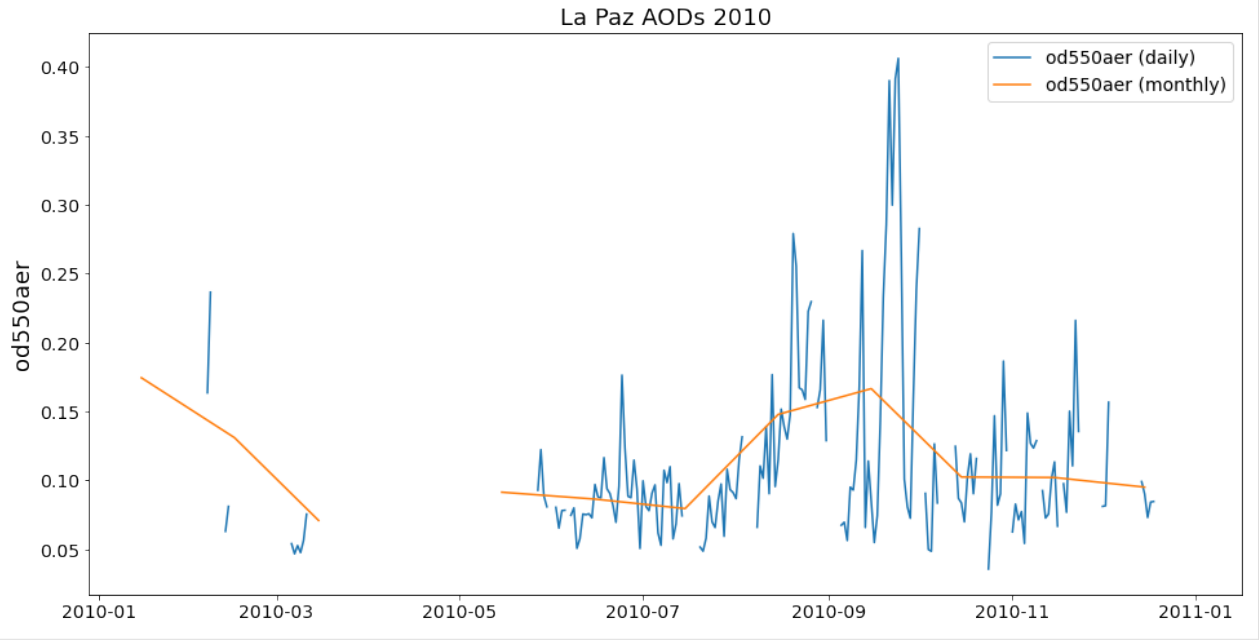
[56]:
```
<AxesSubplot:title={'center':'La_Paz'}, ylabel='od550aer'>
```



Back to `UngriddedData`: You may also retrieve the `StationData` with specifying more constraints using `to_station_data` (e.g. in monthly resolution and only for the year 2010). And you can overlay different curves,

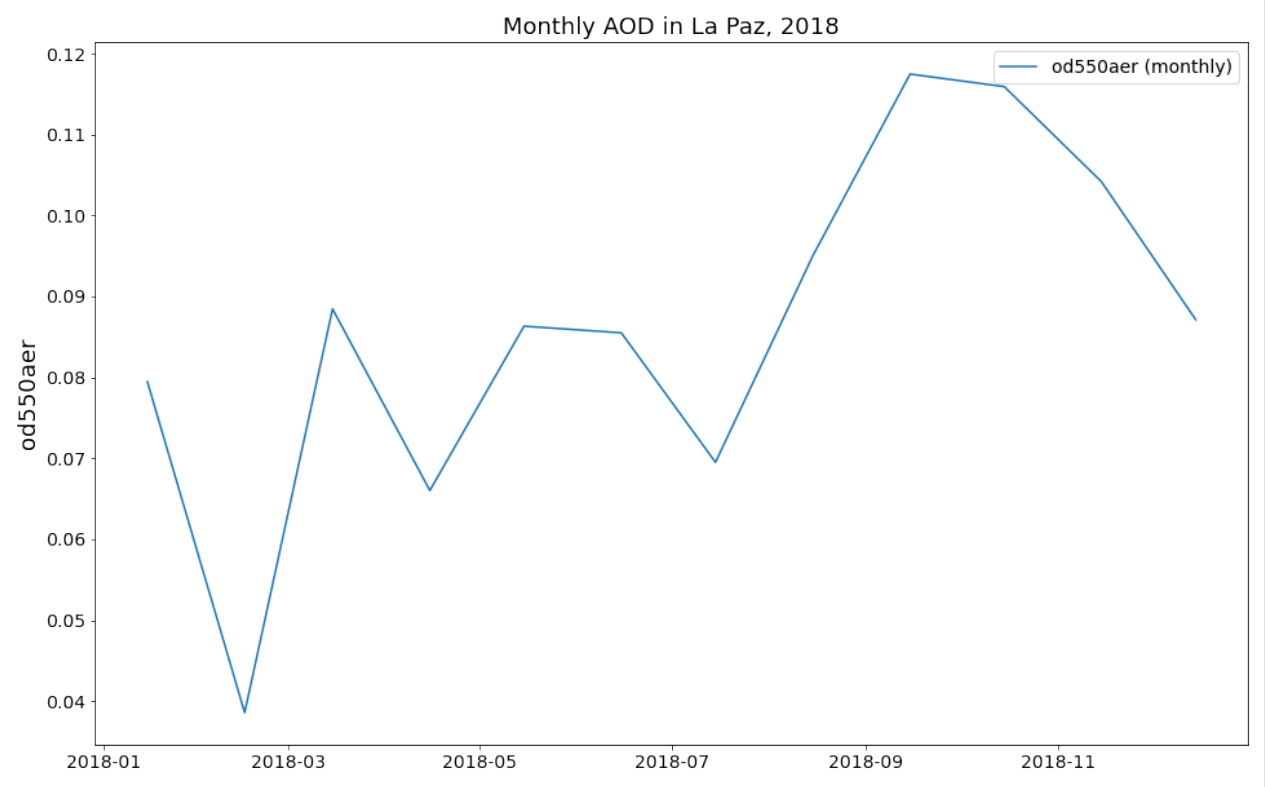
by passing the axes instance returned by the plotting method:

[57]:
```
ax=od550aer_aeronet.to_station_data('La_Paz',
                                    start=2010, stop=2011,
                                    freq='daily').plot_timeseries('od550aer')

ax=od550aer_aeronet.to_station_data('La_Paz',
                                    start=2010,
                                    freq='monthly').plot_timeseries('od550aer', ax=ax)
ax.legend()
ax.set_title('La Paz AODs 2010');
```



You can also plot the time-series directly from `UngriddedData`

[58]:
```
od550aer_aeronet.plot_station_timeseries('La_Paz', 'od550aer', ts_type='monthly',
                                         start=2018).set_title('Monthly AOD in La Paz, 2018
↪');
```



Computing trends (BETA API, will likely see some revisions)

Trends can be computed using the same methodolgy as Mortier et al., 2020, which is also used in the Aerosol trends interface. You may also read about the method in the *methods* section therein.

[59]:
```
te = pya.trends_engine.TrendsEngine
timeseries_monthly = station_data.resample_time('od550aer', ts_type='monthly')['od550aer
↪']
result = te.compute_trend(data=timeseries_monthly, start_year=2008, stop_year=2019, ts_
↪type='monthly', min_num_yrs=7)
result
```

[59]:
```
{'pval': 0.12097096961680295,
 'm': -0.0017985318700550548,
 'm_err': 0.0016087563002642336,
 'n': 11,
 'y_mean': 0.09864413175599861,
 'y_min': 0.07926316386912101,
 'y_max': 0.1684956149220456,
 'slp': -1.782064278905696,
 'slp_err': 1.6112865430184924,
 'reg0': 0.10092407391496962,
 'slp_2008': -1.782064278905696,
 'slp_2008_err': 1.6112865430184924,
 'reg0_2008': 0.10092407391496962,
 'data': 2008-06-15    0.091931
 2009-06-15    0.168496
```

(continues on next page)

(continued from previous page)

```
2010-06-15    0.113513
2011-06-15    0.099361
2012-06-15    0.086237
2013-06-15    0.079263
2014-06-15    0.094324
2015-06-15    0.081144
2016-06-15    0.090727
2017-06-15    0.093958
2018-06-15    0.086130
2019-06-15         NaN
dtype: float64,
'period': '2008-2019',
'season': 'all',
'yoffs': 0.1692682849770617}
```

Colocation of model and obsdata

Now that we have a **gridded model dataset** and an **ungridded observation dataset** loaded we can continue with colocation of both datasets. *Colocation* essentially describes the process of matching observations and model in space and time, which makes it possible to compare both and ultimately, to assess how well the model is performing.

As the observations are usually sparse, they define the set of locations and times to be extracted from the model (for comparison). Essentially, what needs to be done is simple:

1. Decide on a time interval in which you want to colocate the observations with the model data.
2. Decide on an output frequency.
3. Find all site location coordinates from the observations in the time period and extract the model values from the model dataset at these locations.
4. Match the time interval and frequency.

pyaerocom has some methods that can do this for you and these methods return an instance of the `ColocatedData <https://pyaerocom.met.no/api.html#pyaerocom.colocateddata.ColocatedData>`__ object.

Low-level colocation routine(s)

Let's colocate the TM5 model data with the AERONET AOD subset for the year 2010 and in monthly resolution. Since we already have both data objects loaded, we can go straight to the low-level colocation routine:

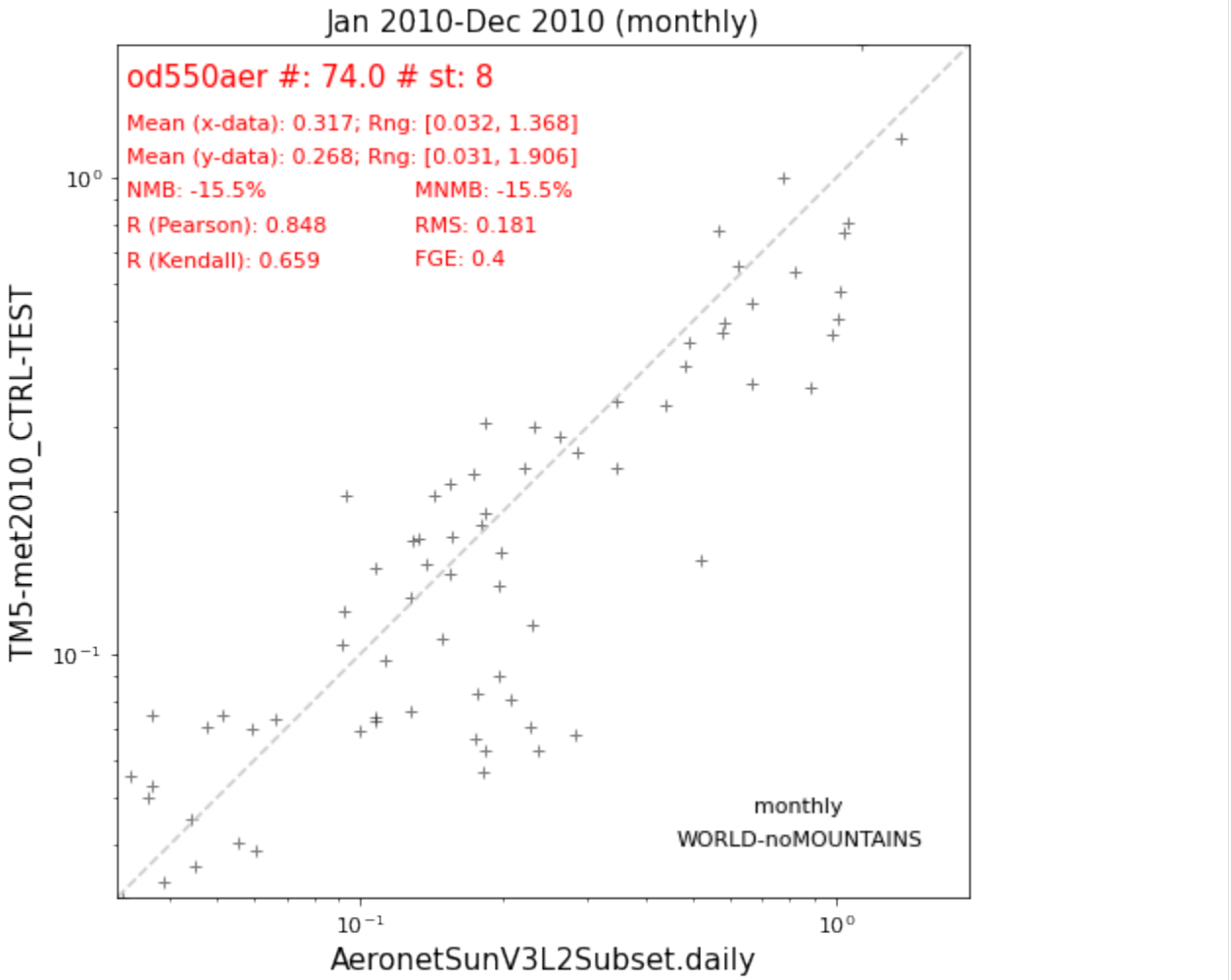
[60]:
```
coldata = pya.colocation.colocate_gridded_ungridded(od550aer_tm5,
                                                    od550aer_aeronet,
                                                    ts_type='monthly',
                                                    start=2010,
                                                    filter_name='WORLD-noMOUNTAINS')
```

```
/home/jonasg/github/pya/pyaerocom/pyaerocom/ungriddeddata.py:137: RuntimeWarning:␣
↪invalid value encountered in multiply
  self._data = np.empty([num_points, self._COLNO]) * np.nan
```

The filter-name `WORLD-noMOUNTAINS` denotes that all available AERONET sites are supposed to be used but high altitude sites (located above 1000m a.s.l). A more detailed introduction into available regions and region filters is provided in the getting_started_setup.ipynb tutorial.

You may create a scatter plot from these colocated monthly means, which includes relevant statistical parameters that help to assess model performance:

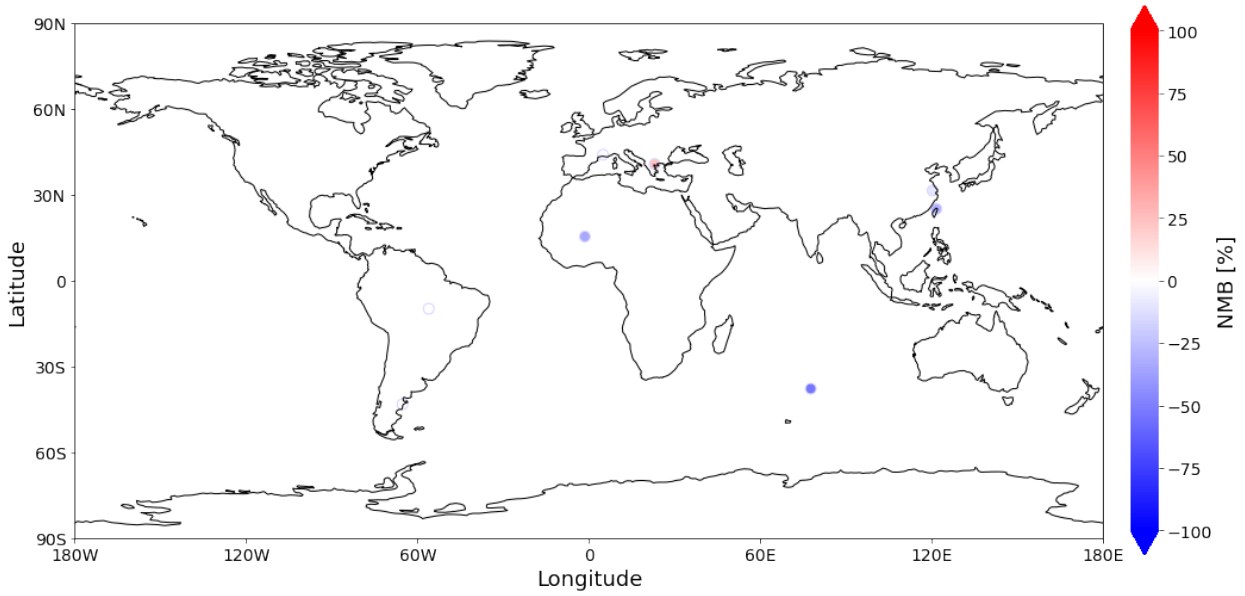
[61]:
```
coldata.plot_scatter(loglog=True);
```



Does not look too bad, you can see that this result is from 8 sites and 62 data points (monthly averages). The normalised-mean-bias (NMB) is -15.5%, which means that the model slightly underestimates AOD at these locations.

A more illustrative view of the model biases can be retrieved by plotting a *bias map*:

[62]:
```
pya.plot.mapping.plot_nmb_map_colocateddata(coldata);
```



The fact that you can barely see most of the sites is a good sign, since 0% bias is mapped to white color which is the same as the background color here. The largest bias is found in Amsterdam Island, in the southern Indian Ocean, which could be an indication that the model is simulating too little sea-salt aerosol in this very remote and clean region.

Under the hood ...

... the `ColocatedData` object is an `xarray.DataArray`:

[63]:
```
coldata.data
```

[63]:
```
<xarray.DataArray 'od550aer' (data_source: 2, time: 12, station_name: 8)>
array([[[       nan, 0.10766512, 0.03649583, 0.11300932,        nan,
                nan, 0.13828996, 0.03593528],
        [       nan, 0.1281071 , 0.05131321, 0.1490295 , 0.58044357,
         0.88151395, 0.09390731, 0.03669082],
        [       nan, 0.06650846, 0.05941147, 0.15400053, 0.77145611,
         0.81960458, 0.15375983, 0.03303893],
        [1.04184348, 0.09195735, 0.17511331, 0.17945707, 0.56584754,
         0.66320834, 0.26381093, 0.03862473],
        [1.05366664, 0.0922461 , 0.18108093, 0.13238964, 0.62159417,
         0.49044907, 0.18389999,        nan],
        [1.00704245,        nan, 0.23711979, 0.12935421, 1.01970277,
         0.43910105, 0.23285789,        nan],
        [0.98468077,        nan, 0.28315152, 0.15569135, 0.57586237,
                nan, 0.22223999,        nan],
        [0.34698175, 1.1291196 , 0.22721504, 0.1078829 , 0.6659021 ,
                nan, 0.28657127, 0.03157442],
        [0.34638257, 1.36759564, 0.22911738, 0.12730093, 0.48163878,
                nan, 0.17283706, 0.05573316],
        [       nan, 0.51974499, 0.19571329, 0.19495548,        nan,
                nan, 0.14301278, 0.06043777],
...
        [0.89635843, 0.07341532, 0.06992208, 0.14702028, 1.00326443,
```

(continues on next page)

(continued from previous page)

```
         0.63314253, 0.22872289, 0.05543199],
        [0.77087325, 0.1048945 , 0.06650965, 0.1865073 , 0.7776224 ,
         0.54631037, 0.28632328, 0.03336016],
        [0.80734402, 0.12378095, 0.05653326, 0.17420147, 0.65254241,
         0.45009974, 0.30526772, 0.03229909],
        [0.50654256, 0.14234523, 0.06274261, 0.17343882, 0.58004749,
         0.33348686, 0.30070606, 0.03174512],
        [0.46850282, 0.1863786 , 0.0675569 , 0.17624108, 0.47164536,
         0.19874078, 0.24717812, 0.04090422],
        [0.34062195, 1.90581691, 0.07018556, 0.15123938, 0.36946237,
         0.18504179, 0.26474383, 0.03088068],
        [0.24573354, 1.21507001, 0.11556577, 0.13191637, 0.4020009 ,
         0.26275966, 0.23841298, 0.04008143],
        [0.2798166 , 0.15733013, 0.08971586, 0.13991331, 0.38702768,
         0.34291422, 0.21531013, 0.03859388],
        [0.35316458, 0.08011972, 0.0626139 , 0.069165  , 0.55920124,
         0.38426778, 0.19802441, 0.03579372],
        [0.28721237, 0.0827378 , 0.07033022, 0.07274448, 0.50464362,
         0.3815313 , 0.1641538 , 0.04502364]]])
Coordinates:
  * data_source   (data_source) <U26 'AeronetSunV3L2Subset.daily' 'TM5-met201...
  * time          (time) datetime64[ns] 2010-01-15 2010-02-15 ... 2010-12-15
  * station_name  (station_name) <U16 'Agoufou' 'Alta_Floresta' ... 'Trelew'
    latitude      (station_name) float64 15.35 -9.871 -37.8 ... 40.63 -43.25
    longitude     (station_name) float64 -1.479 -56.1 77.57 ... 22.96 -65.31
    altitude      (station_name) float64 305.0 277.0 49.0 ... 26.0 60.0 15.0
Attributes: (12/16)
    data_source:     ['AeronetSunV3L2Subset.daily', 'TM5-met2010_CTRL-TEST']
    var_name:        ['od550aer', 'od550aer']
    var_name_input:  ['od550aer', 'od550aer']
    ts_type:         monthly
    filter_name:     WORLD-noMOUNTAINS
    ts_type_src:     ['daily', 'monthly']
    ...              ...
    from_files_ref:  None
    colocate_time:   False
    obs_is_clim:     False
    pyaerocom:       0.12.0.dev1
    min_num_obs:     None
    resample_how:    None
```

As you can see, model and obs (stored in `data_source` dimension) now share the same coordinates (dimension `station_name`) and time stamps (dimension `time`). The `data_source` dimension always contains the observation data at the first index and the model data at the second:

[64]:
```
obsdata = coldata.data[0]
obsdata
```

[64]:
```
<xarray.DataArray 'od550aer' (time: 12, station_name: 8)>
array([[       nan, 0.10766512, 0.03649583, 0.11300932,        nan,
                nan, 0.13828996, 0.03593528],
       [       nan, 0.1281071 , 0.05131321, 0.1490295 , 0.58044357,
        0.88151395, 0.09390731, 0.03669082],
```

(continues on next page)

(continued from previous page)

```
       [       nan, 0.06650846, 0.05941147, 0.15400053, 0.77145611,
        0.81960458, 0.15375983, 0.03303893],
       [1.04184348, 0.09195735, 0.17511331, 0.17945707, 0.56584754,
        0.66320834, 0.26381093, 0.03862473],
       [1.05366664, 0.0922461 , 0.18108093, 0.13238964, 0.62159417,
        0.49044907, 0.18389999,        nan],
       [1.00704245,        nan, 0.23711979, 0.12935421, 1.01970277,
        0.43910105, 0.23285789,        nan],
       [0.98468077,        nan, 0.28315152, 0.15569135, 0.57586237,
               nan, 0.22223999,        nan],
       [0.34698175, 1.1291196 , 0.22721504, 0.1078829 , 0.6659021 ,
               nan, 0.28657127, 0.03157442],
       [0.34638257, 1.36759564, 0.22911738, 0.12730093, 0.48163878,
               nan, 0.17283706, 0.05573316],
       [       nan, 0.51974499, 0.19571329, 0.19495548,        nan,
               nan, 0.14301278, 0.06043777],
       [       nan, 0.20634955, 0.18329662, 0.09995729,        nan,
               nan, 0.18381835, 0.04511684],
       [       nan, 0.17557466, 0.04773335, 0.10730821,        nan,
               nan, 0.19721314, 0.04412376]])
Coordinates:
    data_source   <U26 'AeronetSunV3L2Subset.daily'
  * time          (time) datetime64[ns] 2010-01-15 2010-02-15 ... 2010-12-15
  * station_name  (station_name) <U16 'Agoufou' 'Alta_Floresta' ... 'Trelew'
    latitude      (station_name) float64 15.35 -9.871 -37.8 ... 40.63 -43.25
    longitude     (station_name) float64 -1.479 -56.1 77.57 ... 22.96 -65.31
    altitude      (station_name) float64 305.0 277.0 49.0 ... 26.0 60.0 15.0
Attributes: (12/16)
    data_source:     ['AeronetSunV3L2Subset.daily', 'TM5-met2010_CTRL-TEST']
    var_name:        ['od550aer', 'od550aer']
    var_name_input:  ['od550aer', 'od550aer']
    ts_type:         monthly
    filter_name:     WORLD-noMOUNTAINS
    ts_type_src:     ['daily', 'monthly']
    ...              ...
    from_files_ref:  None
    colocate_time:   False
    obs_is_clim:     False
    pyaerocom:       0.12.0.dev1
    min_num_obs:     None
    resample_how:    None
```

[65]:
```
modeldata = coldata.data[1]
modeldata
```

[65]:
```
<xarray.DataArray 'od550aer' (time: 12, station_name: 8)>
array([[0.15579131, 0.074198  , 0.07480989, 0.09742171, 0.51915187,
        0.32987517, 0.15443519, 0.04990007],
       [0.18340141, 0.0763083 , 0.07439816, 0.10789451, 0.49576023,
        0.36209598, 0.21525712, 0.05284398],
       [0.89635843, 0.07341532, 0.06992208, 0.14702028, 1.00326443,
        0.63314253, 0.22872289, 0.05543199],
```

(continues on next page)

(continued from previous page)

```
       [0.77087325, 0.1048945 , 0.06650965, 0.1865073 , 0.7776224 ,
        0.54631037, 0.28632328, 0.03336016],
       [0.80734402, 0.12378095, 0.05653326, 0.17420147, 0.65254241,
        0.45009974, 0.30526772, 0.03229909],
       [0.50654256, 0.14234523, 0.06274261, 0.17343882, 0.58004749,
        0.33348686, 0.30070606, 0.03174512],
       [0.46850282, 0.1863786 , 0.0675569 , 0.17624108, 0.47164536,
        0.19874078, 0.24717812, 0.04090422],
       [0.34062195, 1.90581691, 0.07018556, 0.15123938, 0.36946237,
        0.18504179, 0.26474383, 0.03088068],
       [0.24573354, 1.21507001, 0.11556577, 0.13191637, 0.4020009 ,
        0.26275966, 0.23841298, 0.04008143],
       [0.2798166 , 0.15733013, 0.08971586, 0.13991331, 0.38702768,
        0.34291422, 0.21531013, 0.03859388],
       [0.35316458, 0.08011972, 0.0626139 , 0.069165  , 0.55920124,
        0.38426778, 0.19802441, 0.03579372],
       [0.28721237, 0.0827378 , 0.07033022, 0.07274448, 0.50464362,
        0.3815313 , 0.1641538 , 0.04502364]])
Coordinates:
    data_source   <U26 'TM5-met2010_CTRL-TEST'
  * time          (time) datetime64[ns] 2010-01-15 2010-02-15 ... 2010-12-15
  * station_name  (station_name) <U16 'Agoufou' 'Alta_Floresta' ... 'Trelew'
    latitude      (station_name) float64 15.35 -9.871 -37.8 ... 40.63 -43.25
    longitude     (station_name) float64 -1.479 -56.1 77.57 ... 22.96 -65.31
    altitude      (station_name) float64 305.0 277.0 49.0 ... 26.0 60.0 15.0
Attributes: (12/16)
    data_source:     ['AeronetSunV3L2Subset.daily', 'TM5-met2010_CTRL-TEST']
    var_name:        ['od550aer', 'od550aer']
    var_name_input:  ['od550aer', 'od550aer']
    ts_type:         monthly
    filter_name:     WORLD-noMOUNTAINS
    ts_type_src:     ['daily', 'monthly']
    ...              ...
    from_files_ref:  None
    colocate_time:   False
    obs_is_clim:     False
    pyaerocom:       0.12.0.dev1
    min_num_obs:     None
    resample_how:    None
```

High-level colocation routine

If it wasn't for the purpose of this notebook, normally, we don't want to go through the hassle of reading the data individually before colocating. Thus, pyaerocom has a high-level interface that can do colocation straight with the observation and model IDs (under the hood, of course, it uses the same routines that have been used here). By default, this high-level interface also stores all produced `ColocatedData` objects as NetCDF files, for later analysis:

[73]:
```
colocator = pya.Colocator(
    model_id=model_id, obs_id=obs_id, obs_vars='od550aer',
    ts_type='monthly',
    model_ts_type_read='monthly',
```

(continues on next page)

(continued from previous page)

```
    filter_name='OCN', #  let's try to better isolate Amsterdam Island
    reanalyse_existing=True,
    save_coldata=True)

colocator
```

[73]:
```
Colocator: {'model_id': 'TM5-met2010_CTRL-TEST', 'obs_id': 'AeronetSunV3L2Subset.daily',
↪'obs_vars': ['od550aer'], 'ts_type': 'monthly', 'start': None, 'stop': None, 'filter_
↪name': 'OCN', 'basedir_coldata': '/home/jonasg/MyPyaerocom/colocated_data', 'save_
↪coldata': True, 'obs_name': None, 'obs_data_dir': None, 'obs_use_climatology': False,
↪'_obs_cache_only': False, 'obs_vert_type': None, 'obs_ts_type_read': None, 'obs_filters
↪': {}, 'read_opts_ungridded': {}, 'model_name': None, 'model_data_dir': None, 'model_
↪vert_type_alt': None, 'model_read_opts': {}, 'model_use_vars': {}, 'model_rename_vars':
↪ {}, 'model_add_vars': {}, 'model_to_stp': False, 'model_ts_type_read': 'monthly',
↪'model_read_aux': {}, 'model_use_climatology': False, 'gridded_reader_id': {'model':
↪'ReadGridded', 'obs': 'ReadGridded'}, 'flex_ts_type': True, 'min_num_obs': None,
↪'resample_how': 'mean', 'obs_remove_outliers': False, 'model_remove_outliers': False,
↪'obs_outlier_ranges': {}, 'model_outlier_ranges': {}, 'zeros_to_nan': False,
↪'harmonise_units': False, 'regrid_res_deg': None, 'colocate_time': False, 'reanalyse_
↪existing': True, 'raise_exceptions': False, 'keep_data': True, 'add_meta': {}, '_log':␣
↪None, 'logging': True, '_loaded_model_data': {}, 'data': {}, '_processing_status': [],
↪'files_written': [], '_model_reader': None, '_obs_reader': None}
```

Quite a few options, a lot of them are for the even higher-level automatic web-processing tools that feed the Aerocom Evaluation websites, so let's not get lost in these details here.

The colocation can be run as follows:

[74]:
```
colocator.run()
```

```
Rearranging longitude dimension from 0 -> 360 definition to -180 -> 180 definition
WARNING: Found definition of outlier ranges for od550aer (TM5-met2010_CTRL-TEST) but␣
↪outlier removal is deactivated. Consider checking your setup (note: model or obs␣
↪outlier removal can be activated via attrs. model_remove_outliers and remove_outliers,␣
↪respectively
The following variable combinations will be colocated
MODEL-VAR       OBS-VAR
od550aer        od550aer
Running TM5-met2010_CTRL-TEST (od550aer) vs. AeronetSunV3L2Subset.daily (od550aer)
WRITE: /home/jonasg/MyPyaerocom/colocated_data/TM5-met2010_CTRL-TEST/od550aer_od550aer_
↪MOD-TM5-met2010_CTRL-TEST_REF-AeronetSunV3L2Subset.daily_20100101_20101231_monthly_OCN.
↪nc

Colocation processing status for TM5-met2010_CTRL-TEST vs. AeronetSunV3L2Subset.daily
  Model Var   Obs Var   Status
0  od550aer  od550aer  SUCCESS
```

[74]:
```
{'od550aer': {'od550aer': pyaerocom.ColocatedData: data: <xarray.DataArray 'od550aer'␣
↪(data_source: 2, time: 12, station_name: 1)>
  array([[[0.03649583],
          [0.05131321],
          [0.05941147],
          [0.17511331],
          [0.18108093],
```

(continues on next page)

(continued from previous page)

```
          [0.23711979],
          [0.28315152],
          [0.22721504],
          [0.22911738],
          [0.19571329],
          [0.18329662],
          [0.04773335]],

         [[0.07480989],
          [0.07439816],
          [0.06992208],
          [0.06650965],
          [0.05653326],
          [0.06274261],
          [0.0675569 ],
          [0.07018556],
          [0.11556577],
          [0.08971586],
          [0.0626139 ],
          [0.07033022]]])
  Coordinates:
    * data_source   (data_source) <U26 'AeronetSunV3L2Subset.daily' 'TM5-met201...
    * time          (time) datetime64[ns] 2010-01-15 2010-02-15 ... 2010-12-15
    * station_name  (station_name) <U16 'Amsterdam_Island'
      latitude      (station_name) float64 -37.8
      longitude     (station_name) float64 77.57
      altitude      (station_name) float64 49.0
  Attributes: (12/19)
      data_source:     ['AeronetSunV3L2Subset.daily', 'TM5-met2010_CTRL-TEST']
      var_name:        ['od550aer', 'od550aer']
      var_name_input:  ['od550aer', 'od550aer']
      ts_type:         monthly
      filter_name:     OCN
      ts_type_src:     ['daily', 'monthly']
      ...              ...
      pyaerocom:       0.12.0.dev1
      min_num_obs:     None
      resample_how:    mean
      model_name:      TM5-met2010_CTRL-TEST
      obs_name:        AeronetSunV3L2Subset.daily
      vert_code:       None}}
```

As you can see in the last line of the output, the colocated data object was stored as NetCDF file. The default direcory for these files can be accessed (and modified) in the `const` class:

[75]:
```
pya.const.COLOCATEDDATADIR
```

[75]:
```
'/home/jonasg/MyPyaerocom/colocated_data'
```

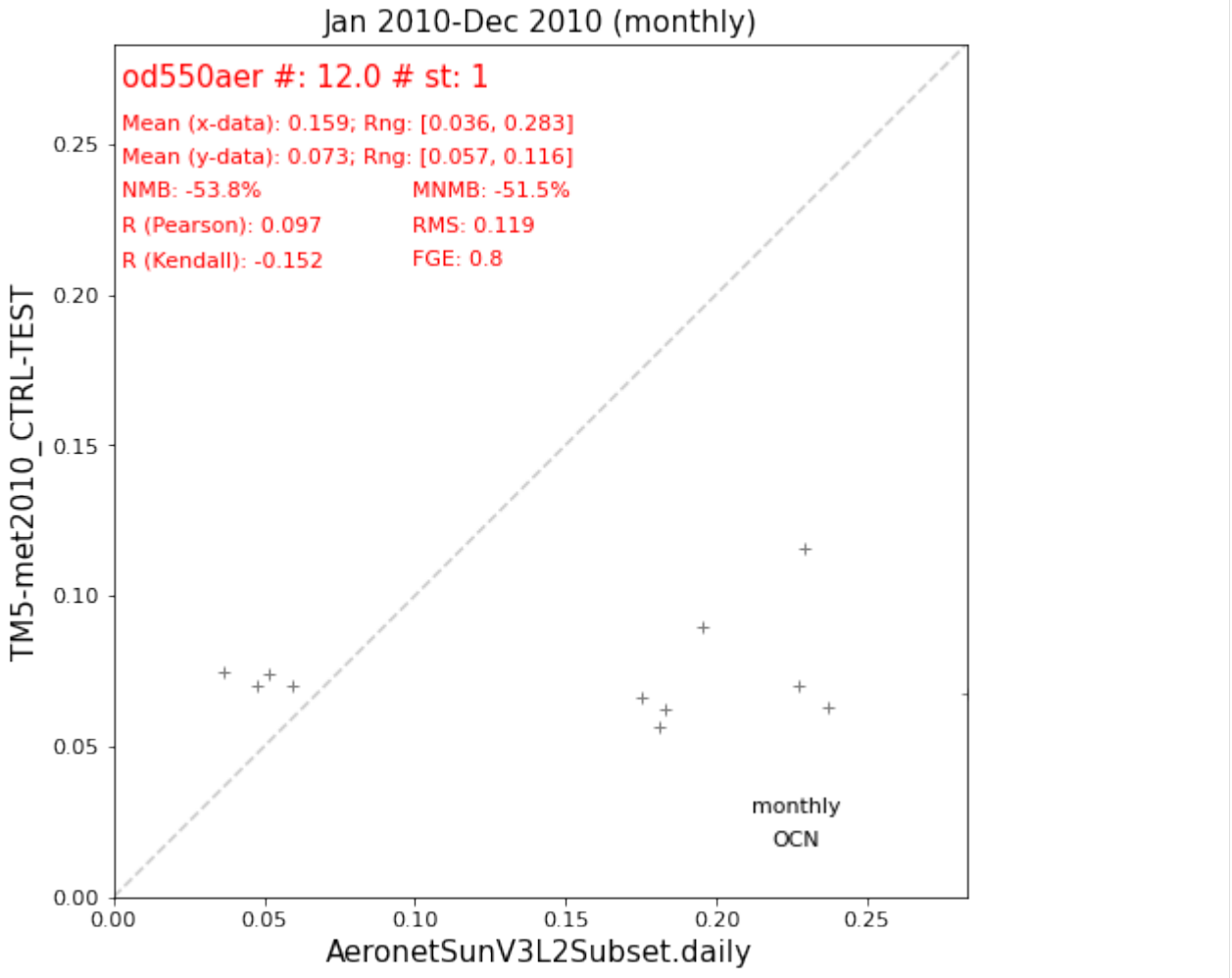
[76]:
```
import os
os.listdir(pya.const.COLOCATEDDATADIR)
```

[76]:
```
['bla',
 'test',
 'logfiles',
 'project',
 'AEROCOM-MEDIAN',
 'dev',
 'proj',
 'TM5-met2010_CTRL-TEST',
 'cams84']
```

And you can see that there is a subdirecory which contains all colocated data objects that have been created for the TM5 model. The loaded colocated data object can also be accessed via:

[85]:
```
coldata2 = colocator.data['od550aer']['od550aer']
```

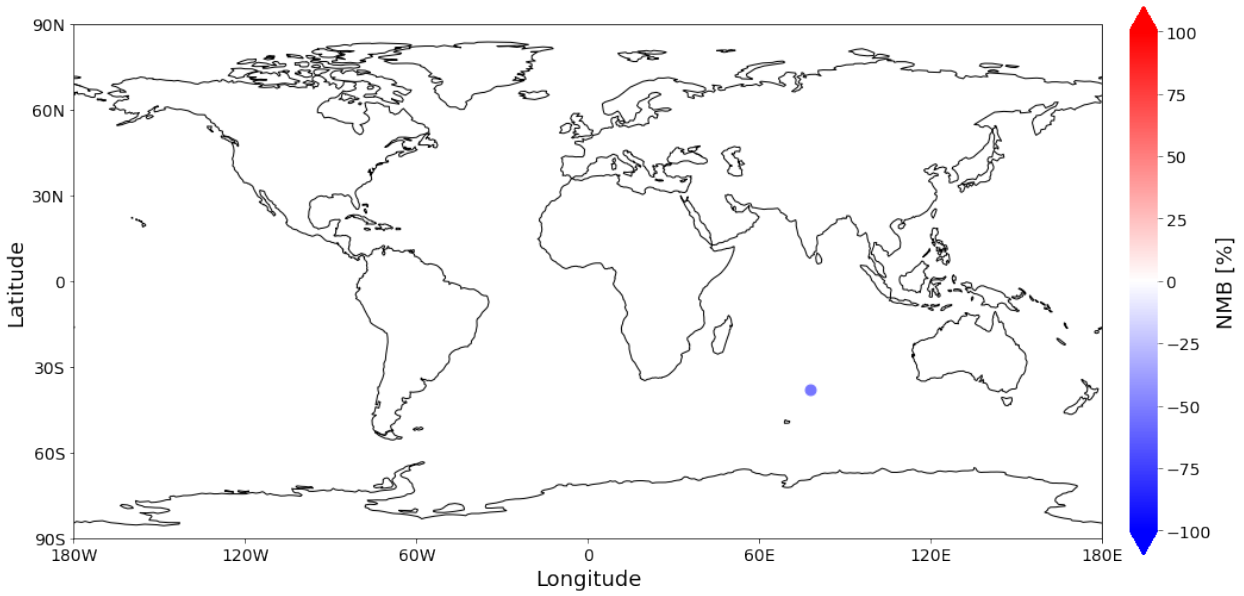
[86]:
```
coldata2.plot_scatter();
```



That looks like we managed to get only Amsterdam Island here (1 site) and 8 months of data. The corresponding model bias suggests that TM5 is underestimating AOD at 550nm at Amsterdam Island by ca 50%.

As a last step for this tutorial, let's make sure it is Amsterdam Island that we got here:

[87]:
```
pya.plot.mapping.plot_nmb_map_colocateddata(coldata2);
```



Looks like it! Ciao!

This section contains tutorials that are meant to help you getting started quickly with pyaerocom.

- Setup and data access | *getting_started_setup.ipynb*
- Basic data analysis | *getting_started_analysis.ipynb*

3.2 Outdated tutorials

3.2.1 Getting started

This notebook is meant to give a quick introduction into the main features and workflows of pyaerocom.

This includes brief introductions into the following features:

- Finding model and observation data on the AeroCom servers (method `browse_database`).
- Reading of **gridded** model data (`ReadGridded` class).
- Reading of **ungridded** observation data (`ReadUngridded` class).
- Working with **gridded** data (`GriddedData` class).
- Working with **ungridded** data (`UngriddedData` class).
- Retrieving and working with data from individual stations (`StationData` class).
- **Colocation** of model data with all stations from an observation network.

It ends with a colocation of CAM53-Oslo model AODs both all-sky and clear-sky with Aeronet Sun V3 level 2 data.

Prerequisites

- In order to run this notebook, you need to be connected to the AeroCom post processing servers (PPI).
- If you have PPI (/lustre/) mounted on your local machine you need to update the basic data directory after importing pyaerocom as follows: `import pyaerocom as pya pya.const.BASEDIR = <path_where_lustre_is_mounted>`

[1]:
```
import pyaerocom as pya
```
```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.021 s
```

Check data directory

By default, pyaerocom assumes that the AEROCOM database can be accessed (cf. top of flowchart), that is, it initiates all data query paths relative to the database server path names.

[2]:
```
pya.const.BASEDIR
```
[2]:
```
'/lustre/storeA/project/aerocom'
```

NOTE: Execution of the following lines will only work if you are connected to the AEROCOM data server or if you have access to the pyaerocom testdataset. The latter can be retrieved upon request (please contact jonasg@met.no).

Reading of and working with *gridded* model data (`ReadGridded` and `GriddedData` classes)

This section illustrates the reading of gridded data as well as some features of the `GriddedData` class of *pyaerocom*. First, however, we have to find a valid model ID for the reading (cf. flow chart).

Find model data

The database contains data from the CAM53-Oslo model, which is used in the following. You can use the `browse_database` function of pyaerocom to find model ID's (which can be quite cryptic sometimes) using wildcard pattern search.

[3]:
```
pya.browse_database('CAM53*-Oslo*UNTUNED*')
```
```

Pyaerocom ReadGridded
---------------------
Data ID: CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PI_UNTUNED
Data directory: /lustre/storeA/project/aerocom/aerocom2/NorESM_SVN_TEST/CAM53-Oslo_7310_
↪MG15CLM45_5feb2017IHK_53OSLO_PI_UNTUNED/renamed
Available experiments: ['UNTUNED']
Available years: [9999]
```

(continues on next page)

(continued from previous page)

```
Available frequencies ['monthly']
Available variables: ['abs440aer', 'abs440aercs', 'abs500aer', 'abs550i3Daer', 'abs550aer
↪', 'abs550bc', 'abs550dryaer', 'abs550dust', 'abs550oa', 'abs550so4', 'abs550ss',
↪'abs670aer', 'abs870aer', 'airmass', 'area', 'asy3Daer', 'bc550i3Daer', 'cheaqpso4',
↪'chegpso4', 'chepso2', 'cl3D', 'clt', 'drybc', 'drydms', 'drydust', 'dryoa', 'dryso2',
↪'dryso4', 'dryss', 'ec550i3Daer', 'ec550dryaer', 'emibc', 'emidms', 'emidust', 'emioa',
↪'emiso2', 'emiso4', 'emiss', 'hus', 'landf', 'loadbc', 'loaddms', 'loaddust', 'loadoa',
↪ 'loadso2', 'loadso4', 'loadss', 'mmraerh2o', 'mmrbc', 'mmrdu', 'mmroa', 'mmrso4',
↪'mmrss', 'od440aer', 'od440csaer', 'od550aer', 'od550aerh2o', 'od550bc', 'od550csaer',
↪'od550dust', 'od550lt1aer', 'od550lt1dust', 'od550oa', 'od550so4', 'od550ss', 'od670aer
↪', 'od870aer', 'od870csaer', 'orog', 'precip', 'pressure', 'ps', 'rlds', 'rlus', 'rlut
↪', 'rlutcs', 'rsds', 'rsdscs', 'rsdt', 'rsus', 'rsut', 'sconcbc', 'sconcdms',
↪'sconcdust', 'sconcoa', 'sconcso2', 'sconcso4', 'sconcss', 'temp', 'vmrdms', 'vmrso2',
↪'wetbc', 'wetdms', 'wetdust', 'wetoa', 'wetso2', 'wetso4', 'wetss']
```

```
Pyaerocom ReadGridded
---------------------
Data ID: CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED
Data directory: /lustre/storeA/project/aerocom/aerocom2/NorESM_SVN_TEST/CAM53-Oslo_7310_
↪MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED/renamed
Available experiments: ['UNTUNED']
Available years: [2004, 2005, 2006, 2007, 2008, 2009, 2010, 9999]
Available frequencies ['monthly']
Available variables: ['abs440aer', 'abs440aercs', 'abs500aer', 'abs550i3Daer', 'abs550aer
↪', 'abs550aercs', 'abs550bc', 'abs550dryaer', 'abs550dust', 'abs550oa', 'abs550so4',
↪'abs550ss', 'abs670aer', 'abs870aer', 'airmass', 'ang4487aer', 'ang4487csaer', 'area',
↪'asy3Daer', 'bc550i3Daer', 'cheaqpso4', 'chegpso4', 'chepso2', 'cl3D', 'clt', 'drybc',
↪'drydms', 'drydust', 'dryoa', 'dryso2', 'dryso4', 'dryss', 'ec550i3Daer', 'ec550dryaer',
↪ 'emibc', 'emidms', 'emidust', 'emioa', 'emiso2', 'emiso4', 'emiss', 'hus', 'landf',
↪'loadbc', 'loaddms', 'loaddust', 'loadoa', 'loadso2', 'loadso4', 'loadss', 'mmraerh2o',
↪ 'mmrbc', 'mmrdu', 'mmroa', 'mmrso4', 'mmrss', 'od440aer', 'od440csaer', 'od550aer',
↪'od550aerh2o', 'od550bc', 'od550csaer', 'od550dust', 'od550lt1aer', 'od550lt1dust',
↪'od550oa', 'od550so4', 'od550ss', 'od670aer', 'od870aer', 'od870csaer', 'orog', 'precip
↪', 'pressure', 'ps', 'rlds', 'rlus', 'rlut', 'rlutcs', 'rsds', 'rsdscs', 'rsdt', 'rsus
↪', 'rsut', 'sconcbc', 'sconcdms', 'sconcdust', 'sconcoa', 'sconcso2', 'sconcso4',
↪'sconcss', 'temp', 'vmrdms', 'vmrso2', 'wetbc', 'wetdms', 'wetdust', 'wetoa', 'wetso2',
↪ 'wetso4', 'wetss']
```

Initiate reader class for the present day (PD) dataset

[4]:
```
import warnings
warnings.filterwarnings('ignore')
reader = pya.io.ReadGridded('CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED')
```

You can have a look at the individual files and corresponding metadata using the `file_info` attribute:

[5]:
```
reader.file_info
```

[5]:
```
      var_name  year  ts_type vert_code  \
345  abs440aer  2004  monthly    Column
```

(continues on next page)

(continued from previous page)

```
506  abs440aer  2005  monthly    Column
340  abs440aer  2006  monthly    Column
12   abs440aer  2007  monthly    Column
685  abs440aer  2008  monthly    Column
..         ...   ...      ...       ...
169      wetss  2007  monthly   Surface
213      wetss  2008  monthly   Surface
36       wetss  2009  monthly   Surface
129      wetss  2010  monthly   Surface
454      wetss  9999  monthly   Surface

                                                data_id   name meteo  \
345  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
506  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
340  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
12   CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
685  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
..                                                 ...    ...   ...
169  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
213  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
36   CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
129  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
454  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo

    experiment  is_at_stations     3D  \
345    UNTUNED           False  False
506    UNTUNED           False  False
340    UNTUNED           False  False
12     UNTUNED           False  False
685    UNTUNED           False  False
..         ...             ...    ...
169    UNTUNED           False  False
213    UNTUNED           False  False
36     UNTUNED           False  False
129    UNTUNED           False  False
454    UNTUNED           False  False

                                             filename
345  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
506  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
340  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
12   aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
685  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
..                                                 ...
169  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
213  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
36   aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
129  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
454  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...

[808 rows x 11 columns]
```

If this contains too much information (as is the case here), you can also filter this attribute based on what you are

interested in. E.g.:

[6]:
```
reader.filter_files(var_name='od550aer')
```

[6]:
```
     var_name  year  ts_type vert_code  \
199  od550aer  2004  monthly    Column
414  od550aer  2005  monthly    Column
44   od550aer  2006  monthly    Column
333  od550aer  2007  monthly    Column
46   od550aer  2008  monthly    Column
679  od550aer  2009  monthly    Column
352  od550aer  2010  monthly    Column

                                              data_id   name meteo  \
199  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
414  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
44   CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
333  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
46   CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
679  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo
352  CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_P...  CAM53  Oslo

    experiment  is_at_stations     3D  \
199    UNTUNED           False  False
414    UNTUNED           False  False
44     UNTUNED           False  False
333    UNTUNED           False  False
46     UNTUNED           False  False
679    UNTUNED           False  False
352    UNTUNED           False  False

                                             filename
199  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
414  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
44   aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
333  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
46   aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
679  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
352  aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK...
```

Read Aerosol optical depth at 550 nm

Import both clear-sky (*cs* in variable name) and all-sky data.

[7]:
```
od550aer = reader.read_var('od550aer')
od550csaer = reader.read_var('od550csaer')
```

Both data objects are instances of class GriddedData which is based on the Cube class (iris library) and features very similar functionality and more.

Some of these features are introduced below.

Overview of what is in the data

Simply print the object.

[8]:
```
print(od550aer)
```

```
pyaerocom.GriddedData: CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED
Grid data: Aerosol optical depth at 500nm / (1) (time: 84; latitude: 192; longitude: 288)
     Dimension coordinates:
          time                           x             -             -
          latitude                       -             x             -
          longitude                      -             -             x
     Attributes:
          Conventions: CF-1.0
          NCO: 4.3.7
          Version: $Name$
          case: 53OSLO_PD_UNTUNED
          computed: False
          concatenated: True
          data_id: CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED
          from_files: ['/lustre/storeA/project/aerocom/aerocom2/NorESM_SVN_TEST/CAM53-
↪Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED/renamed/aerocom3_CAM53-Oslo_7310_
↪MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED_od550aer_Column_2004_monthly.nc',...
          history: Thu Feb  9 11:05:21 2017: ncatted -O -a units,od550aer,o,c,1 /
↪projects/NS2345K/CAM-Oslo/DO_AEROCOM/CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_
↪UNTUNED/renamed/aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED_
↪od550aer_Column_2004_monthly.nc
Thu...
          host: hexagon-2
          initial_file: /work/shared/noresm/inputdata/atm/cam/inic/fv/cami-mam3_0000-01-
↪01_0.9...
          logname: ihkarset
          nco_openmp_thread_number: 1
          outliers_removed: False
          reader: None
          region: None
          regridded: False
          revision_Id: $Id$
          source: CAM
          title: UNSET
          topography_file: /work/shared/noresm/inputdata/noresm-only/inputForNudging/ERA_
↪f09f09_3...
          ts_type: monthly
          var_name_read: n/d
     Cell methods:
          mean: time
```

[9]:
```
print(od550csaer)
```

```
pyaerocom.GriddedData: CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED
Grid data: Clear air Aerosol optical depth at 550nm / (1) (time: 84; latitude: 192;␣
↪longitude: 288)
     Dimension coordinates:
          time                                x             -             -
```

(continues on next page)

(continued from previous page)

```
          latitude                                  -             x             -
          longitude                                 -             -             x
     Attributes:
          Conventions: CF-1.0
          NCO: 4.3.7
          Version: $Name$
          case: 53OSLO_PD_UNTUNED
          computed: False
          concatenated: True
          data_id: CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED
          from_files: ['/lustre/storeA/project/aerocom/aerocom2/NorESM_SVN_TEST/CAM53-
↪Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED/renamed/aerocom3_CAM53-Oslo_7310_
↪MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED_od550csaer_Column_2004_monthly.nc',...
          history: Thu Feb  9 11:05:16 2017: ncatted -O -a units,od550csaer,o,c,1 /
↪projects/NS2345K/CAM-Oslo/DO_AEROCOM/CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_
↪UNTUNED/renamed/aerocom3_CAM53-Oslo_7310_MG15CLM45_5feb2017IHK_53OSLO_PD_UNTUNED_
↪od550csaer_Column_2004_monthly.nc
Thu...
          host: hexagon-2
          initial_file: /work/shared/noresm/inputdata/atm/cam/inic/fv/cami-mam3_0000-01-
↪01_0.9...
          logname: ihkarset
          nco_openmp_thread_number: 1
          outliers_removed: False
          reader: None
          region: None
          regridded: False
          revision_Id: $Id$
          source: CAM
          title: UNSET
          topography_file: /work/shared/noresm/inputdata/noresm-only/inputForNudging/ERA_
↪f09f09_3...
          ts_type: monthly
          var_name_read: n/d
     Cell methods:
          mean: time
```

Access time stamps

Time stamps are represented as numerical values with respect to a reference date and frequency, according to the CF conventions. They can be accessed via the `time` attribute of the data class.

[10]:
```
od550aer.time
```

[10]:
```
DimCoord(array([   0.,   31.,   60.,   91.,  121.,  152.,  182.,  213.,  244.,
        274.,  305.,  335.,  366.,  397.,  425.,  456.,  486.,  517.,
        547.,  578.,  609.,  639.,  670.,  700.,  731.,  762.,  790.,
        821.,  851.,  882.,  912.,  943.,  974., 1004., 1035., 1065.,
       1096., 1127., 1155., 1186., 1216., 1247., 1277., 1308., 1339.,
       1369., 1400., 1430., 1461., 1492., 1521., 1552., 1582., 1613.,
       1643., 1674., 1705., 1735., 1766., 1796., 1827., 1858., 1886.,
```

(continues on next page)

(continued from previous page)

```
       1917., 1947., 1978., 2008., 2039., 2070., 2100., 2131., 2161.,
       2192., 2223., 2251., 2282., 2312., 2343., 2373., 2404., 2435.,
       2465., 2496., 2526.]), standard_name='time', units=Unit('days since 2004-01-01 00:
↪00:00', calendar='gregorian'), long_name='Time', var_name='time')
```

You may also want the time-stamps in the form of actual datetime-like objects. These can be computed using the `time_stamps()` method:

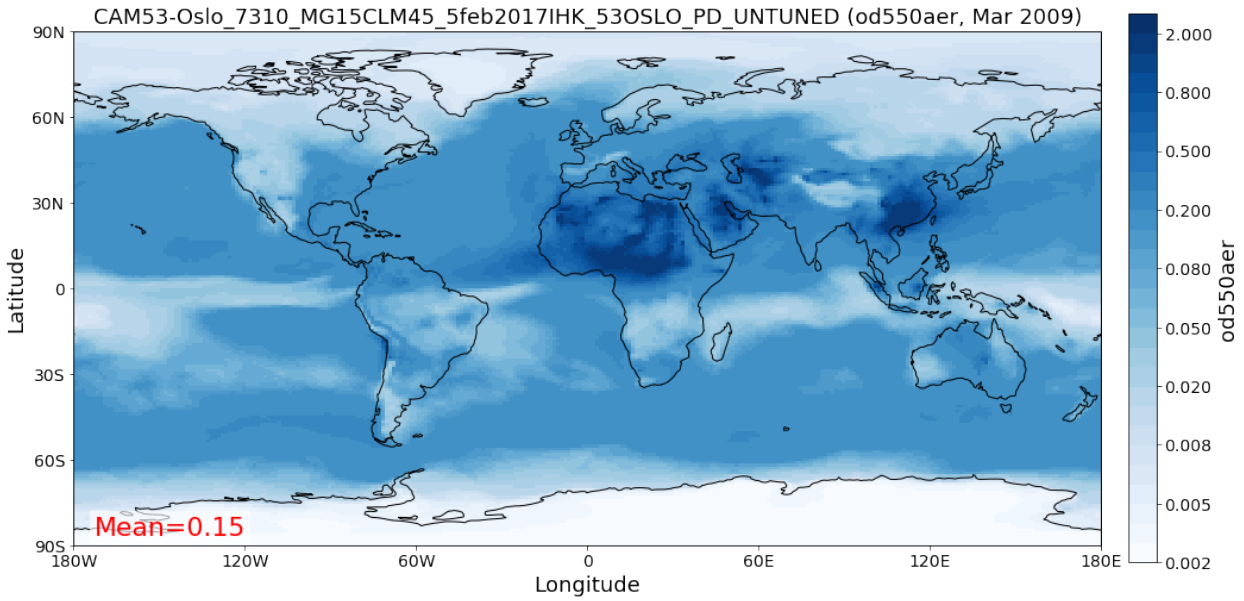
[11]:
```
od550aer.time_stamps()[0:3]
```

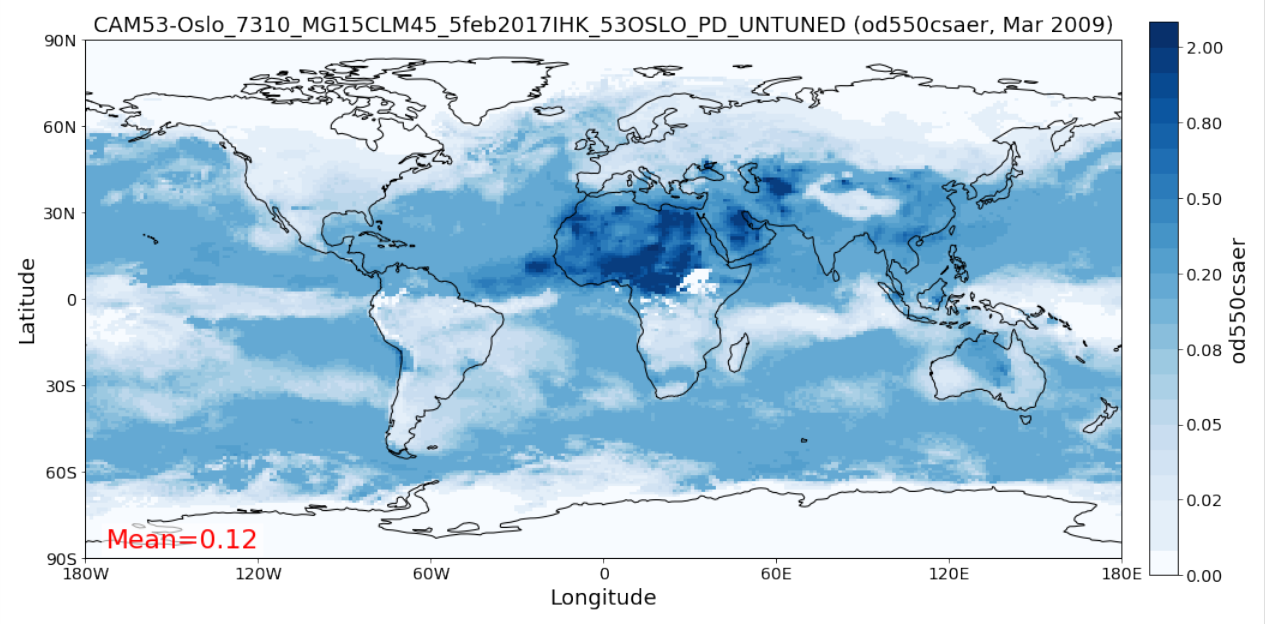
[11]:
```
array(['2004-01-01T00:00:00.000000', '2004-02-01T00:00:00.000000',
       '2004-03-01T00:00:00.000000'], dtype='datetime64[us]')
```

Plotting maps

Maps of individual time stamps can be plotted using the quickplot_map method.

[12]:
```
fig1 = od550aer.quickplot_map('2009-3-15')
fig2 = od550csaer.quickplot_map('2009-3-15')
```





Filtering

Regional filtering can be performed using the Filter class (cf. flowchart above).

An overview of available default regions can be accessed via:

[13]:
```
print(pya.region.get_all_default_region_ids())
```

```
['WORLD', 'EUROPE', 'ASIA', 'AUSTRALIA', 'CHINA', 'INDIA', 'NAFRICA', 'SAFRICA',
↪'SAMERICA', 'NAMERICA']
```

Now let's go for north Africa. Create instance of Filter class:

[14]:
```
f = pya.Filter('NAFRICA')
f
```

[14]:
```
Filter([('_name', 'NAFRICA-wMOUNTAINS'),
        ('_region',
         Region NAFRICA Region([('_name', 'NAFRICA'), ('lon_range', [-20, 50]), ('lat_
↪range', [0, 40]), ('lon_range_plot', [-20, 50]), ('lat_range_plot', [0, 40]), ('lon_
↪ticks', None), ('lat_ticks', None)])),
        ('lon_range', [-20, 50]),
        ('lat_range', [0, 40]),
        ('alt_range', None)])
```

... and apply to the two data objects (this can be done by calling the filter with the corresponding data class as input parameter):

[15]:
```
od550aer_nafrica = f(od550aer)
od550csaer_nafrica = f(od550csaer)
```

```
Applying regional cropping in GriddedData using Filter class. Note that this does not␣
↪yet include potential cropping in the vertical dimension. Coming soon...
```

```
Applying regional cropping in GriddedData using Filter class. Note that this does not␣
↪yet include potential cropping in the vertical dimension. Coming soon...
```

Compare shapes:

[16]:
```
od550aer_nafrica
```

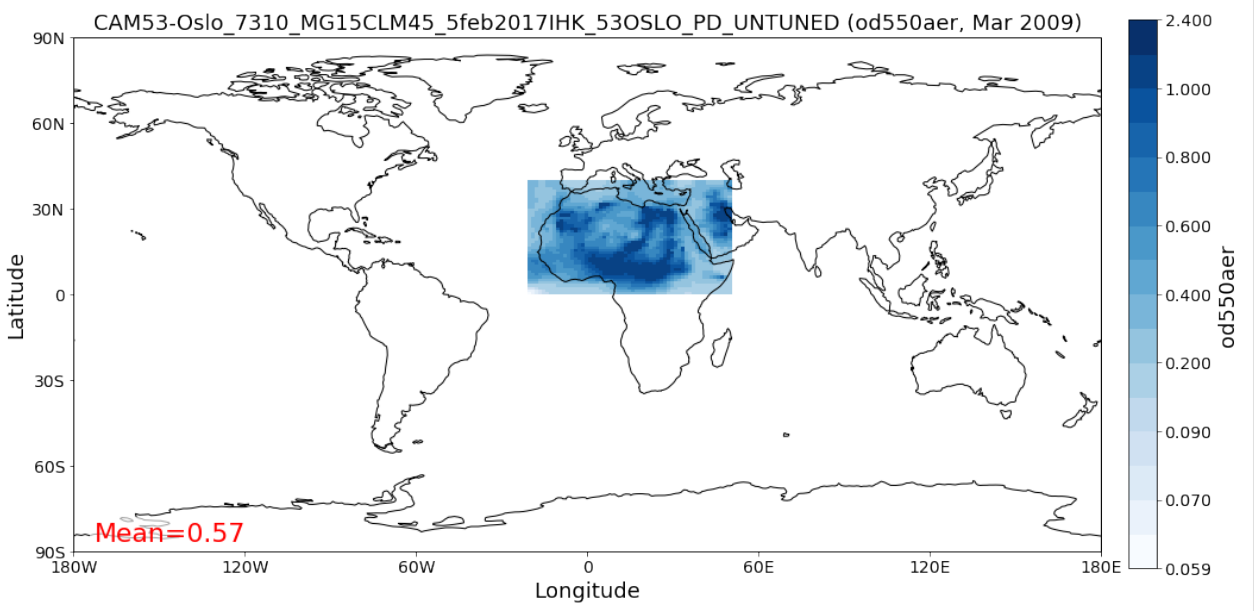
[16]:
```
pyaerocom.GriddedData
Grid data: <iris 'Cube' of Aerosol optical depth at 500nm / (1) (time: 84; latitude: 42;␣
↪longitude: 57)>
```

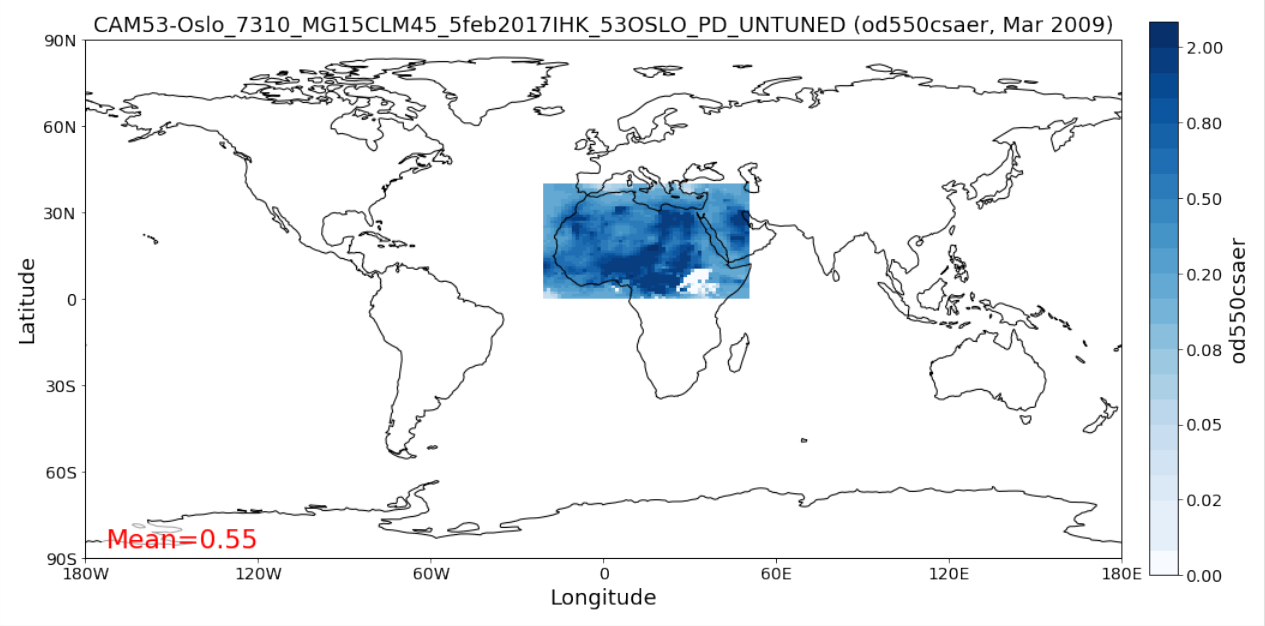
[17]:
```
od550aer
```

[17]:
```
pyaerocom.GriddedData
Grid data: <iris 'Cube' of Aerosol optical depth at 500nm / (1) (time: 84; latitude: 192;
↪ longitude: 288)>
```

As you can see, the filtered object is reduced in the longitude and latitude dimension. Let's plot the two new objects:

[18]:
```
ax1 = od550aer_nafrica.quickplot_map('2009-3-15')
ax2 = od550csaer_nafrica.quickplot_map('2009-3-15')
```





Filtering of time

Filtering of time is not yet included in the Filter class but can be easily performed from the `GriddedData` object directly. If you know the indices of the time stamps you want to crop, you can simply use numpy indexing syntax (remember that we have a 3D array containing time, latitude and lonfgitude).

Let's say we want to filter the **year 2009**.

Since the time dimension corresponds the first index in the 3D data (time, lat, lon), and since we know, that we have monthly data from 2008-2010 (see above), we may use

[19]:
```
od550aer_nafrica_2009 = od550aer_nafrica[12:24]
od550aer_nafrica_2009.time_stamps()
```

[19]:
```
array(['2005-01-01T00:00:00.000000', '2005-02-01T00:00:00.000000',
       '2005-03-01T00:00:00.000000', '2005-04-01T00:00:00.000000',
       '2005-05-01T00:00:00.000000', '2005-06-01T00:00:00.000000',
       '2005-07-01T00:00:00.000000', '2005-08-01T00:00:00.000000',
       '2005-09-01T00:00:00.000000', '2005-10-01T00:00:00.000000',
       '2005-11-01T00:00:00.000000', '2005-12-01T00:00:00.000000'],
      dtype='datetime64[us]')
```

in order to extract the year 2009.

However, this methodology might not always be handy (imagine you have a 10 year dataset of `3hourly` sampled data and want to extract three months in the 6th year ...). In that case, you can perform the cropping using the actual timestamps (for comparibility, let's stick to 2009 here):

[20]:
```
od550aer_nafrica_2009_alt = od550aer_nafrica.crop(time_range=('1-1-2009', '1-1-2010'))
od550aer_nafrica_2009.time_stamps()
```

[20]:
```
array(['2005-01-01T00:00:00.000000', '2005-02-01T00:00:00.000000',
       '2005-03-01T00:00:00.000000', '2005-04-01T00:00:00.000000',
       '2005-05-01T00:00:00.000000', '2005-06-01T00:00:00.000000',
```

(continues on next page)

(continued from previous page)

```
       '2005-07-01T00:00:00.000000', '2005-08-01T00:00:00.000000',
       '2005-09-01T00:00:00.000000', '2005-10-01T00:00:00.000000',
       '2005-11-01T00:00:00.000000', '2005-12-01T00:00:00.000000'],
      dtype='datetime64[us]')
```

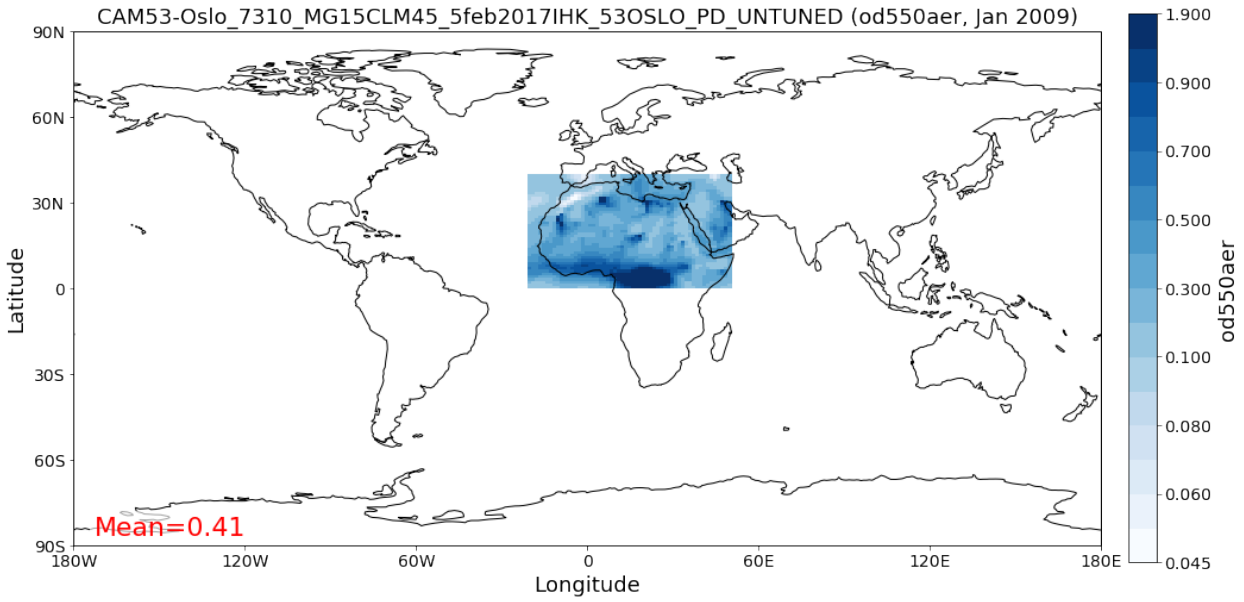
Data aggregation

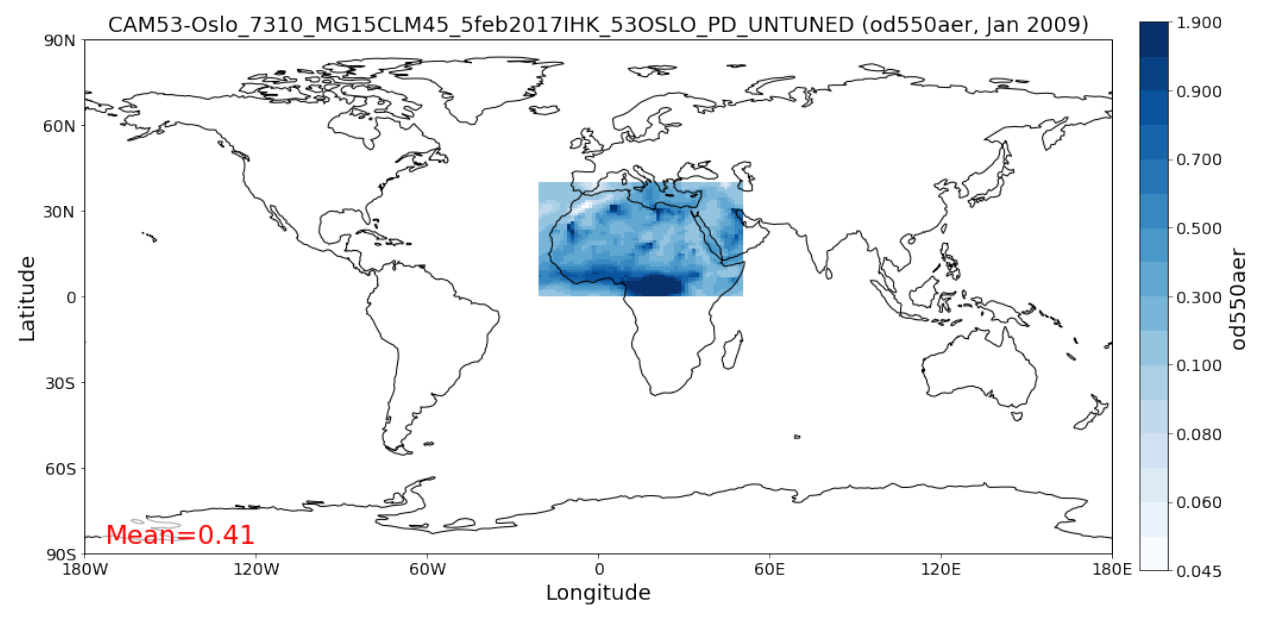
Let's say we want to compute yearly means for each of the 3 years. In this case we can simply call the `downscale_time` method:

[21]:
```
od550aer_nafrica.downscale_time('yearly')
od550aer_nafrica.quickplot_map('2009')
```

```
This method is deprecated. Please use new name resample_time
```

[21]:





Note: seasonal aggregation is not yet implemented in pyaerocom but will follow soon.

In the following section the reading of ungridded data is illustrated based on the example of AERONET version 3 (level 2) data. The test dataset contains a randomly picked subset of 100 Aeronet stations. Aeronet provides different products,

Reading of and working with ungridded data (`ReadUngridded` and `UngriddedData` classes)

Ungridded data in pyaerocom refers to data that is available in the form of *files per station* and that is not sampled in a manner that it would make sense to translate into a rgular gridded format such as the previously introduced `GriddedData` class.

Data from the AERONET network (that is introduced in the following), for instance, is provided in the form of column seperated text files per measurement station, where columns correspond to different variables and data rows to individual time stamps. Needless to say that the time stamps (or the covered periods) vary from station to station.

The basic workflow for reading of ungridded data, such as Aeronet data, is very similar to the reading of gridded data (comprising a reading class that handles a query and returns a data class, here UngriddedData (see also flow chart above).

Before we can continue with the data import, some things need to be said related to the caching of `UngriddedData` objects.

Caching of UngriddedData

Reading of ungridded data is often rather time-consuming. Therefore, pyaerocom uses a caching strategy that stores loaded instances of the `UngriddedData` class as pickle files in a cache directory (illustrated in the left hand side of the flowchart shown above). The loaction of the cache directory can be accessed via:

```
[22]: pya.const.CACHEDIR
```

```
[22]: '/home/jonasg/MyPyaerocom/_cache/jonasg'
```

You may change this directory if required.

[23]:
```
print('Caching is active? {}'.format(pya.const.CACHING))
```

```
Caching is active? True
```

Deactivate / Activate caching

[24]:
```
pya.const.CACHING = False
```

[25]:
```
pya.const.CACHING = True
```

Note: if caching is active, make sure you have enough disk quota or change location where the files are stored.

Read Aeronet Sun v3 level 2 data

As illustrated in the flowchart above, ungridded observation data can be imported using the `ReadUngridded` class. The reading class requires an ID for the observation network that is supposed to be read. Let's find the right ID for these data:

[26]:
```
pya.browse_database('Aeronet*V3*Lev2*')
```

```
Dataset name: AeronetSunV3Lev2.daily
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev2.
↪0.daily/renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang4487aer_calc', 'od550aer']
Last revision: 20190920

Dataset name: AeronetSunV3Lev2.AP
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev2.
↪0.AP/renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang4487aer_calc', 'od550aer']
Last revision: 20190511

Dataset name: AeronetSDAV3Lev2.daily
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Aeronet.SDA.V3L2.
↪0.daily/renamed
Supported variables: ['od500gt1aer', 'od500lt1aer', 'od500aer', 'ang4487aer', 'od550aer',
↪ 'od550gt1aer', 'od550lt1aer']
Last revision: 20190920
Reading failed for AeronetSDAV3Lev2.AP. Error: NetworkNotImplemented('No reading class␣
↪available yet for dataset AeronetSDAV3Lev2.AP')

Dataset name: AeronetInvV3Lev2.daily
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/Aeronet.Inv.V3L2.
↪0.daily/renamed
Supported variables: ['abs440aer', 'angabs4487aer', 'od440aer', 'ang4487aer', 'abs550aer
↪', 'od550aer']
Last revision: 20190914
```

It found one match and the dataset ID is *AeronetSunV3Lev2.daily*. It also tells us what variables can be loaded via the interface.

Note: You can safely ignore all the warnings in the output. These are due to the fact that the testdata set does not contain all observation networks that are available in the AEROCOM database.

[27]:
```
obs_reader = pya.io.ReadUngridded('AeronetSunV3Lev2.daily')
print(obs_reader)
```

```
Dataset name: AeronetSunV3Lev2.daily
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev2.
↪0.daily/renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang4487aer_calc', 'od550aer']
Last revision: 20190920
```

Let's read the data (you can read a single or multiple variables at the same time). For now, we only read the AOD at 550 nm:

[28]:
```
aeronet_data = obs_reader.read(vars_to_retrieve='od550aer')
type(aeronet_data) #displays data type
```

[28]:
```
pyaerocom.ungriddeddata.UngriddedData
```

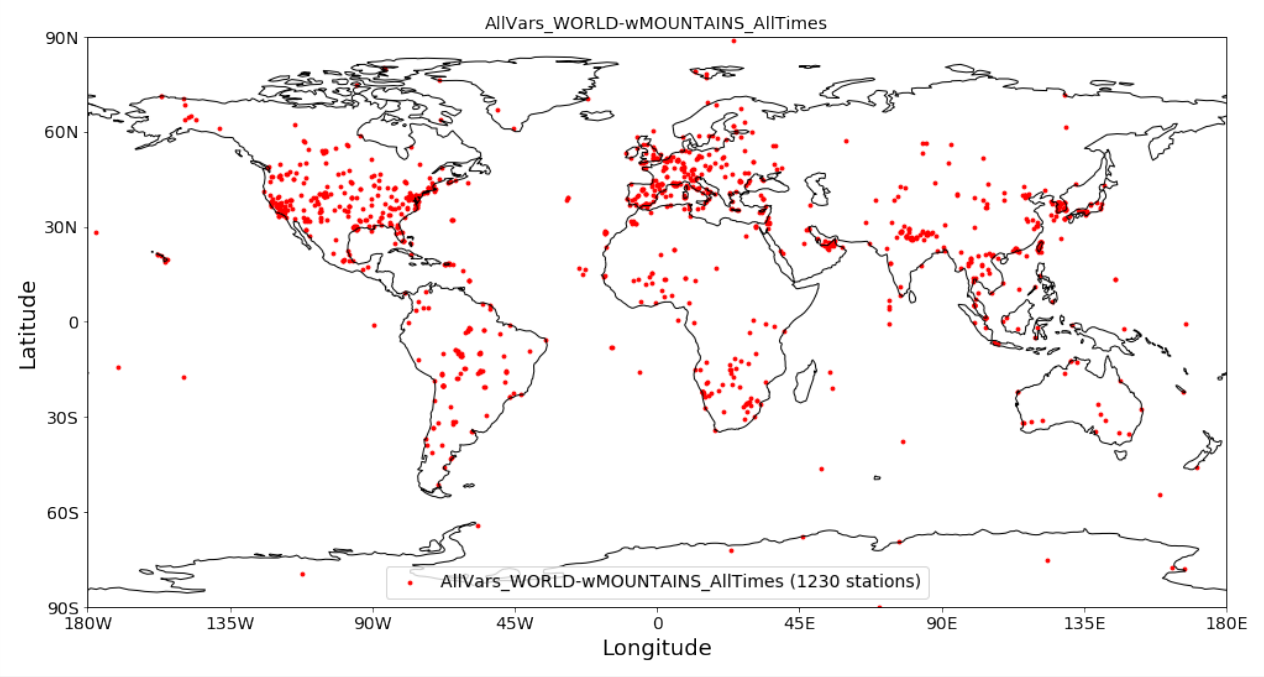
As you can see, the data object is of type `UngriddedData`. Like the `GriddedData` object, also the `UngriddedData` class has an informative string representation (that can be printed):

[29]:
```
print(aeronet_data)
```

```
Pyaerocom UngriddedData
-----------------------
Contains networks: ['AeronetSunV3Lev2.daily']
Contains variables: ['od550aer']
Contains instruments: ['sun_photometer']
Total no. of meta-blocks: 1230
Filters that were applied:
 Filter time log: 20191002122003
        Created od550aer single var object from multivar UngriddedData instance
```

Plot all station coordinates

[30]:
```
aeronet_data.plot_station_coordinates();
```



Access of individual stations

Get all station names:

[31]:
```
all_station_names = aeronet_data.unique_station_names
all_station_names[:10]  #displays first 10 stations
```

[31]:
```
['AAOT',
 'AOE_Baotou',
 'ARM_Ascension_Is',
 'ARM_Barnstable_MA',
 'ARM_Cordoba',
 'ARM_Darwin',
 'ARM_Gan_Island',
 'ARM_Graciosa',
 'ARM_Highlands_MA',
 'ARM_HyytialaFinland']
```

For instance, to access the data for the city of Leipzig, Germany, you can use square brackets with the station name of Leipzig:

[32]:
```
station_data = aeronet_data['Leipzig'] # this is fully equivalent with aeronet_data.to_
↪station_data('Leipzig')
type(station_data)
```

[32]:
```
pyaerocom.stationdata.StationData
```

As you can see, the returned object is of type `StationData`, which is one further data format of pyaerocom (note that this is not displayed in the simplified flowchart above). `StationData` may be useful for individual stations and is an extended Python dictionary (if you are familiar with Python).

You may print it to see what is in there:

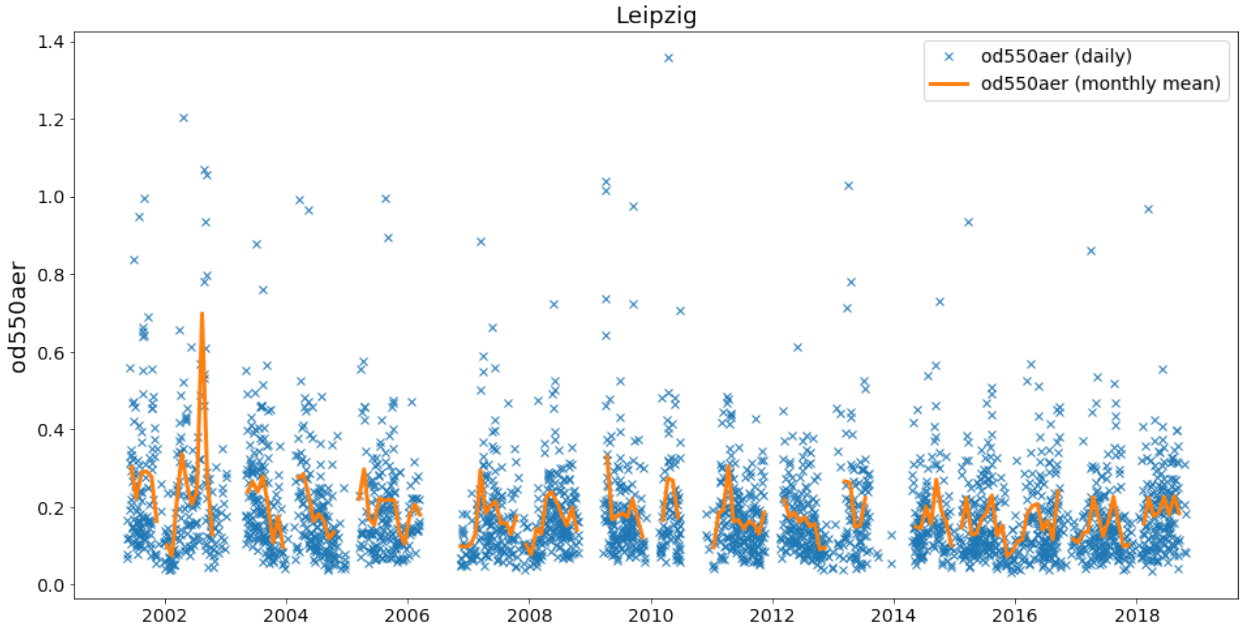
[33]:
```
print(station_data)
```

```
Pyaerocom StationData
---------------------
var_info (BrowseDict):
  od550aer (OrderedDict):
    units: 1
    overlap: False
    ts_type: daily
    apply_constraints: False
    min_num_obs: None
station_coords (dict):
  latitude: 51.352500000000006
  longitude: 12.435277999999998
  altitude: 125.0
data_err (BrowseDict): <empty_dict>
overlap (BrowseDict): <empty_dict>
data_flagged (BrowseDict): <empty_dict>
filename: None
station_id: None
station_name: Leipzig
instrument_name: sun_photometer
PI: Brent_Holben
country: None
ts_type: daily
latitude: 51.352500000000006
longitude: 12.435277999999998
altitude: 125.0
data_id: AeronetSunV3Lev2.daily
dataset_name: None
data_product: None
data_version: None
data_level: None
revision_date: None
website: None
ts_type_src: daily
stat_merge_pref_attr: None
data_revision: 20190920

Data arrays
...
dtime (ndarray, 6363 items): [2001-05-21T00:00:00.000000000, 2001-05-22T00:00:00.
↪000000000, ..., 2018-10-20T00:00:00.000000000, 2018-10-21T00:00:00.000000000]
Pandas Series
...
od550aer (Series, 6363 items)
```

As you can see, this station contains a time-series of the AOD at 550 nm. If you like, you can plot this time-series:

[34]:
```
ax = station_data.insert_nans_timeseries('od550aer').plot_timeseries('od550aer', marker=
↪'x', ls='none')
station_data.plot_timeseries('od550aer', freq='monthly', marker=' ', ls='-', lw=3, ax=ax)
```

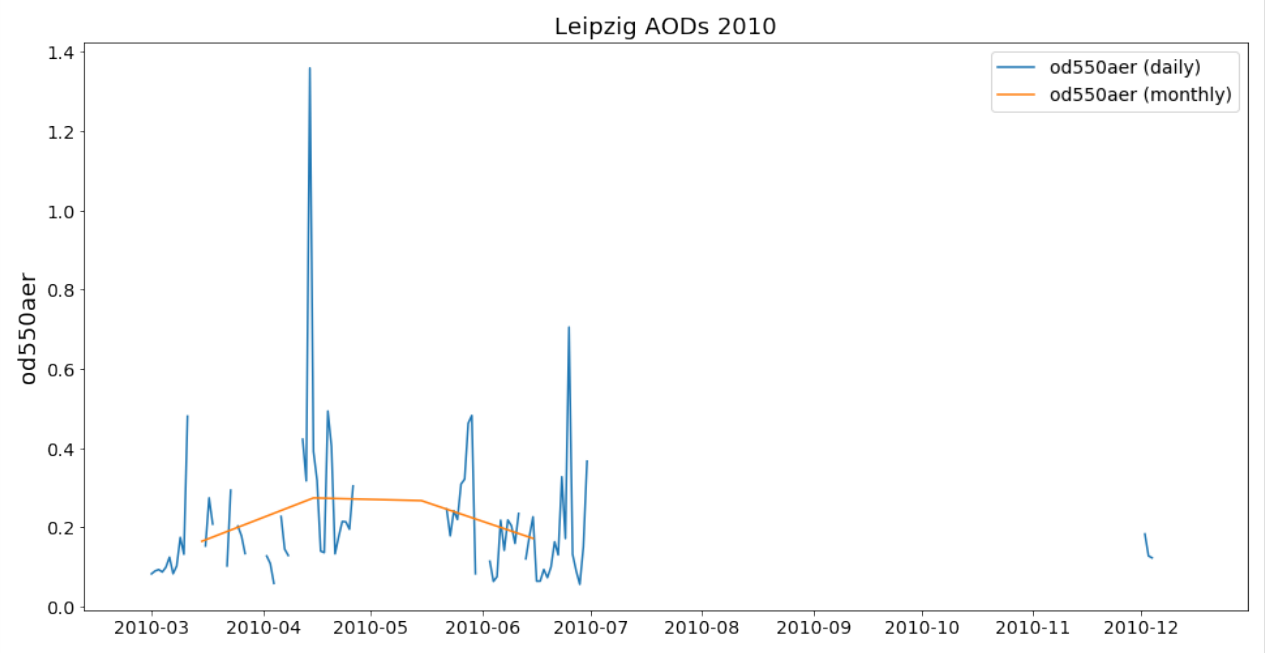
[34]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2c2061e3c8>
```



You can also retrieve the `StationData` with specifying more constraints using `to_station_data` (e.g. in monthly resolution and only for the year 2010). And you can overlay different curves, by passing the axes instance returned by the plotting method:

[35]:
```
ax=aeronet_data.to_station_data('Leipzig',
                                start=2010,
                                freq='daily').plot_timeseries('od550aer')

ax=aeronet_data.to_station_data('Leipzig',
                                start=2010,
                                freq='monthly').plot_timeseries('od550aer', ax=ax)
ax.legend()
ax.set_title('Leipzig AODs 2010')
```

[35]:
```
Text(0.5, 1.0, 'Leipzig AODs 2010')
```



You can also plot the time-series directly

For instance, if you want to do an air-quality check for you next bouldering trip, you may call:

[36]:
```
ts = aeronet_data.to_station_data('Fontainebleau', 'od550aer', 2006, None, 'monthly')
ts
```

[36]:
```
StationData([('dtime',
              array(['2006-01-15T00:00:00.000000000', '2006-02-15T00:00:00.000000000',
                     '2006-03-15T00:00:00.000000000', '2006-04-15T00:00:00.000000000',
                     '2006-05-15T00:00:00.000000000', '2006-06-15T00:00:00.000000000',
                     '2006-07-15T00:00:00.000000000', '2006-08-15T00:00:00.000000000',
                     '2006-09-15T00:00:00.000000000', '2006-10-15T00:00:00.000000000',
                     '2006-11-15T00:00:00.000000000', '2006-12-15T00:00:00.000000000'],
                    dtype='datetime64[ns]')),
             ('var_info',
              BrowseDict([('od550aer',
                           OrderedDict([('units', '1'),
                                        ('overlap', False),
                                        ('ts_type', 'monthly'),
                                        ('apply_constraints', True),
                                        ('min_num_obs',
                                         {'yearly': {'monthly': 3},
                                          'monthly': {'daily': 7},
                                          'daily': {'hourly': 6},
                                          'hourly': {'minutely': 15}})]))])),
             ('station_coords',
              {'latitude': 48.406666999999985,
               'longitude': 2.6802780000000004,
               'altitude': 85.0}),
             ('data_err', BrowseDict()),
```

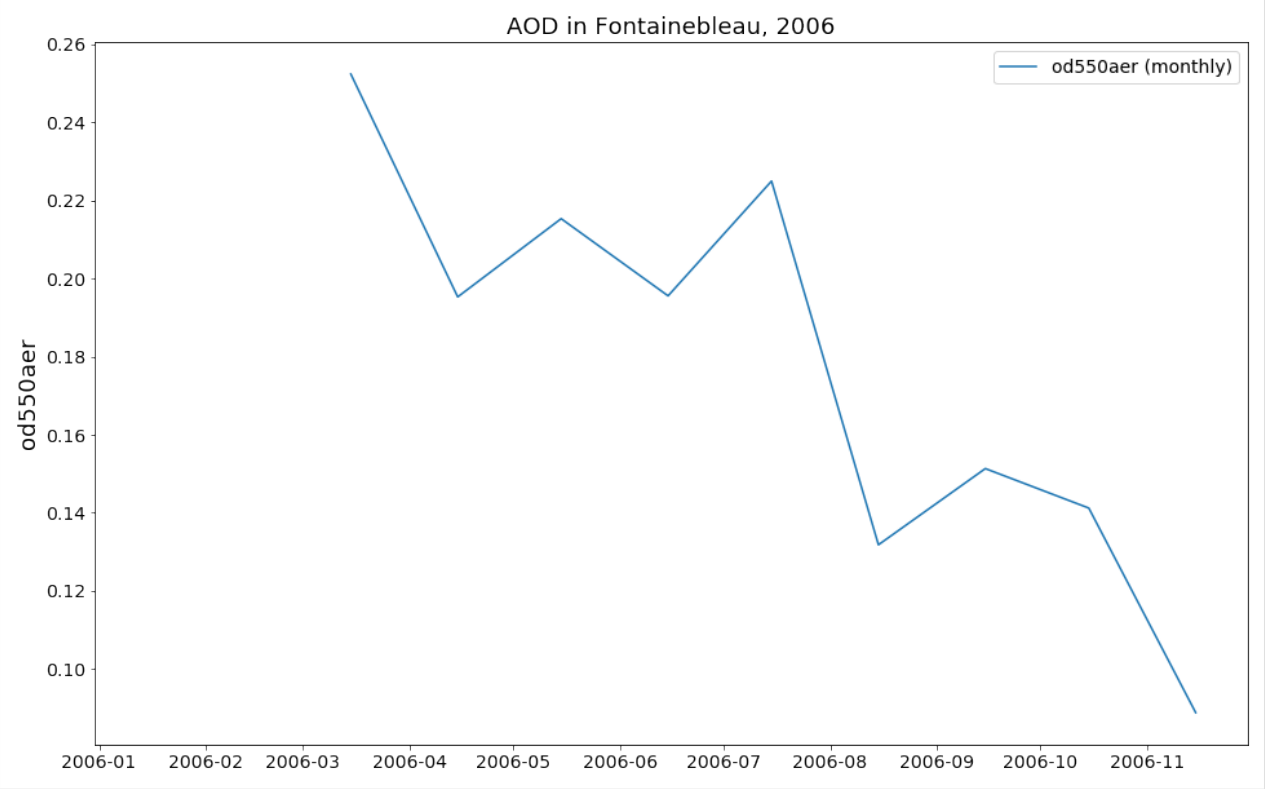
(continues on next page)

(continued from previous page)

```
             ('overlap', BrowseDict()),
             ('data_flagged', BrowseDict()),
             ('filename', None),
             ('station_id', None),
             ('station_name', 'Fontainebleau'),
             ('instrument_name', 'sun_photometer'),
             ('PI', 'Brent_Holben'),
             ('country', None),
             ('ts_type', 'monthly'),
             ('latitude', 48.406666999999985),
             ('longitude', 2.6802780000000004),
             ('altitude', 85.0),
             ('data_id', 'AeronetSunV3Lev2.daily'),
             ('dataset_name', None),
             ('data_product', None),
             ('data_version', None),
             ('data_level', None),
             ('revision_date', None),
             ('website', None),
             ('ts_type_src', 'daily'),
             ('stat_merge_pref_attr', None),
             ('data_revision', '20190920'),
             ('od550aer', 2006-01-15    0.176742
              2006-02-15         NaN
              2006-03-15    0.252403
              2006-04-15    0.195318
              2006-05-15    0.215357
              2006-06-15    0.195586
              2006-07-15    0.224991
              2006-08-15    0.131814
              2006-09-15    0.151338
              2006-10-15    0.141222
              2006-11-15    0.088815
              2006-12-15         NaN
              Name: mean, dtype: float64)])
```

[37]:
```
aeronet_data.plot_station_timeseries('Fontainebleau', 'od550aer', ts_type='monthly',
                                     start=2006).set_title('AOD in Fontainebleau, 2006')
```

[37]:
```
Text(0.5, 1.0, 'AOD in Fontainebleau, 2006')
```



Seems like November is a good time (maybe a bit rainy though)

Colocation of model and obsdata

Now that we have different data objects loaded we can continue with colocation. In the following, both the all-sky and the clear-sky data from CAM53-Oslo will be colocated with the subset of Aeronet stations that we just loaded.

The colocation will be performed for the year of 2010 and two scatter plots will be created.

You have also the option to apply a certain filter when colocating using a valid filter name. Here, we use global data and exclude mountain sides.

[38]:
```
col_all_sky_glob = pya.colocation.colocate_gridded_ungridded(od550aer, aeronet_data,
                                                             ts_type='monthly',
                                                             start=2010,
                                                             filter_name='WORLD-
↪noMOUNTAINS')
type(col_all_sky_glob)
```

```
Setting od550aer outlier lower lim: -1.00
Setting od550aer outlier upper lim: 10.00
Interpolating data of shape (12, 192, 288). This may take a while.
Successfully interpolated cube
```

[38]:
```
pyaerocom.colocateddata.ColocatedData
```

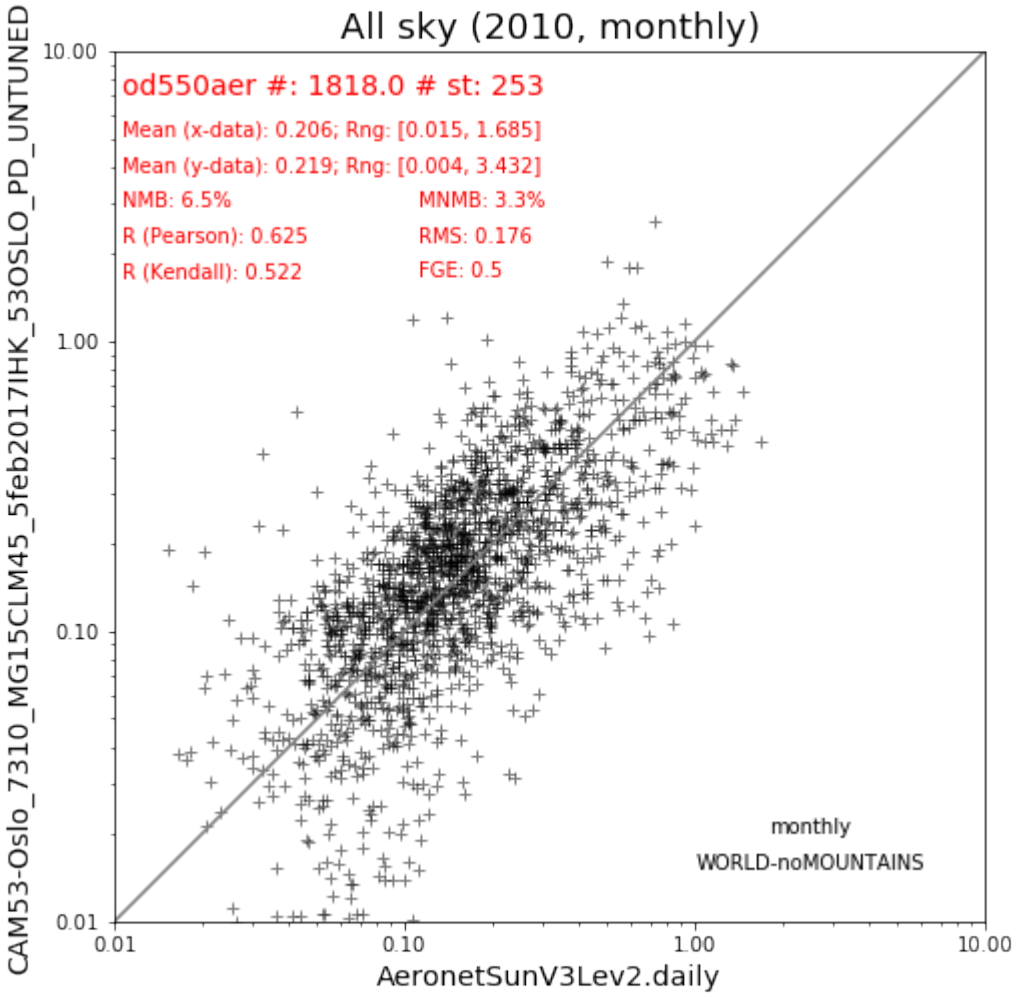
Let's do the same for the clear-sky data.

[39]:
```
col_clear_sky_glob = pya.colocation.colocate_gridded_ungridded(od550csaer, aeronet_data,
                                                               ts_type='monthly',
                                                               start=2010,
                                                               filter_name='WORLD-
↪noMOUNTAINS')
type(col_clear_sky_glob)
```

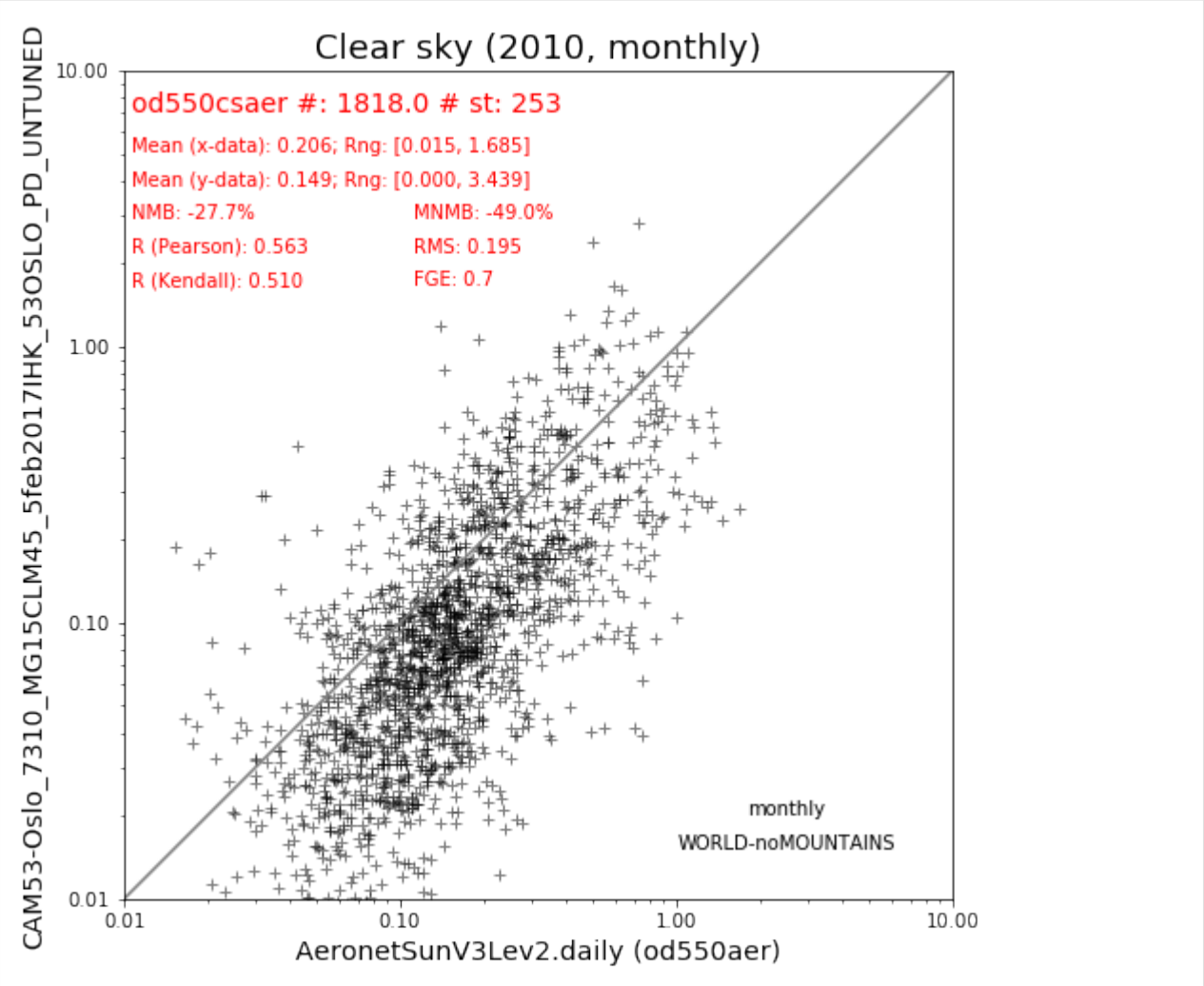
```
Setting od550aer outlier lower lim: -1.00
Setting od550aer outlier upper lim: 10.00
Interpolating data of shape (12, 192, 288). This may take a while.
Successfully interpolated cube
```

[39]:
```
pyaerocom.colocateddata.ColocatedData
```

[40]:
```
ax1 = col_all_sky_glob.plot_scatter()
ax1.set_title('All sky (2010, monthly)');
```



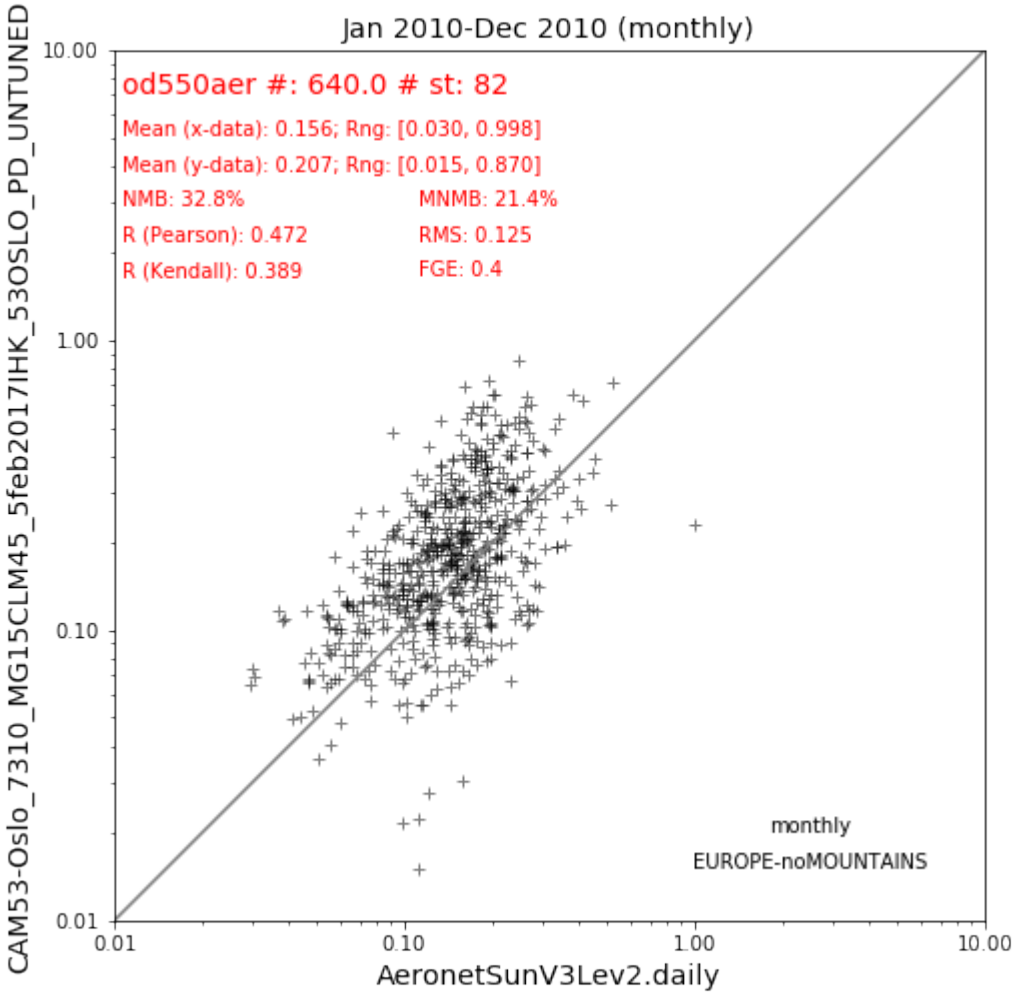
[41]:
```
ax2 = col_clear_sky_glob.plot_scatter()
ax2.set_title('Clear sky (2010, monthly)');
```



... or for EUROPE:

[42]:
```
pya.colocation.colocate_gridded_ungridded(od550aer, aeronet_data,
                                          ts_type='monthly',
                                          start=2010,
                                          filter_name='EUROPE-noMOUNTAINS').plot_
↪scatter();
```

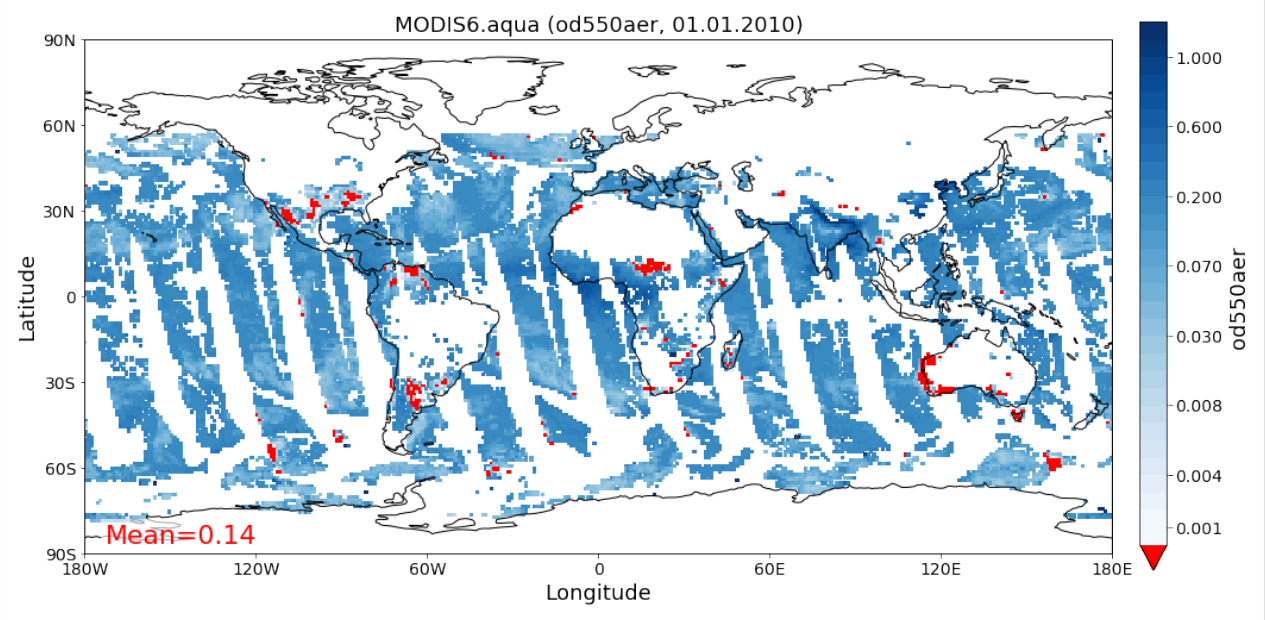
```
Setting od550aer outlier lower lim: -1.00
Setting od550aer outlier upper lim: 10.00
Interpolating data of shape (12, 192, 288). This may take a while.
Successfully interpolated cube
```



Colocation with satellite AODs from MODIS

[43]:
```
sat_data = pya.io.ReadGridded('MODIS6.aqua').read_var('od550aer', start=2010)
sat_data.quickplot_map(vmin=.001, vmax=3, c_under='r');
```

```
Overwriting unit unknown in cube od550aer with value "1"
```



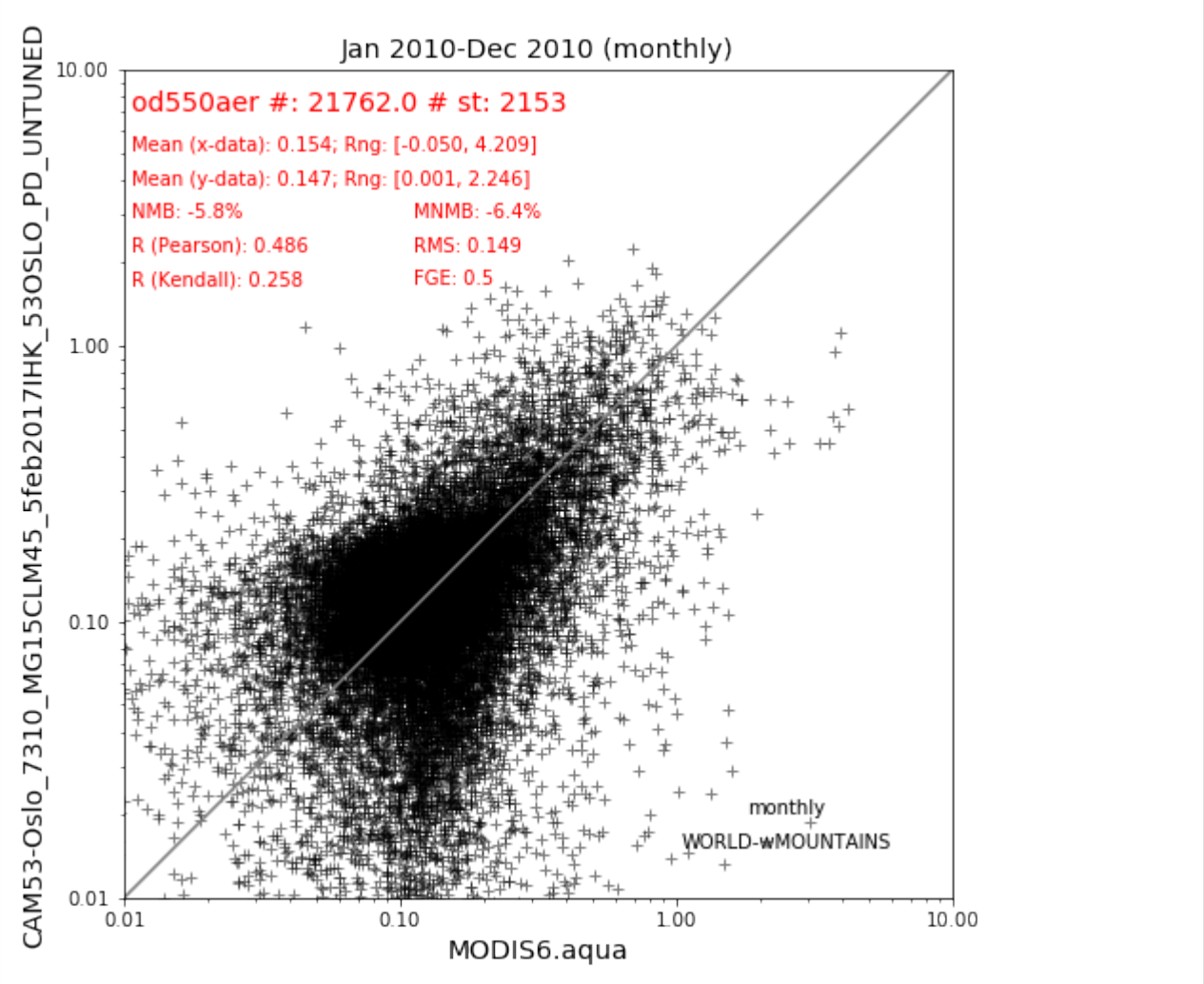
Colocation is straight forward as shown below. The data will be regridded to 5x5 degrees resolution during the colocation.

[44]:
```
col_modis_monthly = pya.colocation.colocate_gridded_gridded(gridded_data=od550aer,
                                                            gridded_data_ref=sat_data,
                                                            ts_type='monthly',
                                                            regrid_res_deg=5,
                                                            remove_outliers=True,
                                                            harmonise_units=False)
```

```
Interpolating data of shape (365, 180, 360). This may take a while.
Successfully interpolated cube
Setting od550aer outlier lower lim: -1.00
Setting od550aer outlier upper lim: 10.00
```

[45]:
```
col_modis_monthly.plot_scatter()
```

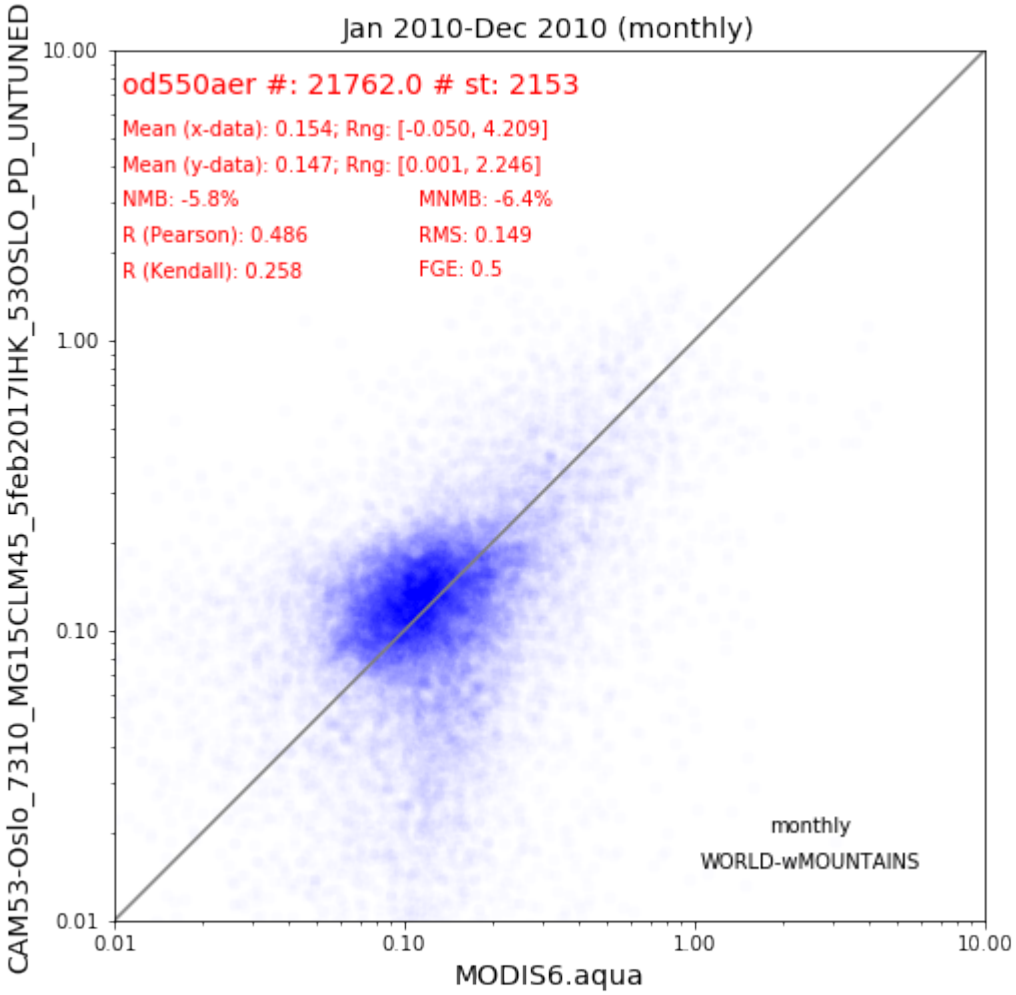
[45]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2c1049a198>
```



You may also change the plot style, to adapt better to the large amount of data points here in these satellite products:

[46]:
```
col_modis_monthly.plot_scatter(ms=6, marker='o',mec='none', color='b', alpha=0.01)
```

[46]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2c10614908>
```



Remark on the ColocatedData object

The `ColocatedData` object has not many features and methods implemented yet. But it builds a good basis for further analysis using features of the underlying data-structure, which is an instance of the xarray.DataArray class and can be accessed via the `data` attribute if the `ColocatedData` object:

[47]:
```
arr = col_all_sky_glob.data
type(arr)
```

[47]:
```
xarray.core.dataarray.DataArray
```

You may want to read more about it here.

3.2.2 AEROCOM default regions (class `Region`)

This notebook introduces how pya handles information related to default regions (e.g. Europe, Asia, ...) as used in the AEROCOM interface. All default regions are defined in the file regions.ini.

[1]:
```
import pyaerocom as pya
```
```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.016 s
```

What regions are available

[2]:
```
pya.region.all()
```
[2]:
```
['WORLD',
 'EUROPE',
 'ASIA',
 'AUSTRALIA',
 'CHINA',
 'INDIA',
 'NAFRICA',
 'SAFRICA',
 'SAMERICA',
 'NAMERICA']
```

These region IDs can be used to access more information about the regions (which is used throughout pyaerocom). For instance:

Create a `Region`

[3]:
```
europe = pya.Region("EUROPE")
print(europe)
```
```
pyaeorocom Region
Name: EUROPE
Longitude range: [-20, 70]
Latitude range: [30, 80]
Longitude range (plots): [-20, 70]
Latitude range (plots): [30, 80]
```

[4]:
```
asia = pya.Region("ASIA")
print(asia)
```
```
pyaeorocom Region
Name: ASIA
Longitude range: [40, 150]
```

(continues on next page)

(continued from previous page)

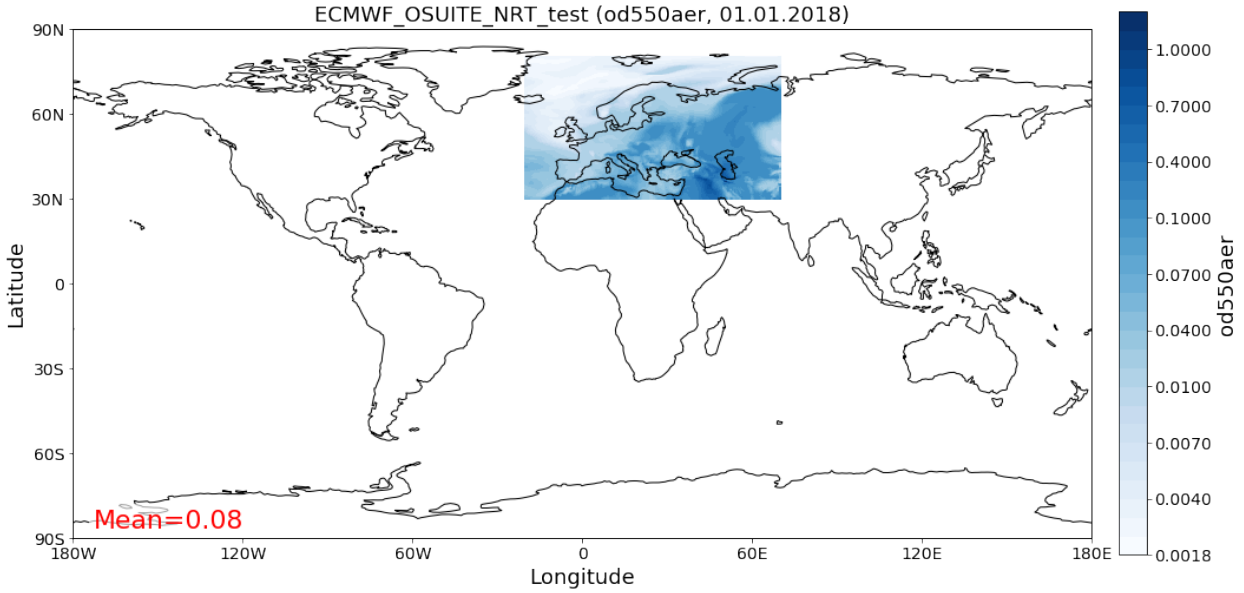
```
Latitude range: [0, 60]
Longitude range (plots): [40, 150]
Latitude range (plots): [0, 60]
```

Load example data and apply region specific crop

In the following cell, we create an instance of the `GriddedData` class (hich is introduced in more detail in a later tutorial), load some test data (from the CAMS ECMWF OSUITE dataset), crop it and plot a map of the results over Europe.

[5]:
```
data = pya.GriddedData()
data._init_testdata_default()
crop = data.crop(region="EUROPE")
fig = crop.quickplot_map()
```

```
Overwriting unit unknown in cube od550aer with value "1"
```



Computing distance to centre of region

For a given lat / lon coordinate, distances to the region centre coordinate can be computed as:

[6]:
```
dc = asia.distance_to_center(lat=45, lon=60)
dc #km
```

[6]:
```
3476.358401761219
```

Access centre coordinate:

[7]:
```
asia.center_coordinate # (lat, lon)
```

[7]:
```
(30.0, 95.0)
```

[8]:
```
asia.distance_to_center(30, 95)
```

[8]:
```
0.0
```

That's it. Not much more to say about regions until now.

3.2.3 The GriddedData class

This notebook introduces the GriddedData class of pyaerocom. The GriddedData class is the fundamental base class for the analysis of model data. The underlying data type is iris.cube.Cube which was extended, for instance by allowing direct imports of netCDF files when creating an instance of GriddedData (i.e. by passing the filename and specifying the variable name on initialisation). This notebook introduces some of the features of the GriddedData class. Starting with some imports:

[1]:
```
import pyaerocom as pya
from warnings import filterwarnings
filterwarnings('ignore')
pya.change_verbosity('critical')
```

```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.016 s
```

Let's get a test file to load

[2]:
```
test_files = pya.io.testfiles.get()
for name, filepath in test_files["models"].items(): print("%s\n%s\n" %(name, filepath))
```

```
aatsr_su_v4.3
/lustre/storeA/project/aerocom/aerocom-users-database/CCI-Aerosol/CCI_AEROSOL_Phase2/
↪AATSR_SU_v4.3/renamed/aerocom.AATSR_SU_v4.3.daily.od550aer.2008.nc

ecmwf_osuite
/lustre/storeA/project/aerocom/aerocom1/ECMWF_OSUITE_NRT_test/renamed/aerocom.ECMWF_
↪OSUITE_NRT_test.daily.od550aer.2018.nc
```

Let's pick out the ECMWF OSUITE test file and load the data directly into an instance of the GriddedData class. The GriddedData class takes either preloaded instances of the iris.cube.Cube class as input, or a valid netCDF file path. The latter requires specification of the variable name which is then filtered from the data stored in the netCDF file (which may contain multiple variables. The following example imports the data for the aerosol optical density at 550 nm. The string representation of the GriddedData class (see print at end of following code cell) was slitghtly adapted from the underlying Cube object.

Read ECMWF OSUITE AOD data

[3]:
```
fpath = test_files["models"]["ecmwf_osuite"]
data = pya.GriddedData(input=fpath, var_name="od550aer", data_id="ECMWF_OSUITE")
print(data)
```

```
Overwriting unit unknown in cube od550aer with value "1"
```

```
pyaerocom.GriddedData: ECMWF_OSUITE
Grid data: Dust Aerosol Optical Depth at 550nm / (1) (time: 365; latitude: 451;␣
↪longitude: 900)
     Dimension coordinates:
          time                                x              -              -
          latitude                            -              x              -
          longitude                           -              -              x
     Attributes:
          Conventions: CF-1.0
          NCO: 4.7.2
          computed: False
          concatenated: False
          data_id: ECMWF_OSUITE
          from_files: ['/lustre/storeA/project/aerocom/aerocom1/ECMWF_OSUITE_NRT_test/
↪rename...
          history: Tue Mar 20 13:08:49 2018: ncks -7 -O -o test.nc -x -v time_bnds␣
↪od550aer.test.orig.nc
Tue...
          history_of_appended_files: Tue Mar 20 02:09:15 2018: Appended file /lustre/
↪storeA/project/aerocom/aerocom1/ECMWF_OSUITE_NRT/renamed//aerocom.ECMWF_OSUITE_NRT.
↪daily.od550bc.2018.nc...
          invalid_units: ~
          is_at_stations: False
          nco_openmp_thread_number: 1
          outliers_removed: False
          reader: None
          region: None
          regridded: False
          ts_type: daily
          var_name: od550aer
          var_name_read: n/d
          vert_code:
          year: 2018
     Cell methods:
          mean: time
```

Remark on longitude definition

If the longitudes in the original NetCDF file are defined as:

$$0 \leq \text{lon} \leq 360$$

then, pyaerocom converts automatically to:

$$-180 \leq \text{lon} \leq 180$$

when an instance of the `GriddedData` class is created (see print statment above *Rolling longitudes to -180 -> 180 definition*). This is, for instance, the case for the ECMWF OSUITE data files.

Basic attributes of the `GriddedData` class

In the following cells, some of the most important attributes are introduced. These are mostly reimplementations of the underlying `iris.Cube` data object, which is stored in and can be accessed via the `GriddedData.cube` attribute. For instance the attribute `GriddedData.longitude` will access `GriddedData.grid.coord("longitude")`, `GriddedData.latitude` will return `GriddedData.grid.coord("latitude")` and `GriddedData.time` will return the time dimension array (`GriddedData.grid.coord("time")`).

[4]:
```
data.data_id
```
[4]:
```
'ECMWF_OSUITE'
```

[5]:
```
data.var_name
```
[5]:
```
'od550aer'
```

[6]:
```
data.units
```
[6]:
```
Unit('1')
```

[7]:
```
data.ts_type
```
[7]:
```
'daily'
```

Side note: the unit is obviously not specified in this dataset, which is part of the game, unfortunately, when working with external data…

[8]:
```
type(data.longitude)
```
[8]:
```
iris.coords.DimCoord
```

[9]:
```
data.longitude.points.min(), data.longitude.points.max()
```
[9]:
```
(-180.0, 179.60000610351562)
```

[10]:
```
type(data.latitude)
```
[10]:
```
iris.coords.DimCoord
```

[11]:
```
data.latitude.points.min(), data.latitude.points.max()
```
[11]:
```
(-90.0, 90.0)
```

[12]:
```
type(data.time)
```

[12]:
```
iris.coords.DimCoord
```

[13]:
```
data.time.points.min(), data.time.points.max()
```

[13]:
```
(0.0, 364.0)
```

[14]:
```
tstamps = data.time_stamps()
print(tstamps[0], tstamps[-1])
```

```
2018-01-01T00:00:00.000000 2018-12-31T00:00:00.000000
```

If you do not specify the variable type, an Exception is raised, that will get you some information about what variables are available in the file (if the file is readable using the `iris.load` method).

[15]:
```
try:
    data = pya.GriddedData(input=fpath)
except pya.exceptions.NetcdfError as e:
    print("This did not work...error message: %s" %repr(e))
```

```
This did not work...error message: NetcdfError("Could not load single cube from /lustre/
↪storeA/project/aerocom/aerocom1/ECMWF_OSUITE_NRT_test/renamed/aerocom.ECMWF_OSUITE_NRT_
↪test.daily.od550aer.2018.nc. Please specify var_name. Input file contains the␣
↪following variables: ['od550dust', 'od550oa', 'od550aer', 'od550so4', 'od550bc']")
```

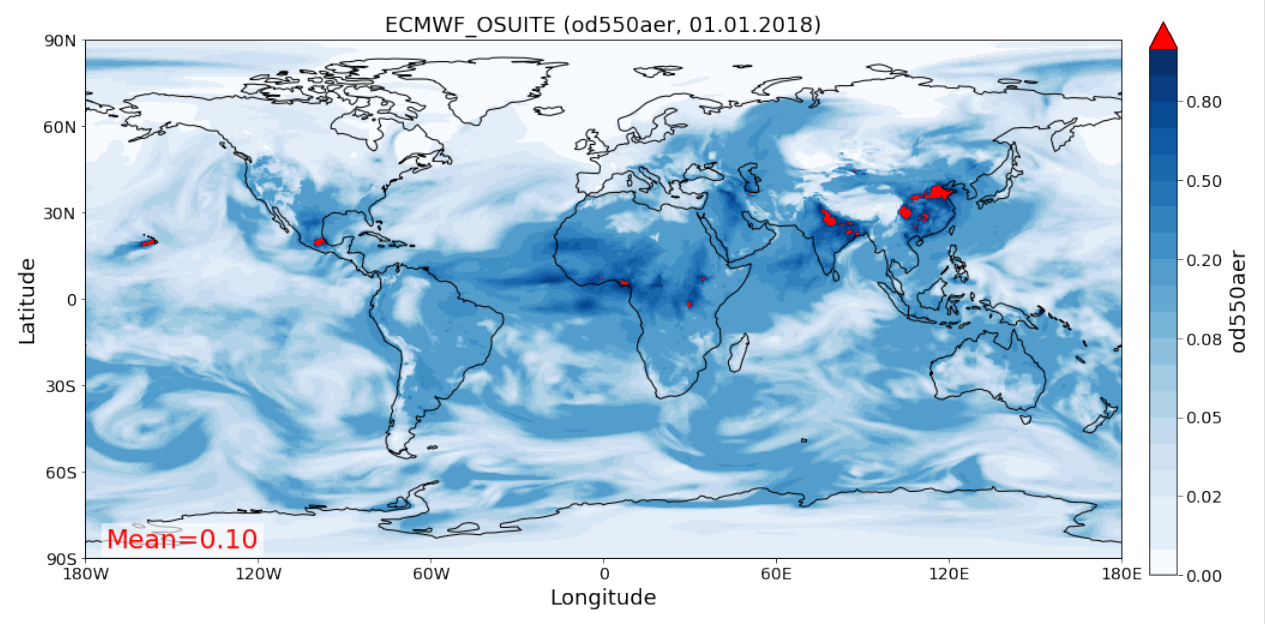
Also, if you parse an invalid variable name, you will get some hint.

[16]:
```
try:
    data = pya.GriddedData(input=fpath, var_name="Blaaa")
except Exception as e:
    print("This also did not work...error message: %s" %repr(e))
```

```
This also did not work...error message: NetcdfError('Variable Blaaa not available in␣
↪file /lustre/storeA/project/aerocom/aerocom1/ECMWF_OSUITE_NRT_test/renamed/aerocom.
↪ECMWF_OSUITE_NRT_test.daily.od550aer.2018.nc')
```

You can have a quick look at the data using the class-own quickplot method

[17]:
```
data.quickplot_map(time_idx=0, vmin=0, vmax=1, c_over="r");
```



Why not load some of the other variables…

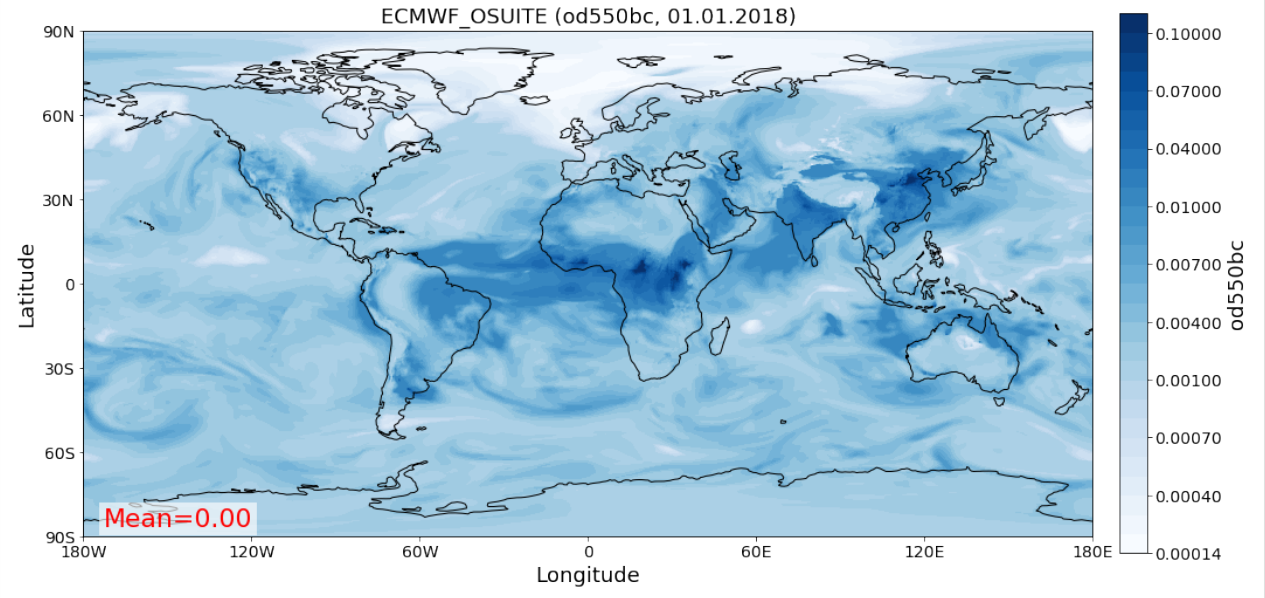
[18]:
```
data_bc = pya.GriddedData(fpath, var_name="od550bc", data_id="ECMWF_OSUITE")
data_so4 = pya.GriddedData(fpath, var_name="od550so4", data_id="ECMWF_OSUITE")
```

```
Overwriting unit unknown in cube od550bc with value "1"
```

```
Overwriting unit unknown in cube od550so4 with value "1"
```

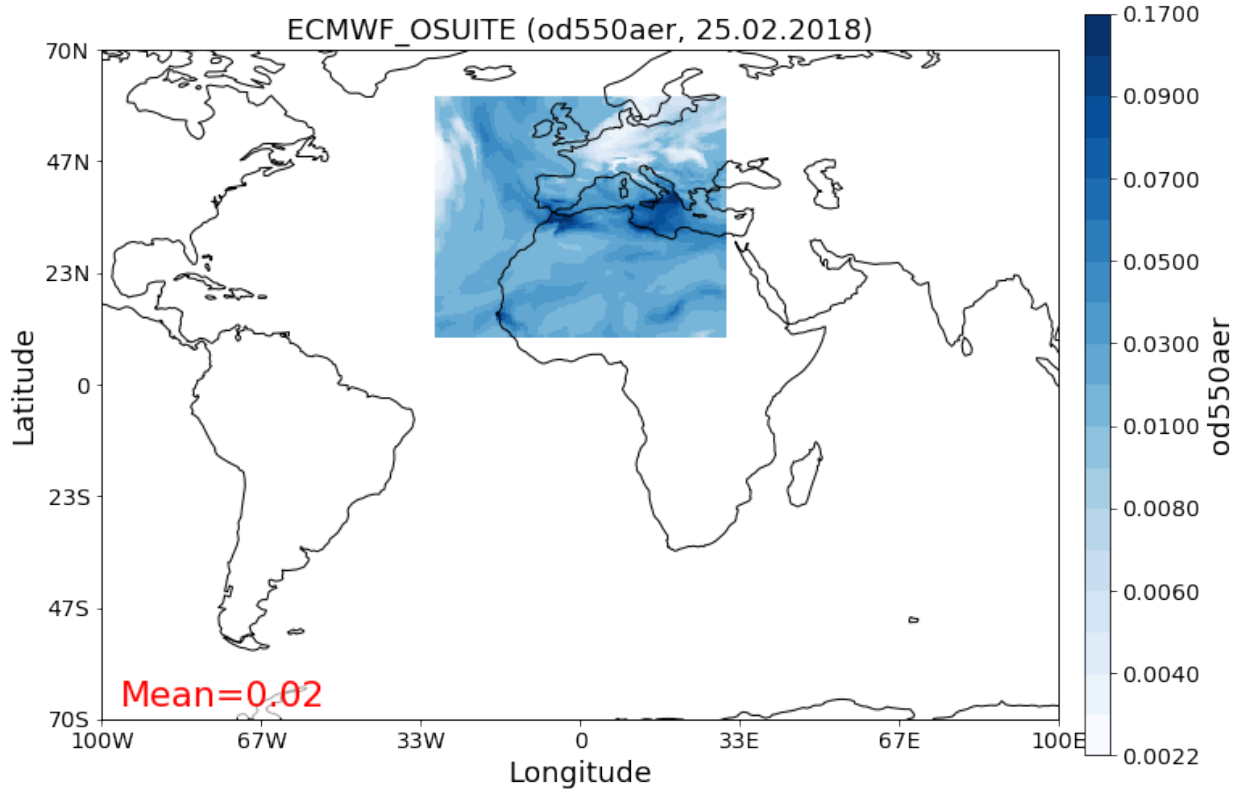
… and plot them as well

[19]:
```
data_bc.quickplot_map();
```



Apply custom crop and plot

[20]:
```
data_so4_cropped = data_so4.crop(lon_range=(-30, 30),
                                 lat_range=(10, 60))

data_so4_cropped.quickplot_map(time_idx='25-2-2018', xlim=(-100, 100), ylim=(-70, 70));
```



Change resolution

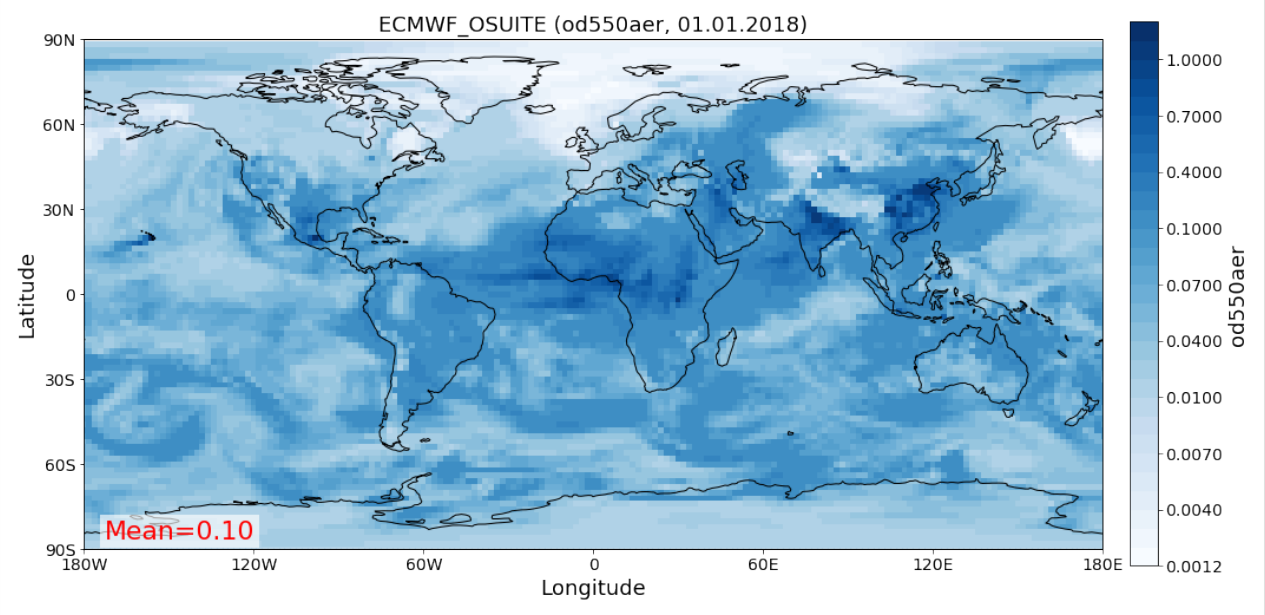
Downscale to 2x2 resolution:

[21]:
```
lons = np.arange(-180, 180, 2)
lats = np.arange(-90, 90, 2)

data_lowres = data.interpolate(longitude=lons, latitude=lats)
```

```
Interpolating data of shape (365, 451, 900). This may take a while.
Successfully interpolated cube
```

And plot:

[22]:
```
fig =data_lowres.quickplot_map()
```

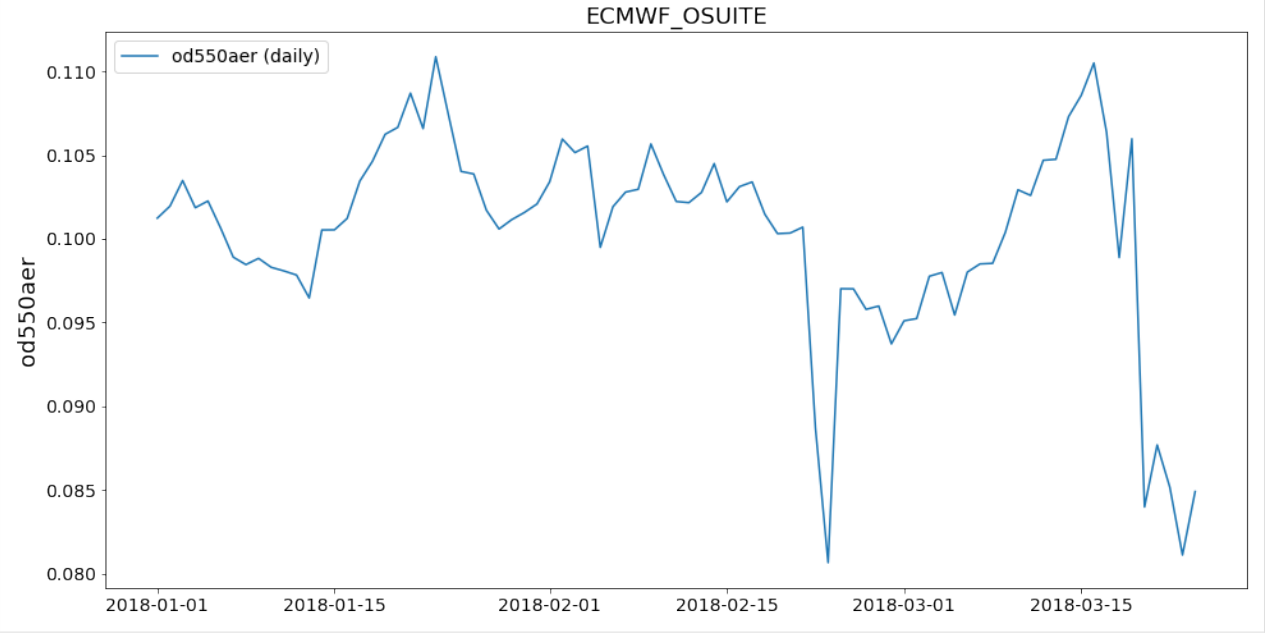


Area weighted mean

Retrieve area weighted mean from data

[23]:
```
area_mean = data.get_area_weighted_timeseries()
area_mean.plot_timeseries('od550aer');
```

```
Trying to infer ts_type in StationData ECMWF_OSUITE for variable od550aer
```



... more to come

This tutorial is not yet completed as the `GriddedData` class is currently under development.

3.2.4 Reading of ungridded observation datasets in pyaerocom

Primer on observation datasets

- Observation data is often provided ungridded in the form of ASCII text files that contain both the data and relevant metadata.
- In **pyaerocom** such datasets are referred to as **ungridded data** since it is sparse observations at different locations and at different times.
- The format of the files can be very different between different observation networks but is usually the same for all data files that belong to one network.
- The data files are often provided **per station**, that is, **one data file** contains data (i.e. time-series data of one or more variables and metadata such as *station name, latitude, longitude, altitude, PI* **from a single station**
  - Other than model data (which often follows certain file standards, e.g. NetCDF files and naming conventions, e.g. CF-conventions
- For some databases, there is exactly one file per station (containing all available data from that station). This is the case in the example dataset shown below (Aeronet Sun version 3 level2 daily data). However, the general case is that there may be more than one file per station (the EBAS database is such an example, where there can be multiple data files per station, see this notebook for an example of how complicated it may get).

Data access

Both the model and the observation datasets related to the AeroCom project are stored on internal servers at the Norwegian Meteorological Institute (METNO). On import, pyaerocom automatically checks if it can access the METNO servers containing the data.

The data directory of each dataset can be accessed via an *unique ID* for the dataset. These ID's can be a little cryptic sometimes and parts of this tutorial show, how to find the data you search for, regardless whether you know the exact pyaerocom ID or not.

For instance, below we are going to work with AERONET Sun photometer data, using the version 3, level 2 daily data product. The corresponding ID for this dataset in pyaerocom is **AeronetSunV3Lev2.daily**.

NOTE

This notebook requires access to the AeroCom database and will not work if you do not have access to the AeroCom servers at METNO

If you run into problems

Please create an issue if you run into problems or have suggestions for improvements.

Reading of and investigating Aeronet Sun AODs (version 3, level 2 data)

[1]:
```
import pyaerocom as pya
```
```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.017 s
```

Check version of pyaerocom:

[2]:
```
pya.__version__ #0.8.0.dev6
```
[2]:
```
'0.8.0.dev30'
```

Search data ID for Aeronet Sun version 2 level 2, daily data

The `browse_database` method helps you to find model or observation datasets.

[3]:
```
pya.browse_database('Aeronet*Sun*V3*')
```
```

Dataset name: AeronetSunV3Lev1.5.daily
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev1.
↪5.daily/renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang4487aer_calc', 'od550aer']
Last revision: 20190920

Dataset name: AeronetSunV3Lev1.5.AP
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev1.
↪5.AP/renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang4487aer_calc', 'od550aer']
Last revision: 20190511

Dataset name: AeronetSunV3Lev2.daily
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev2.
↪0.daily/renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang4487aer_calc', 'od550aer']
Last revision: 20190920

Dataset name: AeronetSunV3Lev2.AP
```

(continues on next page)

(continued from previous page)

```
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev2.
↪0.AP/renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang4487aer_calc', 'od550aer']
Last revision: 20190511
```

You can see that there are three matches that can be read. The attribute `dataset_name` specifies the ID that we are looking for that is required to read the data.

Below, we are interested in the following dataset:

[4]:
```
DATA_ID = 'AeronetSunV3Lev2.daily'
```

Pick one (or more) variable(s) of interest

From the output above, you can also see which variables the individual datasets provide. The variable names follow AeroCom conventions and you can find more information (e.g. CF standard names) about each variable here.

In the following we will import the aerosol optical depth data at 550 nm (`od550aer`)

[5]:
```
VAR_TO_READ = 'od550aer' # you can also use a list of supported variables if you like, e.
↪g. (od550aer, od440aer)
```

Read the AODs from the whole database (all available stations / times) using the ReadUngridded class

Since the dataformats are usually specific for each observation dataset / network, each supported dataset has it's own reading routine in pyaerocom. The individual reading routines can be found in the `pyaerocom.io` module. For instance, the class

[6]:
```
pya.io.ReadAeronetSunV3
```

[6]:
```
pyaerocom.io.read_aeronet_sunv3.ReadAeronetSunV3
```

contains the reading methods for the Aeronet Sun version 3 dataset that we are interested in.

However in order to make life easier for everyone, all implemented *individual reading routines* for *each individual dataset* are registered in the ReadUngridded factory class, which has registered these *individual reading routines* (this works, becuase the invididual reading routines are all based on the same API template).

Here, *registered* means, that the *dataset ID* of one network is linked with the corresponding reading class.

Having said that, it means that calling

```
data = pya.io.ReadAeronetSunV3(vars_to_read='od550aer')
```

will give you exactly the same result as calling:

```
data = pya.io.ReadUngridded(dataset_to_read='AeronetSunV3Lev2.daily',
                            vars_to_read='od550aer')
```

The returned data object is an instance of the UngriddedData class which is the pyaerocom standard object for ungridded data and which is designed to hold a whole dataset of observation records (i.e. data from all stations).

Create instance of ReadUngridded class

[7]:
```
reader = pya.io.ReadUngridded(DATA_ID)
print(reader)
```

```
Dataset name: AeronetSunV3Lev2.daily
Data directory: /lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/AeronetSunV3Lev2.
↪0.daily/renamed
Supported variables: ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer',
↪'ang4487aer_calc', 'od550aer']
Last revision: 20190920
```

You may also check, which other datasets are supported by the `ReadUngridded` class:

[8]:
```
reader.supported_datasets
```

[8]:
```
['AeronetInvV3Lev2.daily',
 'AeronetInvV3Lev1.5.daily',
 'AeronetInvV2Lev2.daily',
 'AeronetInvV2Lev1.5.daily',
 'AeronetSDAV2Lev2.daily',
 'AeronetSDAV3Lev1.5.daily',
 'AeronetSDAV3Lev2.daily',
 'AeronetSunV2Lev2.daily',
 'AeronetSunV2Lev2.AP',
 'AeronetSunV3Lev1.5.daily',
 'AeronetSunV3Lev1.5.AP',
 'AeronetSunV3Lev2.daily',
 'AeronetSunV3Lev2.AP',
 'EARLINET',
 'EBASMC',
 'DMS_AMS_CVO',
 'GAWTADsubsetAasEtAl']
```

Read the dataset

The `read` method loops over all files that exist for this dataset and reads them into one data object (`data`) which contains the whole dataset.

NOTE: this can take a while as it has to read ~1000 files.

[9]:
```
data = reader.read(vars_to_retrieve=VAR_TO_READ)
print(data)
```

```
Pyaerocom UngriddedData
-----------------------
Contains networks: ['AeronetSunV3Lev2.daily']
Contains variables: ['od550aer']
Contains instruments: ['sun_photometer']
Total no. of meta-blocks: 1230
Filters that were applied:
 Filter time log: 20191002122003
        Created od550aer single var object from multivar UngriddedData instance
```

That's it! That is all that is required to import an ungridded dataset.

The data object that is returned by the `read` method is an instance of the pyaerocom.UngriddedData class.

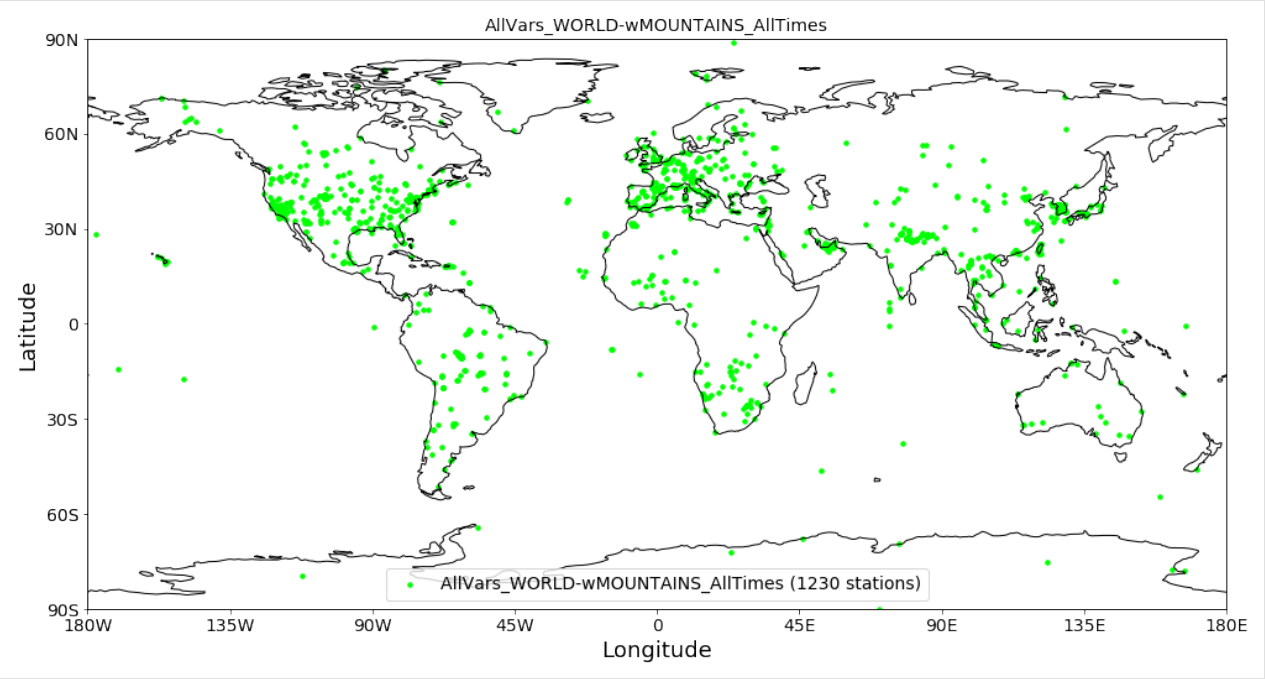
The `UngriddedData` object provides an interface that can be used to work with the data and further analyse it.

From the output above, you can see that this datasets contains 1199 *meta-blocks*, that is, one meta block per data file that was read. Since for this database, each station has exactly one datafile, this means that each meta-block corresponds to one station.

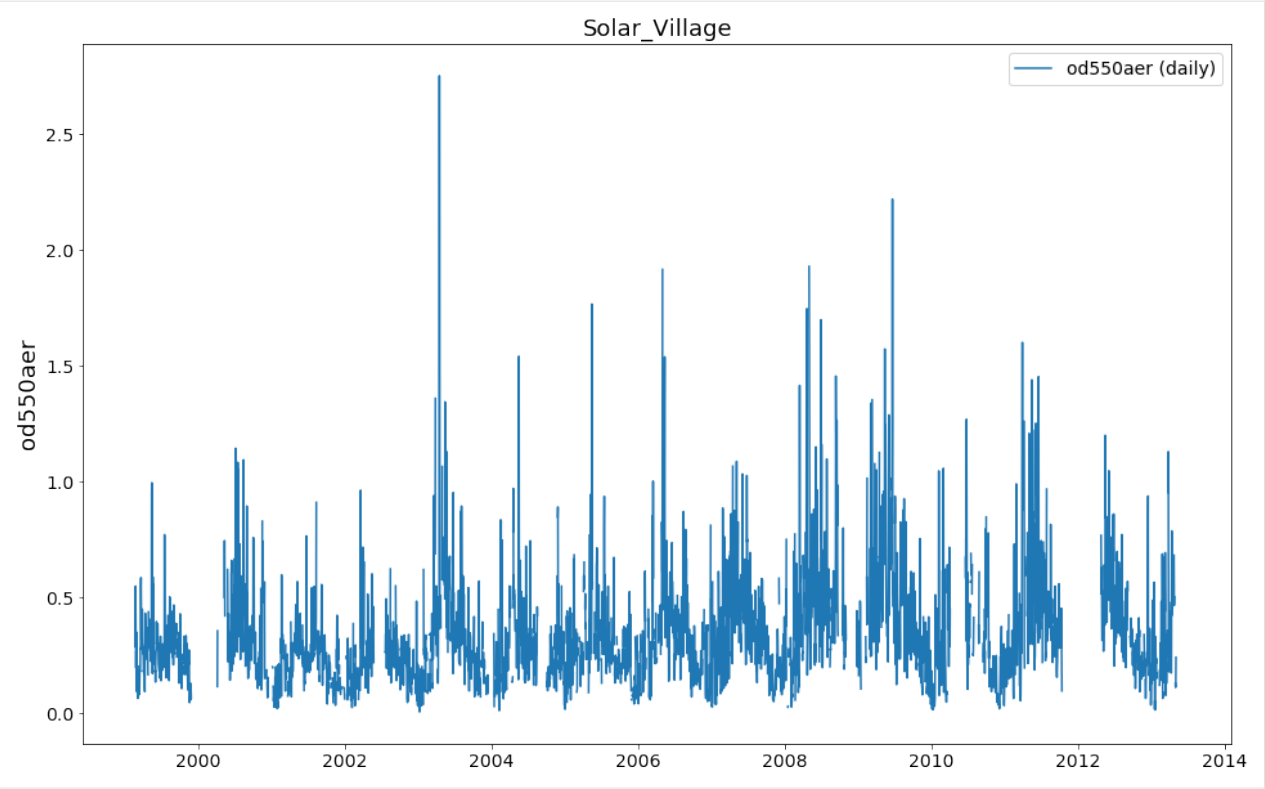
The next tutorial is based on this tutorial and will introduce the `UngriddedData` class and furthermore, the `StationData` class.

However, before ending this tutorial without a plot, let's have a glimpse at the features of the `UngriddedData` object that we just created:

[10]:
```
data.plot_station_coordinates(markersize=12, color='lime');
```



[11]:
```
data.plot_station_timeseries('Solar*', 'od550aer');
```



Finally, store the data object so that it can be used in the next tutorial:

[12]:
```
%store data
```

```
Stored 'data' (UngriddedData)
```

3.2.5 Introduction into the UngriddedData class and the StationData class

This notebook introduces 2 of the most relevant data objects that exist in pyaerocom:

``**UngriddedData**``

- Designed to hold a whole database of observations, that is, timeseries data for multiple variables from multiple stations around the globe.
- Supports also 3D variables (e.g. timeseries of lidar profiles).
- Usually, one instance of this data object contains a single network, but it can also contain more than one network.

``**StationData**``

- Data object that contains data **from a single station**.
- Includes metadata and variable timeseries data.
- Arbitrary number of variables supported.

NOTE

This notebook is currently under development and gives only a brief and incomplete introduction into the two data objects.

The `UngriddedData` object

The first part of the tutorial shows some features of the `UngriddedData` object.

Import the `UngriddedData` object that was created in the previous tutorial

[1]:
```
import pyaerocom as pya
# read the data from the storage
%store -r data

data
```

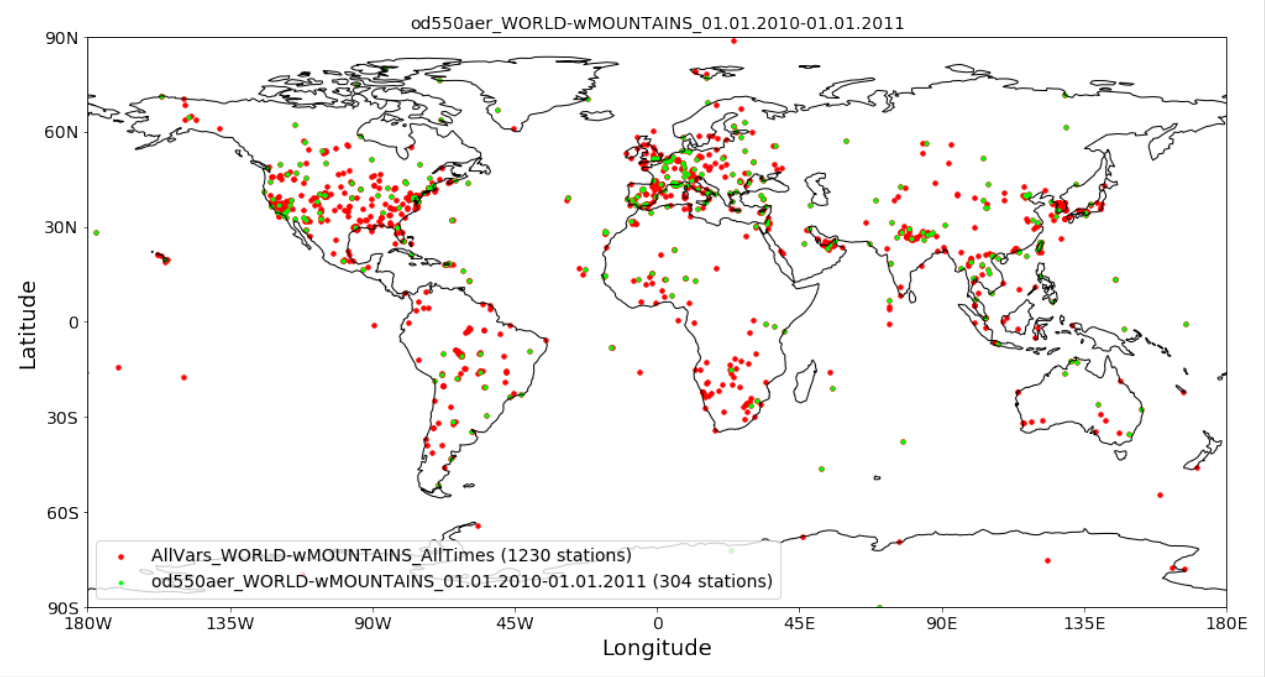
```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.016 s
```

[1]:
```
UngriddedData <networks: ['AeronetSunV3Lev2.daily']; vars: ['od550aer']; instruments: [
↪'sun_photometer'];No. of stations: 1230
```

Create an overview map of all stations

Before digging a little deeper into the `UngriddedData` object, let's get an overview of the bigger picture:

[2]:
```
# plots all stations as red dots
ax = data.plot_station_coordinates(markersize=12, color='r')

# add all stations that contain AOD data in 2010 in green
ax = data.plot_station_coordinates(var_name='od550aer',
                                   start=2010,
                                   stop=2011, color='lime', ax=ax)
```



As you can see, you can specify additional input parameters, e.g. to display only stations that contain variable data or to specify a time interval.

In any case, it is always good to know about the help function:

[3]:
```
help(data.plot_station_coordinates)
```

```
Help on method plot_station_coordinates in module pyaerocom.ungriddeddata:

plot_station_coordinates(var_name=None, filter_name=None, start=None, stop=None, ts_
↪type=None, color='r', marker='o', markersize=8, fontsize_base=10, **kwargs) method of␣
↪pyaerocom.ungriddeddata.UngriddedData instance
    Plot station coordinates on a map

    All input parameters are optional and may be used to add constraints
    related to which stations are plotted. Default is all stations of all
    times.

    Parameters
    ----------

    var_name : :obj:`str`, optional
        name of variable to be retrieved
    filter_name : :obj:`str`, optional
        name of filter (e.g. EUROPE-noMOUNTAINS)
    start
        start time (optional)
    stop
        stop time (optional). If start time is provided and stop time not,
        then only the corresponding year inferred from start time will be
        considered
    ts_type : :obj:`str`, optional
```

(continues on next page)

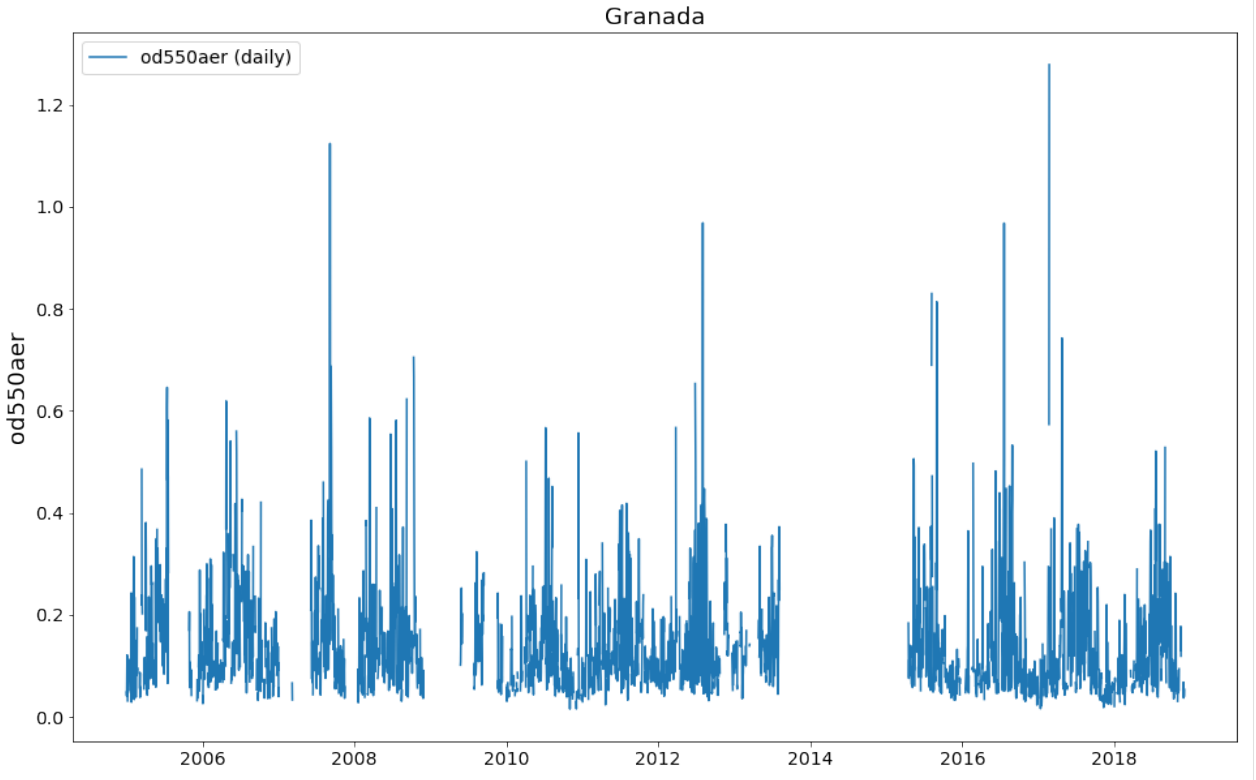
(continued from previous page)

```
        temporal resolution
    color : str
        color of stations on map
    marker : str
        marker type of stations
    markersize : int
        size of station markers
    fontsize_base : int
        basic fontsize
    **kwargs
        Addifional keyword args passed to
        :func:`pyaerocom.plot.plot_coordinates`

    Returns
    -------
    axes
        matplotlib axes instance
```

Quicklook plotting of station timeseries

Time series of individual stations can be plotted as follows:

[4]:
```
data.plot_station_timeseries(station_name='Granada', var_name='od550aer');
```



Access metadata of the data files that were read

Look into the metadata of the different files. Metadata can be accessed via the `metadata` attribute, and there is one metadatadictionary for each file that was read:

[5]:
```
len(data.metadata)
```

[5]:
```
1230
```

Access metadata of first file (index 0):

[6]:
```
data.metadata[0]
```

[6]:
```
{'var_info': OrderedDict([('od550aer', OrderedDict([('units', '1')]))]),
 'latitude': 45.3139,
 'longitude': 12.508299999999998,
 'altitude': 10.0,
 'station_name': 'AAOT',
 'PI': 'Brent_Holben',
 'ts_type': 'daily',
 'data_id': 'AeronetSunV3Lev2.daily',
 'variables': ['od550aer'],
 'instrument_name': 'sun_photometer',
 'data_revision': '20190920'}
```

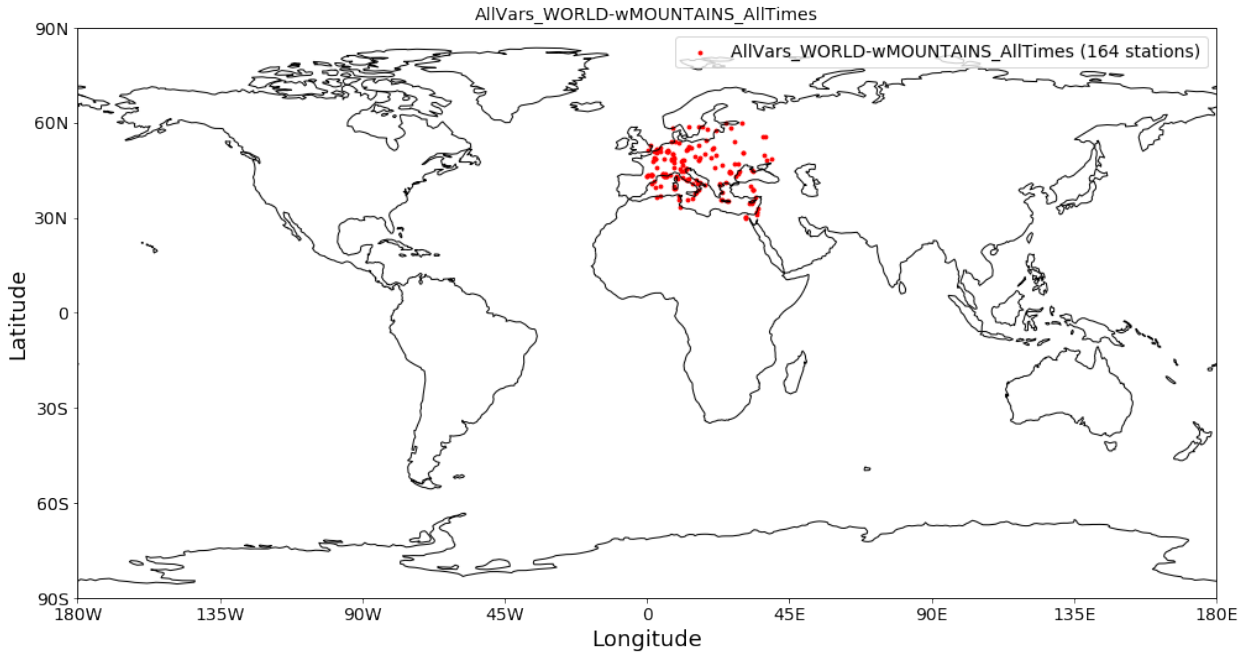
Filtering of the data

So far, you can filter `UngriddedData` objects by common metadata attributes. For instance:

[7]:
```
subset = data.filter_by_meta(latitude=(30, 60), longitude=(0, 45), altitude=(0, 1000))
print(subset)
```

```
Pyaerocom UngriddedData
-----------------------
Contains networks: ['AeronetSunV3Lev2.daily']
Contains variables: ['od550aer']
Contains instruments: ['sun_photometer']
Total no. of meta-blocks: 164
Filters that were applied:
 Filter time log: 20191002122003
        Created od550aer single var object from multivar UngriddedData instance
 Filter time log: 20191003180107
        latitude: (30, 60)
        longitude: (0, 45)
        altitude: (0, 1000)
```

[8]:
```
subset.plot_station_coordinates();
```



Other attributes that may be useful

Access all station names and print the first 4:

[9]:
```
stat_names = data.station_name
print(stat_names[:4])
```

```
['AAOT', 'AOE_Baotou', 'ARM_Ascension_Is', 'ARM_Barnstable_MA']
```

Essentially, what `data.station_name` does is, it iterates over all metadata-dictionaries (that are stored in `data.metadata`, and are organised *per file that was read*) and extracts the `station_name` attribute and appends it to a list which is then returned by the method.

Hence, the list of station names corresponds to the list of metadata-blocks / files that are stored in the data object:

[10]:
```
len(stat_names)
```

[10]:
```
1230
```

In a similar manner, you can access coordinates latitude, longitude and altitude arrays for all files.

[11]:
```
lons = data.longitude
lons[:4]
```

[11]:
```
[12.508299999999998,
 109.62879999999998,
 -14.349805999999996,
 -70.29006100000001]
```

[12]:
```
lats = data.latitude
lats[:4]
```

[12]:
```
[45.3139, 40.851699999999994, -7.966963999999998, 41.669588999999995]
```

[13]:
```
alts = data.altitude
alts[:4]
```

[13]:
```
[10.0, 1314.0, 341.0, 15.0]
```

List of unique station names

As mentioned earlier, some databases provide more than one data file per station. Since the ungridded reading (see previous) tutorial is done *per data file*, this means that their can be more than one metadata-block per station (not the case here, though). In any case, you can get a list of unique station names using:

[14]:
```
unique_names = data.unique_station_names
unique_names[:4]
```

[14]:
```
['AAOT', 'AOE_Baotou', 'ARM_Ascension_Is', 'ARM_Barnstable_MA']
```

`StationData`: Access the data from individual stations

As you could see above the metadata dictionaries in the `UngriddedData` class for each file do only contain the associated metadata. For the sake of performance the actual data arrays are all stored in one big 2D numpy array (which does not need to bother you too much) which is accessible in the `_data` attribute of the `UngriddedData` object (if you like to dive into it).

In most cases that concern model evaluation, the observation data is analysed station-by-station. For this purpose the StationData class was designed, which is introduced below.

Starting from an instance of the `UngriddedData` object, the individual station data (i.e. time series of one or more variables + metadata) can be accessed using the method:

UngriddedData.to_station_data

or using the square brackets `[]` which is equivalent to the former as it is only a wrapper for `to_station_data`. This meants, calling

```
UngriddedData[0]
```

will give you the same output as

```
UngriddedData.to_station_data[0]
```

that is, the data associated with the first file that was read (i.e. the first metadata-block in the object) into the `UngriddedData` object (see previous tutorial for details regarding the reading of ungridded observation networks).

To specify the station, you can either use the metadata index of the corresponding data file (`meta_idx=9`, for 10th file) **or** you can specify the station name or a wildcard specifying the station name.

The method returns a pyaerocom.StationData object, which is a dictionary-like object which contains data vectors and time-stamps as well as metadata.

Below we will illustrate the several options to access station data (and show that they contain the same data):

Option 1. Get station data using the corresponding metadata indices that match the station name

Find index (or indices) that match the station name:

[15]:
```
index = data.find_station_meta_indices('Granada')
index
```

[15]:
```
[497]
```

The result shows that there is one file that matches this station name (as we would expect for AERONET data) and the corresponding metadata index is 488.

To access the data, you can use the method `to_station_data`. It helps to have a look into the options of this method:

[16]:
```
help(data.to_station_data)
```

```
Help on method to_station_data in module pyaerocom.ungriddeddata:

to_station_data(meta_idx, vars_to_convert=None, start=None, stop=None, freq=None, merge_
↪if_multi=True, merge_pref_attr=None, merge_sort_by_largest=True, insert_nans=False,␣
↪**kwargs) method of pyaerocom.ungriddeddata.UngriddedData instance
    Convert data from one station to :class:`StationData`

    Todo
    ----
    - Review for retrieval of profile data (e.g. Lidar data)

    Parameters
    ----------
    meta_idx : float
        index of station or name of station.
    vars_to_convert : :obj:`list` or :obj:`str`, optional
        variables that are supposed to be converted. If None, use all
        variables that are available for this station
    start
        start time, optional (if not None, input must be convertible into
        pandas.Timestamp)
    stop
        stop time, optional (if not None, input must be convertible into
        pandas.Timestamp)
    freq : str
        pandas frequency string (e.g. 'D' for daily, 'M' for month end) or
        valid pyaerocom ts_type
    merge_if_multi : bool
        if True and if data request results in multiple instances of
        StationData objects, then these are attempted to be merged into one
        :class:`StationData` object using :func:`merge_station_data`
    merge_pref_attr
        only relevant for merging of multiple matches: preferred attribute
        that is used to sort the individual StationData objects by relevance.
        Needs to be available in each of the individual StationData objects.
        For details cf. :attr:`pref_attr` in docstring of
        :func:`merge_station_data`. Example could be `revision_date`. If
        None, then the stations will be sorted based on the number of
        available data points (if :attr:`merge_sort_by_largest` is True,
```

(continues on next page)

(continued from previous page)

```
        which is default).
    merge_sort_by_largest : bool
        only relevant for merging of multiple matches: cf. prev. attr. and
        docstring of :func:`merge_station_data` method.
    insert_nans : bool
        if True, then the retrieved :class:`StationData` objects are filled
        with NaNs

    Returns
    -------
    StationData or list
        StationData object(s) containing results. list is only returned if
        input for meta_idx is station name and multiple matches are
        detected for that station (e.g. data from different instruments),
        else single instance of StationData. All variable time series are
        inserted as pandas Series
```

So the first input argument takes either the metadata index, or the name of the station. Here we use the metadata index option using the index that we just retrieved:

[17]:
```
granada_opt1 = data.to_station_data(meta_idx=index[0], insert_nans=True)
type(granada_opt1)
```

[17]:
```
pyaerocom.stationdata.StationData
```

The returned data type is an instance of the pyaerocom.StationData class.

NOTE: if there is more than one index match for one station (i.e. `data.find_station_meta_indices('Granada')` returns more than one match), then, using Option 1, you would need to call `to_station_data` for each of the index matches. Alternatively you could use either of the following methods, which automatically merge the individual `StationData` objects into one, in case of multiple matches for that station name.

Let's have a quick look at the `StationData` object (it is a dictionary-like object and simple to use):

Option 2: Retrieve station data using the station name directly

[18]:
```
granada_opt2 = data.to_station_data('Granada', insert_nans=True)
```

Other than option 1, in case of multiple meta-index matches, this method automatically merges the individual data objects.

Option 3: Use [...] notation

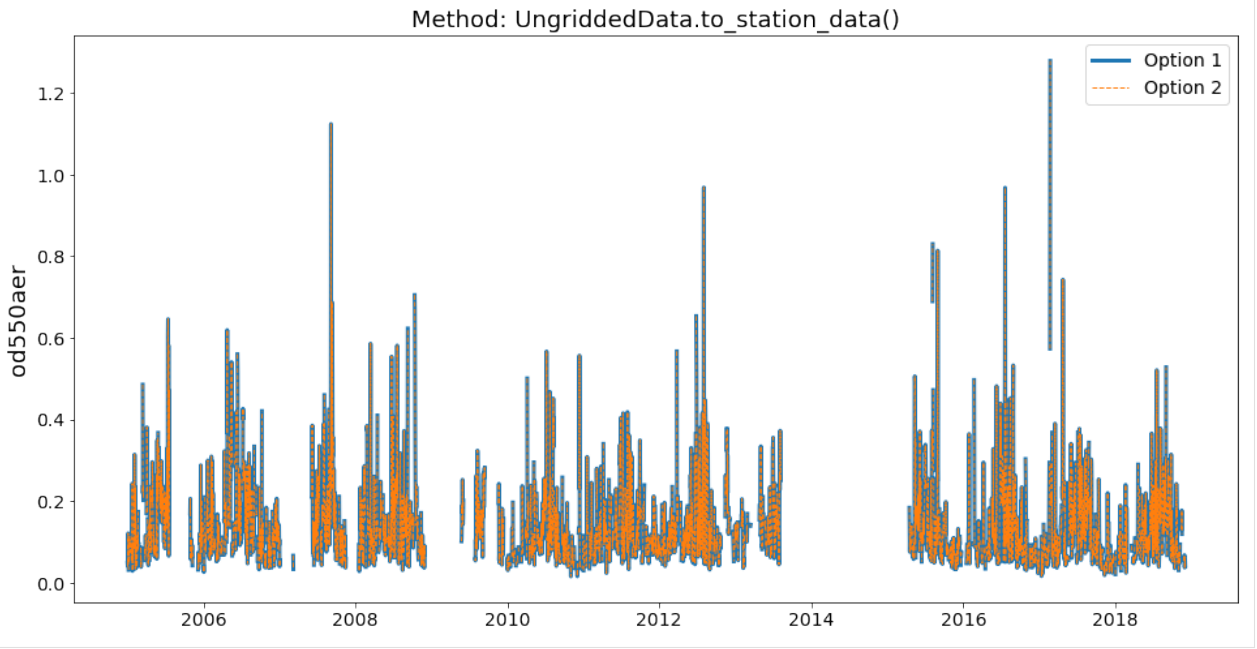
This is a wrapper for the method `to_station_data` so you may use meta-index or station name for access.

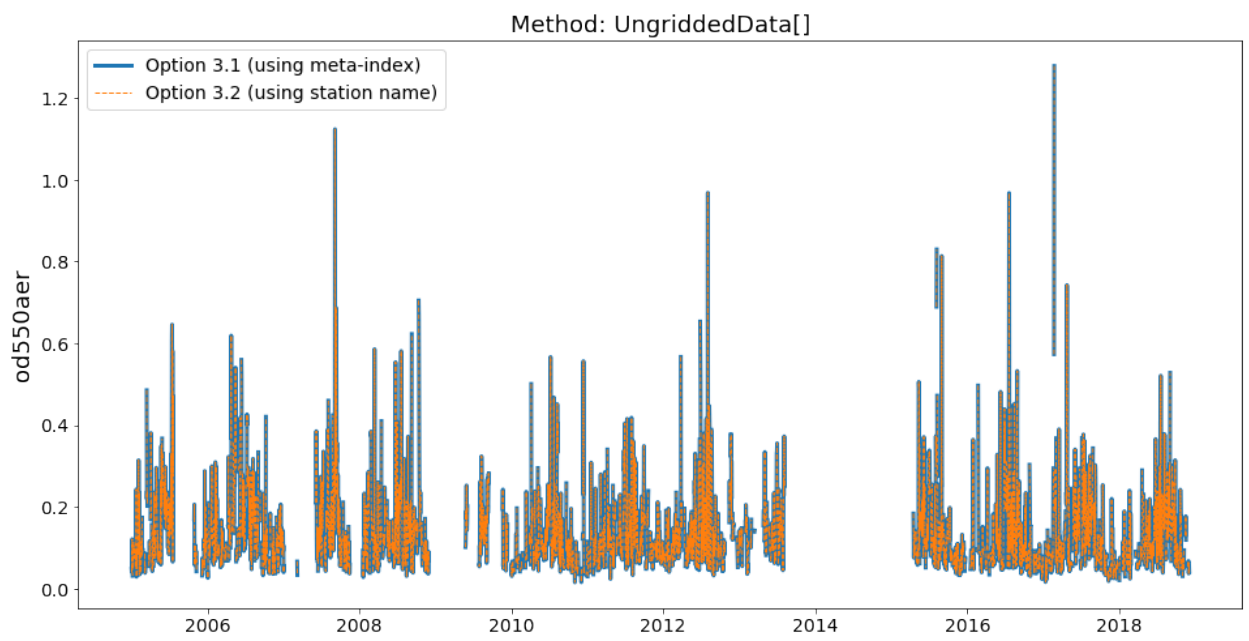
[19]:
```
granada_opt3_1 = data[index[0]]
granada_opt3_2 = data['Granada']
```

Let's have a look if the data objects are really the same (by plotting the AOD timeseries for the 4 different options):

[20]:
```
ax = granada_opt1.plot_timeseries('od550aer', lw=3, label='Option 1', tit='Method:␣
↪UngriddedData.to_station_data()')
granada_opt2.plot_timeseries('od550aer', ls='--', lw=1, label='Option 2',ax=ax)

# plot the results from the [] access option into a new figure (don't pass ax)
ax = granada_opt3_1.plot_timeseries('od550aer', lw=3, label='Option 3.1 (using meta-
↪index)',
                                    tit='Method: UngriddedData[]')
granada_opt3_2.plot_timeseries('od550aer', ls='--', lw=1, label='Option 3.2 (using␣
↪station name)',ax=ax)
```

[20]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3a66706a20>
```





Looks good. Let's explore the `StationData` object a little more (you can print it and it will get you a nice overview):

[21]:
```
print(granada_opt1)
```

```
Pyaerocom StationData
---------------------
var_info (BrowseDict):
  od550aer (OrderedDict):
    units: 1
    overlap: False
    ts_type: daily
    apply_constraints: False
    min_num_obs: None
station_coords (dict):
  latitude: 37.163999999999994
  longitude: -3.6049999999999995
  altitude: 680.0
data_err (BrowseDict): <empty_dict>
overlap (BrowseDict): <empty_dict>
data_flagged (BrowseDict): <empty_dict>
filename: None
station_id: None
station_name: Granada
instrument_name: sun_photometer
PI: Brent_Holben
country: None
ts_type: daily
latitude: 37.163999999999994
longitude: -3.6049999999999995
altitude: 680.0
data_id: AeronetSunV3Lev2.daily
```

(continues on next page)

(continued from previous page)

```
dataset_name: None
data_product: None
data_version: None
data_level: None
revision_date: None
website: None
ts_type_src: daily
stat_merge_pref_attr: None
data_revision: 20190920

Data arrays
...
dtime (ndarray, 5087 items): [2004-12-30T00:00:00.000000000, 2004-12-31T00:00:00.
↪000000000, ..., 2018-12-02T00:00:00.000000000, 2018-12-03T00:00:00.000000000]
Pandas Series
...
od550aer (Series, 5087 items)
```

You can see that the `StationData` object contains both metadata (e.g. `PI`) and data vectors which can be either 1D numpy arrays or python lists (e.g. `dtime`) or `pandas.Series` objects (e.g. variable `od550aer`). All attributes can be accessed and manipulated either using dictionary style access (i.e. `[]` notation), or using the `.` operator.

Here some examples:

[22]:
```
# get longitude using "[]" notation
granada_opt1['longitude']
```

[22]:
```
-3.60499999999999995
```

[23]:
```
# get longitude using "." notation
granada_opt1.longitude
```

[23]:
```
-3.60499999999999995
```

[24]:
```
# assign longitude using "." notation and display new value (again using "[]" notation)
granada_opt1.longitude = 42
granada_opt1['longitude']
```

[24]:
```
42
```

[25]:
```
granada_opt1['station_name']
```

[25]:
```
'Granada'
```

Get station name:

[26]:
```
granada_opt1.station_name
```

[26]:
```
'Granada'
```

Small detour through the pandas world

As you can see in the output above, the time-series data in the `StationData` object is an instance of the `pandas.Series` class.

[27]:
```
aod_data = granada_opt1.od550aer
type(aod_data)
```

[27]:
```
pandas.core.series.Series
```

NOTE: pyaerocom relies on pandas, so if you are not familiar with the pandas library, it is a good advice to make yourself familiar with it (especially if you are interested in timeseries analysis). See here for a short introduction into pandas.

Anyways, you should know about the 2 basic datatypes of pandas which are:

- pandas.Series
- pandas.DataFrame

Both objects are very similar in their handling and the `Series` class can be imagined as a *single column* of the `Dataframe` object which is a table-like object that has columns (e.g. variables) and rows (e.g. time-stamps). It is hence, easy to go back and forth between the two objects.

Anyways, here is some examples what you can do with an instance of the `pandas.Series` object that we just accessed from the `pyaerocom.StationData` object from Granada (and which we named `aod_data`).

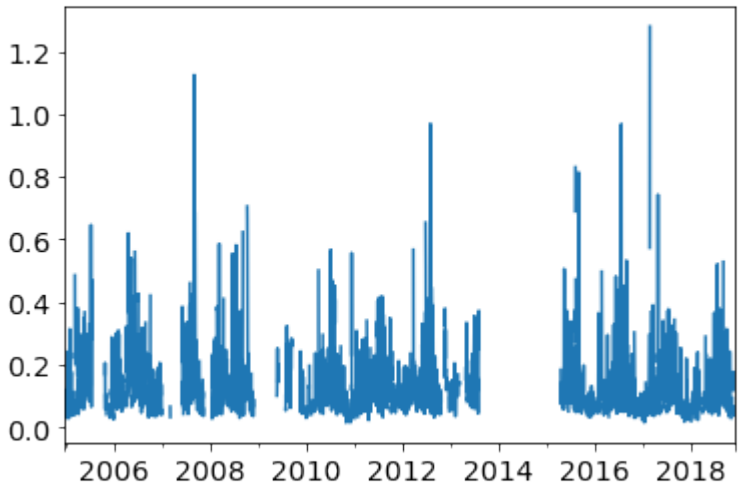
First, you can get a basic clue about the data by using the `describe` method:

[28]:
```
aod_data.describe()
```

[28]:
```
count    3010.000000
mean        0.137759
std         0.103330
min         0.015534
25%         0.070969
50%         0.108200
75%         0.171291
max         1.278507
dtype: float64
```

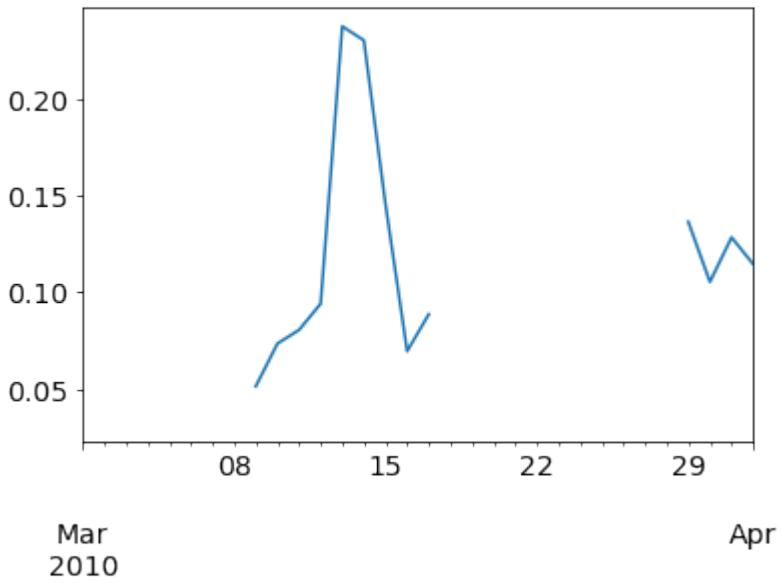
Second, you may plot it:

[29]:
```
aod_data.plot();
```



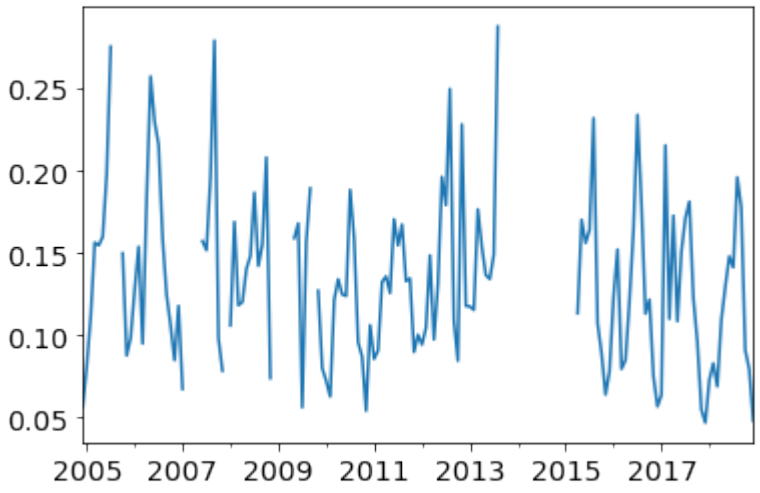
Third, you may extract subsets using *fancy indexing*:

[30]:
```
aod_data_march2010 = aod_data['2010-3-1':'2010-4-1']
aod_data_march2010.plot();
```



Or fourth, resample to another frequency:

[31]:
```
aod_data_monthly = aod_data.resample('M', 'mean')
aod_data_monthly.plot();
```



Or fifth, resample to lower frequency, but require a minimum number of observations per period:

[32]:
```
monthly_with_count = aod_data.resample('M').agg(['mean', 'count'])
monthly_with_count.head()
```

[32]:
```
                mean  count
2004-12-31  0.056462      1
2005-01-31  0.081300     28
2005-02-28  0.111567     24
2005-03-31  0.156312     17
2005-04-30  0.154527     26
```

Now, here you see an example, where `pandas` automatically converted our Series (which is single variable) to a DataFrame (which is a table), since we told the resampler above, to aggregate monthly mean and monthly count.

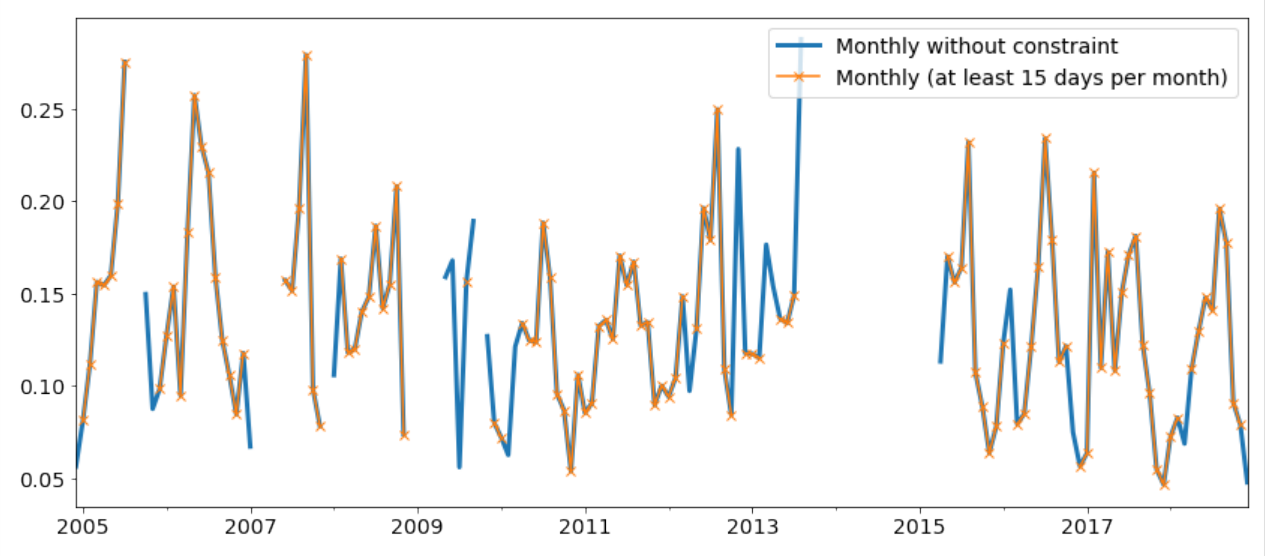
Now let's say we require at least 15 observations (here, days, since our original dataset is in daily resolution) per month:

[33]:
```
invalid_mask = monthly_with_count['count'] < 15
monthly_with_count['mean'][invalid_mask] = np.nan
aod_monthly_min15d = monthly_with_count['mean']
aod_monthly_min15d.head()
```

[33]:
```
2004-12-31         NaN
2005-01-31    0.081300
2005-02-28    0.111567
2005-03-31    0.156312
2005-04-30    0.154527
Freq: M, Name: mean, dtype: float64
```

Now plot both the monthly timeseries from above without constraint and the one with constraint:

[34]:
```
ax = aod_data_monthly.plot(label='Monthly without constraint', lw=3, figsize=(14,6))
aod_monthly_min15d.plot(ax=ax, style='x-', label='Monthly (at least 15 days per month)')
ax.legend();
```



As you can see, there is quite some months missing when applying the filter.

You may also be intersted in a climatology:

[35]:
```
aod_monthly_climatology = aod_data_monthly.groupby(aod_data_monthly.index.month).mean()
aod_monthly_climatology
```

[35]:
```
1     0.091885
2     0.125760
3     0.114174
4     0.132566
5     0.146915
6     0.164997
7     0.174380
8     0.192138
9     0.146022
10    0.115006
11    0.091344
12    0.082346
dtype: float64
```

And do the same for the monthly data with minimum 15 days per month that we created above:

[36]:
```
aod_monthly_climatology_min15d = aod_monthly_min15d.groupby(aod_monthly_min15d.index.
↪month).mean()
aod_monthly_climatology_min15d
```

[36]:
```
1     0.093066
2     0.130359
3     0.119908
4     0.136720
5     0.145820
6     0.164751
7     0.184243
8     0.183453
9     0.141699
10    0.111532
```

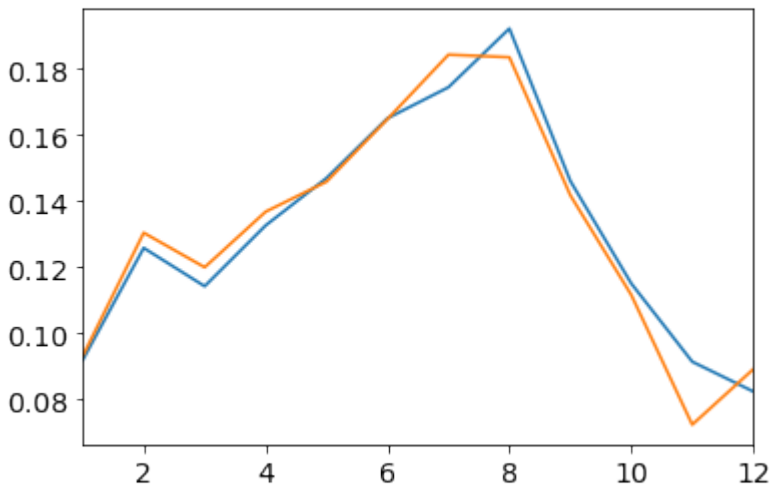
(continues on next page)

(continued from previous page)

```
11    0.072294
12    0.089056
Name: mean, dtype: float64
```

[37]:
```
ax = aod_monthly_climatology.plot(label='Monthly climatology (no constraint)')
aod_monthly_climatology_min15d.plot(label='Monthly climatology (min 15 days/month)')
```

[37]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3a6f89a6a0>
```

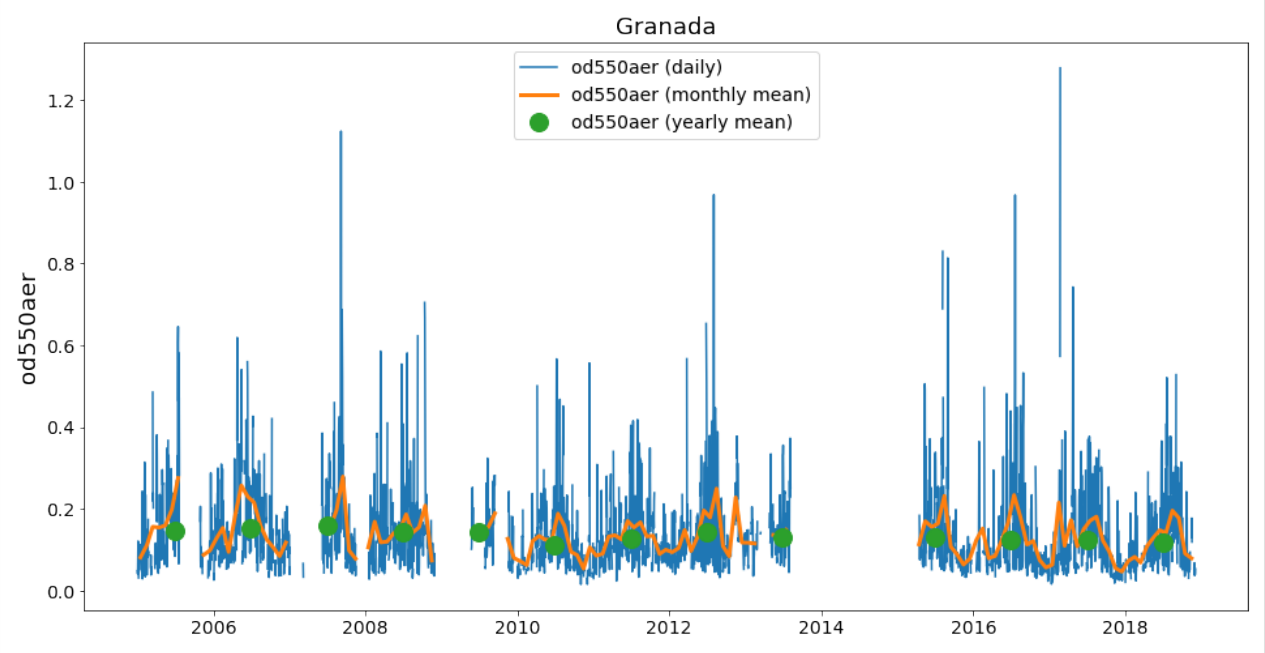


That was enough of a detour into the pandas world. As you shall see below, some of these pandas features are also provided in the pyaerocom data objects (e.g. resampling) and more will follow soon!

Plotting of timeseries data directly from StationData class

Let's come back to the `StationData` object. Below are some more examples that show how you can plot the timeseries directly from the `StationData` object. This includes to do a resampling out of the box when plotting:

[38]:
```
ax = granada_opt1.plot_timeseries('od550aer')
ax = granada_opt1.plot_timeseries('od550aer', freq='monthly', lw=3, ax=ax)
granada_opt1.plot_timeseries('od550aer', freq='yearly', ls='none', marker='o', ms=14,␣
↪ax=ax);
```



The resampling of the timeseries in the plotting method is done automatically (if input `ts_type` is provided).

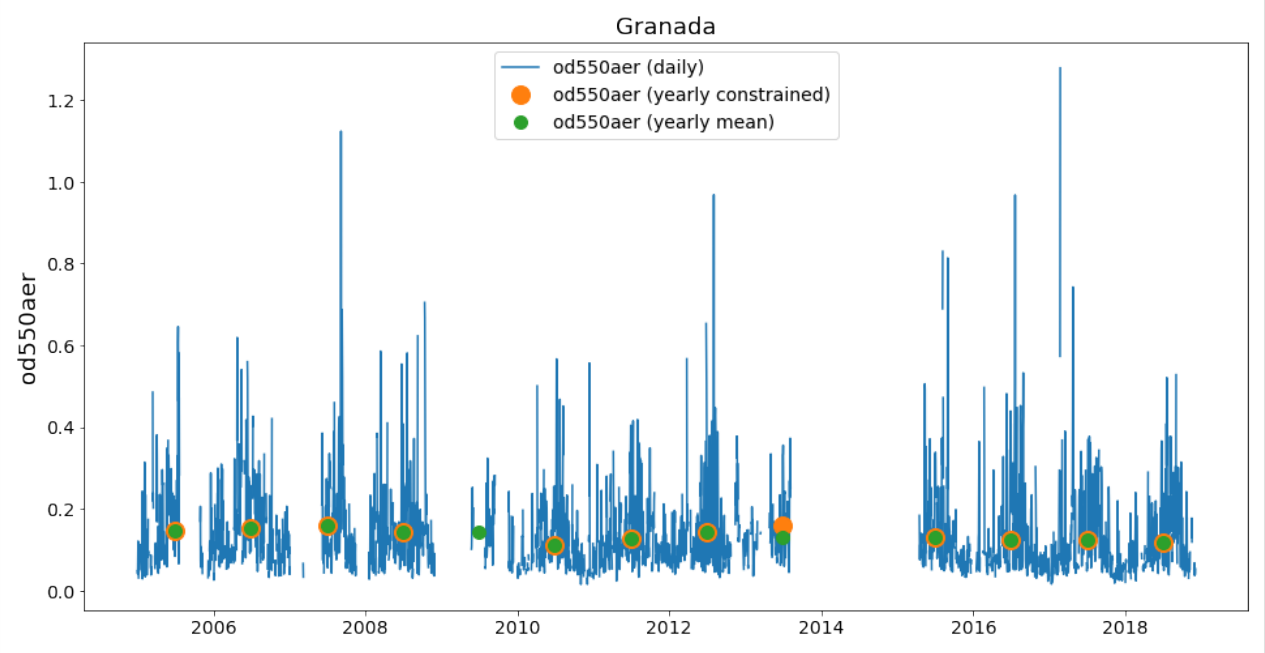
Currently, this does not apply additional constraints such as a minimum number of available observations when down-sampling (like we showed above). From the yearly data (green dots) you can see clearly that this can be an issue, especially for the first and the last year.

In order to account for it, you may to the following:

[39]:
```
# convert to monthly with at least 5 days per month
od550aer_monthly = granada_opt1.resample_timeseries('od550aer', 'monthly', 'mean', min_
↪num_obs=5)

# assign to our StationData
granada_opt1['od550aer_monthly'] = od550aer_monthly

# convert to yearly with at least 6 months per year
od550aer_yearly_constrained = granada_opt1.resample_timeseries('od550aer_monthly',
↪'yearly', 'mean', min_num_obs=6)
```

Compare the result with the yearly product plotted above:

[40]:
```
ax = granada_opt1.plot_timeseries('od550aer')
ax.plot(od550aer_yearly_constrained, ls='none', marker='o', ms=14, label='od550aer␣
↪(yearly constrained)')
ax = granada_opt1.plot_timeseries('od550aer', freq='yearly', ls='none', marker='o',␣
↪ms=10, ax=ax)
```



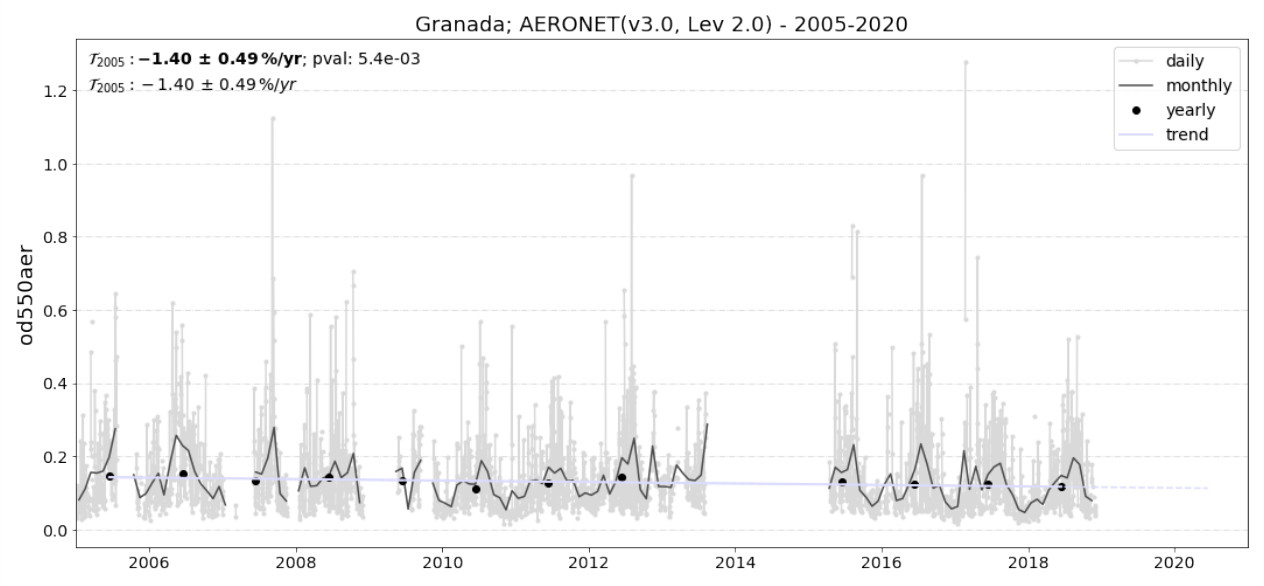
Computing trends from station data

[41]:
```
granada_opt1.compute_trend(var_name='od550aer', start_year=2005, stop_year=2020)
```

[41]:
```
{'pval': 0.005380307706696595,
 'm': -0.0019984909545023464,
 'm_err': 0.0006859706833670536,
 'n': 12,
 'y_mean': 0.13274217888172632,
 'y_min': 0.11068644396371304,
 'y_max': 0.15450395909117362,
 'coverage': None,
 'slp': -1.3972849007872112,
 'slp_err': 0.48509627152652673,
 'reg0': 0.14302673373028105,
 't0': None,
 'slp_2005': -1.3972849007872112,
 'slp_2005_err': 0.48509627152652673,
 'reg0_2005': 0.14302673373028105,
 'yoffs': 0.21297391713786318,
 'period': '2005-2020'}
```

[42]:
```
granada_opt1.trends['od550aer'].plot()
```

[42]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3a6ccd50b8>
```



3.2.6 Colocation of model data with observations

This notebook gives an introduction into collocation of gridded data with observations. Here, the 550 nm AODs of the ECMWF CAMS reanalysis model are compared with global daily AeroNet Sun V2 (Level 2) data for the year 2010. The collocated data will be analysed and visualised in monthly resolution. The analysis results will be plotted in the form of the well known Aerocom loglog scatter plots as can be found in the online interface (see e.g. here).

Import setup and imports

[1]:
```
import pyaerocom as pya
pya.change_verbosity('critical')

YEAR = 2010
VAR = "od550aer"
TS_TYPE = "daily"
MODEL_ID = "ECMWF_CAMS_REAN"
OBS_ID = 'AeronetSunV3Lev2.daily'
```

```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.017 s
```

Import of model data

Create reader instance for model data and print overview of what is in there.

[2]:
```
model_reader = pya.io.ReadGridded(MODEL_ID)
print(model_reader)
```

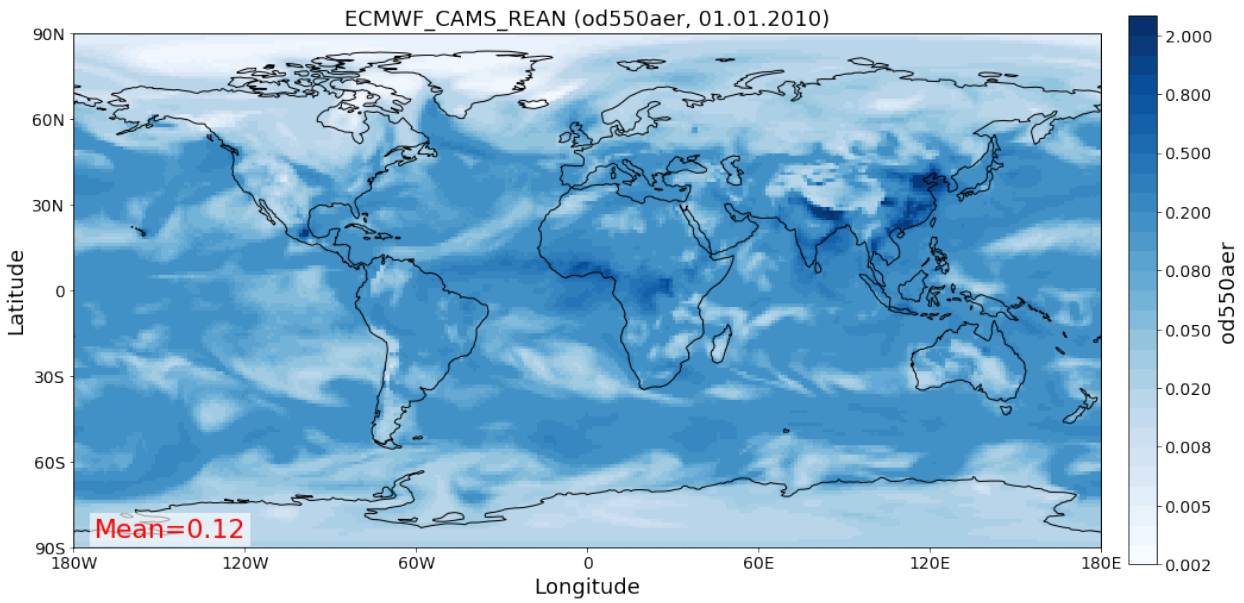
```
Pyaerocom ReadGridded
---------------------
Data ID: ECMWF_CAMS_REAN
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/ECMWF/ECMWF_CAMS_
↪REAN/renamed
Available experiments: ['', 'REAN']
Available years: [2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014,
↪ 2015, 2016, 2017, 2018, 2019, 9999]
Available frequencies ['daily' 'monthly']
Available variables: ['ang4487aer', 'bscatc532aerboa', 'bscatc532aertoa', 'ec532aer',
↪'ec532dryaer', 'od440aer', 'od550aer', 'od550bc', 'od550dust', 'od550oa', 'od550so4',
↪'od550ss', 'od865aer', 'sconcbc', 'sconcdust', 'sconcoa', 'sconcpm10', 'sconcpm25',
↪'sconcso4', 'sconcss', 'time', 'z']
```

Since we are only interested in a single year we can use the method

[3]:
```
model_data = model_reader.read_var(VAR, start=YEAR)
#model_data = read_result[VAR][YEAR]
print(model_data)
```

```
pyaerocom.GriddedData: ECMWF_CAMS_REAN
Grid data: Aerosol optical depth at 550 nm / (1) (time: 365; latitude: 161; longitude:␣
↪320)
    Dimension coordinates:
        time                            x             -             -
        latitude                        -             x             -
        longitude                       -             -             x
    Attributes:
        Conventions: CF-1.6
        NCO: "4.5.4"
        computed: False
        concatenated: False
        data_id: ECMWF_CAMS_REAN
        from_files: ['/lustre/storeA/project/aerocom/aerocom-users-database/ECMWF/
↪ECMWF_CA...
        history: Sat May 26 21:08:48 2018: ncecat -O -u time -n 365,3,1 CAMS_REAN_001.
↪nc...
        nco_openmp_thread_number: 1
        outliers_removed: False
        reader: None
        region: None
        regridded: False
        ts_type: daily
        var_name_read: n/d
    Cell methods:
        mean: step
        mean: time
```

[4]:
```
fig = model_data.quickplot_map(time_idx=0)
```



Import of AeroNet Sun V3 data (Level 2)

Import Aeronet data and apply filter that selects only stations that are located at altitudes between 0 and 1000 m.

[5]:
```
obs_reader = pya.io.ReadUngridded(OBS_ID, [VAR, 'ang4487aer'])
obs_data = obs_reader.read().filter_by_meta(altitude=[0, 1000])
print(obs_data)
```

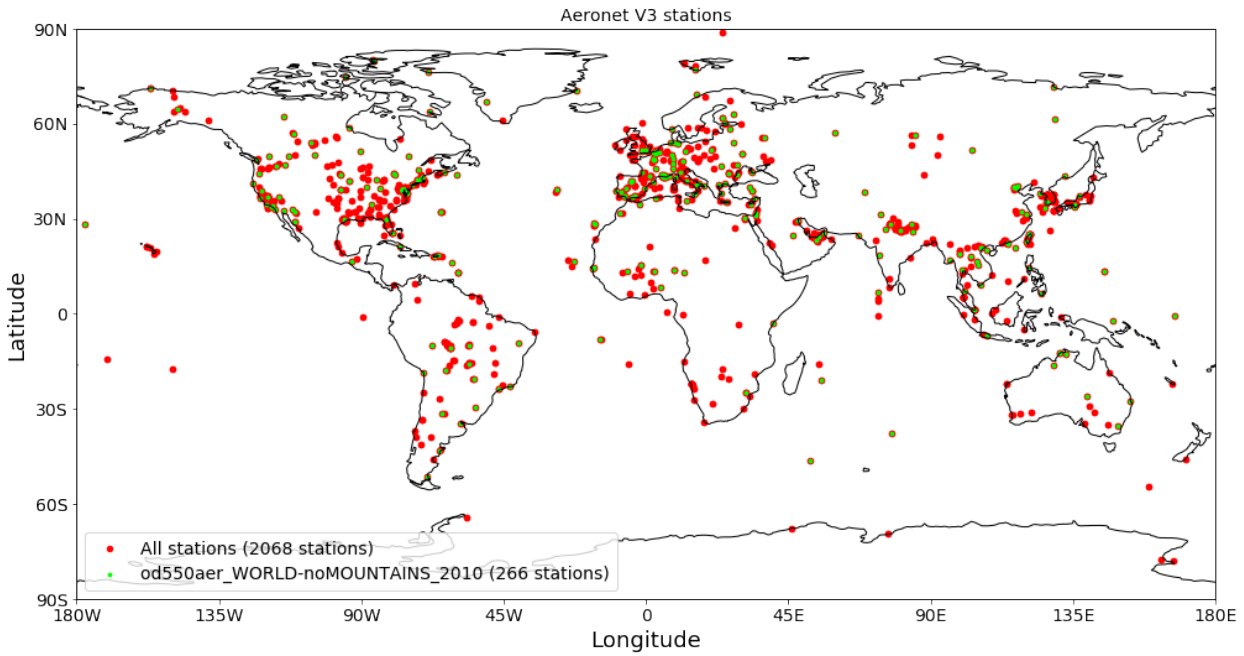
```
Pyaerocom UngriddedData
-----------------------
Contains networks: ['AeronetSunV3Lev2.daily']
Contains variables: ['od550aer', 'ang4487aer']
Contains instruments: ['sun_photometer']
Total no. of meta-blocks: 2068
Filters that were applied:
 Filter time log: 20191002122003
        Created od550aer single var object from multivar UngriddedData instance
 Filter time log: 20191002122002
        Created ang4487aer single var object from multivar UngriddedData instance
 Filter time log: 20191003180122
        altitude: [0, 1000]
```

Plot station coordinates

First, plot all stations that are available at all times (as red dots), then (on top of that in green), plot all stations that provide AODs in 2010.

[6]:
```
ax = obs_data.plot_station_coordinates(color='r', markersize=20,
                                       label='All stations')
ax = obs_data.plot_station_coordinates(var_name='od550aer', start=2010,
                                       filter_name='WORLD-noMOUNTAINS',
                                       color='lime', markersize=8, legend=True,
                                       title='Aeronet V3 stations',
                                       ax=ax) #just pass the GeoAxes instance that was␣
↪created in the first call
```

```
Input filters {'longitude': [-180, 180], 'latitude': [-90, 90], 'altitude': [-1000000.0,␣
↪1000.0]} result in unchanged data object
```



Perform colocation and plot corresponding scatter plots with statistical values

2010 monthly World no mountains

Colocate 2010 data in monthly resolution using (cf. green dots in station plot above).

[7]:
```
obs_data
```

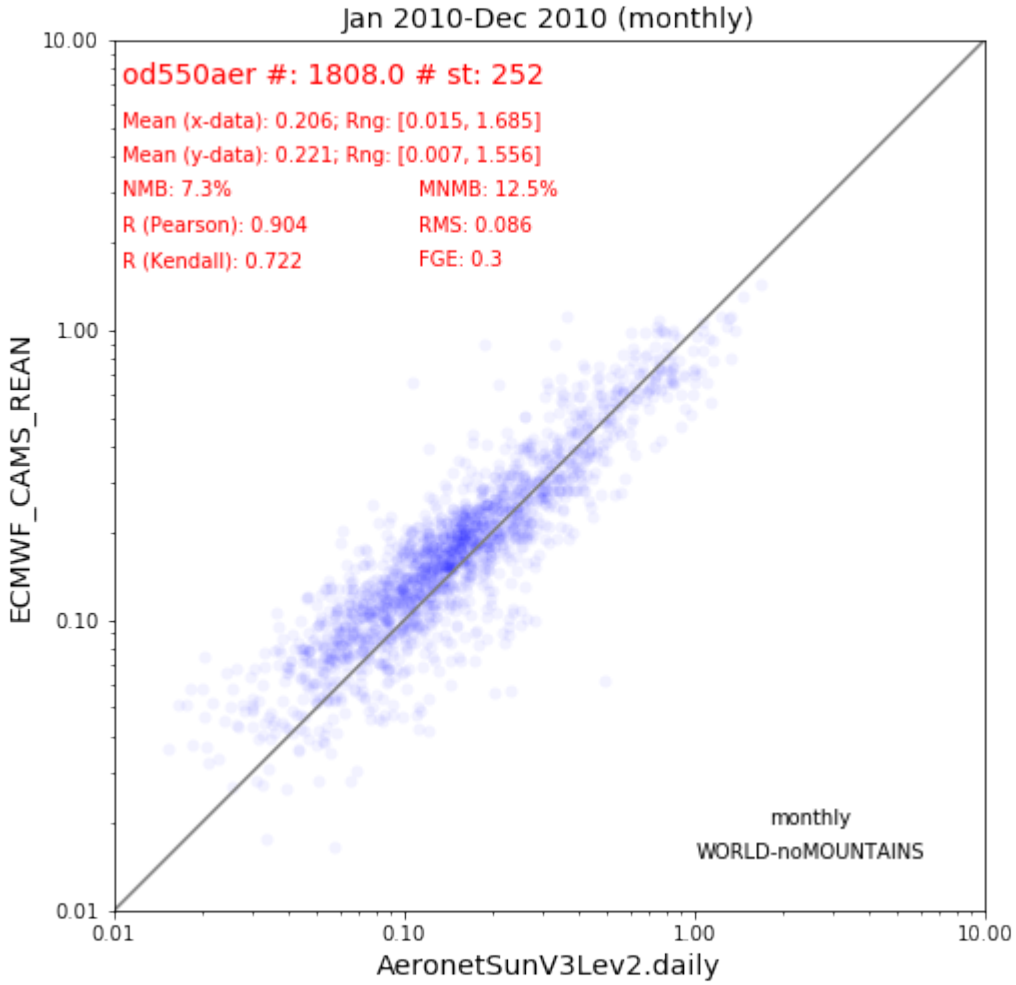
[7]:
```
UngriddedData <networks: ['AeronetSunV3Lev2.daily']; vars: ['od550aer', 'ang4487aer'];␣
↪instruments: ['sun_photometer'];No. of stations: 2068
```

[8]:
```
data_coloc = pya.colocation.colocate_gridded_ungridded(model_data, obs_data, ts_type=
↪'monthly',
                                                       filter_name='WORLD-noMOUNTAINS')
data_coloc
```

```
Input filters {'longitude': [-180, 180], 'latitude': [-90, 90], 'altitude': [-1000000.0,␣
↪1000.0]} result in unchanged data object

Setting od550aer outlier lower lim: -1.00

Setting od550aer outlier upper lim: 10.00

Interpolating data of shape (12, 161, 320). This may take a while.

Successfully interpolated cube
```

[8]:
```
<xarray.DataArray 'od550aer' (data_source: 2, time: 12, station_name: 252)>
array([[[     nan, 0.117588, ...,      nan, 0.222138],
        [     nan, 0.132128, ...,      nan, 0.429762],
        ...,
        [0.132236, 0.195057, ...,      nan, 0.261765],
        [     nan,      nan, ...,      nan, 0.37905 ]],

       [[0.189948, 0.140062, ..., 0.016372, 0.204337],
        [0.150408, 0.190089, ..., 0.035838, 0.257806],
        ...,
        [0.159844, 0.178564, ..., 0.022606, 0.239393],
        [0.147172, 0.138039, ..., 0.015231, 0.19986 ]]])
Coordinates:
  * data_source   (data_source) <U22 'AeronetSunV3Lev2.daily' 'ECMWF_CAMS_REAN'
    var_name      (data_source) <U8 'od550aer' 'od550aer'
    var_units     (data_source) <U1 '1' '1'
    ts_type_src   (data_source) <U5 'daily' 'daily'
  * time          (time) datetime64[ns] 2010-01-01 2010-02-01 ... 2010-12-01
  * station_name  (station_name) <U19 'ARM_Darwin' ... 'Zinder_Airport'
    latitude      (station_name) float64 -12.43 37.97 15.35 ... 62.45 13.78
    longitude     (station_name) float64 130.9 23.72 -1.479 ... -114.4 8.99
    altitude      (station_name) float64 29.9 130.0 305.0 ... 300.0 220.8 456.0
Attributes:
    data_source:        ['AeronetSunV3Lev2.daily', 'ECMWF_CAMS_REAN']
    var_name:           ['od550aer', 'od550aer']
    ts_type:            monthly
    filter_name:        WORLD-noMOUNTAINS
    ts_type_src:        ['daily', 'daily']
    start_str:          20100101
    stop_str:           20101231
    var_units:          ['1', '1']
    vert_scheme:        None
    data_level:         3
    revision_ref:       20190920
    from_files:         ['aerocom.ECMWF_CAMS_REAN.daily.od550aer.2010.nc']
    from_files_ref:     None
    stations_ignored:   None
    colocate_time:      False
    apply_constraints:  True
    min_num_obs:        {'yearly': {'monthly': 3}, 'monthly': {'daily': 7}, '...
    region:             WORLD
    lon_range:          [-180, 180]
    lat_range:          [-90, 90]
    alt_range:          [-1000000.0, 1000.0]
```

[9]:
```
data_coloc.plot_scatter(marker='o', mec='none', color='b', alpha=0.05);
```



Time colocation

The above colocation was performed based on monthly means, both from model and obs, at each station. However, if you look closely in the output you can see that both datasets are provided in daily resolution. You may colocate on a daily basis using the input argument `colocate_time`, in which case the model monthly means correspond to the mean value from the days where there were observations. This can (and most likely will) give you different results, since the observations may miss some days in the month, which is disregarded in the above monthly colocation routine:

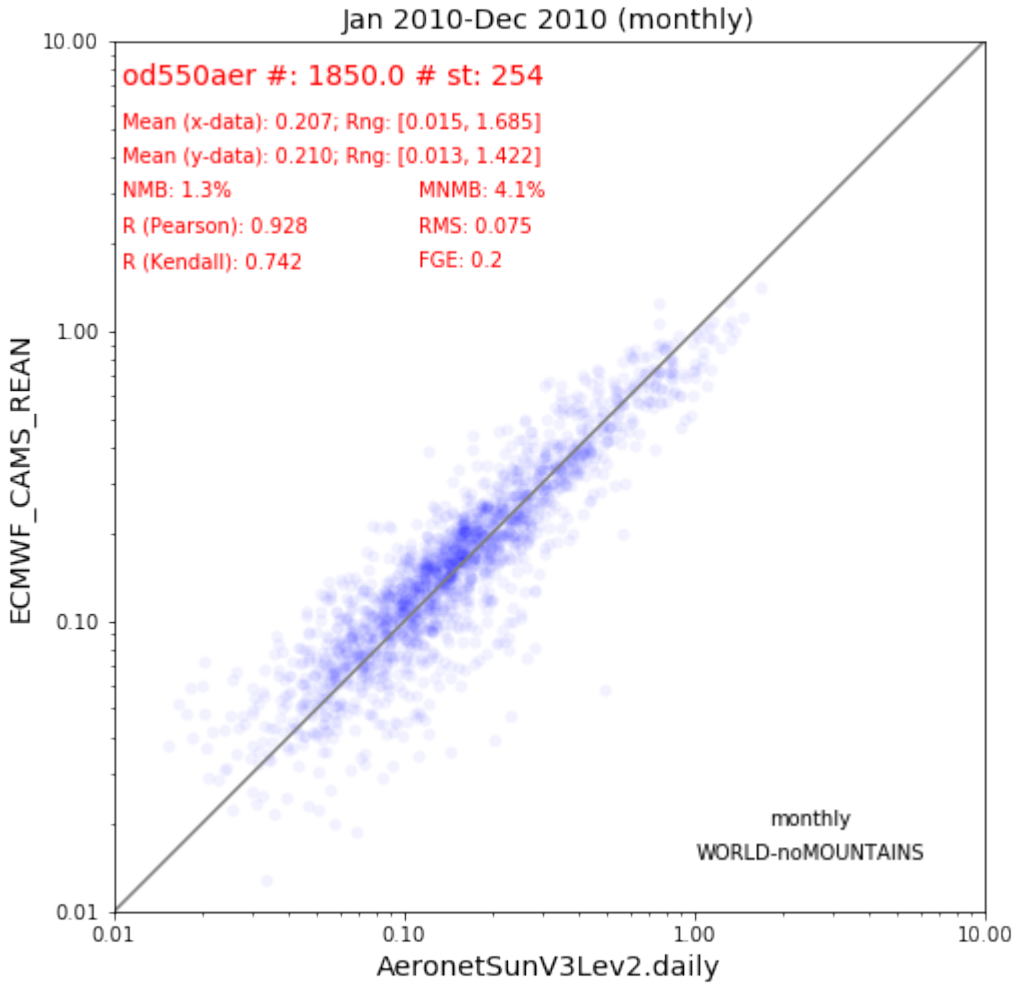
[10]:
```
data_coloc_alt = pya.colocation.colocate_gridded_ungridded(model_data, obs_data, ts_type=
↪'monthly',
                                                           filter_name='WORLD-noMOUNTAINS
↪',
                                                           colocate_time=True)
```

```
Input filters {'longitude': [-180, 180], 'latitude': [-90, 90], 'altitude': [-1000000.0,␣
↪1000.0]} result in unchanged data object
```

```
Setting od550aer outlier lower lim: -1.00
```

```
Setting od550aer outlier upper lim: 10.00
Interpolating data of shape (365, 161, 320). This may take a while.
Successfully interpolated cube
```

[11]:
```
data_coloc_alt.plot_scatter(marker='o', mec='none', color='b', alpha=0.05);
```



The result shows, that time colocation yields better results, with lower biases (NMB and MNMB) and higher correlation, etc.

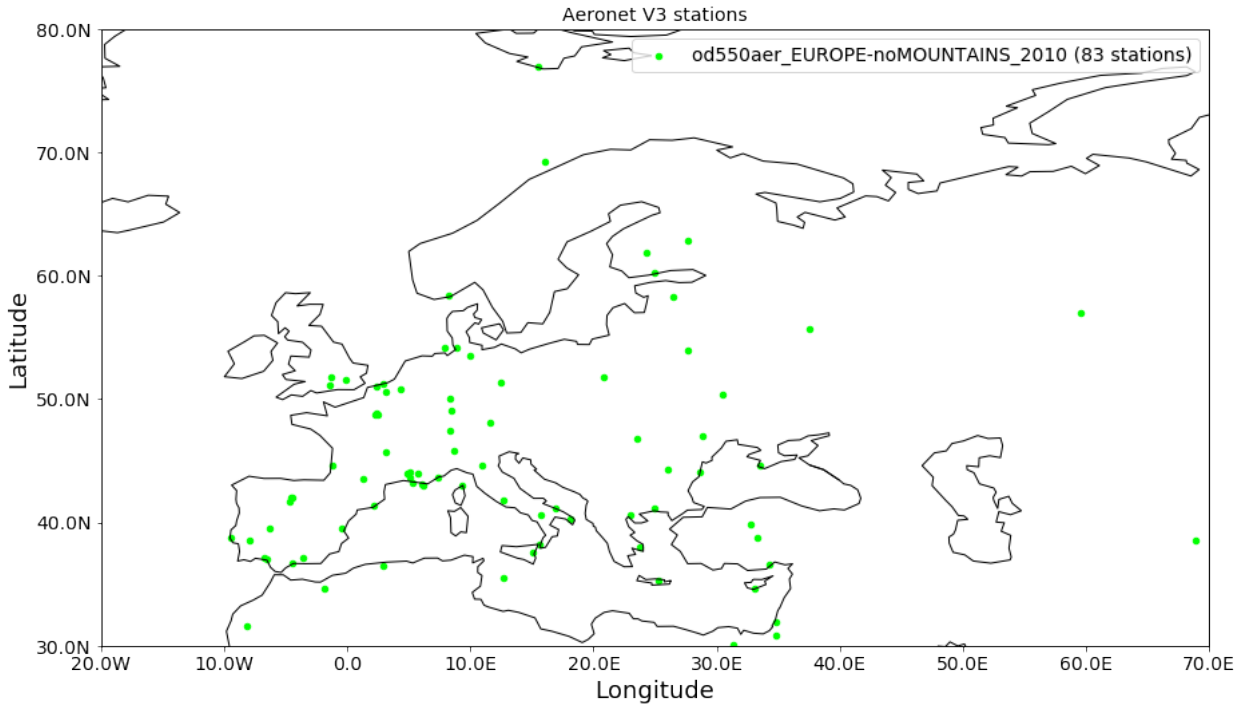
However, in reality and in particular in large model intercomparison studies (involving many variables and model outputs) the model diagnostics output files are submitted in monthly resolution, which does not allow to perform these time colocation on a daily basis.

Note also, that the model data used here is the CAMS reanalysis dataset which assimilates AERONET AODs. It is therefore not surprising, that the results look so shiny.

2010 daily Europe no mountains

Now perform colocation only over Europe. Starting with a station plot.

[12]:
```
obs_data.plot_station_coordinates(var_name='od550aer', start=2010,
                                  filter_name='EUROPE-noMOUNTAINS',
                                  color='lime', markersize=20, legend=True,
                                  title='Aeronet V3 stations');
```



[13]:
```
data_coloc_eur = pya.colocation.colocate_gridded_ungridded(model_data, obs_data, ts_type=
↪'daily',
                                                           filter_name='EUROPE-noMOUNTAINS')
data_coloc_eur
```

```
Setting od550aer outlier lower lim: -1.00
Setting od550aer outlier upper lim: 10.00
Interpolating data of shape (365, 161, 320). This may take a while.
Successfully interpolated cube
```

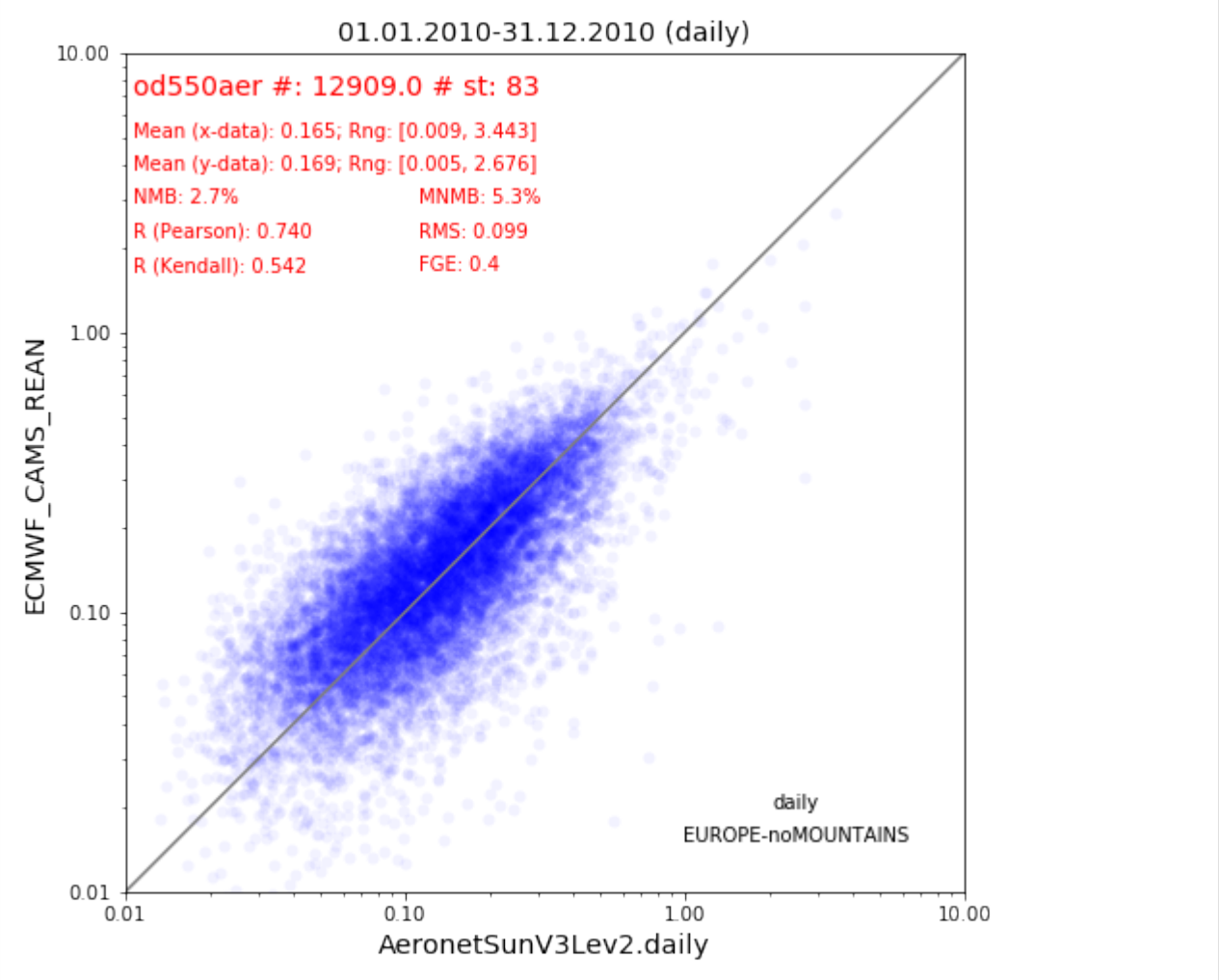
[13]:
```
<xarray.DataArray 'od550aer' (data_source: 2, time: 365, station_name: 83)>
array([[[     nan,      nan, ...,      nan,      nan],
        [0.078648,      nan, ...,      nan,      nan],
        ...,
        [     nan,      nan, ...,      nan,      nan],
        [     nan,      nan, ...,      nan,      nan]],

       [[0.086522, 0.015151, ..., 0.075447, 0.03005 ],
        [0.067198, 0.043074, ..., 0.103671, 0.042999],
        ...,
```

(continues on next page)

(continued from previous page)

```
       [0.242585, 0.186407, ..., 0.053797, 0.011344],
       [0.079498, 0.122098, ..., 0.027066, 0.019639]]])
Coordinates:
  * data_source   (data_source) <U22 'AeronetSunV3Lev2.daily' 'ECMWF_CAMS_REAN'
    var_name      (data_source) <U8 'od550aer' 'od550aer'
    var_units     (data_source) <U1 '1' '1'
    ts_type_src   (data_source) <U5 'daily' 'daily'
  * time          (time) datetime64[ns] 2010-01-01 2010-01-02 ... 2010-12-31
  * station_name  (station_name) <U19 'ATHENS-NOA' 'Andenes' ... 'Yekaterinburg'
    latitude      (station_name) float64 37.97 69.28 44.66 ... 51.77 41.15 57.04
    longitude     (station_name) float64 23.72 16.01 -1.163 ... 24.92 59.54
    altitude      (station_name) float64 130.0 379.0 11.0 ... 160.0 54.0 300.0
Attributes:
    data_source:        ['AeronetSunV3Lev2.daily', 'ECMWF_CAMS_REAN']
    var_name:           ['od550aer', 'od550aer']
    ts_type:            daily
    filter_name:        EUROPE-noMOUNTAINS
    ts_type_src:        ['daily', 'daily']
    start_str:          20100101
    stop_str:           20101231
    var_units:          ['1', '1']
    vert_scheme:        None
    data_level:         3
    revision_ref:       20190920
    from_files:         ['aerocom.ECMWF_CAMS_REAN.daily.od550aer.2010.nc']
    from_files_ref:     None
    stations_ignored:   None
    colocate_time:      False
    apply_constraints:  False
    min_num_obs:        None
    region:             EUROPE
    lon_range:          [-20, 70]
    lat_range:          [30, 80]
    alt_range:          [-1000000.0, 1000.0]
```

[14]:
```
data_coloc_eur.plot_scatter(marker='o', mec='none', color='b', alpha=0.05);
```



Satellite colocation

Below, the same is done for satellite colocation using AODs from the MODIS instrument onboard the Aqua satellite (Collection 6).

[15]:
```
pya.browse_database('MODIS6*aqua')
```

```
Pyaerocom ReadGridded
---------------------
Data ID: MODIS6.aqua
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/SATELLITE-DATA/
↪MODIS6.aqua/renamed
Available experiments: ['MODIS6.aqua']
Available years: [2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013,
↪ 2014]
Available frequencies ['daily']
Available variables: ['od550aer']
```

[16]:
```
modis_aods = pya.io.ReadGridded('MODIS6.aqua').read_var('od550aer', start=2010)
modis_aods
```

```
Overwriting unit unknown in cube od550aer with value "1"
```

[16]:
```
pyaerocom.GriddedData
Grid data: <iris 'Cube' of Aerosol Optical Thickness at 0.55 microns for both Ocean␣
↪(best) and Land (corrected): Mean / (1) (time: 365; latitude: 180; longitude: 360)>
```

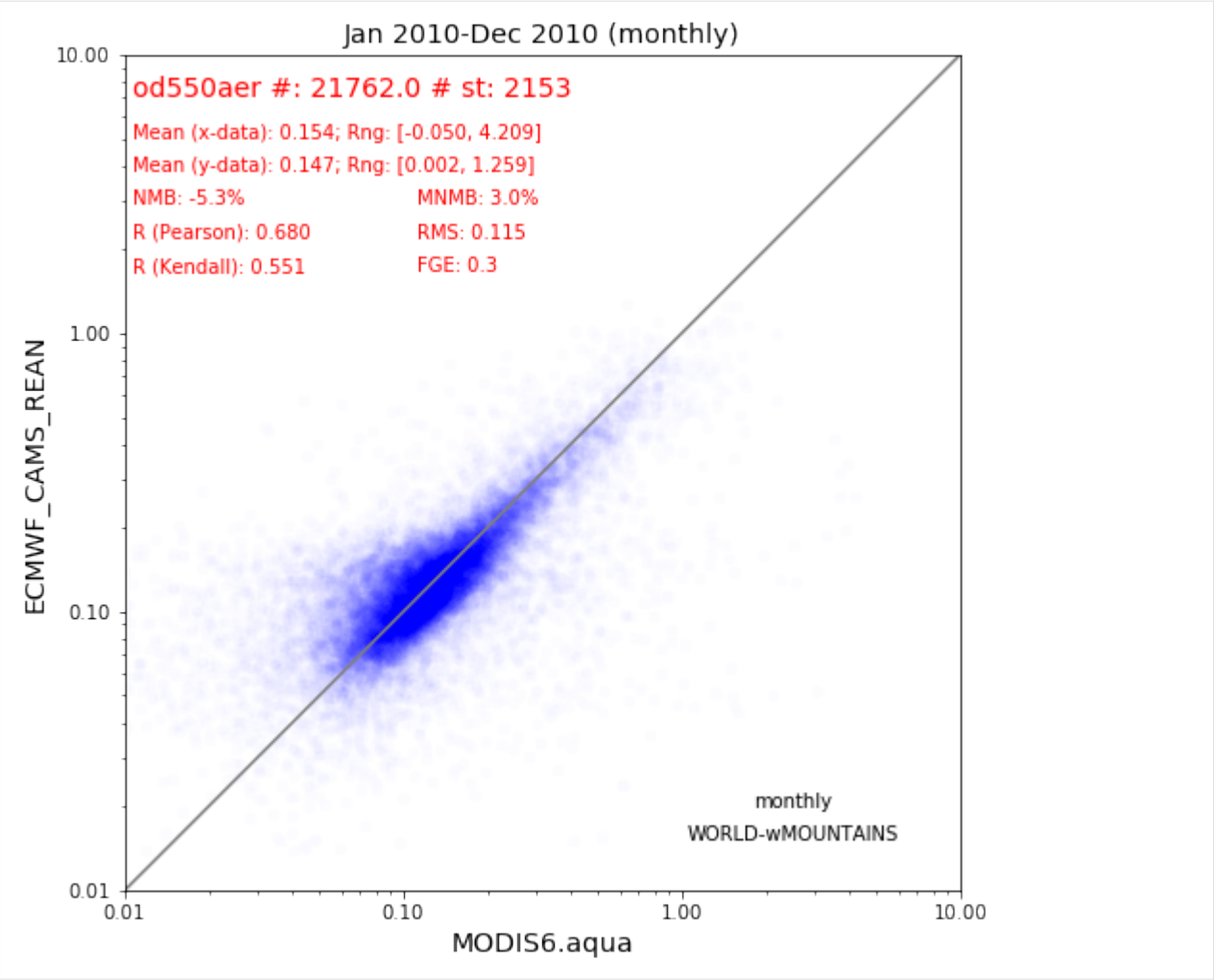
Now the satellite data comes gridded, like the model data. Thus, we use the gridded / gridded colocation routine rather than the gridded / ungridded that we used above when using AERONET station data.

No (daily) time colocation

[17]:
```
coldata_modis =  pya.colocation.colocate_gridded_gridded(model_data,
                                                         modis_aods,
                                                         ts_type='monthly',
                                                         regrid_res_deg=5,
                                                         remove_outliers=True,
                                                         colocate_time=False)
```

```
Interpolating data of shape (365, 180, 360). This may take a while.
Successfully interpolated cube
Setting od550aer outlier lower lim: -1.00
Setting od550aer outlier upper lim: 10.00
```

[18]:
```
coldata_modis.plot_scatter(marker='o', mec='none', color='b', alpha=0.01);
```

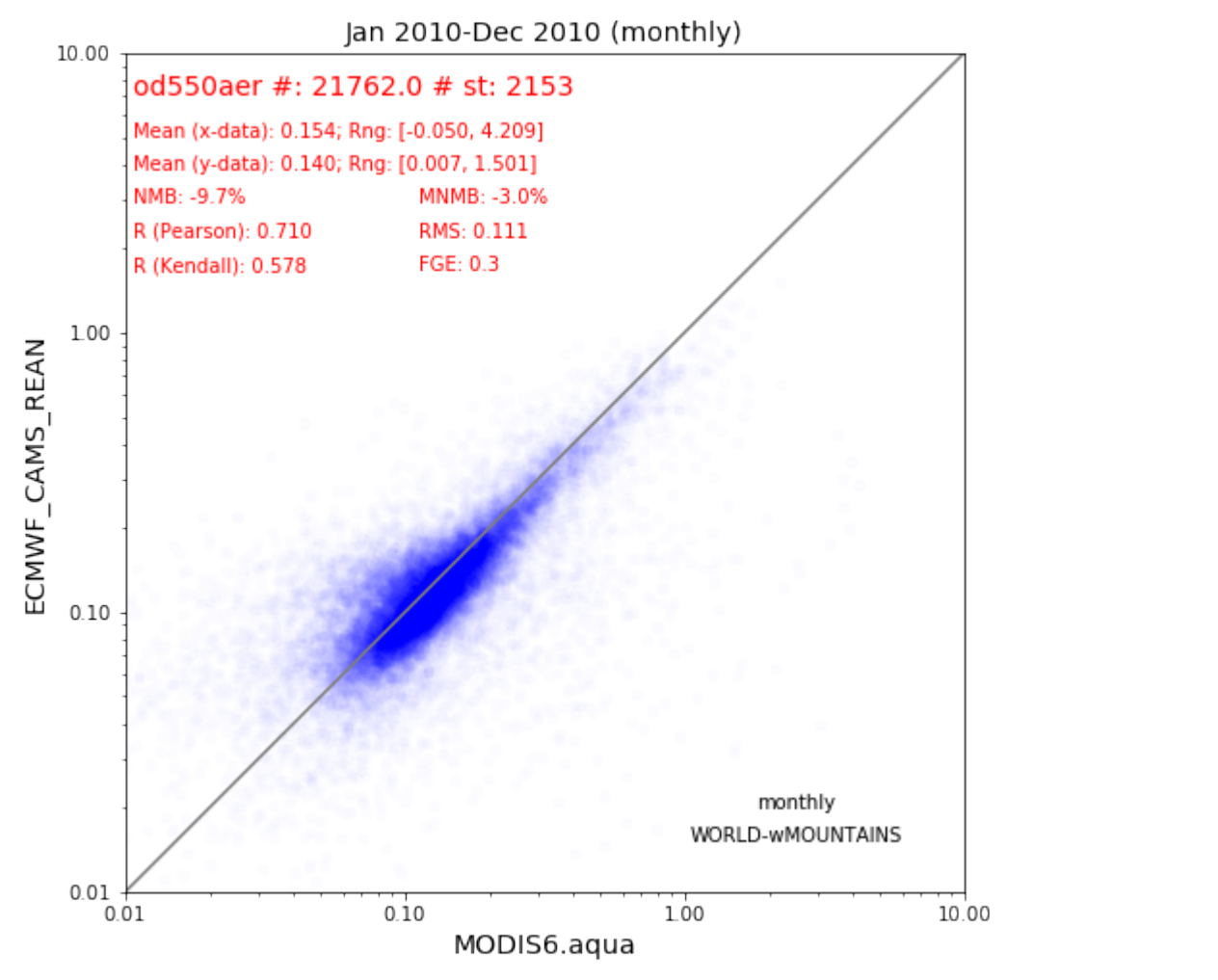


With (daily) time colocation

[19]:
```
coldata_modis_alt =  pya.colocation.colocate_gridded_gridded(model_data,
                                                             modis_aods,
                                                             ts_type='monthly',
                                                             regrid_res_deg=5,
                                                             remove_outliers=True,
                                                             colocate_time=True)
```

```
Interpolating data of shape (365, 180, 360). This may take a while.
Successfully interpolated cube
Setting od550aer outlier lower lim: -1.00
Setting od550aer outlier upper lim: 10.00
```

[20]:
```
coldata_modis_alt.plot_scatter(marker='o', mec='none', color='b', alpha=0.01);
```



3.2.7 Merging of several StationData objects into one

This notebook illustrates to can merge several instances StationData objects into one objects. This merging only works for data from the same station and a typical case is if the data source files for one station are separated into many single files containing only parts of the data (e.g. if the files contain only one year of data).

In the following, the example of the EBAS database is used for illustration. In particular, we will focus on the retrieval of the aerosol light scattering coefficients at 550 nm (**scatc550aer** in AEROCOM naming convention) for the station **Jungfraujoch**, located in Germany.

[1]:
```
import pyaerocom as pya
```

```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
```

```
Expired time: 0.016 s
```

Get list of all files containing scattering data for EBAS station Jungfraujoch

[2]:
```
reader = pya.io.ReadEbas()
data = reader.read(vars_to_retrieve='scatc550aer',
                           datalevel=2, station_names='Jungfraujoch')
print(data)
```

```
Retrieving EBAS files for variables
['scatc550aer']
Reading files 1-3 of 28 (ReadEbas) | 18:10:51 (delta = 0 s')
Reading files 3-5 of 28 (ReadEbas) | 18:10:52 (delta = 0 s')
Reading files 5-7 of 28 (ReadEbas) | 18:10:52 (delta = 0 s')
Reading files 7-9 of 28 (ReadEbas) | 18:10:53 (delta = 0 s')
Reading files 9-11 of 28 (ReadEbas) | 18:10:53 (delta = 0 s')
Reading files 11-13 of 28 (ReadEbas) | 18:10:54 (delta = 0 s')
Reading files 13-15 of 28 (ReadEbas) | 18:10:55 (delta = 0 s')
Reading files 15-17 of 28 (ReadEbas) | 18:10:55 (delta = 0 s')
Reading files 17-19 of 28 (ReadEbas) | 18:10:56 (delta = 0 s')
Reading files 19-21 of 28 (ReadEbas) | 18:10:57 (delta = 0 s')
Reading files 21-23 of 28 (ReadEbas) | 18:10:58 (delta = 0 s')
Reading files 23-25 of 28 (ReadEbas) | 18:10:59 (delta = 1 s')
Reading files 25-27 of 28 (ReadEbas) | 18:11:00 (delta = 1 s')
Reading files 27-29 of 28 (ReadEbas) | 18:11:01 (delta = 1 s')

Pyaerocom UngriddedData
-----------------------
Contains networks: ['EBASMC']
Contains variables: ['scatc550aer']
Contains instruments: ['IN3563', 'TSI_3563_JFJ_dry', 'Ecotech_Aurora3000_JFJ_dry']
Total no. of meta-blocks: 28
Filters that were applied:
 Filter time log: 20191003181051
        prefer_statistics: ['arithmetic mean', 'median']
        wavelength_tol_nm: 50
        datalevel: 2
        station_names: Jungfraujoch
 Filter time log: 20191003181102
        Removed 0 metadata blocks that have no data assigned
```

As you can see, the data has successfully been imported into an instance of the `UngriddedData` class. This class is organised *by file*, that is, for each of the 26 files that were imported, there is one metadata dictionary assigned. Let's look at the metadata from the first file:

[3]:
```
data.metadata[0]
```

[3]:
```
OrderedDict([('latitude', 46.5475),
             ('longitude', 7.985),
             ('altitude', 3578.0),
             ('filename',
              'CH0001G.19950101000000.20181031145000.nephelometer..aerosol.1y.1h.CH02L_
↪IN3563.CH02L_backscat_coef.lev2.nas'),
             ('station_id', 'CH0001G'),
             ('station_name', 'Jungfraujoch'),
             ('instrument_name', 'IN3563'),
             ('PI', 'Baltensperger, Urs; Weingartner, Ernest'),
             ('country', None),
             ('ts_type', 'hourly'),
             ('data_id', 'EBASMC'),
             ('dataset_name', None),
             ('data_product', None),
             ('data_version', None),
             ('data_level', 2),
             ('revision_date', numpy.datetime64('2018-10-31T00:00:00')),
             ('website', None),
             ('ts_type_src', None),
             ('stat_merge_pref_attr', None),
             ('data_revision', '20190701'),
             ('var_info',
              OrderedDict([('scatc550aer',
                            OrderedDict([('name',
                                          'aerosol_light_scattering_coefficient'),
                                         ('units', '1/Mm'),
                                         ('wavelength', '550.0 nm'),
                                         ('method_ref', 'CH02L_scat_coef'),
                                         ('matrix', 'aerosol'),
                                         ('statistics',
                                          'arithmetic mean')]))])),
             ('variables', ['scatc550aer'])])
```

And the last one:

[4]:
```
data.metadata[25]
```

[4]:
```
OrderedDict([('latitude', 46.5475),
             ('longitude', 7.985),
             ('altitude', 3580.0),
             ('filename',
              'CH0001G.20170101000000.20190524143212.nephelometer..aerosol.1y.1h.CH02L_
↪Ecotech_Aurora3000_JFJ_dry.CH02L_Neph_Aurora3000.lev2.nas'),
             ('station_id', 'CH0001G'),
             ('station_name', 'Jungfraujoch'),
             ('instrument_name', 'Ecotech_Aurora3000_JFJ_dry'),
             ('PI', 'Bukowiecki, Nicolas; Baltensperger, Urs'),
             ('country', None),
             ('ts_type', 'hourly'),
             ('data_id', 'EBASMC'),
             ('dataset_name', None),
```

(continues on next page)

(continued from previous page)

```
             ('data_product', None),
             ('data_version', None),
             ('data_level', 2),
             ('revision_date', numpy.datetime64('2019-05-24T00:00:00')),
             ('website', None),
             ('ts_type_src', None),
             ('stat_merge_pref_attr', None),
             ('data_revision', '20190701'),
             ('var_info',
              OrderedDict([('scatc550aer',
                            OrderedDict([('name',
                                          'aerosol_light_scattering_coefficient'),
                                         ('units', '1/Mm'),
                                         ('wavelength', '525.0 nm'),
                                         ('statistics', 'arithmetic mean'),
                                         ('matrix', 'aerosol')]))])),
             ('variables', ['scatc550aer'])])
```

As you can see, both files contain scattering data but do not share all the same metadata attributes (e.g. `instrument_name` is different, which might be due to technological updates over time).

Let's have a look at the respective time-series for both stations. First, convert into instance of `StationData` class and then plot.

[5]:
```
first_file = data.to_station_data(0, vars_to_convert='scatc550aer')
print(first_file)
```

```
Pyaerocom StationData
---------------------
var_info (BrowseDict):
  scatc550aer (OrderedDict):
    name: aerosol_light_scattering_coefficient
    units: 1/Mm
    wavelength: 550.0 nm
    method_ref: CH02L_scat_coef
    matrix: aerosol
    statistics: arithmetic mean
    overlap: False
station_coords (dict):
  latitude: 46.5475
  longitude: 7.985
  altitude: 3578.0
data_err (BrowseDict): <empty_dict>
overlap (BrowseDict): <empty_dict>
data_flagged (BrowseDict):
  scatc550aer (ndarray, 8760 items): [1.00, 1.00, ..., 0.0, 0.0]
filename: CH0001G.19950101000000.20181031145000.nephelometer..aerosol.1y.1h.CH02L_IN3563.
↪CH02L_backscat_coef.lev2.nas
station_id: CH0001G
station_name: Jungfraujoch
instrument_name: IN3563
PI: Baltensperger, Urs; Weingartner, Ernest
```

(continues on next page)

(continued from previous page)

```
country: None
ts_type: hourly
latitude: 46.5475
longitude: 7.985
altitude: 3578.0
data_id: EBASMC
dataset_name: None
data_product: None
data_version: None
data_level: 2
revision_date: 2018-10-31T00:00:00
website: None
ts_type_src: hourly
stat_merge_pref_attr: None
data_revision: 20190701

Data arrays
...
dtime (ndarray, 8760 items): [1995-01-01T00:30:00.000000000, 1995-01-01T01:29:59.
↪000000000, ..., 1995-12-31T22:29:59.000000000, 1995-12-31T23:29:59.000000000]
Pandas Series
...
scatc550aer (Series, 8760 items)
```

[6]:
```
last_file = data.to_station_data(25)
print(last_file)
```

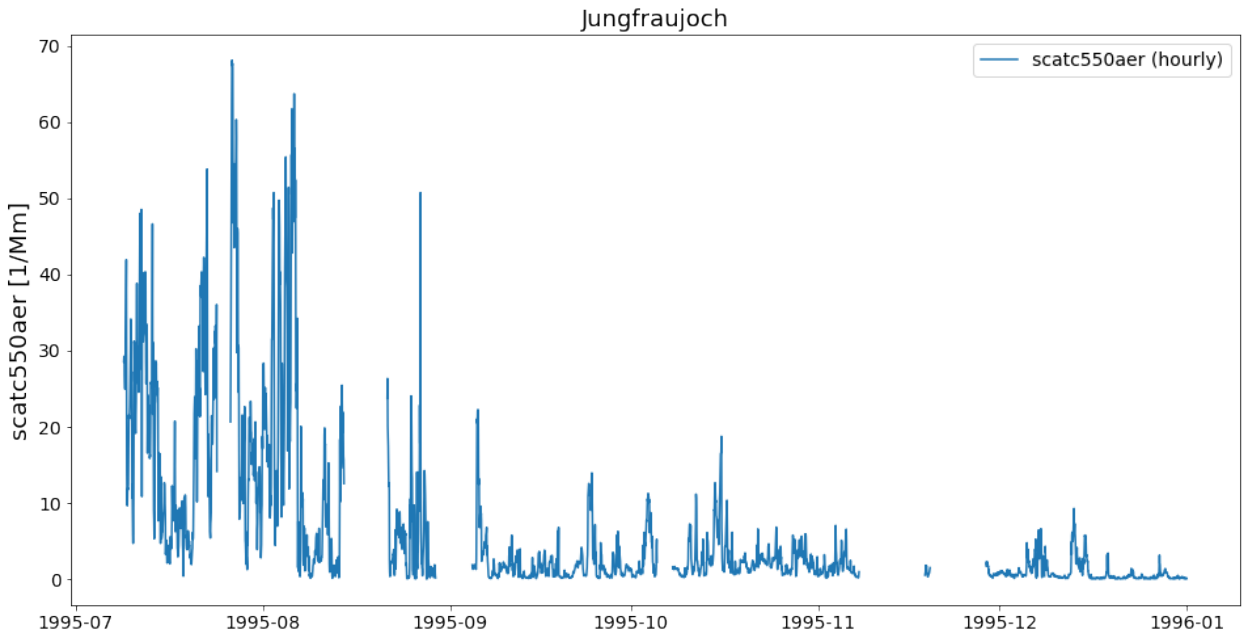
```
Pyaerocom StationData
---------------------
var_info (BrowseDict):
  scatc550aer (OrderedDict):
    name: aerosol_light_scattering_coefficient
    units: 1/Mm
    wavelength: 525.0 nm
    statistics: arithmetic mean
    matrix: aerosol
    overlap: False
station_coords (dict):
  latitude: 46.5475
  longitude: 7.985
  altitude: 3580.0
data_err (BrowseDict): <empty_dict>
overlap (BrowseDict): <empty_dict>
data_flagged (BrowseDict):
  scatc550aer (ndarray, 8760 items): [1.00, 1.00, ..., 1.00, 0.0]
filename: CH0001G.20170101000000.20190524143212.nephelometer..aerosol.1y.1h.CH02L_
↪Ecotech_Aurora3000_JFJ_dry.CH02L_Neph_Aurora3000.lev2.nas
station_id: CH0001G
station_name: Jungfraujoch
instrument_name: Ecotech_Aurora3000_JFJ_dry
PI: Bukowiecki, Nicolas; Baltensperger, Urs
```

(continues on next page)

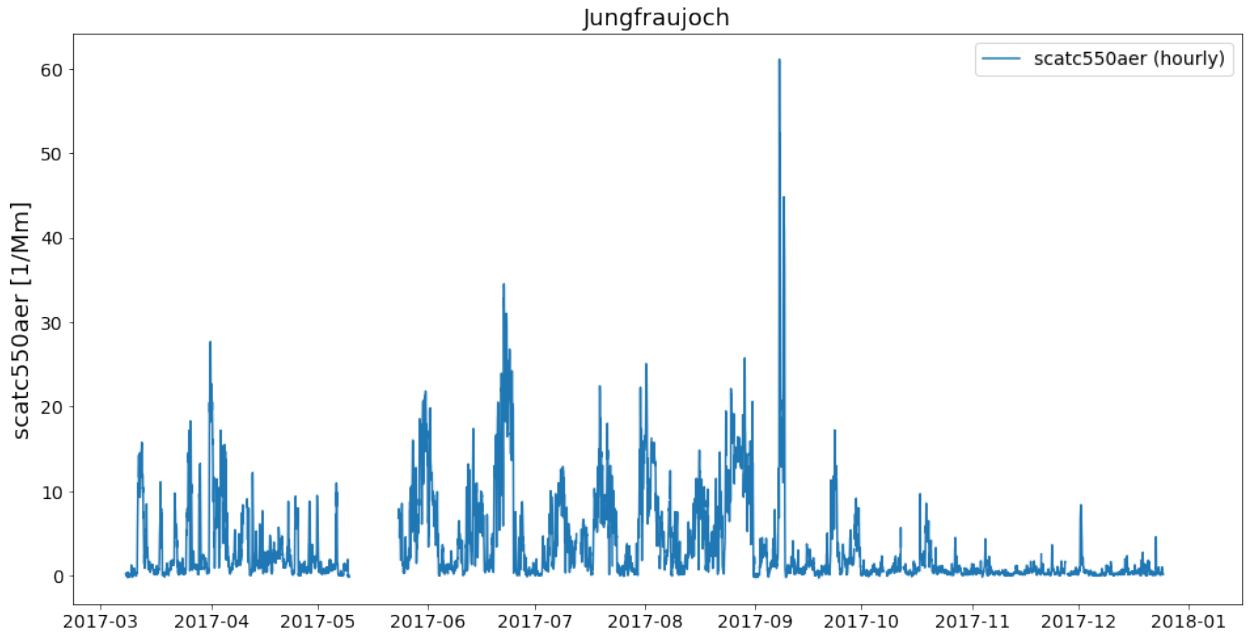
(continued from previous page)

```
country: None
ts_type: hourly
latitude: 46.5475
longitude: 7.985
altitude: 3580.0
data_id: EBASMC
dataset_name: None
data_product: None
data_version: None
data_level: 2
revision_date: 2019-05-24T00:00:00
website: None
ts_type_src: hourly
stat_merge_pref_attr: None
data_revision: 20190701

Data arrays
...
dtime (ndarray, 8760 items): [2017-01-01T00:30:00.000000000, 2017-01-01T01:29:59.
↪000000000, ..., 2017-12-31T22:29:59.000000000, 2017-12-31T23:29:59.000000000]
Pandas Series
...
scatc550aer (Series, 8760 items)
```

[7]:
```
first_file.plot_timeseries('scatc550aer');
```



[8]:
```
last_file.plot_timeseries('scatc550aer');
```



As you can see, the files contain data from different years. Now, how can we get these objects into one object that contains the timeseries of both files from this station?

This is actually very easy:

[9]:
```
merged = first_file.merge_other(last_file, 'scatc550aer')
print(merged)
```

```
Pyaerocom StationData
---------------------
var_info (BrowseDict):
  scatc550aer (OrderedDict):
    name: aerosol_light_scattering_coefficient
    units: 1/Mm
    wavelength: 550.0 nm;525.0 nm
    method_ref: CH02L_scat_coef
    matrix: aerosol
    statistics: arithmetic mean
    overlap: False
    ts_type: hourly
    apply_constraints: False
    min_num_obs: None
station_coords (dict):
  latitude: 46.5475
  longitude: 7.985
  altitude: 3578.0
data_err (BrowseDict): <empty_dict>
overlap (BrowseDict): <empty_dict>
data_flagged (BrowseDict):
  scatc550aer (ndarray, 8760 items): [1.00, 1.00, ..., 0.0, 0.0]
filename: CH0001G.19950101000000.20181031145000.nephelometer..aerosol.1y.1h.CH02L_IN3563.
↪CH02L_backscat_coef.lev2.nas; CH0001G.20170101000000.20190524143212.nephelometer..
```

(continues on next page)

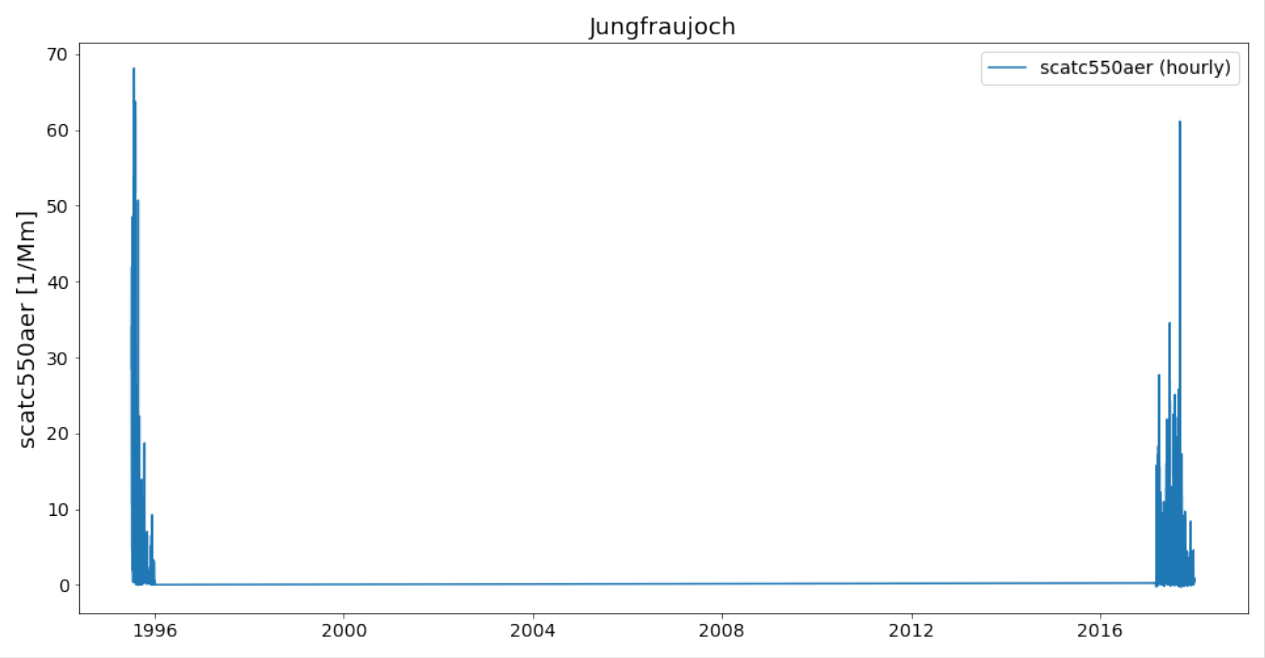
(continued from previous page)

```
↪aerosol.1y.1h.CH02L_Ecotech_Aurora3000_JFJ_dry.CH02L_Neph_Aurora3000.lev2.nas
station_id: CH0001G
station_name: Jungfraujoch
instrument_name: IN3563; Ecotech_Aurora3000_JFJ_dry
PI: Baltensperger, Urs; Weingartner, Ernest; Bukowiecki, Nicolas
country: None
ts_type: hourly
latitude: 46.5475
longitude: 7.985
altitude: 3578.0
data_id: EBASMC
dataset_name: None
data_product: None
data_version: None
data_level: 2
revision_date (list, 2 items): [numpy.datetime64('2018-10-31T00:00:00'), numpy.
↪datetime64('2019-05-24T00:00:00')]
website: None
ts_type_src: hourly
stat_merge_pref_attr: None
data_revision: 20190701

Data arrays
...
dtime (ndarray, 9794 items): [1995-07-08T23:00:00.000000000, 1995-07-09T00:00:00.
↪000000000, ..., 2017-12-24T19:00:00.000000000, 2017-12-31T23:00:00.000000000]
Pandas Series
...
scatc550aer (Series, 9794 items)
```

As you can see in the output, the merging comprises not only the data arrays but also registers any differences in the assoicated metdadata (cf. e.g., sampling wavelength 550 nm vs. 525 nm, instrument name, PI)

Now, have a look at the merged timeseries data.

[10]:
```
merged.plot_timeseries('scatc550aer');
```



Looks okay. Let's merge all 26 files and see if we get a nice long time series.

Retrieve list of `StationData` objects:

[11]:
```
stats = data.to_station_data('Jungfraujoch', 'scatc550aer', merge_if_multi=False)
print('Number of StationData objects retrieved: {}'.format(len(stats)))
```

```
Number of StationData objects retrieved: 28
```

Now merge them into one long time series:

[12]:
```
merged = pya.helpers.merge_station_data(stats, var_name='scatc550aer')
print(merged)
```

```
Pyaerocom StationData
---------------------
var_info (BrowseDict):
  scatc550aer (OrderedDict):
    name: aerosol_light_scattering_coefficient
    units: 1/Mm
    wavelength: 550.0 nm;525.0 nm
    statistics: arithmetic mean
    matrix: aerosol
    overlap: False
    ts_type: hourly
    apply_constraints: False
    min_num_obs: None
    method_ref: CH02L_scat_coef
station_coords (dict):
  latitude: 46.5475
  longitude: 7.985
  altitude: 3580.0
```

(continues on next page)

(continued from previous page)

```
data_err (BrowseDict): <empty_dict>
overlap (BrowseDict):
  scatc550aer: 2015-01-01 01:00:00    0.255198
2015-01-01 02:00:00   -0.284809
2015-01-01 03:00:00    0.123830
2015-01-01 04:00:00    0.127599
2015-01-01 05:00:00    0.052224
                          ...
2018-12-31 19:00:00   -0.030204
2018-12-31 20:00:00    0.625988
2018-12-31 21:00:00    0.178854
2018-12-31 22:00:00    0.175893
2018-12-31 23:00:00    0.329280
Length: 25787, dtype: float64
data_flagged (BrowseDict):
  scatc550aer (ndarray, 8760 items): [0.0, 0.0, ..., 0.0, 0.0]
filename: CH0001G.20010101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_
↪3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.20100101000000.20190524142901.
↪nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.
↪20060101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_
↪Neph_3563.lev2.nas; CH0001G.20180101000000.20190520125514.nephelometer..aerosol.1y.1h.
↪CH02L_Ecotech_Aurora3000_JFJ_dry.CH02L_Neph_Aurora3000.lev2.nas; CH0001G.
↪20040101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_
↪Neph_3563.lev2.nas; CH0001G.20180101000000.20190520124723.nephelometer..aerosol.1y.1h.
↪CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.20080101000000.20190524142901.
↪nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.
↪20050101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_
↪Neph_3563.lev2.nas; CH0001G.20070101000000.20190524142901.nephelometer..aerosol.1y.1h.
↪CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.20110101000000.20190524142901.
↪nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.
↪20030101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_
↪Neph_3563.lev2.nas; CH0001G.20150101000000.20190524143212.nephelometer..aerosol.1y.1h.
↪CH02L_Ecotech_Aurora3000_JFJ_dry.CH02L_Neph_Aurora3000.lev2.nas; CH0001G.
↪20170101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_
↪Neph_3563.lev2.nas; CH0001G.20160101000000.20190524142901.nephelometer..aerosol.1y.1h.
↪CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.20000101000000.20181031145000.
↪nephelometer..aerosol.1y.1h.CH02L_IN3563.CH02L_backscat_coef.lev2.nas; CH0001G.
↪20140101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_
↪Neph_3563.lev2.nas; CH0001G.20020101000000.20190524142901.nephelometer..aerosol.1y.1h.
↪CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.20090101000000.20190524142901.
↪nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.
↪20120101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_dry.CH02L_
↪Neph_3563.lev2.nas; CH0001G.20130101000000.20190524142901.nephelometer..aerosol.1y.1h.
↪CH02L_TSI_3563_JFJ_dry.CH02L_Neph_3563.lev2.nas; CH0001G.19980101000000.20181031145000.
↪nephelometer..aerosol.1y.1h.CH02L_IN3563.CH02L_backscat_coef.lev2.nas; CH0001G.
↪19960101000000.20181031145000.nephelometer..aerosol.1y.1h.CH02L_IN3563.CH02L_backscat_
↪coef.lev2.nas; CH0001G.20170101000000.20190524143212.nephelometer..aerosol.1y.1h.CH02L_
↪Ecotech_Aurora3000_JFJ_dry.CH02L_Neph_Aurora3000.lev2.nas; CH0001G.19990101000000.
↪20181031145000.nephelometer..aerosol.1y.1h.CH02L_IN3563.CH02L_backscat_coef.lev2.nas;␣
↪CH0001G.20150101000000.20190524142901.nephelometer..aerosol.1y.1h.CH02L_TSI_3563_JFJ_
↪dry.CH02L_Neph_3563.lev2.nas; CH0001G.20160101000000.20190524143212.nephelometer..
↪aerosol.1y.1h.CH02L_Ecotech_Aurora3000_JFJ_dry.CH02L_Neph_Aurora3000.lev2.nas; CH0001G.
```

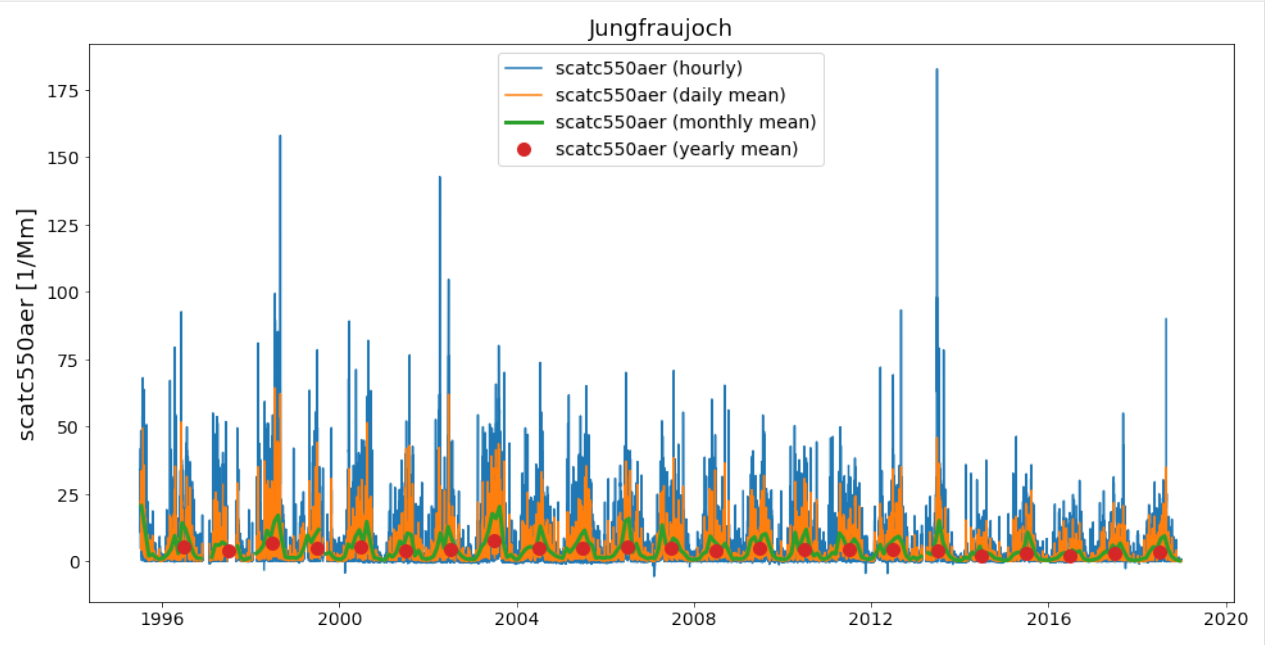
(continues on next page)

(continued from previous page)

```
↪19970101000000.20181031145000.nephelometer..aerosol.1y.1h.CH02L_IN3563.CH02L_backscat_
↪coef.lev2.nas; CH0001G.19950101000000.20181031145000.nephelometer..aerosol.1y.1h.CH02L_
↪IN3563.CH02L_backscat_coef.lev2.nas
station_id: CH0001G
station_name: Jungfraujoch
instrument_name: TSI_3563_JFJ_dry; Ecotech_Aurora3000_JFJ_dry; IN3563
PI: Baltensperger, Urs; Weingartner, Ernest; Bukowiecki, Nicolas
country: None
ts_type: hourly
latitude: 46.5475
longitude: 7.985
altitude: 3580.0
data_id: EBASMC
dataset_name: None
data_product: None
data_version: None
data_level: 2
revision_date (list, 3 items): [numpy.datetime64('2019-05-24T00:00:00'), numpy.
↪datetime64('2019-05-20T00:00:00'), numpy.datetime64('2018-10-31T00:00:00')]
website: None
ts_type_src: hourly
stat_merge_pref_attr: None
data_revision: 20190701

Data arrays
...
dtime (ndarray, 205849 items): [1995-07-08T23:00:00.000000000, 1995-07-09T00:00:00.
↪000000000, ..., 2018-12-31T22:00:00.000000000, 2018-12-31T23:00:00.000000000]
Pandas Series
...
scatc550aer (Series, 205849 items)
```

And plot…

[13]:
```
ax = merged.plot_timeseries('scatc550aer')
merged.plot_timeseries('scatc550aer', freq='daily', ax=ax)
merged.plot_timeseries('scatc550aer', freq='monthly', lw=3, ax=ax)
merged.plot_timeseries('scatc550aer', freq='yearly', ls='none', marker='o', ms=10,␣
↪ax=ax);
```



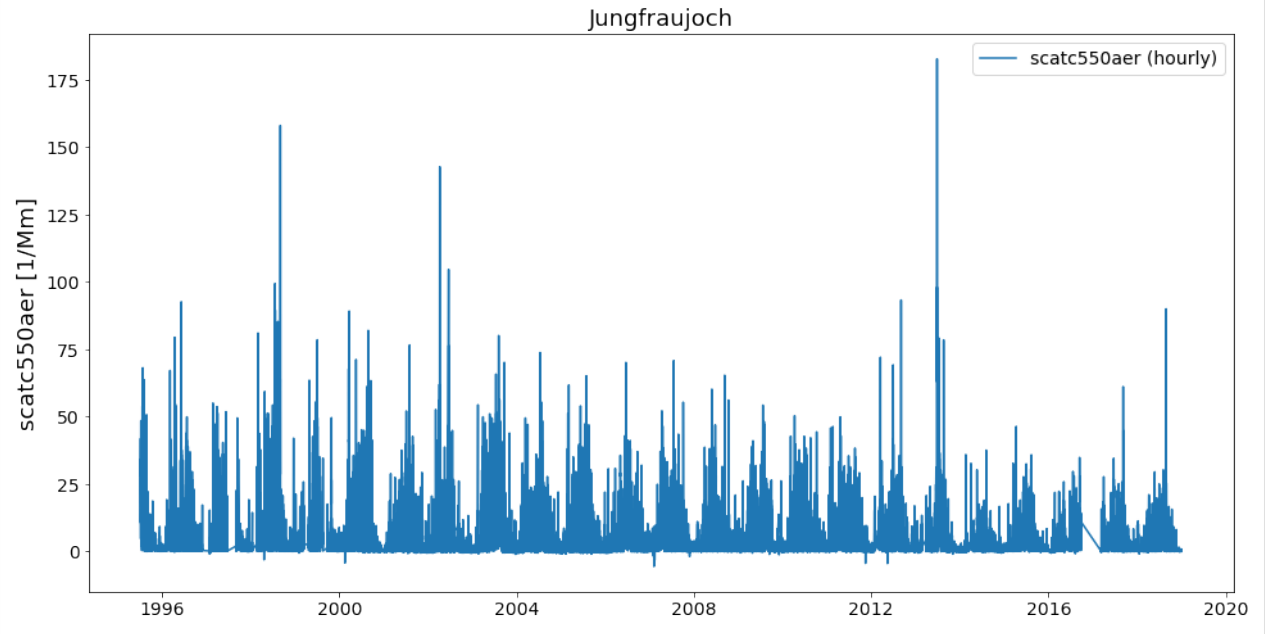
Comment for convenience....

Actually, in the default setup you do not really need to think about all this. As you might have recognised, when creating the list of `StationData` objects from the `UngriddedData` object (using method `to_station_data`) we parsed the argument `merge_if_multi=False`.

The default here is `True`, so you can just go ahead and do:

[14]:
```
data.to_station_data('Jungfraujoch', 'scatc550aer').plot_timeseries('scatc550aer')
```

[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9d348975f8>
```



What's happening here is, that `to_station_data` internally creates a list of `StationData` objects and uses the above illustrated method `merge_station_data` at the end if the input argument `merge_if_multi=True`.

What about overlapping data ??

In some situations, there may be overlapping conflicts when merging multiple time series into one long time-series. In the following, we illustrate how these overlaps are handled if they occur.

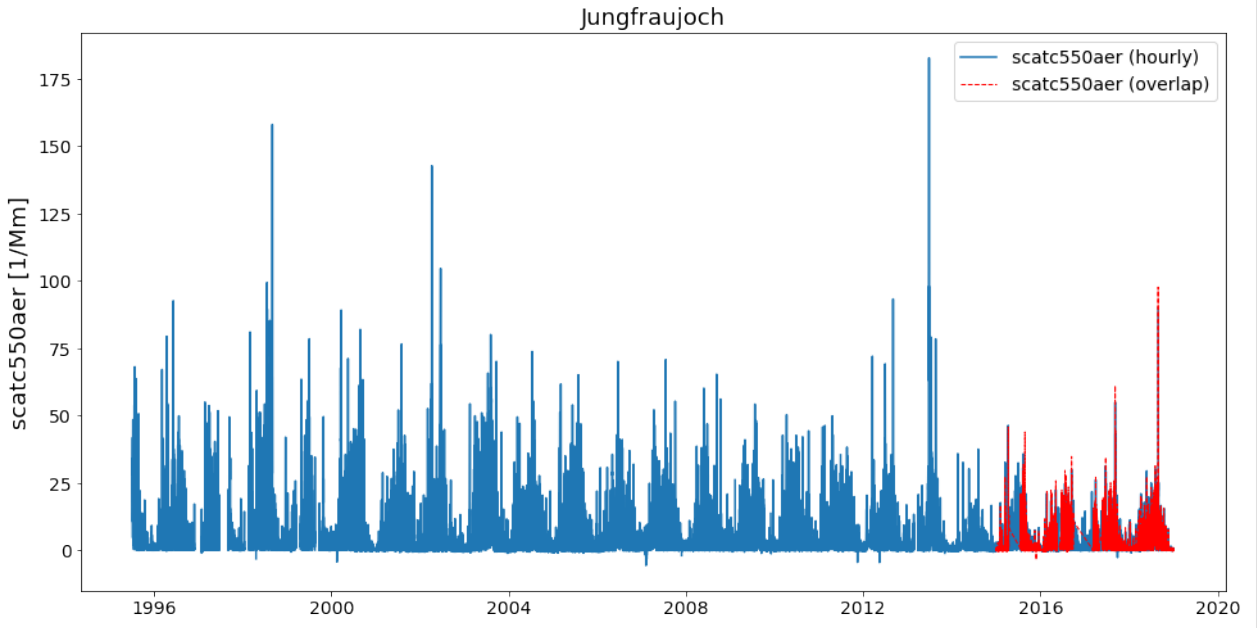
The method `merge_station_data` that is illustrated above has some features to handle overlapping data and in any case, all overlaps that were detected are stored in the `overlap` attribute of the merged `StationData` object. Let's check first if there are any overlaps in the Jungfraujoch data:

[15]:
```
merged.overlap
```

[15]:
```
BrowseDict([('scatc550aer', 2015-01-01 01:00:00    0.255198
             2015-01-01 02:00:00   -0.284809
             2015-01-01 03:00:00    0.123830
             2015-01-01 04:00:00    0.127599
             2015-01-01 05:00:00    0.052224
                                       ...
             2018-12-31 19:00:00   -0.030204
             2018-12-31 20:00:00    0.625988
             2018-12-31 21:00:00    0.178854
             2018-12-31 22:00:00    0.175893
             2018-12-31 23:00:00    0.329280
             Length: 25787, dtype: float64)])
```

Apparently, there is. You can check out these data (in comparison with the retrieved time series) as follows:

[16]:
```
merged.plot_timeseries('scatc550aer', add_overlaps=True);
```



How to prioritise certain stations from others, when deciding what goes into overlap and what into the final timeseries?

The method `merge_station_data` provides a bunch of options to handle that. Things do not be written twice so please read the docstring of the method:

[17]:
```
help(pya.helpers.merge_station_data)
```

```
Help on function merge_station_data in module pyaerocom.helpers:

merge_station_data(stats, var_name, pref_attr=None, sort_by_largest=True, fill_missing_
↪nan=True, **add_meta_keys)
    Merge multiple StationData objects (from one station) into one instance

    Note
    ----
    - all input :class:`StationData` objects need to have same attributes
↪``station_name``, ``latitude``, ``longitude`` and ``altitude``

    Parameters
    ----------
    stats : list
        list containing :class:`StationData` objects (note: all of these
        objects must contain variable data for the specified input variable)
    var_name : str
        data variable name that is to be merged
    pref_attr
        optional argument that may be used to specify a metadata attribute
        that is available in all input :class:`StationData` objects and that
        is used to order the input stations by relevance. The associated values
        of this attribute need to be sortable (e.g. revision_date). This is
        only relevant in case overlaps occur. If unspecified the relevance of
        the stations is sorted based on the length of the associated data
        arrays.
    sort_by_largest : bool
        if True, the result from the sorting is inverted. E.g. if
        ``pref_attr`` is unspecified, then the stations will be sorted based on
        the length of the data vectors, starting with the shortest, ending with
        the longest. This sorting result will then be inverted, if
        ``sort_by_largest=True``, so that the longest time series get's highest
        importance. If, e.g. ``pref_attr='revision_date'``, then the stations
        are sorted by the associated revision date value, starting with the
        earliest, ending with the latest (which will also be inverted if
        this argument is set to True)
    fill_missing_nan : bool
        if True, the resulting time series is filled with NaNs. NOTE: this
        requires that information about the temporal resolution (ts_type) of
        the data is available in each of the StationData objects.
```

In particular, `pref_attr` and `sort_by_largest` are relevant here.

NOTE: if `pref_attr` is unspecified, then the stations are sorted based on the number of valid measurement points for the input variable. This was done in the merged time series that we retrieved above.

Now, in the following, let's not use the number of available data points (to sort the stations by relevance) but prefer

stations that have a more recent data revision date.

[18]:
```
try:
    merged_pref_awesomeness = pya.helpers.merge_station_data(stats, 'scatc550aer', pref_
↪attr='awesomeness')
except pya.exceptions.MetaDataError as e:
    print('Failed merging, error: {}'.format(repr(e)))
```

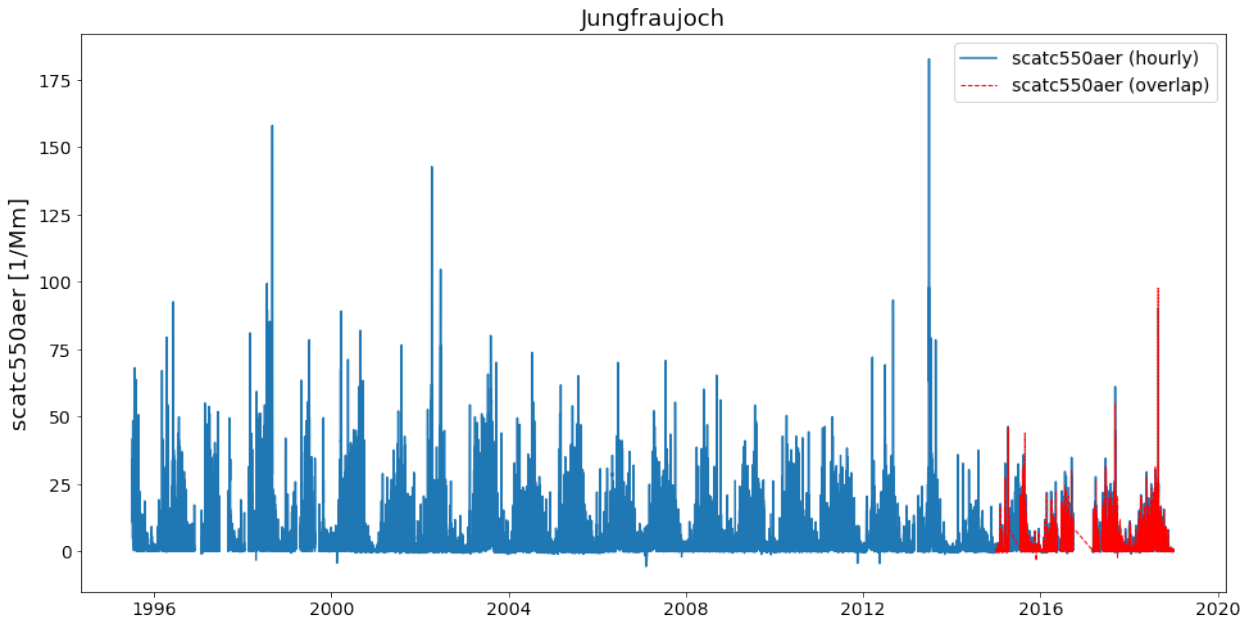
```
Failed merging, error: MetaDataError('Cannot sort station relevance by attribute␣
↪awesomeness. At least one of the input stations does not contain this attribute')
```

Unfortunately, the `StationData` objects do not contain an attribute `awesomeness` by which we could sort. Let's go with `revision_date` instead:

[19]:
```
# recompute stations, since we overwrote one above
stats = data.to_station_data('Jungfraujoch', 'scatc550aer', merge_if_multi=False)
merged_pref_recent_revision = pya.helpers.merge_station_data(stats, 'scatc550aer', pref_
↪attr='revision_date')
```

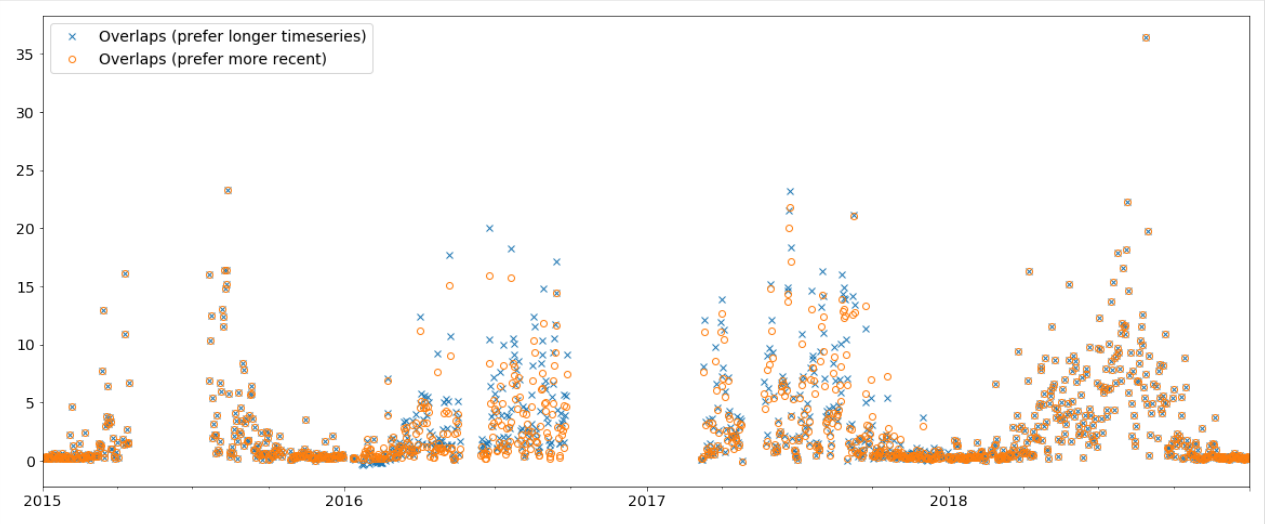
[20]:
```
merged_pref_recent_revision.plot_timeseries('scatc550aer', add_overlaps=True)
```

[20]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9d341f8d68>
```



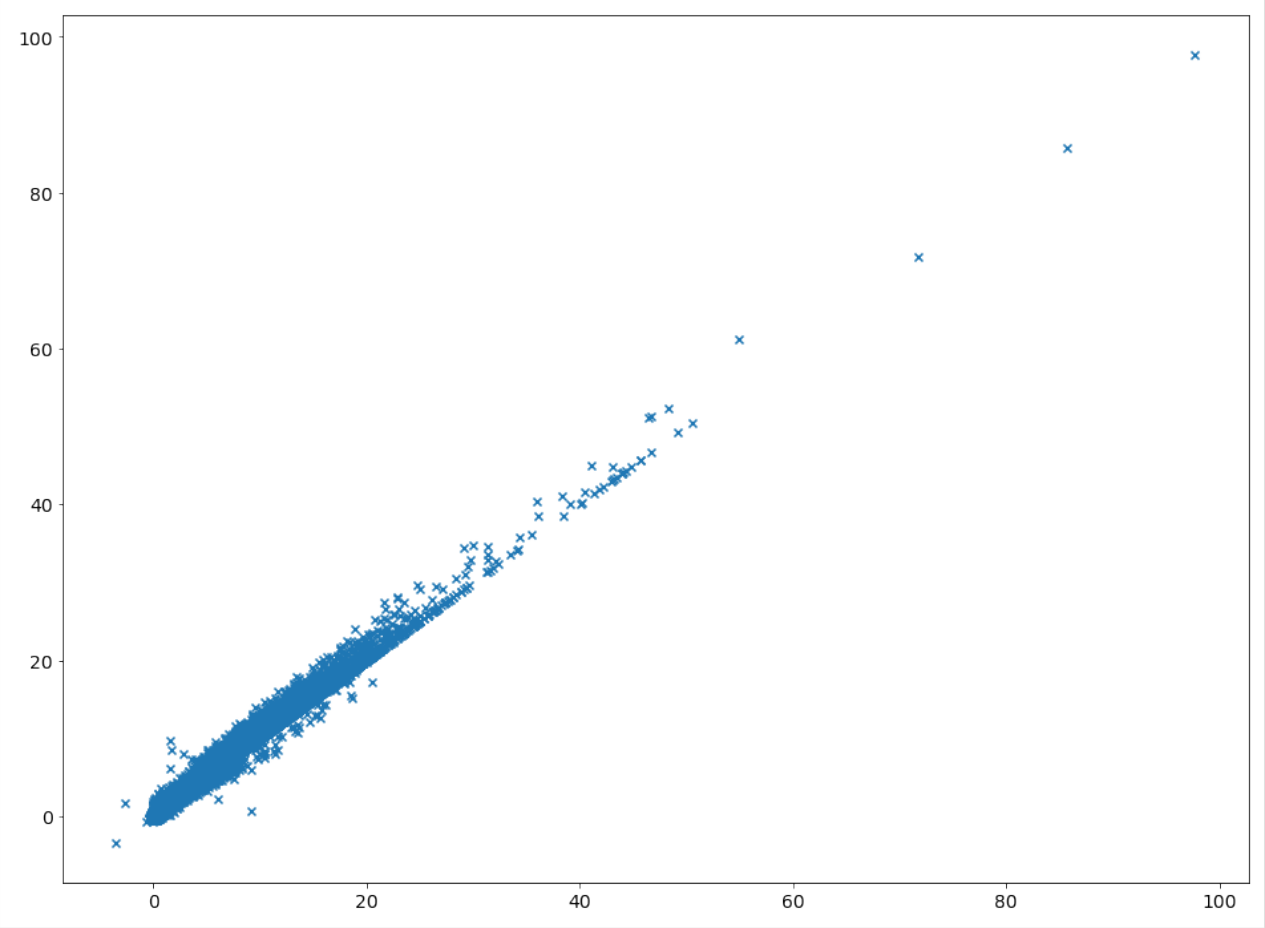
Let's see if there is any difference to the previous method (resample to daily resolution):

[21]:
```
ax = merged.overlap['scatc550aer'].resample('D').mean().plot(style='x', label='Overlaps␣
↪(prefer longer timeseries)',
                                                             figsize=(20, 8))

merged_pref_recent_revision.overlap['scatc550aer'].resample('D').mean().plot(style='o',
                                                                             label=
↪'Overlaps (prefer more recent)',
                                                                             ax=ax,␣
↪markerfacecolor='none')
ax.legend();
```



As you can see, the merging strategy can make an impact and it is important to define a reasonable strategy. In this case, it is certainly more reliable to use the data with the more recent revision date.

[22]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(16, 12))
ax = plt.scatter(merged_pref_recent_revision.overlap['scatc550aer'],
                 merged.overlap['scatc550aer'], marker='x')
```



3.2.8 Short tutorial showing how trends can be computed within pyaerocom

Details regarding the trends computation routine can be found here (methods section):

https://aerocom-trends.met.no/

[1]:
```
import pyaerocom as pya
```
```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.016 s
```

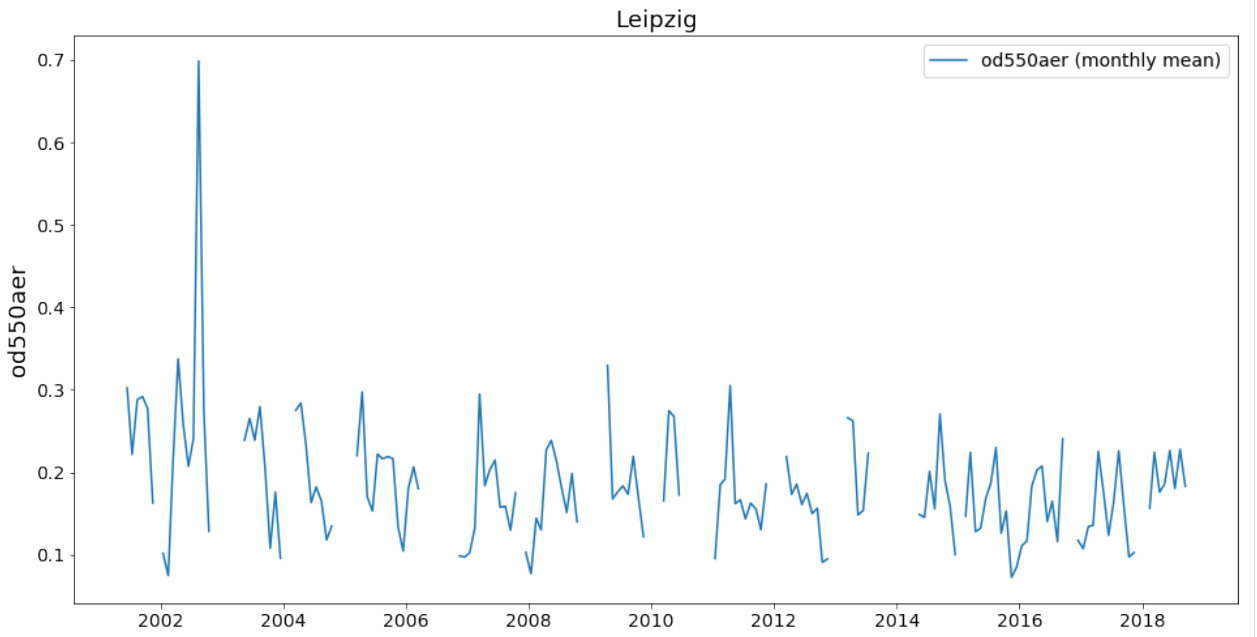
[2]:
```
obsdata = pya.io.ReadUngridded().read('AeronetSunV3Lev2.daily', 'od550aer')
obsdata
```

[2]:
```
UngriddedData <networks: ['AeronetSunV3Lev2.daily']; vars: ['od550aer']; instruments: [
↪'sun_photometer'];No. of stations: 1230
```

[3]:
```
trends_engine = pya.trends_helpers.TrendsEngine()
```

[4]:
```
leipzig = obsdata.to_station_data('Leipzig')
leipzig.plot_timeseries('od550aer', freq='monthly')
```

[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f585d4a27f0>
```



[5]:
```
trend_info = leipzig.compute_trend('od550aer')
trend_info
```

[5]:
```
{'pval': 0.07433498084899375,
 'm': -0.003086717318214402,
 'm_err': 0.001221428677173954,
 'n': 15,
 'y_mean': 0.1770474435541329,
 'y_min': 0.1434634180318197,
 'y_max': 0.24099296150814278,
 'coverage': None,
 'slp': -1.5142800744263278,
 'slp_err': 0.6112855317780068,
 'reg0': 0.20384058209203992,
 't0': None,
 'slp_2001': -1.4916916844764267,
 'slp_2001_err': 0.603290719703447,
 'reg0_2001': 0.20692729941025434,
 'yoffs': 0.3026155362749008,
 'period': '2001-2018'}
```

These results are stored in the station data object:

[6]:
```
leipzig.trends
```

[6]:
```
OrderedDict([('od550aer',
              <pyaerocom.trends_helpers.TrendsEngine at 0x7f585d901630>)])
```

The actual trends result data can be accessed via the instance of the `TrendsEngine` class that is stored in the `trends` attribute of the `StationData` object:

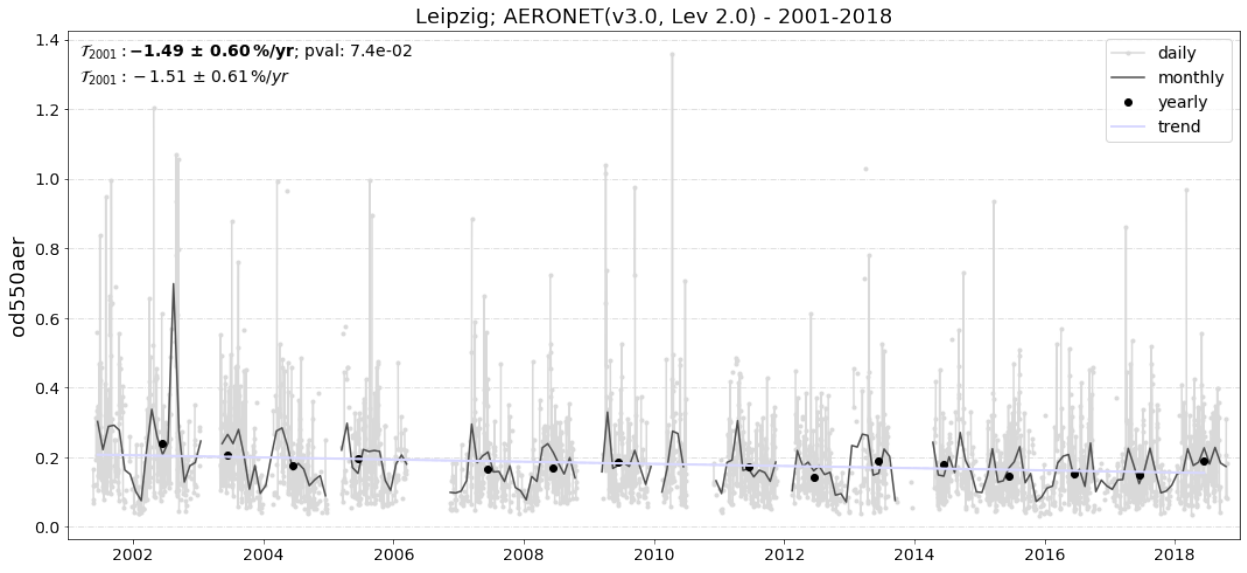
[7]:
```
leipzig.trends['od550aer'].results
```

[7]:
```
OrderedDict([('all',
              OrderedDict([('2001-2018',
                            {'pval': 0.07433498084899375,
                             'm': -0.003086717318214402,
                             'm_err': 0.001221428677173954,
                             'n': 15,
                             'y_mean': 0.1770474435541329,
                             'y_min': 0.1434634180318197,
                             'y_max': 0.24099296150814278,
                             'coverage': None,
                             'slp': -1.5142800744263278,
                             'slp_err': 0.6112855317780068,
                             'reg0': 0.20384058209203992,
                             't0': None,
                             'slp_2001': -1.4916916844764267,
                             'slp_2001_err': 0.603290719703447,
                             'reg0_2001': 0.20692729941025434,
                             'yoffs': 0.3026155362749008,
                             'period': '2001-2018'})]))])
```

Here, the first layer corresponds to the season, and the second layer corresponds to the period.

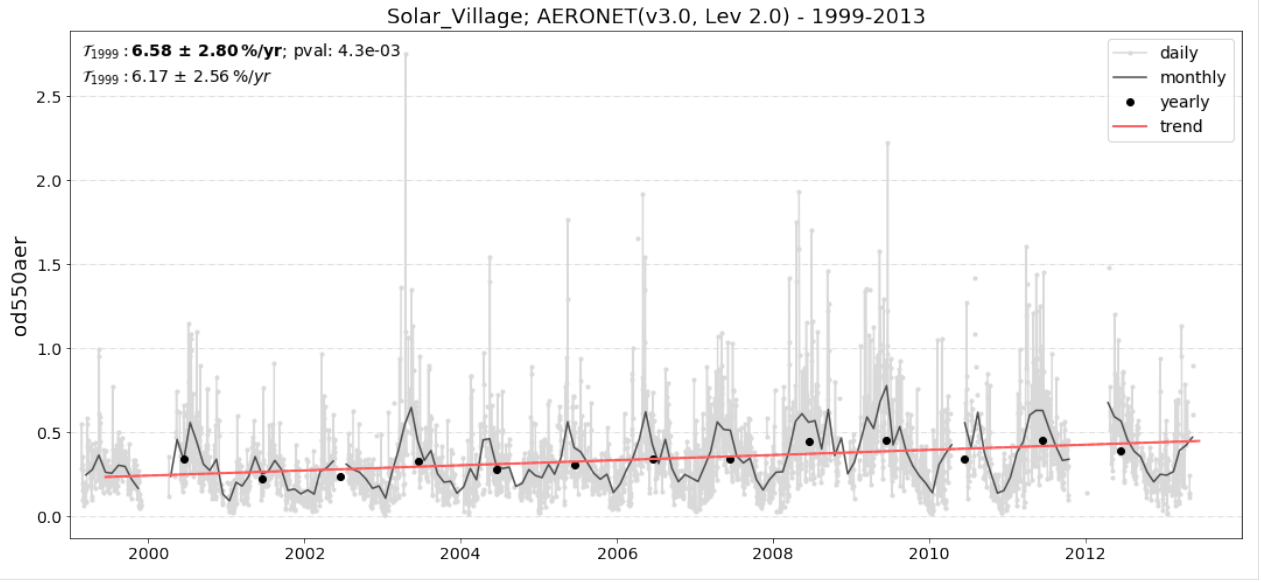
Plotting the trend can be done by using the plotting method of the `TrendsEngine` class:

[8]:
```
leipzig.trends['od550aer'].plot();
```

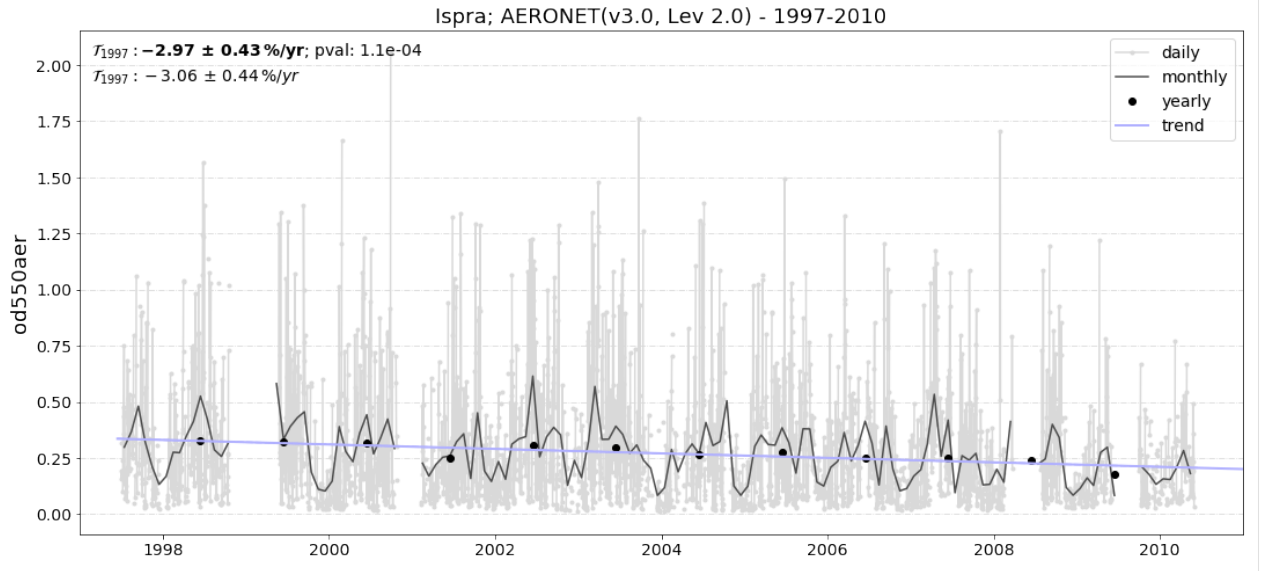


Wrap up: Do the same for other stations:

[9]:
```
sv = obsdata.to_station_data('Solar_Village')
sv.compute_trend('od550aer')
sv.trends['od550aer'].plot();
```



[10]:
```
sv = obsdata.to_station_data('Ispra')
sv.compute_trend('od550aer')
sv.trends['od550aer'].plot();
```



3.2.9 EBAS file query and database browser

The previous tutorial showed how to read EBAS NASA Ames files and gave insights into the structure of these files. However, this did not include the specification of an actual data request (for instance: get data from all stations in the arctic that contain measurements of the AOD between 2010 and 2016).

In this tutorial, we show, how such requests can be specified easily in pyaerocom and how the database can be browsed for instance by variable name, type of instrument, location or other relevant parameters.

The notebook is categorised as follows:

- Defining a request
- Retrieve all files for a request
- Browse the database

Specifying a request

Request parameters can be specified using the `EbasSQLRequest` class. E.g.:

[1]:
```
import pyaerocom as pya

request = pya.io.ebas_file_index.EbasSQLRequest(variables=('aerosol_light_scattering_
↪coefficient',
                                                           'aerosol_light_
↪backscattering_coefficient',
                                                           'pressure'),
                                                start_date="2010-01-01",
                                                stop_date="2011-01-01")
print(request)
```

```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.016 s

Pyaerocom EbasSQLRequest
------------------------
variables: ('aerosol_light_scattering_coefficient', 'aerosol_light_backscattering_
↪coefficient', 'pressure')
start_date: 2010-01-01
stop_date: 2011-01-01
station_names: None
matrices: None
altitude_range: None
lon_range: None
lat_range: None
instrument_types: None
statistics: None
```

(continues on next page)

(continued from previous page)

```
datalevel: None
Filename request string:
select distinct filename from variable join station on station.station_code=variable.
↪station_code where comp_name in ('aerosol_light_scattering_coefficient', 'aerosol_
↪light_backscattering_coefficient', 'pressure') and first_end < '2011-01-01' and last_
↪start > '2010-01-01';
```

You can also output the actual SQL query string:

[2]:
```
print(request.make_query_str())
```

```
select distinct filename from variable join station on station.station_code=variable.
↪station_code where comp_name in ('aerosol_light_scattering_coefficient', 'aerosol_
↪light_backscattering_coefficient', 'pressure') and first_end < '2011-01-01' and last_
↪start > '2010-01-01';
```

The request class is an extended dictionary and can thus be used like a dictionary:

[3]:
```
request.update(instrument_types=("nephelometer"))
print(request)
```

```

Pyaerocom EbasSQLRequest
------------------------
variables: ('aerosol_light_scattering_coefficient', 'aerosol_light_backscattering_
↪coefficient', 'pressure')
start_date: 2010-01-01
stop_date: 2011-01-01
station_names: None
matrices: None
altitude_range: None
lon_range: None
lat_range: None
instrument_types: nephelometer
statistics: None
datalevel: None
Filename request string:
select distinct filename from variable join station on station.station_code=variable.
↪station_code where instr_type in ('nephelometer') and comp_name in ('aerosol_light_
↪scattering_coefficient', 'aerosol_light_backscattering_coefficient', 'pressure') and␣
↪first_end < '2011-01-01' and last_start > '2010-01-01';
```

Execution of file request query

Now that we have defined which files we would like to look into, we can execute the query and retrieve all files that match our specifications. This can be done with the `EbasFileIndex` class:

[4]:
```
db = pya.io.EbasFileIndex()
files = db.execute_request(request)
```

[5]:
```
print("Total number of files found:{}\nRequest:\n{}".format(len(files), request))
```

```
Total number of files found:78
Request:

Pyaerocom EbasSQLRequest
------------------------
variables: ('aerosol_light_scattering_coefficient', 'aerosol_light_backscattering_
↪coefficient', 'pressure')
start_date: 2010-01-01
stop_date: 2011-01-01
station_names: None
matrices: None
altitude_range: None
lon_range: None
lat_range: None
instrument_types: nephelometer
statistics: None
datalevel: None
Filename request string:
select distinct filename from variable join station on station.station_code=variable.
↪station_code where instr_type in ('nephelometer') and comp_name in ('aerosol_light_
↪scattering_coefficient', 'aerosol_light_backscattering_coefficient', 'pressure') and␣
↪first_end < '2011-01-01' and last_start > '2010-01-01';
```

Browsing the database

The `EbasFileIndex` class provides some convenience function that allow to browse meta information for a given request. These are illustrated in the following:

For instance, we can check, what variables could actually be retrieved in the request:

[6]:
```
print(db.contains_variables(request))
```

```
[('pressure',), ('aerosol_light_backscattering_coefficient',), ('aerosol_light_
↪scattering_coefficient',)]
```

Or what matrices the data contains:

[7]:
```
print(db.contains_matrices(request))
```

```
[('instrument',), ('aerosol',), ('pm10',), ('pm1',)]
```

Or which station coordinates (lon, lat) the dataset contains:

[8]:
```
print(db.contains_coordinates(request))
```

```
[(23.583333, 42.166667), (-79.783839, 44.231006), (-122.9576034546, 50.059299469), (-104.
↪986864, 54.353743), (-62.3415260315, 82.4991455078), (7.985, 46.5475), (11.0096197128,␣
↪47.8014984131), (12.93386, 51.53014), (10.97964, 47.4165), (-8.266, -70.666), (-16.
↪4994, 28.309), (-3.605, 37.164), (-6.733333, 37.1), (2.35, 41.766667), (24.283333, 61.
↪85), (24.116111111, 67.973333333), (2.964886, 45.772223), (25.666667, 35.316667), (19.
↪583333, 46.966667), (-9.89944, 53.32583), (77.151389, 28.427778), (8.633333, 45.8),␣
↪(10.7, 44.183333), (126.3300018311, 36.5383338928), (126.17, 33.28), (4.926389, 51.
↪970278), (8.252, 58.38853), (11.88668, 78.90715), (2.533333, -72.016667), (-65.618, 18.
↪381), (13.15, 56.016667), (120.87, 23.47), (-156.6114654541, 71.3230133057), (-111.
```

(continues on next page)

(continued from previous page)

```
↪9841, 35.9731), (-112.1288, 36.0778), (-111.6832, 34.3405), (-88.36667, 40.05), (-83.
↪9416, 35.6334), (-112.8, 31.9506), (-109.3889, 32.0097), (-155.5761566162, 19.
↪5362300873), (-86.148, 37.1317), (-68.2608, 44.3772), (-113.9958, 48.5103), (-114.2158,
↪ 39.005), (-81.7, 36.2), (-103.1772, 29.3022), (-78.4358, 38.5225), (-122.1243, 46.
↪7582), (-97.484999, 36.605), (-24.7999992371, -89.9969482422), (-124.1510009766, 41.
↪0541000366), (-105.5457, 40.2783), (-111.9692, 35.1406), (-77.04, 38.9), (-109.7958,␣
↪34.9139), (18.48968, -34.35348)]
```

Now, let's narrow this down:

[9]:
```
request.update(lat_range=(60, 90))
print(db.contains_coordinates(request))
```

```
[(-62.3415260315, 82.4991455078), (24.283333, 61.85), (24.116111111, 67.973333333), (11.
↪88668, 78.90715), (-156.6114654541, 71.3230133057)]
```

Print all station names:

[10]:
```
print(db.contains_station_names(request))
```

```
[('Alert',), ('Hyytiälä',), ('Pallas (Sammaltunturi)',), ('Zeppelin mountain (Ny-Ålesund)
↪',), ('Barrow',)]
```

Custom browsing

The previous browsing methods (e.g. `contains_coordinates()`, `contains_matrices`, `contains_variables`) are all just simple wrappers for the general query method `make_query_str` of the `EbasSQLRequest` class, that is then called by the `EbasFileIndex` class using the method `execute_request`). Thus, if needed, you may define your own request simply by using the provided interface. Here an example using the request constraints specified above. Let's say we want to retrieve a list of station names and their coordinates (lon, lat, alt). This can be done by calling (we store the results in a list named `station_info`):

[11]:
```
query_str = request.make_query_str(what=("station_name",
                                         "station_longitude",
                                         "station_latitude",
                                         "station_altitude"))

station_info = db.execute_request(query_str)

for item in station_info:
    print(item)
```

```
('Alert', -62.3415260315, 82.4991455078, 210.0)
('Hyytiälä', 24.283333, 61.85, 181.0)
('Pallas (Sammaltunturi)', 24.116111111, 67.973333333, 565.0)
('Zeppelin mountain (Ny-Ålesund)', 11.88668, 78.90715, 474.0)
('Barrow', -156.6114654541, 71.3230133057, 11.0)
```

You can see that the results for each station are stored in tuples in the order of the request.

Read all files

Let's update the file list and read all files.

[12]:
```
files = db.execute_request(request)
print("Total number of files found: {}".format(len(files)))
```

```
Total number of files found: 5
```

Let's read the files:

[13]:
```
import os
data = []
data_dir = os.path.join(pya.const.OBSCONFIG["EBASMC"]["PATH"], 'data')
for f in files:
    data.append(pya.io.EbasNasaAmesFile(os.path.join(data_dir, f[0])))
```

[14]:
```
len(data)
```

[14]:
```
5
```

3.2.10 Tutorial showing how to read EBAS NASA Ames files

This low-level tutorial shows how to read an EBAS NASA Ames file using the class EbasNasaAmesFile of pyaerocom and how to access the import data and metadata.

NOTE: variable names and names of metadata attributes below use the EBAS conventions and **not the AeroCom naming conventions**, since the purpose of the `EbasNasaAmesFile` reading routine is to solely import the content of the original data files (provided by EBAS) into a python interface. If you intend to use EBAS data for AeroCom purposes (e.g. model intercomparison), please use the ReadEbas routine (or the ReadUngridded factory class) which is doing the mapping to AeroCom naming conventions.

Please see here for information related to the EBAS NASA Ames file format.

[1]:
```
import pyaerocom as pya
import glob
```

```
Initating pyaerocom configuration
Checking database access...
Checking access to: /lustre/storeA
Access to lustre database: True
Init data paths for lustre
Expired time: 0.022 s
```

[2]:
```
ebasdir = pya.const.EBASMC_DATA_DIR
ebasdir
```

[2]:
```
'/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/EBASMultiColumn/data/data/'
```

[3]:
```
files = glob.glob('{}DE0043G*2010*nephelometer*lev2.nas'.format(ebasdir))
```

[4]:
```
print('No. of files found: {}'.format(len(files)))
```

```
No. of files found: 2
```

[5]:
```
files[0]
```

[5]:
```
'/lustre/storeA/project/aerocom/aerocom1/AEROCOM_OBSDATA/EBASMultiColumn/data/data/
↪DE0043G.20100201000000.20150304123917.nephelometer..pm10.11mo.1h.DE09L_TSI_Neph_3563.
↪DE09L_scatt_NEPH.lev2.nas'
```

Read the first file that was found:

[6]:
```
mc = pya.io.EbasNasaAmesFile(file=files[0],
                             only_head=False,          #set True if you only want to␣
↪import header
                             replace_invalid_nan=True, #replace invalid values with NaNs
                             convert_timestamps=True,  #compute datetime64 timestamps␣
↪from numerical values
                             decode_flags=True)        #decode all flags (e.g. 0.
↪111222333 -> 111 222 333)

print(mc)
```

```
Pyaerocom EbasNasaAmesFile
--------------------------

num_head_lines: 91
num_head_fmt: 1001
data_originator: Flentje, Harald
sponsor_organisation: DE09L, Deutscher Wetterdienst, DWD, Met. Obs., Hohenspeissenberg, ,
↪ 82283, Hohenspeissenberg, Germany
submitter: Flentje, Harald
project_association: ACTRIS EMEP GAW-WDCA
vol_num: 1
vol_totnum: 1
ref_date: 2010-01-01T00:00:00
revision_date: 2015-03-04T00:00:00
freq: 0.041667
descr_time_unit: days from file reference point
num_cols_dependent: 23
mul_factors (list, 23 items): [1.00, 1.00, ..., 1.00, 1.00]
vals_invalid (list, 23 items): [1000, 9999.0, ..., 9999.999999, 10.00]
descr_first_col: end_time of measurement, days from the file reference point

   Column variable definitions
   -------------------------------
   EbasColDef: name=starttime, unit=days, is_var=False, is_flag=False, flag_col=23,
   EbasColDef: name=endtime, unit=days, is_var=False, is_flag=False, flag_col=23,
   EbasColDef: name=pressure, unit=hPa, is_var=True, is_flag=False, flag_col=23,␣
↪location=instrument internal, statistics=arithmetic mean, matrix=instrument, detection_
↪limit=, detection_limit_expl.=, measurement_uncertainty=, measurement_uncertainty_expl.
↪=,
   EbasColDef: name=relative_humidity, unit=%, is_var=True, is_flag=False, flag_col=23,␣
↪location=instrument internal, statistics=arithmetic mean, matrix=instrument, detection_
```

(continues on next page)

(continued from previous page)

```
↪limit=, detection_limit_expl.=, measurement_uncertainty=, measurement_uncertainty_expl.
↪=,
   EbasColDef: name=temperature, unit=K, is_var=True, is_flag=False, flag_col=23,␣
↪location=instrument internal, statistics=arithmetic mean, matrix=instrument, detection_
↪limit=, detection_limit_expl.=, measurement_uncertainty=, measurement_uncertainty_expl.
↪=,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=450.0 nm, statistics=arithmetic mean,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=450.0 nm, statistics=percentile:15.87,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=450.0 nm, statistics=percentile:84.13,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=550.0 nm, statistics=arithmetic mean,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=550.0 nm, statistics=percentile:15.87,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=550.0 nm, statistics=percentile:84.13,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=700.0 nm, statistics=arithmetic mean,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=700.0 nm, statistics=percentile:15.87,
   EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=700.0 nm, statistics=percentile:84.13,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=450.0 nm, statistics=arithmetic mean,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=450.0 nm, statistics=percentile:15.87,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=450.0 nm, statistics=percentile:84.13,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=550.0 nm, statistics=arithmetic mean,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=550.0 nm, statistics=percentile:15.87,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=550.0 nm, statistics=percentile:84.13,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=700.0 nm, statistics=arithmetic mean,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=700.0 nm, statistics=percentile:15.87,
   EbasColDef: name=aerosol_light_scattering_coefficient, unit=1/Mm, is_var=True, is_
↪flag=False, flag_col=23, wavelength=700.0 nm, statistics=percentile:84.13,
   EbasColDef: name=numflag, unit=no unit, is_var=False, is_flag=True, flag_col=None,

   EBAS meta data
   ------------------
decode_flags: True
data_definition: EBAS_1.1
set_type_code: TU
timezone: UTC
file_name: DE0043G.20100201000000.20150304123917.nephelometer..pm10.11mo.1h.DE09L_TSI_
↪Neph_3563.DE09L_scatt_NEPH.lev2.nas
```

(continues on next page)

(continued from previous page)

```
file_creation: 20190701194839
startdate: 20100201000000
revision_date: 20150304123917
version: 1
version_description: initial revision
data_level: 2
period_code: 11mo
resolution_code: 1h
sample_duration: 1h
orig._time_res.: 10mn
station_code: DE0043G
platform_code: DE0043S
station_name: Hohenpeissenberg
station_wdca-id: GAWADE__HPB
station_gaw-id: HPB
station_gaw-name: Hohenpeissenberg
station_land_use: Grassland
station_setting: Mountain
station_gaw_type: G
station_wmo_region: 6
station_latitude: 47.8014984131
station_longitude: 11.0096197128
station_altitude: 985.0 m
measurement_height: 15.0 m
regime: IMG
component:
unit: 1/Mm
matrix: pm10
laboratory_code: DE09L
instrument_type: nephelometer
instrument_name: TSI_Neph_3563
instrument_manufacturer: TSI
instrument_model: 3563
method_ref: DE09L_scatt_NEPH
standard_method: cal-gas=CO2+AIR_truncation-correction=Anderson1998
inlet_type: Impactor--direct
inlet_description: PM10 at ambient humidity inlet, Digitel, flow 170 l/min
humidity/temperature_control: Nafion dryer
volume_std._temperature: 273.15 K
volume_std._pressure: 1013.25 hPa
detection_limit: 0.3 1/Mm
detection_limit_expl.: Determined only by instrument counting statistics, no detection␣
↪limit flag used
measurement_uncertainty: 0.3 1/Mm
measurement_uncertainty_expl.: Determined only by instrument counting statistics, no␣
↪detection limit flag used
zero/negative_values: Zero values may appear due to statistical variations at very low␣
↪concentrations
originator: Flentje, Harald, Harald.Flentje@dwd.de, , , , , , , ,
submitter: Flentje, Harald, Harald.Flentje@dwd.de, , , , , , , ,
acknowledgement: Request acknowledgement details from data originator
comment: Angstrom-based Anderson & Ogren 1998 corr used for truncation correction
```

(continues on next page)

(continued from previous page)

```
   Data
   --------
[[3.10000000e+01 3.10416660e+01            nan ...            nan
            nan 9.99000000e-01]
 [3.10416670e+01 3.10833330e+01            nan ...            nan
            nan 9.99000000e-01]
 [3.10833330e+01 3.11249990e+01            nan ...            nan
            nan 9.99000000e-01]
 ...
 [3.64875000e+02 3.64916666e+02 9.04000000e+02 ... 1.35433110e+01
  1.62446480e+01 1.00000000e-01]
 [3.64916667e+02 3.64958333e+02 9.04000000e+02 ... 1.13367710e+01
  1.42932090e+01 1.00000000e-01]
 [3.64958333e+02 3.64999999e+02 9.03000000e+02 ... 1.13635590e+01
  1.40839410e+01 1.00000000e-01]]
Colnum: 24
Timestamps: 8016
```

The NASA Ames files are strucured in the same way as they are represented by in the instance of the EbasNasaAmesFile class.

- A header with global metadata
- One row that specifies multiplication factors for each data column (`mul_factors`)
- One row that specifies NaN-equivalent values for each data column (`vals_invalid`)
- A number of rows specifying metainformation for each data column in the file (i.e. 12 rows, if the data has 12 columns)
- Dataset specific metadata
- Data block: rows are timestamps, columns are different columns specified in the header (cf. 2 points above)
  - Represented by 2D numpy array (`data` attribute) where first index is row and second index is column

For details related to the file format see here.

[7]:
```
print(mc.shape)
```

```
(8016, 24)
```

Data array

The data is imported as a 2D numpy array which is accessible via the `data` attribute:

[8]:
```
mc.data
```

[8]:
```
array([[3.10000000e+01, 3.10416660e+01,            nan, ...,
                   nan,            nan, 9.99000000e-01],
       [3.10416670e+01, 3.10833330e+01,            nan, ...,
                   nan,            nan, 9.99000000e-01],
       [3.10833330e+01, 3.11249990e+01,            nan, ...,
                   nan,            nan, 9.99000000e-01],
       ...,
```

(continues on next page)

(continued from previous page)

```
       [3.64875000e+02, 3.64916666e+02, 9.04000000e+02, ...,
        1.35433110e+01, 1.62446480e+01, 1.00000000e-01],
       [3.64916667e+02, 3.64958333e+02, 9.04000000e+02, ...,
        1.13367710e+01, 1.42932090e+01, 1.00000000e-01],
       [3.64958333e+02, 3.64999999e+02, 9.03000000e+02, ...,
        1.13635590e+01, 1.40839410e+01, 1.00000000e-01]])
```

The first index corresponds to the individual measurements (rows in file) and the second index corresponds to the individual columns that are stored in the file.

Column information

Detailed information about each column can be accessed via the `var_defs` attribute, the first two columns are always the start and stop timestamps:

```
[9]: mc.var_defs[0]
```

```
[9]: EbasColDef: name=starttime, unit=days, is_var=False, is_flag=False, flag_col=23,
```

```
[10]: mc.var_defs[1]
```

```
[10]: EbasColDef: name=endtime, unit=days, is_var=False, is_flag=False, flag_col=23,
```

After the start / stop columns follow the individual data columns.

```
[11]: mc.var_defs[2]
```

```
[11]: EbasColDef: name=pressure, unit=hPa, is_var=True, is_flag=False, flag_col=23,␣
      ↪location=instrument internal, statistics=arithmetic mean, matrix=instrument, detection_
      ↪limit=, detection_limit_expl.=, measurement_uncertainty=, measurement_uncertainty_expl.
      ↪=,
```

```
[12]: mc.var_defs[3]
```

```
[12]: EbasColDef: name=relative_humidity, unit=%, is_var=True, is_flag=False, flag_col=23,␣
      ↪location=instrument internal, statistics=arithmetic mean, matrix=instrument, detection_
      ↪limit=, detection_limit_expl.=, measurement_uncertainty=, measurement_uncertainty_expl.
      ↪=,
```

```
[13]: mc.var_defs[4]
```

```
[13]: EbasColDef: name=temperature, unit=K, is_var=True, is_flag=False, flag_col=23,␣
      ↪location=instrument internal, statistics=arithmetic mean, matrix=instrument, detection_
      ↪limit=, detection_limit_expl.=, measurement_uncertainty=, measurement_uncertainty_expl.
      ↪=,
```

```
[14]: mc.var_defs[5]
```

```
[14]: EbasColDef: name=aerosol_light_backscattering_coefficient, unit=1/Mm, is_var=True, is_
      ↪flag=False, flag_col=23, wavelength=450.0 nm, statistics=arithmetic mean,
```

In addition to the data columns in the files (such as time stamps, or measured values of a certain variable) there is **at least one** flag column in the data array and each data column has assigned one flag column (cf. output above where the index of the flag column for each data column is provided `flag_col=23`, i.e. column 23 is the flag column assigned to each of the 5 data columns that were displayed exemplary above:

[15]:
```
mc.var_defs[23]
```

[15]:
```
EbasColDef: name=numflag, unit=no unit, is_var=False, is_flag=True, flag_col=None,
```

The `is_var` attribute specifies, whether this column contains actual variable data or if it is a flag column. A NASA Ames file can have one or more flag columns that can be used to identify valid or invalid measurments. Each flag in a flag column comprises a floating point number that has encoded up to 3 3-digit numerical flags which are specified here:

https://github.com/metno/pyaerocom/blob/master/pyaerocom/data/ebas_flags.csv

More info about the flags follows below. You can see, that the column 4 printed above has assigned column 12 (index 11) as flag column.

If you want to see an overview of all available columns in the file you may use the following command:

[16]:
```
mc.print_col_info()
```

```
Column 0
Pyaerocom EbasColDef
--------------------
name: starttime
unit: days
is_var: False
is_flag: False
flag_col: 23

Column 1
Pyaerocom EbasColDef
--------------------
name: endtime
unit: days
is_var: False
is_flag: False
flag_col: 23

Column 2
Pyaerocom EbasColDef
--------------------
name: pressure
unit: hPa
is_var: True
is_flag: False
flag_col: 23
location: instrument internal
statistics: arithmetic mean
matrix: instrument
detection_limit:
detection_limit_expl.:
measurement_uncertainty:
measurement_uncertainty_expl.:

Column 3
Pyaerocom EbasColDef
--------------------
```

(continues on next page)

(continued from previous page)

```
name: relative_humidity
unit: %
is_var: True
is_flag: False
flag_col: 23
location: instrument internal
statistics: arithmetic mean
matrix: instrument
detection_limit:
detection_limit_expl.:
measurement_uncertainty:
measurement_uncertainty_expl.:

Column 4
Pyaerocom EbasColDef
--------------------
name: temperature
unit: K
is_var: True
is_flag: False
flag_col: 23
location: instrument internal
statistics: arithmetic mean
matrix: instrument
detection_limit:
detection_limit_expl.:
measurement_uncertainty:
measurement_uncertainty_expl.:

Column 5
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 450.0 nm
statistics: arithmetic mean

Column 6
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 450.0 nm
statistics: percentile:15.87

Column 7
```

(continues on next page)

(continued from previous page)

```
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 450.0 nm
statistics: percentile:84.13

Column 8
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 550.0 nm
statistics: arithmetic mean

Column 9
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 550.0 nm
statistics: percentile:15.87

Column 10
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 550.0 nm
statistics: percentile:84.13

Column 11
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 700.0 nm
```

(continues on next page)

(continued from previous page)

```
statistics: arithmetic mean

Column 12
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 700.0 nm
statistics: percentile:15.87

Column 13
Pyaerocom EbasColDef
--------------------
name: aerosol_light_backscattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 700.0 nm
statistics: percentile:84.13

Column 14
Pyaerocom EbasColDef
--------------------
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 450.0 nm
statistics: arithmetic mean

Column 15
Pyaerocom EbasColDef
--------------------
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 450.0 nm
statistics: percentile:15.87

Column 16
Pyaerocom EbasColDef
--------------------
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
```

(continues on next page)

(continued from previous page)

```
is_flag: False
flag_col: 23
wavelength: 450.0 nm
statistics: percentile:84.13

Column 17
Pyaerocom EbasColDef
--------------------
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 550.0 nm
statistics: arithmetic mean

Column 18
Pyaerocom EbasColDef
--------------------
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 550.0 nm
statistics: percentile:15.87

Column 19
Pyaerocom EbasColDef
--------------------
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 550.0 nm
statistics: percentile:84.13

Column 20
Pyaerocom EbasColDef
--------------------
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 700.0 nm
statistics: arithmetic mean

Column 21
Pyaerocom EbasColDef
--------------------
```

(continues on next page)

(continued from previous page)

```
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 700.0 nm
statistics: percentile:15.87

Column 22
Pyaerocom EbasColDef
--------------------
name: aerosol_light_scattering_coefficient
unit: 1/Mm
is_var: True
is_flag: False
flag_col: 23
wavelength: 700.0 nm
statistics: percentile:84.13

Column 23
Pyaerocom EbasColDef
--------------------
name: numflag
unit: no unit
is_var: False
is_flag: True
flag_col: None
```

You can see that all variable columns were assigned the same flag column, since there is only one flag column at the end (index 23). This would be different if there were multiple flag columns (e.g. one for each variable).

Access flag information

You can access the flags for each column using the `flag_col_info` attribute of the file (and the key of the respective flag column, that you want to access, here->11).

[17]:
```
flagcol = mc.flag_col_info[23]
flagcol
```

[17]:
```
<pyaerocom.io.ebas_nasa_ames.EbasFlagCol at 0x7f11494eb2b0>
```

The raw flags can be accessed via:

[18]:
```
flagcol.raw_data
```

[18]:
```
array([0.999, 0.999, 0.999, ..., 0.1  , 0.1  , 0.1  ])
```

And the processed flags are in stored in a (Nx3) numpy array where N is the total number of timestamps.

[19]:
```
flagcol.decoded
```

[19]:
```
array([[999,   0,   0],
       [999,   0,   0],
       [999,   0,   0],
       ...,
       [100,   0,   0],
       [100,   0,   0],
       [100,   0,   0]])
```

For instance, access the flags of the 5 timestamp:

[20]:
```
flagcol.decoded[4]
```

[20]:
```
array([999,   0,   0])
```

This timestamp contains 1 (of the possible up to 3) flags: 999.

Validity of a combination of the flags can be directly accessed via:

[21]:
```
flagcol.valid[4]
```

[21]:
```
False
```

This flag (999) evaluates to an invalid measurement. Looking into the flag definition file we see that these two flags have the following meaning:

- 999,'Missing measurement, unspecified reason','M'

where the last string specifies if this flag is valid (V) or invalid (I) or missing (M).

Convert object to pandas Dataframe

[22]:
```
df = mc.to_dataframe()
df.head()
```

[22]:
```
                     starttime_days  endtime_days  \
2010-02-01 00:29:59       31.000000     31.041666
2010-02-01 01:29:59       31.041667     31.083333
2010-02-01 02:29:59       31.083333     31.124999
2010-02-01 03:29:59       31.125000     31.166666
2010-02-01 04:29:59       31.166667     31.208333

                     pressure_hPa_instrument_arithmetic mean  \
2010-02-01 00:29:59                                      NaN
2010-02-01 01:29:59                                      NaN
2010-02-01 02:29:59                                      NaN
2010-02-01 03:29:59                                      NaN
2010-02-01 04:29:59                                      NaN

                     relative_humidity_%_instrument_arithmetic mean  \
2010-02-01 00:29:59                                             NaN
2010-02-01 01:29:59                                             NaN
2010-02-01 02:29:59                                             NaN
2010-02-01 03:29:59                                             NaN
2010-02-01 04:29:59                                             NaN
```

(continues on next page)

(continued from previous page)

```
                     temperature_K_instrument_arithmetic mean  \
2010-02-01 00:29:59                                        NaN
2010-02-01 01:29:59                                        NaN
2010-02-01 02:29:59                                        NaN
2010-02-01 03:29:59                                        NaN
2010-02-01 04:29:59                                        NaN

                     aerosol_light_backscattering_coefficient_1/Mm_450.0nm_arithmetic␣
↪mean  \
2010-02-01 00:29:59                                                 NaN
2010-02-01 01:29:59                                                 NaN
2010-02-01 02:29:59                                                 NaN
2010-02-01 03:29:59                                                 NaN
2010-02-01 04:29:59                                                 NaN

                     aerosol_light_backscattering_coefficient_1/Mm_450.0nm_percentile:15.
↪87  \
2010-02-01 00:29:59                                                 NaN
2010-02-01 01:29:59                                                 NaN
2010-02-01 02:29:59                                                 NaN
2010-02-01 03:29:59                                                 NaN
2010-02-01 04:29:59                                                 NaN

                     aerosol_light_backscattering_coefficient_1/Mm_450.0nm_percentile:84.
↪13  \
2010-02-01 00:29:59                                                 NaN
2010-02-01 01:29:59                                                 NaN
2010-02-01 02:29:59                                                 NaN
2010-02-01 03:29:59                                                 NaN
2010-02-01 04:29:59                                                 NaN

                     aerosol_light_backscattering_coefficient_1/Mm_550.0nm_arithmetic␣
↪mean  \
2010-02-01 00:29:59                                                 NaN
2010-02-01 01:29:59                                                 NaN
2010-02-01 02:29:59                                                 NaN
2010-02-01 03:29:59                                                 NaN
2010-02-01 04:29:59                                                 NaN

                     aerosol_light_backscattering_coefficient_1/Mm_550.0nm_percentile:15.
↪87  \
2010-02-01 00:29:59                                                 NaN
2010-02-01 01:29:59                                                 NaN
2010-02-01 02:29:59                                                 NaN
2010-02-01 03:29:59                                                 NaN
2010-02-01 04:29:59                                                 NaN

                     ...  \
2010-02-01 00:29:59  ...
2010-02-01 01:29:59  ...
2010-02-01 02:29:59  ...
2010-02-01 03:29:59  ...
```

(continues on next page)

(continued from previous page)

```
2010-02-01 04:29:59  ...

                     aerosol_light_scattering_coefficient_1/Mm_450.0nm_arithmetic mean  \
2010-02-01 00:29:59                                                  NaN
2010-02-01 01:29:59                                                  NaN
2010-02-01 02:29:59                                                  NaN
2010-02-01 03:29:59                                                  NaN
2010-02-01 04:29:59                                                  NaN

                     aerosol_light_scattering_coefficient_1/Mm_450.0nm_percentile:15.87 ␣
↪\
2010-02-01 00:29:59                                                  NaN
2010-02-01 01:29:59                                                  NaN
2010-02-01 02:29:59                                                  NaN
2010-02-01 03:29:59                                                  NaN
2010-02-01 04:29:59                                                  NaN

                     aerosol_light_scattering_coefficient_1/Mm_450.0nm_percentile:84.13 ␣
↪\
2010-02-01 00:29:59                                                  NaN
2010-02-01 01:29:59                                                  NaN
2010-02-01 02:29:59                                                  NaN
2010-02-01 03:29:59                                                  NaN
2010-02-01 04:29:59                                                  NaN

                     aerosol_light_scattering_coefficient_1/Mm_550.0nm_arithmetic mean  \
2010-02-01 00:29:59                                                  NaN
2010-02-01 01:29:59                                                  NaN
2010-02-01 02:29:59                                                  NaN
2010-02-01 03:29:59                                                  NaN
2010-02-01 04:29:59                                                  NaN

                     aerosol_light_scattering_coefficient_1/Mm_550.0nm_percentile:15.87 ␣
↪\
2010-02-01 00:29:59                                                  NaN
2010-02-01 01:29:59                                                  NaN
2010-02-01 02:29:59                                                  NaN
2010-02-01 03:29:59                                                  NaN
2010-02-01 04:29:59                                                  NaN

                     aerosol_light_scattering_coefficient_1/Mm_550.0nm_percentile:84.13 ␣
↪\
2010-02-01 00:29:59                                                  NaN
2010-02-01 01:29:59                                                  NaN
2010-02-01 02:29:59                                                  NaN
2010-02-01 03:29:59                                                  NaN
2010-02-01 04:29:59                                                  NaN

                     aerosol_light_scattering_coefficient_1/Mm_700.0nm_arithmetic mean  \
2010-02-01 00:29:59                                                  NaN
2010-02-01 01:29:59                                                  NaN
2010-02-01 02:29:59                                                  NaN
```

(continues on next page)

(continued from previous page)

```
2010-02-01 03:29:59                                                NaN
2010-02-01 04:29:59                                                NaN

                    aerosol_light_scattering_coefficient_1/Mm_700.0nm_percentile:15.87 ␣
↪\
2010-02-01 00:29:59                                                NaN
2010-02-01 01:29:59                                                NaN
2010-02-01 02:29:59                                                NaN
2010-02-01 03:29:59                                                NaN
2010-02-01 04:29:59                                                NaN

                    aerosol_light_scattering_coefficient_1/Mm_700.0nm_percentile:84.13 ␣
↪\
2010-02-01 00:29:59                                                NaN
2010-02-01 01:29:59                                                NaN
2010-02-01 02:29:59                                                NaN
2010-02-01 03:29:59                                                NaN
2010-02-01 04:29:59                                                NaN

                    numflag_no unit
2010-02-01 00:29:59           0.999
2010-02-01 01:29:59           0.999
2010-02-01 02:29:59           0.999
2010-02-01 03:29:59           0.999
2010-02-01 04:29:59           0.999

[5 rows x 24 columns]
```

You may also apply selection constraints when converting to a DataFrame

[23]:
```
scattering = mc.to_dataframe('aerosol_light_scattering_coefficient', statistics=
↪'arithmetic mean')
scattering
```

[23]:
```
                    aerosol_light_scattering_coefficient_1/Mm_450.0nm_arithmetic mean  \
2010-02-01 00:29:59                                                NaN
2010-02-01 01:29:59                                                NaN
2010-02-01 02:29:59                                                NaN
2010-02-01 03:29:59                                                NaN
2010-02-01 04:29:59                                                NaN
...                                                                ...
2010-12-31 19:29:59                                          29.312950
2010-12-31 20:29:59                                          28.166000
2010-12-31 21:29:59                                          36.854919
2010-12-31 22:29:59                                          30.724499
2010-12-31 23:29:59                                          28.431919

                    aerosol_light_scattering_coefficient_1/Mm_550.0nm_arithmetic mean  \
2010-02-01 00:29:59                                                NaN
2010-02-01 01:29:59                                                NaN
2010-02-01 02:29:59                                                NaN
```

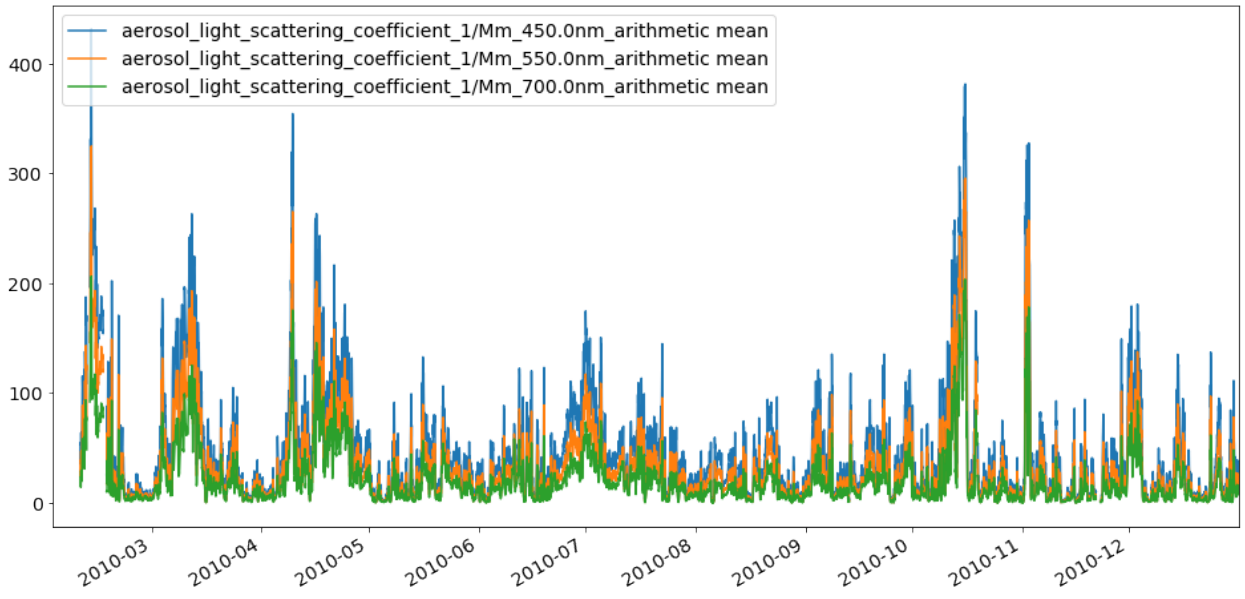
(continues on next page)

(continued from previous page)

```
2010-02-01 03:29:59                                                NaN
2010-02-01 04:29:59                                                NaN
...                                                                ...
2010-12-31 19:29:59                                          19.707840
2010-12-31 20:29:59                                          19.270330
2010-12-31 21:29:59                                          25.116589
2010-12-31 22:29:59                                          21.249210
2010-12-31 23:29:59                                          20.387381

                     aerosol_light_scattering_coefficient_1/Mm_700.0nm_arithmetic mean
2010-02-01 00:29:59                                                NaN
2010-02-01 01:29:59                                                NaN
2010-02-01 02:29:59                                                NaN
2010-02-01 03:29:59                                                NaN
2010-02-01 04:29:59                                                NaN
...                                                                ...
2010-12-31 19:29:59                                          13.09690
2010-12-31 20:29:59                                          11.51522
2010-12-31 21:29:59                                          14.89398
2010-12-31 22:29:59                                          12.81499
2010-12-31 23:29:59                                          12.72375

[8016 rows x 3 columns]
```

[24]:
```
scattering.plot(figsize=(16,8))
```

[24]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f11494752e8>
```



Note: These are old tutorials that are not tested against the latest pyaerocom version. Thus, some of the notebooks may not work properly anymore with the most recent pyaerocom version.

Also, most of these tutorials rely on access to internal servers of the Norwegian Meteorological institute and can, thus, not be executed interactively from the outside world.

However, the tutorials provide a comprehensive introduction into pyaerocom and most of the introduced features will work with the current pyaerocom version. Thus they are a useful addition to the updated notebooks above, which rely on publicly accessible data and can thus be run interactively.

- Getting started | *tut00_get_started.ipynb*
- Regions in pyaerocom | *tut01_intro_regions.ipynb*
- Reading of gridded data | *tut02_intro_ReadGridded.ipynb*
- Working with gridded data | *tut04_intro_class_GriddedData.ipynb*
- Reading of ungridded data | *tut05_intro_ungridded_reading.ipynb*
- Working with ungridded observations | *tut06_intro_UngriddedData_and_StationData_classes.ipynb*
- Colocation of models and observations | *tut07_intro_colocation.ipynb*
- Merging of StationData | *add04_stationdata_merging.ipynb*
- Computation of trends | *tut08_trends_computation.ipynb*
- EBAS data: finding the right data files | *add03_ebas_database_browser.ipynb*
- EBAS data: low-level reading of NASA Ames files | *add02_read_ebas_nasa_ames.ipynb*

CHAPTER

FOUR

CORE API

Documentation of the core API of pyaerocom.

4.1 Logging

`pyaerocom` initializes logging automatically on import in the following way.

1. `info`-messages or worse are logged to `logs/pyaerocom.log.$PID` or (dynamic feature) the file given in the environment variable `PYAEROCOM_LOG_FILE` - (dynamic feature) these log-files will be deleted after 7 days.
2. `warning`-messages or worse are also printed on stdout. (dynamic feature) Output to stdout is disabled if the script is called non-interactive.

Putting a file with the name `logging.ini` in the scripts current working directory will use that configuration instead of above described default. An example `logging.ini` doing about the same as described above, except for the dynamic features, and enable `debug` logging on one package (`pyaerocom.io.ungridded`), is provided here:

```
[loggers]
keys=root,pyaerocom-ungridded

[handlers]
keys=console,file

[formatters]
keys=plain,detailed

[formatter_plain]
format=%(message)s

[formatter_detailed]
format=%(asctime)s:%(name)s:%(levelname)s:%(message)s
datefmt=%F %T

[handler_console]
class=StreamHandler
formatter=plain
args=(sys.stdout,)
level=WARN

[handler_file]
class=FileHandler
```

(continues on next page)

(continued from previous page)

```
formatter=detailed
level=DEBUG
file_name=logs/pyaerocom.log.%(pid)s
args=('%(file_name)s', "w")


[logger_root]
handlers=file,console
level=INFO

[logger_pyaerocom-ungridded]
handlers=file
qualname=pyaerocom.io.readungriddedbase
level=DEBUG
propagate=0
```

4.2 Data classes

4.2.1 Gridded data

class pyaerocom.griddeddata.**GriddedData**(*input=None*, *var_name=None*, *check_unit=True*, *convert_unit_on_init=True*, ***meta*)

pyaerocom object representing gridded data (e.g. model diagnostics)

Gridded data refers to data that can be represented on a regular, multidimensional grid. In `pyaerocom` this comprises both model output and diagnostics as well as gridded level 3 satellite data, typically with dimensions *latitude, longitude, time* (for surface or columnar data) and an additional dimension *lev* (or similar) for vertically resolved data.

Under the hood, this data object is based on (but not inherited from) the `iris.cube.Cube` object, and makes large use of the therein implemented functionality (many methods implemented here in *GriddedData* are simply wrappers for *Cube* methods.

Note: Note that the implemented functionality in this class is mostly limited to what is needed in the pyaerocom API (e.g. for *pyaerocom.colocation* routines or data import) and is not aimed at replacing or competing with similar data classes such as `iris.cube.Cube` or `xarray.DataArray`. Rather, dependent on the use case, one or another of such gridded data objects is needed for optimal processing, which is why *GriddedData* provides methods and / or attributes to convert to or from other such data classes (e.g. *GriddedData.cube* is an instance of `iris.cube.Cube` and method *GriddedData.to_xarray()* can be used to convert to `xarray.DataArray`). Thus, *GriddedData* can be considered rather high-level as compared to the other mentioned data classes from iris or xarray.

Note: Since *GriddedData* object is based on the `iris.cube.Cube` object it is optimised for netCDF files that follow the CF conventions and may not work out of the box for files that do not follow this standard.

Parameters

- **input** (`str:` or `Cube`) – data input. Can be a single .nc file or a preloaded iris Cube.

- **var_name** (str, optional) – variable name that is extracted if *input* is a file path. Irrelevant if *input* is preloaded Cube
- **check_unit** (*bool*) – if True, the assigned unit is checked and if it is an alias to another unit the unit string will be updated. It will print a warning if the unit is invalid or not equal the associated AeroCom unit for the input variable. Set *convert_unit_on_init* to True, if you want an automatic conversion to AeroCom units. Defaults to True.
- **convert_unit_on_init** (*bool*) – if True and if unit check indicates non-conformity with AeroCom unit it will be converted automatically, and warning will be printed if that conversion fails. Defaults to True.

COORDS_ORDER_TSERIES = ['time', 'latitude', 'longitude']

Req. order of dimension coordinates for time-series computation

SUPPORTED_VERT_SCHEMES = ['mean', 'max', 'min', 'surface', 'altitude', 'profile']

property TS_TYPES

List with valid filename encryptions specifying temporal resolution

aerocom_filename(*at_stations=False*)

Filename of data following Aerocom 3 conventions

Parameters

at_stations (*str*) – if True, then AtStations string will be included in filename

Returns

generated file name based on what is in this object

Return type

str

aerocom_savename(*data_id=None*, *var_name=None*, *vert_code=None*, *year=None*, *ts_type=None*)

Get filename for saving following AeroCom conventions

Parameters

- **data_id** (*str, optional*) – data ID used in output filename. Defaults to None, in which case *data_id* is used.
- **var_name** (*str, optional*) – variable name used in output filename. Defaults to None, in which case *var_name* is used.
- **vert_code** (*str, optional*) – vertical code used in output filename (e.g. Surface, Column, ModelLevel). Defaults to None, in which case assigned value in *metadata* is used.
- **year** (*str, optional*) – year to be used in filename. If None, then it is attempted to be inferred from values in time dimension.
- **ts_type** (*str, optional*) – frequency string to be used in filename. If None, then *ts_type* is used.

Raises

ValueError – if vertical code is not provided and cannot be inferred or if year is not provided and data is not single year. Note that if year is provided, then no sanity checking is done against time dimension.

Returns

output filename following AeroCom Phase 3 conventions.

Return type

str

property altitude_access

apply_region_mask(*region_id*, *thresh_coast=0.5*, *inplace=False*)

Apply a masked region filter

area_weighted_mean()

Get area weighted mean

property area_weights

Area weights of lat / lon grid

property base_year

Base year of time dimension

Note: Changing this attribute will update the time-dimension.

calc_area_weights()

Calculate area weights for grid

change_base_year(*new_year*, *inplace=True*)

Changes base year of time dimension

Relevant, e.g. for climatological analyses.

Note: This method does not account for offsets arising from leap years (affecting daily or higher resolution data). It is thus recommended to use this method with care. E.g. if you use this method on a 2016 daily data object, containing a calendar that supports leap years, you'll end up with 366 time stamps also in the new data object.

Parameters

- **new_year** (*int*) – new base year (can also be other than integer if it is convertible)
- **inplace** (*bool*) – if True, modify this object, else, use a copy

Returns
modified data object

Return type
GriddedData

check_altitude_access()

Checks if altitude levels can be accessed

Returns
True, if altitude access is provided, else False

Return type
bool

check_dimcoords_tseries() → None

Check order of dimension coordinates for time series retrieval

For computation of time series at certain lon / lat coordinates, the data dimensions have to be in a certain order specified by *COORDS_ORDER_TSERIES*.

This method checks the current order (and dimensionality) of data and raises appropriate errors.

Raises

- ***DataDimensionError*** – if dimension of data is not supported (currently, 3D or 4D data is supported)
- **NotImplementedError** – if one of the required coordinates is associated with more than one dimension.
- ***DimensionOrderError*** – if dimensions are not in the right order (in which case *reorder_dimensions_tseries()* may be used to catch the Exception)

check_frequency()

Check if all datapoints are sampled at the same time frequency

check_lon_circular()

Check if latitude and longitude coordinates are circular

check_unit(*try_convert_if_wrong=False*)

Check if unit is correct

collapsed(*coords*, *aggregator*, ***kwargs*)

Collapse cube

Reimplementation of method `iris.cube.Cube.collapsed()`, for details see here

Parameters

- **coords** (*str or list*) – string IDs of coordinate(s) that are to be collapsed (e.g. `["longitude", "latitude"]`)
- **aggregator** (*str or Aggregator or WeightedAggretor*) – the aggregator used. If input is string, it is converted into the corresponding iris Aggregator object, see `str_to_iris()` for valid strings
- ****kwargs** – additional keyword args (e.g. `weights`)

Returns

collapsed data object

Return type

GriddedData

property computed

property concatenated

convert_unit(*new_unit*, *inplace=True*)

Convert unit of data to new unit

Parameters

- **new_unit** (*str or cf_units.Unit*) – new unit of data
- **inplace** (*bool*) – convert in this instance or create a new one

property coord_names

List containing coordinate names

property coords_order

Array containing the order of coordinates

copy()

Copy this data object

copy_coords(*other*, *inplace=True*)

Copy all coordinates from other data object

Requires the underlying data to be the same shape.

> **Warning:** This operation will delete all existing coordinates and auxiliary coordinates and will then copy the ones from the input data object. No checks of any kind will be performed

Parameters

- **other** (*GriddedData or Cube*) – other data object (needs to be same shape as this object)
- **inplace** (*bool*) – if True, then this object will be modified and returned, else a copy.

Returns

data object containing coordinates from other object

Return type

GriddedData

crop(*lon_range=None*, *lat_range=None*, *time_range=None*, *region=None*)

High level function that applies cropping along multiple axes

---

Note: 1. For cropping of longitudes and latitudes, the method `iris.cube.Cube.intersection()` is used since it automatically accepts and understands longitude input based on definition 0 <= lon <= 360 as well as for -180 <= lon <= 180 2. Time extraction may be provided directly as index or in form of `pandas.Timestamp` objects.

Parameters

- **lon_range** (`tuple`, optional) – 2-element tuple containing longitude range for cropping. If None, the longitude axis remains unchanged. Example input to crop around meridian: *lon_range=(-30, 30)*
- **lat_range** (`tuple`, optional) – 2-element tuple containing latitude range for cropping. If None, the latitude axis remains unchanged
- **time_range** (`tuple`, optional) – 2-element tuple containing time range for cropping. Allowed data types for specifying the times are

  1. a combination of 2 `pandas.Timestamp` instances or
  2. a combination of two strings that can be directly converted into `pandas.Timestamp` instances (e.g. *time_range=("2010-1-1", "2012-1-1")*) or
  3. directly a combination of indices (`int`).

  If None, the time axis remains unchanged.
- **region** (`str` or `Region`, optional) – string ID of pyaerocom default region or directly an instance of the `Region` object. May be used instead of `lon_range` and `lat_range`, if these are unspecified.

Returns

new data object containing cropped grid

Return type
GriddedData

property cube

Instance of underlying cube object

property data

Data array (n-dimensional numpy array)

Note: This is a pointer to the data object of the underlying iris.Cube instance and will load the data into memory. Thus, in case of large datasets, this may lead to a memory error

---

property data_id

ID of data object (e.g. model run ID, obsnetwork ID)

Note: This attribute was formerly named `name` which is alse the corresponding attribute name in *metadata*

---

property data_revision

Revision string from file Revision.txt in the main data directory

delete_all_coords(*inplace=True*)

Deletes all coordinates (dimension + auxiliary) in this object

delete_aux_vars()

Delete auxiliary variables and iris AuxFactories

property delta_t

Array containing timedelta values for each time stamp

property dimcoord_names

List containing coordinate names

estimate_value_range_from_data(*extend_percent=5*)

Estimate lower and upper end of value range for these data

Parameters
extend_percent (*int*) – percentage specifying to which extend min and max values are to be extended to estimate the value range. Defaults to 5.

Returns

- *float* – lower end of estimated value range
- *float* – upper end of estimated value range

extract(*constraint*, *inplace=False*)

Extract subset

Parameters
constraint (*iris.Constraint*) – constraint that is to be applied

Returns
new data object containing cropped data

Return type
GriddedData

extract_surface_level()

Extract surface level from 4D field

filter_altitude(*alt_range=None*)

Currently dummy method that makes life easier in `Filter`

Returns
current instance

Return type
GriddedData

filter_region(*region_id*, *inplace=False*, ***kwargs*)

Filter region based on ID

This works both for rectangular regions and mask regions

Parameters

- **region_id** (*str*) – name of region
- **inplace** (*bool*) – if True, the current data object is modified, else a new object is returned
- ****kwargs** – additional keyword args passed to *apply_region_mask()* if input region is a mask.

Returns
filtered data object

Return type
GriddedData

find_closest_index(***dimcoord_vals*)

Find the closest indices for dimension coordinate values

property from_files

List of file paths from which this data object was created

get_altitude(***coords*)

Extract (or try to compute) altitude values at input coordinates

get_area_weighted_timeseries(*region=None*)

Helper method to extract area weighted mean timeseries

Parameters
region – optional, name of AeroCom default region for which the mean is to be calculated (e.g. EUROPE)

Returns
station data containing area weighted mean

Return type
StationData

property grid

Underlying grid data object

property has_data

True if sum of shape of underlying Cube instance is > 0, else False

property has_latlon_dims

Boolean specifying whether data has latitude and longitude dimensions

property has_time_dim

Boolean specifying whether data has latitude and longitude dimensions

infer_ts_type()

Try to infer sampling frequency from time dimension data

Returns
ts_type that was inferred (is assigned to metadata too)

Return type
str

Raises
DataDimensionError – if data object does not contain a time dimension

interpolate(*sample_points=None*, *scheme='nearest'*, *collapse_scalar=True*, ***coords*)

Interpolate cube at certain discrete points

Reimplementation of method `iris.cube.Cube.interpolate()`, for details see here

Note: The input coordinates may also be provided using the input arg ***coords* which provides a more intuitive option (e.g. input `(sample_points=[("longitude", [10, 20]), ("latitude", [1, 2])])` is the same as input `(longitude=[10, 20], latitude=[1,2])`

Parameters

- **sample_points** (*list*) – sequence of coordinate pairs over which to interpolate
- **scheme** (*str or iris interpolator object*) – interpolation scheme, pyaerocom default is nearest. If input is string, it is converted into the corresponding iris Interpolator object, see `str_to_iris()` for valid strings
- **collapse_scalar** (*bool*) – Whether to collapse the dimension of scalar sample points in the resulting cube. Default is True.
- ****coords** – additional keyword args that may be used to provide the interpolation coordinates in an easier way than using the `Cube` argument *sample_points*. May also be a combination of both.

Returns
new data object containing interpolated data

Return type
GriddedData

Examples

```
>>> from pyaerocom import GriddedData
>>> data = GriddedData()
>>> data._init_testdata_default()
>>> itp = data.interpolate([("longitude", (10)),
...                         ("latitude" , (35))])
>>> print(itp.shape)
(365, 1, 1)
```

intersection(**args*, ***kwargs*)

Ectract subset using `iris.cube.Cube.intersection()`

See here for details related to method and input parameters.

Note: Only works if underlying grid data type is `iris.cube.Cube`

Parameters

- ***args** – non-keyword args
- ****kwargs** – keyword args

Returns
new data object containing cropped data

Return type
GriddedData

property is_climatology

property is_masked

Flag specifying whether data is masked or not

Note: This method only works if the data is loaded.

isel(***kwargs*)

property lat_res

load_input(*input*, *var_name=None*, *perform_fmt_checks=None*)

Import input as cube

Parameters

- **input** (`str:` or `Cube`) – data input. Can be a single .nc file or a preloaded iris Cube.
- **var_name** (`str`, optional) – variable name that is extracted if *input* is a file path . Irrelevant if *input* is preloaded Cube
- **perform_fmt_checks** (*bool, optional*) – perform formatting checks based on information in filenames. Only relevant if input is a file

property lon_res

property long_name

Long name of variable

max()

Maximum value

Return type
float

mean(*areaweighted=True*)

Mean value of data array

Note: Corresponds to numerical mean of underlying N-dimensional numpy array. Does not consider area-weights or any other advanced averaging.

---

mean_at_coords(*latitude=None*, *longitude=None*, *time_resample_kwargs=None*, ***kwargs*)

Compute mean value at all input locations

Parameters

- **latitude** (*1D list or similar*) – list of latitude coordinates of coordinate locations. If None, please provided coords in iris style as list of (lat, lon) tuples via *coords* (handled via arg kwargs)
- **longitude** (*1D list or similar*) – list of longitude coordinates of coordinate locations. If None, please provided coords in iris style as list of (lat, lon) tuples via *coords* (handled via arg kwargs)
- **time_resample_kwargs** (*dict, optional*) – time resampling arguments passed to `StationData.resample_time()`
- ****kwargs** – additional keyword args passed to *to_time_series()*

Returns

mean value at coordinates over all times available in this object

Return type

float

property metadata

min()

Minimum value

Return type

float

property name

ID of model to which data belongs

nanmax()

Maximum value excluding NaNs

Return type

float

nanmin()

Minimum value excluding NaNs

Return type

float

property ndim

Number of dimensions

property plot_settings

`Variable` instance that contains plot settings

The settings can be specified in the variables.ini file based on the unique var_name, see e.g. here

If no default settings can be found for this variable, all parameters will be initiated with None, in which case the Aerocom plot method uses

quickplot_map(*time_idx=0*, *xlim=(-180, 180)*, *ylim=(-90, 90)*, *add_mean=True*, ***kwargs*)

Make a quick plot onto a map

Parameters

- **time_idx** (*int*) – index in time to be plotted
- **xlim** (*tuple*) – 2-element tuple specifying plotted longitude range
- **ylim** (*tuple*) – 2-element tuple specifying plotted latitude range
- **add_mean** (*bool*) – if True, the mean value over the region and period is inserted
- ****kwargs** – additional keyword arguments passed to pyaerocom.quickplot.plot_map()

Returns
matplotlib figure instance containing plot

Return type
fig

property reader

Instance of reader class from which this object was created

Note: Currently only supports instances of ReadGridded.

---

register_var_glob(*delete_existing=True*)

regrid(*other=None*, *lat_res_deg=None*, *lon_res_deg=None*, *scheme='areaweighted'*, ***kwargs*)

Regrid this grid to grid resolution of other grid

Parameters

- **other** (*GriddedData or Cube, optional*) – other data object to regrid to. If None, then input args *lat_res* and *lon_res* are used to regrid.
- **lat_res_deg** (*float or int, optional*) – latitude resolution in degrees (is only used if input arg *other* is None)
- **lon_res_deg** (*float or int, optional*) – longitude resolution in degrees (is only used if input arg *other* is None)
- **scheme** (*str*) – regridding scheme (e.g. linear, neirest, areaweighted)

Returns
regridded data object (new instance, this object remains unchanged)

Return type
GriddedData

remove_outliers(*low=None*, *high=None*, *inplace=True*)

Remove outliers from data

Parameters

- **low** (*float*) – lower end of valid range for input variable. If None, then the corresponding value from the default settings for this variable are used (cf. minimum attribute of available variables)

- **high** (*float*) – upper end of valid range for input variable. If None, then the corresponding value from the default settings for this variable are used (cf. maximum attribute of available variables)
- **inplace** (*bool*) – if True, this object is modified, else outliers are removed in a copy of this object

Returns
modified data object

Return type
GriddedData

reorder_dimensions_tseries() → None

Transpose dimensions of data such that *to_time_series()* works

Raises

- ***DataDimensionError*** – if not all needed coordinates are available
- **NotImplementedError** – if one of the required coordinates is associated with more than one dimension.

resample_time(*to_ts_type*, *how=None*, *min_num_obs=None*, *use_iris=False*)

Resample time to input resolution

Parameters

- **to_ts_type** (*str*) – either of the supported temporal resolutions (cf. IRIS_AGGREGATORS in helpers, e.g. "monthly")
- **how** (*str*) – string specifying how the data is to be aggregated, default is mean
- **min_num_obs** (*dict or int, optional*) – integer or nested dictionary specifying minimum number of observations required to resample from higher to lower frequency. For instance, if *input_data* is hourly and *to_ts_type* is monthly, you may specify something like:

```
min_num_obs =
    {'monthly'  :   {'daily'  : 7},
     'daily'    :   {'hourly' : 6}}
```

to require at least 6 hours per day and 7 days per month.

- **use_iris** (*bool*) – option to use resampling scheme from iris library rather than xarray.

Returns
new data object containing downscaled data

Return type
GriddedData

Raises
TemporalResolutionError – if input resolution is not provided, or if it is higher temporal resolution than this object

search_other(*var_name*)

Searches data for another variable

The search is constrained to the time period spanned by this object and it is attempted to load the same frequency. Uses *reader* (instance of ReadGridded to search for the other variable data).

Parameters
var_name (*str*) – variable to be searched

Raises
VariableNotFoundError – if data for input variable cannot be found.

Returns
input variable data

Return type
GriddedData

sel(*use_neirest=True*, ***dimcoord_vals*)

Select subset by dimension names

Note: This is a BETA version, please use with care

Parameters
****dimcoord_vals** – key / value pairs specifying coordinate values to be extracted

Returns
subset data object

Return type
GriddedData

property shape

short_str()

Short string representation

split_years(*years=None*)

Generator to split data object into individual years

Note: This is a generator method and thus should be looped over

Parameters
years (*list, optional*) – List of years that should be excluded. If None, it uses output from *years_avail()*.

Yields
GriddedData – single year data object

property standard_name

Standard name of variable

property start

Start time of dataset as datetime64 object

std()

Standard deviation of values

property stop

Start time of dataset as datetime64 object

property suppl_info

time_stamps()

Convert time stamps into list of numpy datetime64 objects

The conversion is done using method `cfunit_to_datetime64()`

Returns
list containing all time stamps as datetime64 objects

Return type
list

to_netcdf(*out_dir*, *savename=None*, ***kwargs*)

Save as NetCDF file

Parameters

- **out_dir** (*str*) – output direcory (must exist)
- **savename** (*str, optional*) – name of file. If None, *aerocom_savename()* is used which is generated automatically and may be modified via ***kwargs*
- ****kwargs** – keywords for name

Returns
list of output files created

Return type
list

to_time_series(*sample_points=None*, *scheme='nearest'*, *vert_scheme=None*, *add_meta=None*, *use_iris=False*, ***coords*)

Extract time-series for provided input coordinates (lon, lat)

Extract time series for each lon / lat coordinate in this cube or at predefined sample points (e.g. station data). If sample points are provided, the cube is interpolated first onto the sample points.

Parameters

- **sample_points** (*list*) – coordinates (e.g. lon / lat) at which time series is supposed to be retrieved
- **scheme** (*str or iris interpolator object*) – interpolation scheme (for details, see *interpolate()*)
- **vert_scheme** (*str*) – string specifying how to treat vertical coordinates. This is only relevant for data that contains vertical levels. It will be ignored otherwise. Note that if the input coordinate specifications contain altitude information, this parameter will be set automatically to 'altitude'. Allowed inputs are all data collapse schemes that are supported by *pyaerocom.helpers.str_to_iris()* (e.g. *mean, median, sum*). Further valid schemes are *altitude, surface, profile*. If not other specified and if *altitude* coordinates are provided via sample_points (or **coords parameters) then, vert_scheme will be set to *altitude*. Else, *profile* is used.
- **add_meta** (*dict, optional*) – dictionary specifying additional metadata for individual input coordinates. Keys are meta attribute names (e.g. station_name) and corresponding values are lists (with length of input coords) or single entries that are supposed to be assigned to each station. E.g. *add_meta=dict(station_name=[<list_of_station_names>])*.
- ****coords** – additional keyword args that may be used to provide the interpolation coordinates (for details, see *interpolate()*)

Returns
list of result dictionaries for each coordinate. Dictionary keys are: `longitude, latitude, var_name`

Return type
list

`to_xarray()`

Convert this object to an xarray.DataArray

Return type
DataArray

`transpose(new_order)`

Re-order data dimensions in object

Wrapper for `iris.cube.Cube.transpose()`

Note: Changes THIS object (i.e. no new instance of *GriddedData* will be created)

Parameters
order (*list*) – new index order

`property ts_type`

Temporal resolution of data

`property unit`

Unit of data

`property unit_ok`

Boolean specifying if variable unit is AeroCom default

`property units`

Unit of data

`update_meta(**kwargs)`

Update metadata dictionary

Parameters
****kwargs** – metadata to be added to *metadata*.

`property var_info`

Print information about variable

`property var_name`

Name of variable

`property var_name_aerocom`

AeroCom variable name

`property vert_code`

Vertical code of data (e.g. Column, Surface, ModelLevel)

`years_avail()`

Generate list of years that are available in this dataset

Return type
list

4.2.2 Ungridded data

class pyaerocom.ungriddeddata.**UngriddedData**(*num_points=None*, *add_cols=None*)

Class representing point-cloud data (ungridded)

The data is organised in a 2-dimensional numpy array where the first index (rows) axis corresponds to individual measurements (i.e. one timestamp of one variable) and along the second dimension (containing 11 columns) the actual values are stored (in column 6) along with additional information, such as metadata index (can be used as key in *metadata* to access additional information related to this measurement), timestamp, latitude, longitude, altitude of instrument, variable index and, in case of 3D data (e.g. LIDAR profiles), also the altitude corresponding to the data value.

Note: That said, let's look at two examples.

Example 1: Suppose you load 3 variables from 5 files, each of which contains 30 timestamps. This corresponds to a total of 3*5*30=450 data points and hence, the shape of the underlying numpy array will be 450x11.

Example 2: 3 variables, 5 files, 30 timestamps, but each variable is height resolved, containing 100 altitudes => 3*5*30*100=4500 data points, thus, the final shape will be 4500x11.

---

metadata

dictionary containing meta information about the data. Keys are floating point numbers corresponding to each station, values are corresponding dictionaries containing station information.

Type
dict

meta_idx

dictionary containing index mapping for each station and variable. Keys correspond to metadata key (float -> station, see *metadata*) and values are dictionaries containing keys specifying variable name and corresponding values are arrays or lists, specifying indices (rows) of these station / variable information in `_data`. Note: this information is redunant and is there to accelarate station data extraction since the data index matches for a given metadata block do not need to be searched in the underlying numpy array.

Type
dict

var_idx

mapping of variable name (keys, e.g. od550aer) to numerical variable index of this variable in data numpy array (in column specified by `_VARINDEX`)

Type
dict

Parameters

- **num_points** (int, optional) – inital number of total datapoints (number of rows in 2D dataarray)
- **add_cols** (list, optional) – list of additional index column names of 2D datarray.

ALLOWED_VERT_COORD_TYPES = ['altitude']

STANDARD_META_KEYS = ['filename', 'station_id', 'station_name', 'instrument_name', 'PI', 'country', 'country_code', 'ts_type', 'latitude', 'longitude', 'altitude', 'data_id', 'dataset_name', 'data_product', 'data_version', 'data_level', 'framework', 'instr_vert_loc', 'revision_date', 'website', 'ts_type_src', 'stat_merge_pref_attr']

add_chunk(*size=None*)

Extend the size of the data array

Parameters
size (int, optional) – number of additional rows. If None (default) or smaller than minimum chunksize specified in attribute _CHUNKSIZE, then the latter is used.

add_station_data(*stat*, *meta_idx=None*, *data_idx=None*, *check_index=False*)

all_datapoints_var(*var_name*)

Get array of all data values of input variable

Parameters
var_name (*str*) – variable name

Returns
1-d numpy array containing all values of this variable

Return type
ndarray

Raises
AttributeError – if variable name is not available

property altitude

Altitudes of stations

append(*other*)

Append other instance of *UngriddedData* to this object

Note: Calls `merge(other, new_obj=False)`()

Parameters
other (UngriddedData) – other data object

Returns
merged data object

Return type
UngriddedData

Raises
ValueError – if input object is not an instance of *UngriddedData*

apply_filters(*var_outlier_ranges=None*, ***filter_attributes*)

Extended filtering method

Combines *filter_by_meta()* and adds option to also remove outliers (keyword *remove_outliers*), set flagged data points to NaN (keyword *set_flags_nan*) and to extract individual variables (keyword *var_name*).

Parameters

- **var_outlier_ranges** (*dict, optional*) – dictionary specifying custom outlier ranges for individual variables.

- ****filter_attributes** (*dict*) – filters that are supposed to be applied to the data. To remove outliers, use keyword *remove_outliers*, to set flagged values to NaN, use keyword *set_flags_nan*, to extract single or multiple variables, use keyword *var_name*. Further filter keys are assumed to be metadata specific and are passed to *filter_by_meta()*.

Returns
filtered data object

Return type
UngriddedData

apply_region_mask(*region_id=None*)

TODO : Write documentations

Parameters
region_id (*str or list (of strings)*) – ID of region or IDs of multiple regions to be combined

property available_meta_keys

List of all available metadata keys

Note: This is a list of all metadata keys that exist in this dataset, but it does not mean that all of the keys are registered in all metadata blocks, especially if the data is merged from different sources with different metadata availability

change_var_idx(*var_name*, *new_idx*)

Change index that is assigned to variable

Each variable in this object has assigned a unique index that is stored in the dictionary *var_idx* and which is used internally to access data from a certain variable from the data array `_data` (the indices are stored in the data column specified by `_VARINDEX`, cf. class header).

This index thus needs to be unique for each variable and hence, may need to be updated, when two instances of *UngriddedData* are merged (cf. *merge()*).

And the latter is exactrly what this function does.

Parameters

- **var_name** (*str*) – name of variable
- **new_idx** (*int*) – new index of variable

Raises
ValueError – if input `new_idx` already exist in this object as a variable index

check_convert_var_units(*var_name*, *to_unit=None*, *inplace=True*)

check_set_country()

CHecks all metadata entries for availability of country information

Metadata blocks that are missing country entry will be updated based on country inferred from corresponding lat / lon coordinate. Uses *pyaerocom.geodesy.get_country_info_coords()* (library reverse-geocode) to retrieve countries. This may be errouneous close to country borders as it uses eucledian distance based on a list of known locations.

Note: Metadata blocks that do not contain latitude and longitude entries are skipped.

Returns

- *list* – metadata entries where country was added
- *list* – corresponding countries that were inferred from lat / lon

check_unit(*var_name*, *unit=None*)

Check if variable unit corresponds to AeroCom unit

Parameters

- **var_name** (*str*) – variable name for which unit is to be checked
- **unit** (str, optional) – unit to be checked, if None, AeroCom default unit is used

Raises

MetaDataError – if unit information is not accessible for input variable name

clear_meta_no_data(*inplace=True*)

Remove all metadata blocks that do not have data associated with it

Parameters

inplace (*bool*) – if True, the changes are applied to this instance directly, else to a copy

Returns

cleaned up data object

Return type

UngriddedData

Raises

DataCoverageError – if filtering results in empty data object

code_lat_lon_in_float()

method to code lat and lon in a single number so that we can use np.unique to determine single locations

colocate_vardata(*var1*, *data_id1=None*, *var2=None*, *data_id2=None*, *other=None*, ***kwargs*)

property contains_datasets

List of all datasets in this object

property contains_instruments

List of all instruments in this object

property contains_vars: list[str]

List of all variables in this dataset

copy()

Make a copy of this object

Returns

copy of this object

Return type

UngriddedData

Raises

MemoryError – if copy is too big to fit into memory together with existing instance

property countries_available

Alphabetically sorted list of country names available

decode_lat_lon_from_float()

method to decode lat and lon from a single number calculated by code_lat_lon_in_float

empty_trash()

Set all values in trash column to NaN

extract_dataset(*data_id*)

Extract single dataset into new instance of *UngriddedData*

Calls *filter_by_meta()*.

Parameters
data_id (*str*) – ID of dataset

Returns
new instance of ungridded data containing only data from specified input network

Return type
UngriddedData

extract_var(*var_name*, *check_index=True*)

Split this object into single-var UngriddedData objects

Parameters

- **var_name** (*str*) – name of variable that is supposed to be extracted
- **check_index** (*Bool*) – Call _check_index() in the new data object.

Returns
new data object containing only input variable data

Return type
UngriddedData

extract_vars(*var_names*, *check_index=True*)

Extract multiple variables from dataset

Loops over input variable names and calls *extract_var()* to retrieve single variable UngriddedData objects for each variable and then merges all of these into one object

Parameters

- **var_names** (*list or str*) – list of variables to be extracted
- **check_index** (*Bool*) – Call _check_index() in the new data object.

Returns
new data object containing input variables

Return type
UngriddedData

Raises
VarNotAvailableError – if one of the input variables is not available in this data object

filter_altitude(*alt_range*)

Filter altitude range

Parameters
alt_range (*list or tuple*) – 2-element list specifying altitude range to be filtered in m

Returns
filtered data object

Return type
UngriddedData

filter_by_meta(*negate=None*, ***filter_attributes*)

Flexible method to filter these data based on input meta specs

Parameters

- **negate** (*list or str, optional*) – specified meta key(s) provided via *filter_attributes* that are supposed to be treated as 'not valid'. E.g. if *station_name="bad_site"* is input in *filter_attributes* and if *station_name* is listed in *negate*, then all metadata blocks containing "bad_site" as station_name will be excluded in output data object.
- ****filter_attributes** – valid meta keywords that are supposed to be filtered and the corresponding filter values (or value ranges) Only valid meta keywords are considered (e.g. data_id, longitude, latitude, altitude, ts_type)

Returns
filtered ungridded data object

Return type
UngriddedData

Raises

- **NotImplementedError** – if attempt variables are supposed to be filtered (not yet possible)
- **IOError** – if any of the input keys are not valid meta key

Example

```
>>> import pyaerocom as pya
>>> r = pya.io.ReadUngridded(['AeronetSunV2Lev2.daily',
                              'AeronetSunV3Lev2.daily'], 'od550aer')
>>> data = r.read()
>>> data_filtered = data.filter_by_meta(data_id='AeronetSunV2Lev2.daily',
...                                     longitude=[-30, 30],
...                                     latitude=[20, 70],
...                                     altitude=[0, 1000])
```

filter_region(*region_id*, *check_mask=True*, *check_country_meta=False*, ***kwargs*)

Filter object by a certain region

Parameters

- **region_id** (*str*) – name of region (must be valid AeroCom region name or HTAP region)
- **check_mask** (*bool*) – if True and region_id a valid name for a binary mask, then the filtering is done based on that binary mask.
- **check_country_meta** (*bool*) – if True, then the input region_id is first checked against available country names in metadata. If that fails, it is assumed that this regions is either a valid name for registered rectangular regions or for available binary masks.
- ****kwargs** – currently not used in method (makes usage in higher level classes such as `Filter` easier as other data objects have the same method with possibly other input possibilities)

Returns
filtered data object (containing only stations that fall into input region)

Return type
UngriddedData

find_common_data_points(*other*, *var_name*, *sampling_freq='daily'*)

find_common_stations(*other:* UngriddedData, *check_vars_available=None*, *check_coordinates:* *bool* = *True*, *max_diff_coords_km:* *float* = *0.1*) → dict

Search common stations between two UngriddedData objects

This method loops over all stations that are stored within this object (using *metadata*) and checks if the corresponding station exists in a second instance of *UngriddedData* that is provided. The check is performed on basis of the station name, and optionally, if desired, for each station name match, the lon lat coordinates can be compared within a certain radius (defaul 0.1 km).

Note: This is a beta version and thus, to be treated with care.

Parameters

- **other** (UngriddedData) – other object of ungridded data
- **check_vars_available** (list (or similar), optional) – list of variables that need to be available in stations of both datasets
- **check_coordinates** (*bool*) – if True, check that lon and lat coordinates of station candidates match within a certain range, specified by input parameter max_diff_coords_km

Returns
dictionary where keys are meta_indices of the common station in this object and corresponding values are meta indices of the station in the other object

Return type
dict

find_station_meta_indices(*station_name_or_pattern*, *allow_wildcards=True*)

Find indices of all metadata blocks matching input station name

You may also use wildcard pattern as input (e.g. *Potenza*)

Parameters

- **station_pattern** (*str*) – station name or wildcard pattern
- **allow_wildcards** (*bool*) – if True, input station_pattern will be used as wildcard pattern and all matches are returned.

Returns
list containing all metadata indices that match the input station name or pattern

Return type
list

Raises
StationNotFoundError – if no such station exists in this data object

property first_meta_idx

static from_cache(*data_dir*, *file_name*)

Load pickled instance of *UngriddedData*

Parameters

- **data_dir** (*str*) – directory where pickled object is stored
- **file_name** (*str*) – file name of pickled object (needs to end with pkl)

Raises
ValueError – if loading failed

Returns
loaded UngriddedData object. If this method is called from an instance of *UngriddedData*, this instance remains unchanged. You may merge the returned reloaded instance using *merge()*.

Return type
UngriddedData

static from_station_data(*stats*, *add_meta_keys=None*)

Create UngriddedData from input station data object(s)

Parameters

- **stats** (*list or* StationData) – input data object(s)
- **add_meta_keys** (*list, optional*) – list of metadata keys that are supposed to be imported from the input *StationData* objects, in addition to the default metadata retrieved via `StationData.get_meta()`.

Raises
ValueError – if any of the input data objects is not an instance of `StationData`.

Returns
ungridded data object created from input station data objects

Return type
UngriddedData

get_variable_data(*variables*, *start=None*, *stop=None*, *ts_type=None*, ***kwargs*)

Extract all data points of a certain variable

Parameters
vars_to_extract (str or list) – all variables that are supposed to be accessed

property has_flag_data

Boolean specifying whether this object contains flag data

property index

property is_empty

Boolean specifying whether this object contains data or not

property is_filtered

Boolean specifying whether this data object has been filtered

Note: Details about applied filtering can be found in `filter_hist`

property is_vertical_profile

Boolean specifying whether is vertical profile

last_filter_applied()

Returns the last filter that was applied to this dataset

To see all filters, check out `filter_hist`

property last_meta_idx

Index of last metadata block

property latitude

Latitudes of stations

property longitude

Longitudes of stations

merge(*other*, *new_obj=True*)

Merge another data object with this one

Parameters

- **other** (UngriddedData) – other data object
- **new_obj** (*bool*) – if True, this object remains unchanged and the merged data objects are returned in a new instance of *UngriddedData*. If False, then this object is modified

Returns
merged data object

Return type
UngriddedData

Raises
ValueError – if input object is not an instance of *UngriddedData*

merge_common_meta(*ignore_keys=None*)

Merge all meta entries that are the same

Note: If there is an overlap in time between the data, the blocks are not merged

Parameters
ignore_keys (*list*) – list containing meta keys that are supposed to be ignored

Returns
merged data object

Return type
UngriddedData

property nonunique_station_names

List of station names that occur more than once in metadata

num_obs_var_valid(*var_name*)

Number of valid observations of variable in this dataset

Parameters
var_name (*str*) – name of variable

Returns
number of valid observations (all values that are not NaN)

Return type
int

plot_station_coordinates(*var_name=None*, *start=None*, *stop=None*, *ts_type=None*, *color='r'*, *marker='o'*, *markersize=8*, *fontsize_base=10*, *legend=True*, *add_title=True*, ***kwargs*)

Plot station coordinates on a map

All input parameters are optional and may be used to add constraints related to which stations are plotted. Default is all stations of all times.

Parameters

- **var_name** (str, optional) – name of variable to be retrieved
- **start** – start time (optional)
- **stop** – stop time (optional). If start time is provided and stop time not, then only the corresponding year inferred from start time will be considered
- **ts_type** (str, optional) – temporal resolution
- **color** (*str*) – color of stations on map
- **marker** (*str*) – marker type of stations
- **markersize** (*int*) – size of station markers
- **fontsize_base** (*int*) – basic fontsize
- **legend** (*bool*) – if True, legend is added
- **add_title** (*bool*) – if True, title will be added
- ****kwargs** – Addifional keyword args passed to `pyaerocom.plot.plot_coordinates()`

Returns
matplotlib axes instance

Return type
axes

plot_station_timeseries(*station_name*, *var_name*, *start=None*, *stop=None*, *ts_type=None*, *insert_nans=True*, *ax=None*, ***kwargs*)

Plot time series of station and variable

Parameters

- **station_name** (str or int) – station name or index of station in metadata dict
- **var_name** (*str*) – name of variable to be retrieved
- **start** – start time (optional)
- **stop** – stop time (optional). If start time is provided and stop time not, then only the corresponding year inferred from start time will be considered
- **ts_type** (str, optional) – temporal resolution
- ****kwargs** – Addifional keyword args passed to method `pandas.Series.plot()`

Returns
matplotlib axes instance

Return type
axes

remove_outliers(*var_name*, *inplace=False*, *low=None*, *high=None*, *unit_ref=None*, *move_to_trash=True*)

Method that can be used to remove outliers from data

Parameters

- **var_name** (*str*) – variable name
- **inplace** (*bool*) – if True, the outliers will be removed in this object, otherwise a new oject will be created and returned
- **low** (*float*) – lower end of valid range for input variable. If None, then the corresponding value from the default settings for this variable are used (cf. minimum attribute of available variables)
- **high** (*float*) – upper end of valid range for input variable. If None, then the corresponding value from the default settings for this variable are used (cf. maximum attribute of available variables)
- **unit_ref** (*str*) – reference unit for assessment of input outlier ranges: all data needs to be in that unit, else an Exception will be raised
- **move_to_trash** (*bool*) – if True, then all detected outliers will be moved to the trash column of this data object (i.e. column no. specified at `UngriddedData._TRASHINDEX`).

Returns

ungridded data object that has all outliers for this variable removed.

Return type

UngriddedData

Raises

ValueError – if input `move_to_trash` is True and in case for some of the measurements there is already data in the trash.

save_as(*file_name*, *save_dir*)

Save this object to disk

Note: So far, only storage as pickled object via *CacheHandlerUngridded* is supported, so input file_name must end with .pkl

---

Parameters

- **file_name** (*str*) – name of output file
- **save_dir** (*str*) – name of output directory

Returns

file path

Return type

str

set_flags_nan(*inplace=False*)

Set all flagged datapoints to NaN

Parameters

inplace (*bool*) – if True, the flagged datapoints will be set to NaN in this object, otherwise a new oject will be created and returned

Returns
data object that has all flagged data values set to NaN

Return type
UngriddedData

Raises
AttributeError – if no flags are assigned

property shape

Shape of data array

property station_coordinates

dictionary with station coordinates

Returns
dictionary containing station coordinates (latitude, longitude, altitude -> values) for all stations (keys) where these parameters are accessible.

Return type
dict

property station_name

Latitudes of data

property time

Time dimension of data

to_station_data(*meta_idx*, *vars_to_convert=None*, *start=None*, *stop=None*, *freq=None*, *ts_type_preferred=None*, *merge_if_multi=True*, *merge_pref_attr=None*, *merge_sort_by_largest=True*, *insert_nans=False*, *allow_wildcards_station_name=True*, *add_meta_keys=None*, *resample_how=None*, *min_num_obs=None*)

Convert data from one station to `StationData`

Parameters

- **meta_idx** (*float*) – index of station or name of station.
- **vars_to_convert** (`list` or `str`, optional) – variables that are supposed to be converted. If None, use all variables that are available for this station
- **start** – start time, optional (if not None, input must be convertible into pandas.Timestamp)
- **stop** – stop time, optional (if not None, input must be convertible into pandas.Timestamp)
- **freq** (*str*) – pandas frequency string (e.g. 'D' for daily, 'M' for month end) or valid pyaerocom ts_type
- **merge_if_multi** (*bool*) – if True and if data request results in multiple instances of StationData objects, then these are attempted to be merged into one `StationData` object using `merge_station_data()`
- **merge_pref_attr** – only relevant for merging of multiple matches: preferred attribute that is used to sort the individual StationData objects by relevance. Needs to be available in each of the individual StationData objects. For details cf. `pref_attr` in docstring of `merge_station_data()`. Example could be *revision_date*. If None, then the stations will be sorted based on the number of available data points (if `merge_sort_by_largest` is True, which is default).
- **merge_sort_by_largest** (*bool*) – only relevant for merging of multiple matches: cf. prev. attr. and docstring of `merge_station_data()` method.

- **insert_nans** (*bool*) – if True, then the retrieved `StationData` objects are filled with NaNs
- **allow_wildcards_station_name** (*bool*) – if True and if input *meta_idx* is a string (i.e. a station name or pattern), metadata matches will be identified applying wildcard matches between input *meta_idx* and all station names in this object.

Returns

StationData object(s) containing results. list is only returned if input for meta_idx is station name and multiple matches are detected for that station (e.g. data from different instruments), else single instance of StationData. All variable time series are inserted as pandas Series

Return type

StationData or list

to_station_data_all(*vars_to_convert=None*, *start=None*, *stop=None*, *freq=None*, *ts_type_preferred=None*, *by_station_name=True*, *ignore_index=None*, ***kwargs*)

Convert all data to `StationData` objects

Creates one instance of `StationData` for each metadata block in this object.

Parameters

- **vars_to_convert** (`list` or `str`, optional) – variables that are supposed to be converted. If None, use all variables that are available for this station
- **start** – start time, optional (if not None, input must be convertible into pandas.Timestamp)
- **stop** – stop time, optional (if not None, input must be convertible into pandas.Timestamp)
- **freq** (*str*) – pandas frequency string (e.g. 'D' for daily, 'M' for month end) or valid pyaerocom ts_type (e.g. 'hourly', 'monthly').
- **by_station_name** (*bool*) – if True, then iter over unique_station_name (and merge multiple matches if applicable), else, iter over metadata index
- ****kwargs** – additional keyword args passed to *to_station_data()* (e.g. *merge_if_multi, merge_pref_attr, merge_sort_by_largest, insert_nans*)

Returns

4-element dictionary containing following key / value pairs:

- stats: list of `StationData` objects
- station_name: list of corresponding station names
- latitude: list of latitude coordinates
- longitude: list of longitude coordinates

Return type

dict

property unique_station_names

List of unique station names

pyaerocom.ungriddeddata.**reduce_array_closest**(*arr_nominal*, *arr_to_be_reduced*)

4.2.3 Co-located data

class pyaerocom.colocateddata.**ColocatedData**(*, *data: Path | str | DataArray | ndarray | None = None*, ***extra_data: Any*)

Class representing colocated and unified data from two sources

Sources may be instances of `UngriddedData` or `GriddedData` that have been compared to each other.

Note: It is intended that this object can either be instantiated from scratch OR created in and returned by pyaerocom objects / methods that perform colocation. This is particauarly true as pyaerocom will now be expected to read in colocated files created outside of pyaerocom. (Related CAMS2_82 development)

The purpose of this object is not the creation of colocated objects, but solely the analysis of such data as well as I/O features (e.g. save as / read from .nc files, convert to pandas.DataFrame, plot station time series overlays, scatter plots, etc.).

In the current design, such an object comprises 3 or 4 dimensions, where the first dimension (*data_source*, index 0) is ALWAYS length 2 and specifies the two datasets that were co-located (index 0 is obs, index 1 is model). The second dimension is *time* and in case of 3D colocated data the 3rd dimension is *station_name* while for 4D colocated data the 3rd and 4th dimension are latitude and longitude, respectively.

3D colocated data is typically created when a model is colocated with station based ground based observations (cf *pyaerocom.colocation.colocate_gridded_ungridded()*) while 4D colocated data is created when a model is colocated with another model or satellite observations, that cover large parts of Earth's surface (other than discrete lat/lon pairs in the case of ground based station locations).

Parameters

- **data** (*xarray.DataArray or numpy.ndarray or str, optional*) – Colocated data. If str, then it is attempted to be loaded from file. Else, it is assumed that data is numpy array and that all further supplementary inputs (e.g. coords, dims) for the instantiation of `DataArray` is provided via **kwargs.
- ****kwargs** – Additional keyword args that are passed to init of `DataArray` in case input *data* is numpy array.

Raises

`IOError` – if init fails

apply_country_filter(*region_id*, *use_country_code=False*, *inplace=False*)

Apply country filter

Parameters

- **region_id** (*str*) – country name or code.
- **use_country_code** (*bool, optional*) – If True, input value for *country* is evaluated against country codes rather than country names. Defaults to False.
- **inplace** (*bool, optional*) – Apply filter to this object directly or to a copy. The default is False.

Raises

`NotImplementedError` – if data is 4D (i.e. it has latitude and longitude dimensions).

Returns

filtered data object.

Return type
ColocatedData

apply_latlon_filter(*lat_range=None*, *lon_range=None*, *region_id=None*, *inplace=False*)

Apply rectangular latitude/longitude filter

Parameters

- **lat_range** (*list, optional*) – latitude range that is supposed to be applied. If specified, then also lon_range need to be specified, else, region_id is checked against AeroCom default regions (and used if applicable)
- **lon_range** (*list, optional*) – longitude range that is supposed to be applied. If specified, then also lat_range need to be specified, else, region_id is checked against AeroCom default regions (and used if applicable)
- **region_id** (*str*) – name of region to be applied. If provided (i.e. not None) then input args *lat_range* and *lon_range* are ignored
- **inplace** (*bool, optional*) – Apply filter to this object directly or to a copy. The default is False.

Raises
ValueError – if lower latitude bound exceeds upper latitude bound.

Returns
filtered data object

Return type
ColocatedData

apply_region_mask(*region_id*, *inplace=False*)

Apply a binary regions mask filter to data object. Available binary regions IDs can be found at *pyaerocom.const.HTAP_REGIONS*.

Parameters

- **region_id** (*str*) – ID of binary regions.
- **inplace** (*bool, optional*) – If True, the current instance, is modified, else a new instance of *ColocatedData* is created and filtered. The default is False.

Raises
DataCoverageError – if filtering results in empty data object.

Returns
data – Filtered data object.

Return type
ColocatedData

property area_weights

Wrapper for *calc_area_weights()*

calc_area_weights()

Calculate area weights

Note: Only applies to colocated data that has latitude and longitude dimension.

Returns
array containing weights for each datapoint (same shape as *self.data[0]*)

Return type
ndarray

`calc_nmb_array`()

Calculate data array with normalised bias (NMB) values

Returns
NMBs at each coordinate

Return type
DataArray

`calc_spatial_statistics`(*aggr=None*, *use_area_weights=False*, ***kwargs*)

Calculate *spatial* statistics from model and obs data

Spatial statistics is computed by averaging first the time dimension and then, if data is 4D, flattening lat / lon dimensions into new station_name dimension, so that the resulting dimensions are *data_source* and *station_name*. These 2D data are then used to calculate standard statistics using `pyaerocom.stats.stats.calculate_statistics()`.

See also *calc_statistics()* and *calc_temporal_statistics()*.

Parameters

- **aggr** (*str, optional*) – aggreagator to be used, currently only mean and median are supported. Defaults to mean.
- **use_area_weights** (*bool*) – if True and if data is 4D (i.e. has lat and lon dimension), then area weights are applied when caluclating the statistics based on the coordinate cell sizes. Defaults to False.
- ****kwargs** – additional keyword args passed to `pyaerocom.stats.stats.calculate_statistics()`

Returns
dictionary containing statistical parameters

Return type
dict

`calc_statistics`(*use_area_weights=False*, ***kwargs*)

Calculate statistics from model and obs data

Calculate standard statistics for model assessment. This is done by taking all model and obs data points in this object as input for `pyaerocom.stats.stats.calculate_statistics()`. For instance, if the object is 3D with dimensions *data_source* (obs, model), *time* (e.g. 12 monthly values) and *station_name* (e.g. 4 sites), then the input arrays for model and obs into `pyaerocom.stats.stats.calculate_statistics()` will be each of size 12x4.

See also *calc_temporal_statistics()* and *calc_spatial_statistics()*.

Parameters

- **use_area_weights** (*bool*) – if True and if data is 4D (i.e. has lat and lon dimension), then area weights are applied when caluclating the statistics based on the coordinate cell sizes. Defaults to False.
- ****kwargs** – additional keyword args passed to `pyaerocom.stats.stats.calculate_statistics()`

Returns
dictionary containing statistical parameters

Return type
dict

calc_temporal_statistics(*aggr=None*, ***kwargs*)

Calculate *temporal* statistics from model and obs data

Temporal statistics is computed by averaging first the spatial dimension(s) (that is, *station_name* for 3D data, and *latitude* and *longitude* for 4D data), so that only *data_source* and *time* remains as dimensions. These 2D data are then used to calculate standard statistics using `pyaerocom.stats.stats.calculate_statistics()`.

See also *calc_statistics()* and *calc_spatial_statistics()*.

Parameters

- **aggr** (*str, optional*) – aggreagator to be used, currently only mean and median are supported. Defaults to mean.
- ****kwargs** – additional keyword args passed to `pyaerocom.stats.stats.calculate_statistics()`

Returns
dictionary containing statistical parameters

Return type
dict

check_set_countries(*inplace=True*, *assign_to_dim=None*)

Checks if country information is available and assigns if not

If not country information is available, countries will be assigned for each lat / lon coordinate using *pyaerocom.geodesy.get_country_info_coords()*.

Parameters

- **inplace** (*bool, optional*) – If True, modify and return this object, else a copy. The default is True.
- **assign_to_dim** (*str, optional*) – name of dimension to which the country coordinate is assigned. Default is None, in which case station_name is used.

Raises
DataDimensionError – If data is 4D (i.e. if latitude and longitude are othorgonal dimensions)

Returns
data object with countries assigned

Return type
ColocatedData

property coords

Coordinates of data array

copy()

Copy this object

property countries_available

Alphabetically sorted list of country names available

Raises
MetaDataError – if no country information is available

Returns
list of countries available in these data

Return type
list

property country_codes_available

Alphabetically sorted list of country codes available

Raises
MetaDataError – if no country information is available

Returns
list of countries available in these data

Return type
list

data: Path | str | xr.DataArray | np.ndarray | None

property data_source

Coordinate array containing data sources (z-axis)

property dims

Names of dimensions

filter_altitude(*alt_range*, *inplace=False*)

Apply altitude filter

Parameters

- **alt_range** (*list or tuple*) – altitude range to be applied to data (2-element list)
- **inplace** (*bool, optional*) – Apply filter to this object directly or to a copy. The default is False.

Raises
NotImplementedError – If data is 4D, i.e. it contains latitude and longitude dimensions.

Returns
Filtered data object .

Return type
ColocatedData

filter_region(*region_id*, *check_mask=True*, *check_country_meta=False*, *inplace=False*)

Filter object by region

Parameters

- **region_id** (*str*) – ID of region
- **inplace** (*bool*) – if True, the filtering is done directly in this instance, else a new instance is returned
- **check_mask** (*bool*) – if True and region_id a valid name for a binary mask, then the filtering is done based on that binary mask.
- **check_country_meta** (*bool*) – if True, then the input region_id is first checked against available country names in metadata. If that fails, it is assumed that this regions is either a valid name for registered rectangular regions or for available binary masks.

Returns
filtered data object

Return type
ColocatedData

`flatten_latlondim_station_name`()

Stack (flatten) lat / lon dimension into new dimension station_name

Returns
new colocated data object with dimension station_name and lat lon arrays as additional coordinates

Return type
ColocatedData

`from_csv`(*file_path*)

Read data from CSV file

`from_dataframe`(*df*)

Create colocated Data object from dataframe

Note: This is intended to be used as back-conversion from *to_dataframe()* and methods that use the latter (e.g. *to_csv()*).

`get_coords_valid_obs`()

Get latitude / longitude coordinates where obsdata is available

Returns

- *list* – latitute coordinates
- *list* – longitude coordinates

`get_country_codes`()

Get country names and codes for all locations contained in these data

Raises
MetaDataError – if no country information is available

Returns
dictionary of unique country names (keys) and corresponding country codes (values)

Return type
dict

`static get_meta_from_filename`(*file_path*)

Get meta information from file name

Note: This does not yet include IDs of model and obs data as these should be included in the data anyways (e.g. column names in CSV file) and may include the delimiter _ in their name.

Returns
dicitonary with meta information

Return type
dict

get_meta_item(*key: str*)

Get metadata value

Parameters
key (*str*) – meta item key.

Raises
AttributeError – If key is not available.

Returns
value of metadata.

Return type
object

get_regional_timeseries(*region_id*, ***filter_kwargs*)

Compute regional timeseries both for model and obs

Parameters

- **region_id** (*str*) – name of region for which regional timeseries is supposed to be retrieved
- ****filter_kwargs** – additional keyword args passed to *filter_region()*.

Returns
dictionary containing regional timeseries for model (key *mod*) and obsdata (key *obs*) and name of region.

Return type
dict

get_time_resampling_settings()

Returns a dictionary with relevant settings for temporal resampling

Return type
dict

property has_latlon_dims

Boolean specifying whether data has latitude and longitude dimensions

property has_time_dim

Boolean specifying whether data has a time dimension

property lat_range

Latitude range covered by this data object

property latitude

Array of latitude coordinates

property lon_range

Longitude range covered by this data object

property longitude

Array of longitude coordinates

max()

Wrapper for `xarray.DataArray.max()` called from *data*

Returns
maximum of data

Return type
xarray.DataArray

property metadata

Meta data dictionary (wrapper to `data.attrs`

min()

Wrapper for `xarray.DataArray.min()` called from *data*

Returns
minimum of data

Return type
xarray.DataArray

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'allow', 'protected_namespaces': (), 'validate_assignment': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data': FieldInfo(annotation=Union[Path, str, DataArray, ndarray, NoneType], required=False, default=None)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

property model_name

property ndim

Dimension of data array

property num_coords

Total number of lat/lon coordinate pairs

property num_coords_with_data

Number of lat/lon coordinate pairs that contain at least one datapoint

---

Note: Occurrence of valid data is only checked for obsdata (first index in data_source dimension).

---

property obs_name

open(*file_path*)

High level helper for reading from supported file sources

Parameters
file_path (*str*) – file path

plot_coordinates(*marker='x'*, *markersize=12*, *fontsize_base=10*, ***kwargs*)

Plot station coordinates

Uses *pyaerocom.plot.plotcoordinates.plot_coordinates()*.

Parameters

- **marker** (*str, optional*) – matplotlib marker name used to plot site locations. The default is 'x'.
- **markersize** (*int, optional*) – Size of site markers. The default is 12.
- **fontsize_base** (*int, optional*) – Basic fontsize. The default is 10.
- ****kwargs** – additional keyword args passed to *pyaerocom.plot.plotcoordinates.plot_coordinates()*

Return type
matplotlib.axes.Axes

plot_scatter(***kwargs*)

Create scatter plot of data

Parameters
****kwargs** – keyword args passed to *pyaerocom.plot.plotscatter.plot_scatter()*

Returns
matplotlib axes instance

Return type
Axes

read_netcdf(*file_path*)

Read data from NetCDF file

Parameters
file_path (*str*) – file path

rename_variable(*var_name*, *new_var_name*, *data_source*, *inplace=True*)

Rename a variable in this object

Parameters

- **var_name** (*str*) – current variable name
- **new_var_name** (*str*) – new variable name
- **data_source** (*str*) – name of data source (along data_source dimension)
- **inplace** (*bool*) – replace here or create new instance

Returns
instance with renamed variable

Return type
ColocatedData

Raises

- ***VarNotAvailableError*** – if input variable is not available in this object
- ***DataSourceError*** – if input data_source is not available in this object

resample_time(*to_ts_type*, *how=None*, *min_num_obs=None*, *colocate_time=False*, *settings_from_meta=False*, *inplace=False*, ***kwargs*)

Resample time dimension

The temporal resampling is done using `TimeResampler`

Parameters

- **to_ts_type** (*str*) – desired output frequency.

- **how** (*str or dict, optional*) – aggregator used for resampling (e.g. max, min, mean, median). Can also be hierarchical scheme via *dict*, similar to *min_num_obs*. The default is None.
- **min_num_obs** (*int or dict, optional*) – Minimum number of observations required to resample from current frequency (*ts_type*) to desired output frequency.
- **colocate_time** (*bool, optional*) – If True, the modeldata is invalidated where obs is NaN, before resampling. The default is False (updated in v0.11.0, before was True).
- **settings_from_meta** (*bool*) – if True, then input args *how*, *min_num_obs* and *colocate_time* are ignored and instead the corresponding values set in *metadata* are used. Defaults to False.
- **inplace** (*bool, optional*) – If True, modify this object directly, else make a copy and resample that one. The default is False (updated in v0.11.0, before was True).
- ****kwargs** – Addtitional keyword args passed to `TimeResampler.resample()`.

Returns
Resampled colocated data object.

Return type
ColocatedData

property savename_aerocom

Default save name for data object following AeroCom convention

set_zeros_nan(*inplace=True*)

Replace all 0's with NaN in data

Parameters
inplace (*bool*) – Whether to modify this object or a copy. The default is True.

Returns
cd – modified data object

Return type
ColocatedData

property shape

Shape of data array

stack(*inplace=False*, ***kwargs*)

Stack one or more dimensions

For details see `xarray.DataArray.stack()`.

Parameters
- **inplace** (*bool*) – modify this object or a copy.
- ****kwargs** – input arguments passed to `DataArray.stack()`

Returns
stacked data object

Return type
ColocatedData

property start

Start datetime of data

property start_str

Start date of data as str with format YYYYMMDD

Type
str

property stop

Stop datetime of data

property stop_str

Stop date of data as str with format YYYYMMDD

Type
str

property time

Array containing time stamps

to_csv(*out_dir*, *savename=None*)

Save data object as .csv file

Converts data to pandas.DataFrame and then saves as csv

Parameters

- **out_dir** (*str*) – output directory
- **savename** (str, optional) – name of file, if None, the default save name is used (cf. *savename_aerocom*)

to_dataframe()

Convert this object into pandas.DataFrame

Note: This does not include meta information

to_netcdf(*out_dir*, *savename=None*, ***kwargs*)

Save data object as NetCDF file

Wrapper for method `xarray.DataArray.to_netcdf()`

Parameters

- **out_dir** (*str*) – output directory
- **savename** (*str, optional*) – name of file, if None, the default save name is used (cf. *savename_aerocom*)
- ****kwargs** – additional, optional keyword arguments passed to `xarray.DataArray.to_netcdf()`

Returns
file path of stored object.

Return type
str

property ts_type

String specifying temporal resolution of data

property units

Unit of data

property unitstr

String representation of obs and model units in this object

unstack(*inplace=False*, ***kwargs*)

Unstack one or more dimensions

For details see `xarray.DataArray.unstack()`.

Parameters

- **inplace** (*bool*) – modify this object or a copy.
- ****kwargs** – input arguments passed to `DataArray.unstack()`

Returns

unstacked data object

Return type

ColocatedData

validate_data()

property var_name

Coordinate array containing data sources (z-axis)

pyaerocom.colocateddata.**ensure_correct_dimensions**(*data: ndarray | DataArray*)

Ensure the dimensions on either a numpy aray or xarray passed to ColocatedData. If a ColocatedData object is created outside of pyaerocom, this checking is needed. This function is used as part of the model validator.

4.2.4 Station data

class pyaerocom.stationdata.**StationData**(***meta_info*)

Dict-like base class for single station data

ToDo: write more detailed introduction

Note: Variable data (e.g. numpy array or pandas Series) can be directly assigned to the object. When assigning variable data it is recommended to add variable metadata (e.g. unit, ts_type) in *var_info*, where key is variable name and value is dict with metadata entries.

---

dtime

list / array containing time index values

Type

list

var_info

dictionary containing information about each variable

Type

dict

data_err

dictionary that may be used to store uncertainty timeseries or data arrays associated with the different variable data.

Type

dict

overlap

dictionary that may be filled to store overlapping timeseries data associated with one variable. This is, for instance, used in *merge_vardata()* to store overlapping data from another station.

Type
dict

PROTECTED_KEYS = ['dtime', 'var_info', 'station_coords', 'data_err', 'overlap', 'numobs', 'data_flagged']

Keys that are ignored when accessing metadata

STANDARD_COORD_KEYS = ['latitude', 'longitude', 'altitude']

List of keys that specify standard metadata attribute names. This is used e.g. in *get_meta()*

STANDARD_META_KEYS = ['filename', 'station_id', 'station_name', 'instrument_name', 'PI', 'country', 'country_code', 'ts_type', 'latitude', 'longitude', 'altitude', 'data_id', 'dataset_name', 'data_product', 'data_version', 'data_level', 'framework', 'instr_vert_loc', 'revision_date', 'website', 'ts_type_src', 'stat_merge_pref_attr']

VALID_TS_TYPES = ['minutely', 'hourly', 'daily', 'weekly', 'monthly', 'yearly', 'native']

calc_climatology(*var_name*, *start=None*, *stop=None*, *min_num_obs=None*, *clim_mincount=None*, *clim_freq=None*, *set_year=None*, *resample_how=None*)

Calculate climatological timeseries for input variable

Parameters

- **var_name** (*str*) – name of data variable
- **start** – start time of data used to compute climatology
- **stop** – start time of data used to compute climatology
- **min_num_obs** (*dict or int, optional*) – minimum number of observations required per period (when downsampling). For details see *pyaerocom.time_resampler.TimeResampler.resample()*)
- **clim_micount** (*int, optional*) – minimum number of of monthly values required per month of climatology
- **set_year** (*int, optional*) – if specified, the output data will be assigned the input year. Else the middle year of the climatological interval is used.
- **resample_how** (*str*) – how should the resampled data be averaged (e.g. mean, median)
- ****kwargs** – Additional keyword args passed to *pyaerocom.time_resampler.TimeResampler.resample()*

Returns
new instance of StationData containing climatological data

Return type
StationData

check_dtime()

Checks if dtime attribute is array or list

check_if_3d(*var_name*)

Checks if altitude data is available in this object

check_unit(*var_name*, *unit=None*)

Check if variable unit corresponds to a certain unit

Parameters

- **var_name** (*str*) – variable name for which unit is to be checked
- **unit** (str, optional) – unit to be checked, if None, AeroCom default unit is used

Raises

- *MetaDataError* – if unit information is not accessible for input variable name
- *UnitConversionError* – if current unit cannot be converted into specified unit (e.g. 1 vs m-1)
- *DataUnitError* – if current unit is not equal to input unit but can be converted (e.g. 1/Mm vs 1/m)

check_var_unit_aerocom(*var_name*)

Check if unit of input variable is AeroCom default, if not, convert

Parameters
var_name (*str*) – name of variable

Raises

- *MetaDataError* – if unit information is not accessible for input variable name
- *UnitConversionError* – if current unit cannot be converted into specified unit (e.g. 1 vs m-1)
- *DataUnitError* – if current unit is not equal to AeroCom default and cannot be converted.

convert_unit(*var_name*, *to_unit*)

Try to convert unit of data

Requires that unit of input variable is available in *var_info*

Parameters

- **var_name** (*str*) – name of variable
- **to_unit** (*str*) – new unit

Raises

- *MetaDataError* – if variable unit cannot be accessed
- *UnitConversionError* – if conversion failed

copy()

property default_vert_grid

AeroCom default grid for vertical regridding

For details, see DEFAULT_VERT_GRID_DEF in Config

Returns
numpy array specifying default coordinates

Return type
ndarray

dist_other(*other*)

Distance to other station in km

Parameters
other (StationData) – other data object

Returns
distance between this and other station in km

Return type
float

get_meta(*force_single_value=True*, *quality_check=True*, *add_none_vals=False*, *add_meta_keys=None*)

Return meta-data as dictionary

By default, only default metadata keys are considered, use parameter *add_meta_keys* to add additional metadata.

Parameters

- **force_single_value** (*bool*) – if True, then each meta value that is list or array,is converted to a single value.
- **quality_check** (*bool*) – if True, and coordinate values are lists or arrays, then the standarad deviation in the values is compared to the upper limits allowed in the local variation. The upper limits are specified in attr. COORD_MAX_VAR.
- **add_none_vals** (*bool*) – Add metadata keys which have value set to None.
- **add_meta_keys** (*str or list, optional*) – Add none-standard metadata.

Returns
dictionary containing the retrieved meta-data

Return type
dict

Raises

- **AttributeError** – if one of the meta entries is invalid
- ***MetaDataError*** – in case of consistencies in meta data between individual time-stamps

get_station_coords(*force_single_value=True*)

Return coordinates as dictionary

This method uses the standard coordinate names defined in *STANDARD_COORD_KEYS* (latitude, longitude and altitude) to get the station coordinates. For each of these parameters tt first looks in `station_coords` if the parameter is defined (i.e. it is not None) and if not it checks if this object has an attribute that has this name and uses that one.

Parameters
force_single_value (*bool*) – if True and coordinate values are lists or arrays, then they are collapsed to single value using mean

Returns
dictionary containing the retrieved coordinates

Return type
dict

Raises

- **AttributeError** – if one of the coordinate values is invalid

- *CoordinateError* – if local variation in either of the three spatial coordinates is found too large

get_unit(*var_name*)

Get unit of variable data

Parameters
var_name (*str*) – name of variable

Returns
unit of variable

Return type
str

Raises
MetaDataError – if unit cannot be accessed for variable

get_var_ts_type(*var_name*, *try_infer=True*)

Get ts_type for a certain variable

Note: Converts to ts_type string if assigned ts_type is in pandas format

Parameters

- **var_name** (*str*) – data variable name for which the ts_type is supposed to be retrieved
- **try_infer** (*bool*) – if ts_type is not available, try inferring it from data

Returns
the corresponding data time resolution

Return type
str

Raises
MetaDataError – if no metadata is available for this variable (e.g. if `var_name` cannot be found in *var_info*)

has_var(*var_name*)

Checks if input variable is available in data object

Parameters
var_name (*str*) – name of variable

Returns
True, if variable data is available, else False

Return type
bool

insert_nans_timeseries(*var_name*)

Fill up missing values with NaNs in an existing time series

Note: This method does a resample of the data onto a regular grid. Thus, if the input `ts_type` is different from the actual current `ts_type` of the data, this method will not only insert NaNs but at the same.

Parameters

- **var_name** (*str*) – variable name
- **inplace** (*bool*) – if True, the actual data in this object will be overwritten with the new data that contains NaNs

Returns
the modified station data object

Return type
StationData

merge_meta_same_station(*other*, *coord_tol_km=None*, *check_coords=True*, *inplace=True*, *add_meta_keys=None*, *raise_on_error=False*)

Merge meta information from other object

Note: Coordinate attributes (latitude, longitude and altitude) are not copied as they are required to be the same in both stations. The latter can be checked and ensured using input argument `check_coords`

Parameters

- **other** (StationData) – other data object
- **coord_tol_km** (*float*) – maximum distance in km between coordinates of input StationData object and self. Only relevant if `check_coords` is True. If None, then `_COORD_MAX_VAR` is used which is defined in the class header.
- **check_coords** (*bool*) – if True, the coordinates are compared and checked if they are lying within a certain distance to each other (cf. `coord_tol_km`).
- **inplace** (*bool*) – if True, the metadata from the other station is added to the metadata of this station, else, a new station is returned with the merged attributes.
- **add_meta_keys** (*str or list, optional*) – additional non-standard metadata keys that are supposed to be considered for merging.
- **raise_on_error** (*bool*) – if True, then an Exception will be raised in case one of the metadata items cannot be merged, which is most often due to unresolvable type differences of metadata values between the two objects

merge_other(*other*, *var_name*, *add_meta_keys=None*, ***kwargs*)

Merge other station data object

Parameters

- **other** (StationData) – other data object
- **var_name** (*str*) – variable name for which info is to be merged (needs to be both available in this object and the provided other object)
- **add_meta_keys** (*str or list, optional*) – additional non-standard metadata keys that are supposed to be considered for merging.
- **kwargs** – keyword args passed on to *merge_vardata()* (e.g time resampling settings)

Returns
this object that has merged the other station

Return type
StationData

merge_vardata(*other*, *var_name*, ***kwargs*)

Merge variable data from other object into this object

Note: This merges also the information about this variable in the dict *var_info*. It is required, that variable meta-info is specified in both StationData objects.

Note: This method removes NaN's from the existing time series in the data objects. In order to fill up the time-series with NaNs again after merging, call *insert_nans_timeseries()*

Parameters

- **other** (StationData) – other data object
- **var_name** (*str*) – variable name for which info is to be merged (needs to be both available in this object and the provided other object)
- **kwargs** – keyword args passed on to `_merge_vardata_2d()`

Returns

this object merged with other object

Return type

StationData

merge_varinfo(*other*, *var_name*)

Merge variable specific meta information from other object

Parameters

- **other** (StationData) – other data object
- **var_name** (*str*) – variable name for which info is to be merged (needs to be both available in this object and the provided other object)

plot_timeseries(*var_name*, *add_overlaps=False*, *legend=True*, *tit=None*, ***kwargs*)

Plot timeseries for variable

Note: If you set input arg `add_overlaps = True` the overlapping timeseries data - if it exists - will be plotted on top of the actual timeseries using red colour and dashed line. As the overlapping data may be identical with the actual data, you might want to increase the line width of the actual timeseries using an additional input argument `lw=4`, or similar.

Parameters

- **var_name** (*str*) – name of variable (e.g. "od550aer")
- **add_overlaps** (*bool*) – if True and if overlapping data exists for this variable, it will be added to the plot.
- **tit** (str, optional) – title of plot, if None, default title is used
- ****kwargs** – additional keyword args passed to matplotlib `plot` method

Returns

matplotlib.axes instance of plot

Return type
axes

Raises

- **KeyError** – if variable key does not exist in this dictionary
- **ValueError** – if length of data array does not equal the length of the time array

remove_outliers(*var_name*, *low=None*, *high=None*, *check_unit=True*)

Remove outliers from one of the variable timeseries

Parameters

- **var_name** (*str*) – variable name
- **low** (*float*) – lower end of valid range for input variable. If None, then the corresponding value from the default settings for this variable are used (cf. minimum attribute of available variables)
- **high** (*float*) – upper end of valid range for input variable. If None, then the corresponding value from the default settings for this variable are used (cf. maximum attribute of available variables)
- **check_unit** (*bool*) – if True, the unit of the data is checked against AeroCom default

remove_variable(*var_name*)

Remove variable data

Parameters
var_name (*str*) – name of variable that is to be removed

Returns
current instance of this object, with data removed

Return type
StationData

Raises
VarNotAvailableError – if the input variable is not available in this object

resample_time(*var_name*, *ts_type*, *how=None*, *min_num_obs=None*, *inplace=False*, ***kwargs*)

Resample one of the time-series in this object

Parameters

- **var_name** (*str*) – name of data variable
- **ts_type** (*str*) – new frequency string (can be pyaerocom ts_type or valid pandas frequency string)
- **how** (*str*) – how should the resampled data be averaged (e.g. mean, median)
- **min_num_obs** (*dict or int, optional*) – minimum number of observations required per period (when downsampling). For details see *pyaerocom.time_resampler.TimeResampler.resample()*)
- **inplace** (*bool*) – if True, then the current data object stored in self, will be overwritten with the resampled time-series
- ****kwargs** – Additional keyword args passed to *pyaerocom.time_resampler.TimeResampler.resample()*

Returns
with resampled variable timeseries

Return type
StationData

resample_timeseries(*var_name*, ***kwargs*)

Wrapper for *resample_time()* (for backwards compatibility)

Note: For backwards compatibility, this method will return a pandas Series instead of the actual StationData object

---

same_coords(*other*, *tol_km=None*)

Compare station coordinates of other station with this station

Parameters

- **other** (StationData) – other data object
- **tol_km** (*float*) – distance tolerance in km

Returns
if True, then the two object are located within the specified tolerance range

Return type
bool

select_altitude(*var_name*, *altitudes*)

Extract variable data within certain altitude range

Note: Beta version

Parameters

- **var_name** (*str*) – name of variable for which metadata is supposed to be extracted
- **altitudes** (*list*) – altitude range in m, e.g. [0, 1000]

Returns
data object within input altitude range

Return type
pandas. Series or xarray.DataArray

to_timeseries(*var_name*, ***kwargs*)

Get pandas.Series object for one of the data columns

Parameters
var_name (*str*) – name of variable (e.g. "od550aer")

Returns
time series object

Return type
Series

Raises

- **KeyError** – if variable key does not exist in this dictionary

- **ValueError** – if length of data array does not equal the length of the time array

property units

Dictionary containing units of all variables in this object

property vars_available

Number of variables available in this data object

4.2.5 Other data classes

class pyaerocom.vertical_profile.**VerticalProfile**(*data: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *altitude: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *dtime*, *var_name: str*, *data_err: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*, *var_unit: str*, *altitude_unit: str*)

Object representing single variable profile data

property altitude

Array containing altitude values corresponding to data

property data

Array containing data values corresponding to data

property data_err

Array containing data values corresponding to data

plot(*plot_errs=True*, *whole_alt_range=False*, *rot_xlabels=30*, *errs_shaded=True*, *errs_alpha=0.1*, *add_vertbar_zero=True*, *figsize=None*, *ax=None*, ***kwargs*)

Simple plot method for vertical profile

4.3 Co-location routines

4.3.1 High-level co-location engine

Classes and methods to perform high-level colocation.

class pyaerocom.colocation_auto.**ColocationSetup**(*model_id=None*, *obs_config: PyaroConfig | None = None*, *obs_id=None*, *obs_vars=None*, *ts_type=None*, *start=None*, *stop=None*, *basedir_coldata=None*, *save_coldata=False*, ***kwargs*)

Setup class for high-level model / obs co-location.

An instance of this setup class can be used to run a colocation analysis between a model and an observation network and will create a number of `pya.ColocatedData` instances, which can be saved automatically as NetCDF files.

Apart from co-location, this class also handles reading of the input data for co-location. Supported co-location options are:

1. gridded vs. ungridded data For instance 3D model data (instance of `GriddedData`) with lat, lon and time dimension that is co-located with station based observations which are represented in pyaerocom through `UngriddedData` objects. The co-location function used is `pyaerocom.colocation.colocated_gridded_ungridded()`. For this type of co-location, the output co-located data object will be 3-dimensional, with dimensions *data_source* (index 0: obs, index 1: model), *time* and *station_name*.

2. gridded vs. gridded data For instance 3D model data that is co-located with 3D satellite data (both instances of `GriddedData`), both objects with lat, lon and time dimensions. The co-location function used is `pyaerocom.colocation.colocated_gridded_gridded()`. For this type of co-location, the output co-located data object will be 4-dimensional, with dimensions *data_source* (index 0: obs, index 1: model), *time* and *latitude* and *longitude*.

`model_id`

ID of model to be used.

Type
str

`obs_config`

In the case Pyaro is used, a config must be provided. In that case obs_id(see below) is ignored and only the config is used.

Type
PyaroConfig

`obs_id`

ID of observation network to be used.

Type
str

`obs_vars`

Variables to be analysed (need to be available in input obs dataset). Variables that are not available in the model data output will be skipped. Alternatively, model variables to be used for a given obs variable can also be specified via attributes *`model_use_vars`* and *`model_add_vars`*.

Type
list

`ts_type`

String specifying colocation output frequency.

Type
str

`start`

Start time of colocation. Input can be integer denoting the year or anything that can be converted into `pandas.Timestamp` using *`pyaerocom.helpers.to_pandas_timestamp()`*. If None, than the first available date in the model data is used.

`stop`

stop time of colocation. int or anything that can be converted into `pandas.Timestamp` using *`pyaerocom.helpers.to_pandas_timestamp()`* or None. If None and if `start` is on resolution of year (e.g.

start=2010) then stop will be automatically set to the end of that year. Else, it will be set to the last available timestamp in the model data.

filter_name

name of filter to be applied. If None, no filter is used (to be precise, if None, then

pyaerocom.const.DEFAULT_REG_FILTER is used which should default to *ALL-wMOUNTAINS*, that is, no filtering).

Type
str

basedir_coldata

Base directory for storing of colocated data files.

Type
str

save_coldata

if True, colocated data objects are saved as NetCDF file.

Type
bool

obs_name

if provided, this string will be used in colocated data filename to specify obsnetwork, else obs_id will be used.

Type
str, optional

obs_data_dir

location of obs data. If None, attempt to infer obs location based on obs ID.

Type
str, optional

obs_use_climatology

BETA if True, pyaerocom default climatology is computed from observation stations (so far only possible for unrgidded / gridded colocation).

Type
bool

obs_vert_type

AeroCom vertical code encoded in the model filenames (only AeroCom 3 and later). Specifies which model file should be read in case there are multiple options (e.g. surface level data can be read from a *Surface*.nc file as well as from a *ModelLevel*.nc file). If input is string (e.g. 'Surface'), then the corresponding vertical type code is used for reading of all variables that are colocated (i.e. that are specified in *obs_vars*).

Type
str

obs_ts_type_read

may be specified to explicitly define the reading frequency of the observation data (so far, this does only apply to gridded obsdata such as satellites), either as str (same for all obs variables) or variable specific as dict. For ungridded reading, the frequency may be specified via *obs_id*, where applicable (e.g. Aeronet-SunV3Lev2.daily). Not to be confused with *ts_type*, which specifies the frequency used for colocation. Can be specified variable specific in form of dictionary.

Type
str or dict, optional

`obs_filters`

filters applied to the observational dataset before co-location. In case of gridded / gridded, these are filters that can be passed to `pyaerocom.io.ReadGridded.read_var()`, for instance, *flex_ts_type*, or *constraints*. In case the obsdata is ungridded (gridded / ungridded co-locations) these are filters that are handled through keyword *filter_post* in `pyaerocom.io.ReadUngridded.read()`. These filters are applied to the `UngriddedData` objects after reading and caching the data, so changing them, will not invalidate the latest cache of the `UngriddedData`.

Type
dict

`read_opts_ungridded`

dictionary that specifies reading constraints for ungridded reading, and are passed as ***kwargs* to `pyaerocom.io.ReadUngridded.read()`. Note that - other than for *obs_filters* these filters are applied during the reading of the `UngriddedData` objects and specifying them will deactivate caching.

Type
dict, optional

`model_name`

if provided, this string will be used in colocated data filename to specify model, else obs_id will be used.

Type
str, optional

`model_data_dir`

Location of model data. If None, attempt to infer model location based on model ID.

Type
str, optional

`model_read_opts`

options for model reading (passed as keyword args to `pyaerocom.io.ReadUngridded.read()`).

Type
dict, optional

`model_use_vars`

dictionary that specifies mapping of model variables. Keys are observation variables, values are the corresponding model variables (e.g. model_use_vars=dict(od550aer='od550csaer')). Example: your observation has var *od550aer* but your model model uses a different variable name for that variable, say *od550*. Then, you can specify this via *model_use_vars = {'od550aer' : 'od550'}*. NOTE: in this case, a model variable *od550aer* will be ignored, even if it exists (cf `model_add_vars`).

Type
dict, optional

`model_rename_vars`

rename certain model variables **after** co-location, before storing the associated `ColocatedData` object on disk. Keys are model variables, values are new names (e.g. *model_rename_vars={'od550aer':'MyAOD'}*). Note: this does not impact which variables are read from the model.

Type
dict, optional

model_add_vars

additional model variables to be processed for one obs variable. E.g. *model_add_vars={'od550aer': ['od550so4', 'od550gt1aer']}* would co-locate both model SO4 AOD (od550so4) and model coarse mode AOD (od550gt1aer) with total AOD (od550aer) from obs (in addition to od550aer vs od550aer if applicable).

Type
dict, optional

model_to_stp

ALPHA (please do not use): convert model data values to STP conditions after co-location. Note: this only works for very particular settings at the moment and needs revision, as it relies on access to meteorological data.

Type
bool

model_ts_type_read

may be specified to explicitly define the reading frequency of the model data, either as str (same for all obs variables) or variable specific as dict. Not to be confused with *ts_type*, which specifies the output frequency of the co-located data.

Type
str or dict, optional

model_read_aux

may be used to specify additional computation methods of variables from models. Keys are variables to be computed, values are dictionaries with keys *vars_required* (list of required variables for computation of var and *fun* (method that takes list of read data objects and computes and returns var).

Type
dict, optional

model_use_climatology

if True, attempt to use climatological model data field. Note: this only works if model data is in AeroCom conventions (climatological fields are indicated with 9999 as year in the filename) and if this is active, only single year analysis are supported (i.e. provide int to *start* to specify the year and leave *stop* empty).

Type
bool

gridded_reader_id

BETA: dictionary specifying which gridded reader is supposed to be used for model (and gridded obs) reading. Note: this is a workaround solution and will likely be removed in the future when the gridded reading API is more harmonised (see https://github.com/metno/pyaerocom/issues/174).

Type
dict

flex_ts_type

Bboolean specifying whether reading frequency of gridded data is allowed to be flexible. This includes all gridded data, whether it is model or gridded observation (e.g. satellites). Defaults to True.

Type
bool

min_num_obs

time resampling constraints applied, defaults to None, in which case no constraints are applied. For instance, say your input is in daily resolution and you want output in monthly and you want to make sure to have

roughly 50% daily coverage for the monthly averages. Then you may specify *min_num_obs=15* which will ensure that at least 15 daily averages are available to compute a monthly average. However, you may also define a hierarchical scheme that first goes from daily to weekly and then from weekly to monthly, via a dict. E.g. *min_num_obs=dict(monthly=dict(weekly=4), weekly=dict(daily=3))* would ensure that each week has at least 3 daily values, as well as that each month has at least 4 weekly values.

Type
dict or int, optional

resample_how

string specifying how data should be aggregated when resampling in time. Default is "mean". Can also be a nested dictionary, e.g. *resample_how={'conco3': 'daily': {'hourly' : 'max'}}* would use the maximum value to aggregate from hourly to daily for variable conco3, rather than the mean.

Type
str or dict, optional

obs_remove_outliers

if True, outliers are removed from obs data before colocation, else not. Default is False. Custom outlier ranges for each variable can be specified via *obs_outlier_ranges*, and for all other variables, the pyaerocom default outlier ranges are used. The latter are specified in *variables.ini* file via *minimum* and *maximum* attributes and can also be accessed through *pyaerocom.variable.Variable.minimum* and `pyaerocom.variable.Variable.maximum`, respectively.

Type
bool

model_remove_outliers

if True, outliers are removed from model data (normally this should be set to False, as the models are supposed to be assessed, including outlier cases). Default is False. Custom outlier ranges for each variable can be specified via *model_outlier_ranges*, and for all other variables, the pyaerocom default outlier ranges are used. The latter are specified in *variables.ini* file via *minimum* and *maximum* attributes and can also be accessed through *pyaerocom.variable.Variable.minimum* and `pyaerocom.variable.Variable.maximum`, respectively.

Type
bool

obs_outlier_ranges

dictionary specifying outlier ranges for individual obs variables. (e.g. dict(od550aer = [-0.05, 10], ang4487aer=[0,4])). Only relevant if *obs_remove_outliers* is True.

Type
dict, optional

model_outlier_ranges

like *obs_outlier_ranges* but for model variables. Only relevant if *model_remove_outliers* is True.

Type
dict, optional

zeros_to_nan

If True, zero's in output co-located data object will be converted to NaN. Default is False.

Type
bool

harmonise_units

if True, units are attempted to be harmonised during co-location (note: raises Exception if True and in case units cannot be harmonised).

Type
bool

`regrid_res_deg`

resolution in degrees for regridding of model grid (done before co-location). Default is None.

Type
int, optional

`colocate_time`

if True and if obs and model sampling frequency (e.g. daily) are higher than output colocation frequency (e.g. monthly), then the datasets are first colocated in time (e.g. on a daily basis), before the monthly averages are calculated. Default is False.

Type
bool

`reanalyse_existing`

if True, always redo co-location, even if there is already an existing co-located NetCDF file (under the output location specified by *basedir_coldata*) for the given variable combination to be co-located. If False and output already exists, then co-location is skipped for the associated variable. Default is True.

Type
bool

`raise_exceptions`

if True, Exceptions that may occur for individual variables to be processed, are raised, else the analysis is skipped for such cases.

Type
bool

`keep_data`

if True, then all colocated data objects computed when running `run()` will be stored in `data`. Defaults to True.

Type
bool

`add_meta`

additional metadata that is supposed to be added to each output `ColocatedData` object.

Type
dict

`CRASH_ON_INVALID = False`

do not raise Exception if invalid item is attempted to be assigned (Overwritten from base class)

`OBS_VERT_TYPES_ALT = {'2D': '2D', 'Surface': 'ModelLevel'}`

Dictionary specifying alternative vertical types that may be used to read model data. E.g. consider the variable is ec550aer, obs_vert_type='Surface' and obs_vert_type_alt=dict(Surface='ModelLevel'). Now, if a model that is used for the analysis does not contain a data file for ec550aer at the surface ('*ec550aer*Surface*.nc'), then, the colocation routine will look for '*ec550aer*ModelLevel*.nc' and if this exists, it will load it and extract the surface level.

`add_glob_meta`(***kwargs*)

Add global metadata to *add_meta*

Parameters
kwargs – metadata to be added

Return type
None

property basedir_logfiles

Base directory for storing logfiles

class pyaerocom.colocation_auto.**Colocator**(***kwargs*)

High level class for running co-location

Note: This object inherits from *ColocationSetup* and is also instantiated as such. For setup attributes, please see base class.

get_model_name()

Get name of model

Note: Not to be confused with `model_id` which is always the database ID of the model, while model_name can differ from that and is used for output files, etc.

Raises
AttributeError – If neither model_id or model_name are set

Returns
preferably `model_name`, else `model_id`

Return type
str

get_nc_files_in_coldatadir()

Get list of NetCDF files in colocated data directory

Returns
list of NetCDF file paths found

Return type
list

get_obs_name()

Get name of obsdata source

Note: Not to be confused with `obs_id` which is always the database ID of the observation dataset, while obs_name can differ from that and is used for output files, etc.

Raises
AttributeError – If neither obs_id or obs_name are set

Returns
preferably `obs_name`, else `obs_id`

Return type
str

property model_reader

Model data reader

property model_vars

List of all model variables specified in config

Note: This method does not check if the variables are valid or available.

Returns
list of all model variables specified in this setup.

Return type
list

property obs_is_ungridded

True if obs_id refers to an ungridded observation, else False

Type
bool

property obs_is_vertical_profile

True if obs_id refers to a VerticalProfile, else False

Type
bool

property obs_reader

Observation data reader

property output_dir

Output directory for colocated data NetCDF files

Type
str

prepare_run(*var_list: list | None = None*) → dict

Prepare colocation run for current setup.

Parameters
var_name (*str, optional*) – Variable name that is supposed to be analysed. The default is None, in which case all defined variables are attempted to be colocated.

Raises
AttributeError – If no observation variables are defined (`obs_vars` empty).

Returns
vars_to_process – Mapping of variables to be processed, keys are model vars, values are obs vars.

Return type
dict

run(*var_list: list | None = None*, ***opts*)

Perform colocation for current setup

See also *prepare_run()*.

Parameters

- **var_list** (*list, optional*) – list of variables supposed to be analysed. The default is None, in which case all defined variables are attempted to be colocated.

- ****opts** – keyword args that may be specified to change the current setup before colocation

Returns
nested dictionary, where keys are model variables, values are dictionaries comprising key / value pairs of obs variables and associated instances of `ColocatedData`.

Return type
dict

4.3.2 Low-level co-location functions

Methods and / or classes to perform colocation

pyaerocom.colocation.**check_time_ival**(*data*, *start*, *stop*)

pyaerocom.colocation.**check_ts_type**(*data*, *ts_type*)

pyaerocom.colocation.**colocate_gridded_gridded**(*data*, *data_ref*, *ts_type=None*, *start=None*, *stop=None*, *filter_name=None*, *regrid_res_deg=None*, *harmonise_units=True*, *regrid_scheme='areaweighted'*, *update_baseyear_gridded=None*, *min_num_obs=None*, *colocate_time=False*, *resample_how=None*, ***kwargs*)

Colocate 2 gridded data objects

Parameters

- **data** (GriddedData) – gridded data (e.g. model results)
- **data_ref** (GriddedData) – reference data (e.g. gridded satellite object) that is co-located with *data*. observation data or other model)
- **ts_type** (*str, optional*) – desired temporal resolution of output colocated data (e.g. "monthly"). Defaults to None, in which case the highest possible resolution is used.
- **start** (*str or datetime64 or similar, optional*) – start time for colocation, if None, the start time of the input `GriddedData` object is used
- **stop** (*str or datetime64 or similar, optional*) – stop time for colocation, if None, the stop time of the input `GriddedData` object is used
- **filter_name** (*str, optional*) – string specifying filter used (cf. *pyaerocom.filter.Filter* for details). If None, then it is set to 'ALL-wMOUNTAINS', which corresponds to no filtering (world with mountains). Use ALL-noMOUNTAINS to exclude mountain sites.
- **regrid_res_deg** (*int or dict, optional*) – regrid resolution in degrees. If specified, the input gridded data objects will be regridded in lon / lat dimension to the input resolution (if input is integer, both lat and lon are regridded to that resolution, if input is dict, use keys *lat_res_deg* and *lon_res_deg* to specify regrid resolutions, respectively).
- **harmonise_units** (*bool*) – if True, units are attempted to be harmonised (note: raises Exception if True and units cannot be harmonised). Defaults to True.
- **regrid_scheme** (*str*) – iris scheme used for regridding (defaults to area weighted regridding)
- **update_baseyear_gridded** (*int, optional*) – optional input that can be set in order to redefine the time dimension in the first gridded data object `data`to be analysed. E.g., if the data object is a climatology (one year of data) that has set the base year of the time dimension to a value other than the specified input start / stop time this may be used to update the time in order to make co-location possible.

- **min_num_obs** (*int or dict, optional*) – minimum number of observations for resampling of time
- **colocate_time** (*bool*) – if True and if original time resolution of data is higher than desired time resolution (*ts_type*), then both datasets are colocated in time *before* resampling to lower resolution.
- **resample_how** (*str or dict*) – string specifying how data should be aggregated when resampling in time. Default is "mean". Can also be a nested dictionary, e.g. resample_how={'daily': {'hourly' : 'max'}} would use the maximum value to aggregate from hourly to daily, rather than the mean.
- ****kwargs** – additional keyword args (not used here, but included such that factory class can handle different methods with different inputs)

Returns
instance of colocated data

Return type
ColocatedData

pyaerocom.colocation.**colocate_gridded_ungridded**(*data*, *data_ref*, *ts_type=None*, *start=None*, *stop=None*, *filter_name=None*, *regrid_res_deg=None*, *harmonise_units=True*, *regrid_scheme='areaweighted'*, *var_ref=None*, *update_baseyear_gridded=None*, *min_num_obs=None*, *colocate_time=False*, *use_climatology_ref=False*, *resample_how=None*, ***kwargs*)

Colocate gridded with ungridded data (low level method)

For high-level colocation see *pyaerocom.colocation_auto.Colocator* and *pyaerocom.colocation_auto.ColocationSetup*

Note: Uses the variable that is contained in input `GriddedData` object (since these objects only contain a single variable). If this variable is not contained in observation data (or contained but using a different variable name) you may specify the obs variable to be used via input arg *var_ref*

Parameters

- **data** (GriddedData) – gridded data object (e.g. model results).
- **data_ref** (UngriddedData) – ungridded data object (e.g. observations).
- **ts_type** (*str*) – desired temporal resolution of colocated data (must be valid AeroCom ts_type str such as daily, monthly, yearly.).
- **start** (str or datetime64 or similar, optional) – start time for colocation, if None, the start time of the input `GriddedData` object is used.
- **stop** (str or datetime64 or similar, optional) – stop time for colocation, if None, the stop time of the input `GriddedData` object is used
- **filter_name** (*str*) – string specifying filter used (cf. *pyaerocom.filter.Filter* for details). If None, then it is set to 'ALL-wMOUNTAINS', which corresponds to no filtering (world with mountains). Use ALL-noMOUNTAINS to exclude mountain sites.
- **regrid_res_deg** (*int or dict, optional*) – regrid resolution in degrees. If specified, the input gridded data object will be regridded in lon / lat dimension to the input resolution

(if input is integer, both lat and lon are regridded to that resolution, if input is dict, use keys *lat_res_deg* and *lon_res_deg* to specify regrid resolutions, respectively).

- **harmonise_units** (*bool*) – if True, units are attempted to be harmonised (note: raises Exception if True and units cannot be harmonised).
- **var_ref** (str, optional) – variable against which data in arg *data* is supposed to be compared. If None, then the same variable is used (i.e. *data.var_name*).
- **update_baseyear_gridded** (*int, optional*) – optional input that can be set in order to re-define the time dimension in the gridded data object to be analysed. E.g., if the data object is a climatology (one year of data) that has set the base year of the time dimension to a value other than the specified input start / stop time this may be used to update the time in order to make colocation possible.
- **min_num_obs** (*int or dict, optional*) – minimum number of observations for resampling of time
- **colocate_time** (*bool*) – if True and if original time resolution of data is higher than desired time resolution (*ts_type*), then both datasets are colocated in time *before* resampling to lower resolution.
- **use_climatology_ref** (*bool*) – if True, climatological timeseries are used from observations
- **resample_how** (*str or dict*) – string specifying how data should be aggregated when resampling in time. Default is "mean". Can also be a nested dictionary, e.g. resample_how={'daily': {'hourly' : 'max'}} would use the maximum value to aggregate from hourly to daily, rather than the mean.
- ****kwargs** – additional keyword args (passed to `UngriddedData.to_station_data_all()`)

Returns
instance of colocated data

Return type
ColocatedData

Raises

- ***VarNotAvailableError*** – if grid data variable is not available in ungridded data object
- **AttributeError** – if instance of input `UngriddedData` object contains more than one dataset
- ***TimeMatchError*** – if gridded data time range does not overlap with input time range
- ***ColocationError*** – if none of the data points in input `UngriddedData` matches the input colocation constraints

pyaerocom.colocation.**correct_model_stp_coldata**(*coldata*, *p0=None*, *t0=273.15*, *inplace=False*)

Correct modeldata in colocated data object to STP conditions

Note: BETA version, quite unelegant coded (at 8pm 3 weeks before IPCC deadline), but should do the job for 2010 monthly colocated data files (AND NOTHING ELSE)!

pyaerocom.colocation.**resolve_var_name**(*data*)

Check variable name of *GriddedData* against AeroCom default

Checks whether the variable name set in the data corresponds to the AeroCom variable name, or whether it is an alias. Returns both the variable name set and the AeroCom variable name.

Parameters
data (GriddedData) – Data to be checked.

Returns

- *str* – variable name as set in data (may be alias, but may also be AeroCom variable name, in which case first and second return parameter are the same).
- *str* – corresponding AeroCom variable name

4.3.3 Co-locating ungridded observations

pyaerocom.combine_vardata_ungridded.**combine_vardata_ungridded**(*data_ids_and_vars*, *match_stats_how='closest'*, *match_stats_tol_km=1*, *merge_how='combine'*, *merge_eval_fun=None*, *var_name_out=None*, *data_id_out=None*, *var_unit_out=None*, *resample_how=None*, *min_num_obs=None*, *add_meta_keys=None*)

Combine and colocate different variables from UngriddedData

This method allows to combine different variable timeseries from different ungridded observation records in multiple ways. The source data may be all included in a single instance of *UngriddedData* or in multiple, for details see first input parameter **:param:`data_ids_and_vars`**. Merging can be done in flexible ways, e.g. by combining measurements of the same variable from 2 different datasets or by computing new variables based on 2 measured variables (e.g. concox=concno2+conco3). Doing this requires colocation of site locations and timestamps of both input observation records, which is done in this method.

It comprises 2 major steps:

1. **Compute list of StationData objects for both input data combinations (data_id1 & var1; data_id2 & var2) and based on these, find the coincident locations. Finding coincident sites can either be done based on site location name or based on**
   their lat/lon locations. The method to use can be specified via input arg **:param:`match_stats_how`**.
2. **For all coincident locations, a new instance of StationData is computed that has merged the 2 timeseries in the way**
   that can be specified through input args **:param:`merge_how`** and **:param:`merge_eval_fun`**. If the 2 original timeseries from both sites come in different temporal resolutions, they will be resampled to the lower of both resolutions. Resampling constraints that are supposed to be applied in that case can be provided via the respective input args for temporal resampling. Default is pyaerocom default, which corresponds to ~25% coverage constraint (as of 22.10.2020) for major resolution steps, such as daily->monthly.

Note: Currently, only 2 variables can be combined to a new one (e.g. concox=conco3+concno2).

Note: Be aware of unit conversion issues that may arise if your input data is not in AeroCom default units. For

details see below.

Parameters

- **data_ids_and_vars** (*list*) – list of 3 element tuples, each containing, in the following order 1. instance of UngriddedData; 2. dataset ID (remember that UngriddedData can contain more than one dataset); and 3. variable name. Note that currently only 2 of such tuples can be combined.
- **match_stats_how** (*str, optional*) – String specifying how site locations are supposed to be matched. The default is 'closest'. Supported are 'closest' and 'station_name'.
- **match_stats_tol_km** (*float, optional*) – radius tolerance in km for matching site locations when using 'closest' for site location matching. The default is 1.
- **merge_how** (*str, optional*) – String specifying how to merge variable data at site locations. The default is 'combine'. If both input variables are the same and *combine* is used, then the first input variable will be preferred over the other. Supported are 'combine', 'mean' and 'eval', for the latter, *merge_eval_fun* needs to be specified explicitly.
- **merge_eval_fun** (*str, optional*) – String specifying how *var1* and *var2* data should be evaluated (only relevant if *merge_how='eval'* is used) . The default is None. E.g. if one wants to retrieve the column aerosol fine mode fraction at 550nm (fmf550aer) through AERONET, this could be done through the SDA product by prodiding data_id1 and var1 are 'AeronetSDA' and 'od550aer' and second input data_id2 and var2 are 'AeronetSDA' and 'od550lt1aer' and merge_eval_fun could then be 'fmf550aer=(AeronetSDA;od550lt1aer/AeronetSDA;od550aer)*100'. Note that the input variables will be converted to their AeroCom default units, so the specification of *merge_eval_fun* should take that into account in case the originally read obsdata is not in default units.
- **var_name_out** (*str, optional*) – Name of output variable. Default is None, in which case it is attempted to be inferred.
- **data_id_out** (*str, optional*) – *data_id* set in output *StationData* objects. Default is None, in which case it is inferred from input data_ids (e.g. in above example of merge_eval_fun, the output data_id would be 'AeronetSDA' since both input IDs are the same.
- **var_unit_out** (*str*) – unit of output variable.
- **resample_how** (*str, optional*) – String specifying how temporal resampling should be done. The default is 'mean'.
- **min_num_obs** (*int or dict, optional*) – Minimum number of observations for temporal resampling. The default is None in which case pyaerocom default is used, which is available via pyaerocom.const.OBS_MIN_NUM_RESAMPLE.
- **add_meta_keys** (*list, optional*) – additional metadata keys to be added to output *StationData* objects from input data. If None, then only the pyaerocom default keys are added (see *StationData.STANDARD_META_KEYS*).

Raises

- **ValueError** – If input for *merge_how* or *match_stats_how* is invalid.
- **NotImplementedError** – If one of the input UngriddedData objects contains more than one dataset.

Returns
merged_stats – list of *StationData* objects containing the colocated and combined variable data.

Return type
list

4.4 Reading of gridded data

Gridded data specifies any dataset that can be represented and stored on a regular grid within a certain domain (e.g. lat, lon time), for instance, model output or level 3 satellite data, stored, for instance, as NetCDF files. In pyaerocom, the underlying data object is `GriddedData` and pyaerocom supports reading of such data for different file naming conventions.

4.4.1 Gridded data using AeroCom conventions

class pyaerocom.io.readgridded.**ReadGridded**(*data_id=None*, *data_dir=None*, *file_convention='aerocom3'*)

Class for reading gridded files using AeroCom file conventions

data_id
string ID for model or obsdata network (see e.g. Aerocom interface map plots lower left corner)

Type
str

data
imported data object

Type
GriddedData

data_dir
directory containing result files for this model

Type
str

start
start time for data import

Type
pandas.Timestamp

stop
stop time for data import

Type
pandas.Timestamp

file_convention
class specifying details of the file naming convention for the model

Type
FileConventionRead

files

list containing all filenames that were found. Filled, e.g. in `ReadGridded.get_model_files()`

Type
list

from_files

List of all netCDF files that were used to concatenate the current data cube (i.e. that can be based on certain matching settings such as var_name or time interval).

Type
list

ts_types

list of all sampling frequencies (e.g. hourly, daily, monthly) that were inferred from filenames (based on Aerocom file naming convention) of all files that were found

Type
list

vars

list containing all variable names (e.g. od550aer) that were inferred from filenames based on Aerocom model file naming convention

Type
list

years

list of available years as inferred from the filenames in the data directory.

Type
list

Parameters

- **data_id** (*str*) – string ID of model (e.g. "AATSR_SU_v4.3","CAM5.3-Oslo_CTRL2016")
- **data_dir** (*str, optional*) – directory containing data files. If provided, only this directory is considered for data files, else the input *data_id* is used to search for the corresponding directory.
- **file_convention** (*str*) – string ID specifying the file convention of this model (cf. installation file file_conventions.ini)
- **init** (*bool*) – if True, the model directory is searched (*search_data_dir()*) on instantiation and if it is found, all valid files for this model are searched using *search_all_files()*.

AUX_ADD_ARGS = {'concprcpoxn': {'prlim': 0.0001, 'prlim_set_under': nan, 'prlim_units': 'm d-1', 'ts_type': 'daily'}, 'concprcpoxs': {'prlim': 0.0001, 'prlim_set_under': nan, 'prlim_units': 'm d-1', 'ts_type': 'daily'}, 'concprcprdn': {'prlim': 0.0001, 'prlim_set_under': nan, 'prlim_units': 'm d-1', 'ts_type': 'daily'}}

Additional arguments passed to computation methods for auxiliary data This is optional and defined per-variable like in AUX_FUNS

AUX_ALT_VARS = {'ac550dryaer': ['ac550aer'], 'od440aer': ['od443aer'], 'od870aer': ['od865aer']}

AUX_FUNS = {'ang4487aer': <function compute_angstrom_coeff_cubes>, 'angabs4487aer': <function compute_angstrom_coeff_cubes>, 'conc*': <function multiply_cubes>, 'concNhno3': <function calc_concNhno3_from_vmr>, 'concNnh3': <function calc_concNnh3_from_vmr>, 'concNnh4': <function calc_concNnh4>, 'concNno3pm10': <function calc_concNno3pm10>, 'concNno3pm25': <function calc_concNno3pm25>, 'concNtnh': <function calc_concNtnh>, 'concNtno3': <function calc_concNtno3>, 'concno3': <function add_cubes>, 'concno3pm10': <function calc_concno3pm10>, 'concno3pm25': <function calc_concno3pm25>, 'concox': <function add_cubes>, 'concprcpoxn': <function compute_concprcp_from_pr_and_wetdep>, 'concprcpoxs': <function compute_concprcp_from_pr_and_wetdep>, 'concprcprdn': <function compute_concprcp_from_pr_and_wetdep>, 'concsspm10': <function add_cubes>, 'concsspm25': <function calc_sspm25>, 'dryoa': <function add_cubes>, 'fmf550aer': <function divide_cubes>, 'mmr*': <function mmr_from_vmr>, 'od550gt1aer': <function subtract_cubes>, 'sc550dryaer': <function subtract_cubes>, 'vmrox': <function add_cubes>, 'wetoa': <function add_cubes>}

AUX_REQUIRES = {'ang4487aer': ('od440aer', 'od870aer'), 'angabs4487aer': ('abs440aer', 'abs870aer'), 'conc*': ('mmr*', 'rho'), 'concNhno3': ('vmrhno3',), 'concNnh3': ('vmrnh3',), 'concNnh4': ('concnh4',), 'concNno3pm10': ('concno3f', 'concno3c'), 'concNno3pm25': ('concno3f', 'concno3c'), 'concNtnh': ('concnh4', 'vmrnh3'), 'concNtno3': ('concno3f', 'concno3c', 'vmrhno3'), 'concno3': ('concno3c', 'concno3f'), 'concno3pm10': ('concno3f', 'concno3c'), 'concno3pm25': ('concno3f', 'concno3c'), 'concox': ('concno2', 'conco3'), 'concprcpoxn': ('wetoxn', 'pr'), 'concprcpoxs': ('wetoxs', 'pr'), 'concprcprdn': ('wetrdn', 'pr'), 'concsspm10': ('concss25', 'concsscoarse'), 'concsspm25': ('concss25', 'concsscoarse'), 'dryoa': ('drypoa', 'drysoa'), 'fmf550aer': ('od550lt1aer', 'od550aer'), 'mmr*': ('vmr*',), 'od550gt1aer': ('od550aer', 'od550lt1aer'), 'rho': ('ts', 'ps'), 'sc550dryaer': ('ec550dryaer', 'ac550dryaer'), 'vmrox': ('vmrno2', 'vmro3'), 'wetoa': ('wetpoa', 'wetsoa')}

CONSTRAINT_OPERATORS = {'!=': <ufunc 'not_equal'>, '<': <ufunc 'less'>, '<=': <ufunc 'less_equal'>, '==': <ufunc 'equal'>, '>': <ufunc 'greater'>, '>=': <ufunc 'greater_equal'>}

property TS_TYPES

List with valid filename encryptions specifying temporal resolution

Update 7.11.2019: not in use anymore due to improved handling of all possible frequencies now using TsType class.

VERT_ALT = {'Surface': 'ModelLevel'}

add_aux_compute(*var_name*, *vars_required*, *fun*)

Register new variable to be computed

Parameters

- **var_name** (*str*) – variable name to be computed
- **vars_required** (*list*) – list of variables to read, that are required to compute *var_name*
- **fun** (*callable*) – function that takes a list of *GriddedData* objects as input and that are read using variable names specified by *vars_required*.

apply_read_constraint(*data*, *constraint*, ***kwargs*)

Filter a *GriddeData* object by value in another variable

Note: BETA version, that was hacked down in a rush to be able to apply AOD>0.1 threshold when reading AE.

Parameters

- **data** (GriddedData) – data object to which constraint is applied
- **constraint** (*dict*) – dictionary defining read constraint (see *check_constraint_valid()* for minimum requirement). If constraint contains key var_name (not mandatory), then the corresponding variable is attemted to be read and is used to evaluate constraint and the corresponding boolean mask is then applied to input *data*. Wherever this mask is True (i.e. constraint is met), the current value in input *data* will be replaced with *numpy.ma.masked* or, if specified, with entry *new_val* in input constraint dict.
- ****kwargs** (*TYPE*) – reading arguments in case additional variable data needs to be loaded, to determine filter mask (i.e. if *var_name* is specified in input constraint). Parse to *read_var()*.

Raises

ValueError – If constraint is invalid (cf. *check_constraint_valid()* for details).

Returns

modified data objects (all grid-points that met constraint are replaced with either *numpy.ma.masked* or with a value that can be specified via key *new_val* in input constraint).

Return type

GriddedData

browser

This object can be used to

check_compute_var(*var_name*)

Check if variable name belongs to family that can be computed

For instance, if input var_name is *concdust* this method will check *AUX_REQUIRES* to see if there is a variable family pattern (*conc**) defined that specifies how to compute these variables. If a match is found, the required variables and computation method is added via *add_aux_compute()*.

Parameters

var_name (*str*) – variable name to be checked

Returns

True if match is found, else False

Return type

bool

check_constraint_valid(*constraint*)

Check if reading constraint is valid

Parameters

constraint (*dict*) – reading constraint. Requires at lest entries for following keys: - operator (str): for valid operators see *CONSTRAINT_OPERATORS* - filter_val (float): value against which data is evaluated wrt to operator

Raises

ValueError – If constraint is invalid

Return type
None.

compute_var(*var_name*, *start=None*, *stop=None*, *ts_type=None*, *experiment=None*, *vert_which=None*, *flex_ts_type=True*, *prefer_longer=False*, *vars_to_read=None*, *aux_fun=None*, *try_convert_units=True*, *aux_add_args=None*, *rename_var=None*, ***kwargs*)

Compute auxiliary variable

Like *read_var()* but for auxiliary variables (cf. AUX_REQUIRES)

Parameters

- **var_name** (*str*) – variable that are supposed to be read
- **start** (*Timestamp or str, optional*) – start time of data import (if valid input, then the current *start* will be overwritten)
- **stop** (*Timestamp or str, optional*) – stop time of data import
- **ts_type** (*str*) – string specifying temporal resolution (choose from hourly, 3hourly, daily, monthly). If None, prioritised of the available resolutions is used
- **experiment** (*str*) – name of experiment (only relevant if this dataset contains more than one experiment)
- **vert_which** (*str*) – valid AeroCom vertical info string encoded in name (e.g. Column, ModelLevel)
- **flex_ts_type** (*bool*) – if True and if applicable, then another ts_type is used in case the input ts_type is not available for this variable
- **prefer_longer** (*bool*) – if True and applicable, the ts_type resulting in the longer time coverage will be preferred over other possible frequencies that match the query.
- **try_convert_units** (*bool*) – if True, units of GriddedData objects are attempted to be converted to AeroCom default. This applies both to the GriddedData objects being read for computation as well as the variable computed from the forme objects. This is, for instance, useful when computing concentration in precipitation from wet deposition and precipitation amount.
- **rename_var** (*str*) – if this is set, the *var_name* attribute of the output *GriddedData* object will be updated accordingly.
- ****kwargs** – additional keyword args passed to `_load_var()`

Returns
loaded data object

Return type
GriddedData

concatenate_cubes(*cubes*)

Concatenate list of cubes into one cube

Parameters
CubeList – list of individual cubes

Returns
Single cube that contains concatenated cubes from input list

Return type
Cube

Raises
iris.exceptions.ConcatenateError – if concatenation of all cubes failed

property data_dir: str

Directory where data files are located

property data_id: str

Data ID of dataset

property experiments: list

List of all experiments that are available in this dataset

property file_type

File type of data files

property files: list

List of data files

filter_files(*var_name=None*, *ts_type=None*, *start=None*, *stop=None*, *experiment=None*, *vert_which=None*, *is_at_stations=False*, *df=None*)

Filter file database

Parameters

- **var_name** (*str*) – variable that are supposed to be read
- **ts_type** (*str*) – string specifying temporal resolution (choose from "hourly", "3hourly", "daily", "monthly"). If None, prioritised of the available resolutions is used
- **start** (*Timestamp or str, optional*) – start time of data import
- **stop** (*Timestamp or str, optional*) – stop time of data import
- **experiment** (*str*) – name of experiment (only relevant if this dataset contains more than one experiment)
- **vert_which** (*str or dict, optional*) – valid AeroCom vertical info string encoded in name (e.g. Column, ModelLevel) or dictionary containing var_name as key and vertical coded string as value, accordingly
- **flex_ts_type** (*bool*) – if True and if applicable, then another ts_type is used in case the input ts_type is not available for this variable
- **prefer_longer** (*bool*) – if True and applicable, the ts_type resulting in the longer time coverage will be preferred over other possible frequencies that match the query.

filter_query(*var_name*, *ts_type=None*, *start=None*, *stop=None*, *experiment=None*, *vert_which=None*, *is_at_stations=False*, *flex_ts_type=True*, *prefer_longer=False*)

Filter files for read query based on input specs

Returns
dataframe containing filtered dataset

Return type
DataFrame

find_common_ts_type(*vars_to_read*, *start=None*, *stop=None*, *ts_type=None*, *experiment=None*, *vert_which=None*, *flex_ts_type=True*)

Find common ts_type for list of variables to be read

Parameters

- **vars_to_read** (*list*) – list of variables that is supposed to be read
- **start** (*Timestamp or str, optional*) – start time of data import (if valid input, then the current start will be overwritten)
- **stop** (*Timestamp or str, optional*) – stop time of data import (if valid input, then the current *start* will be overwritten)
- **ts_type** (*str*) – string specifying temporal resolution (choose from hourly, 3hourly, daily, monthly). If None, prioritised of the available resolutions is used
- **experiment** (*str*) – name of experiment (only relevant if this dataset contains more than one experiment)
- **vert_which** (*str*) – valid AeroCom vertical info string encoded in name (e.g. Column, ModelLevel)
- **flex_ts_type** (*bool*) – if True and if applicable, then another ts_type is used in case the input ts_type is not available for this variable

Returns
common ts_type for input variable

Return type
str

Raises
DataCoverageError – if no match can be found

get_files(*var_name*, *ts_type=None*, *start=None*, *stop=None*, *experiment=None*, *vert_which=None*, *is_at_stations=False*, *flex_ts_type=True*, *prefer_longer=False*)

Get data files based on input specs

get_var_info_from_files() → dict

Creates dicitonary that contains variable specific meta information

Returns
dictionary where keys are available variables and values (for each variable) contain information about available ts_types, years, etc.

Return type
dict

has_var(*var_name*)

Check if variable is available

Parameters
var_name (*str*) – variable to be checked

Return type
bool

property name

Deprecated name of attribute data_id

read(*vars_to_retrieve=None*, *start=None*, *stop=None*, *ts_type=None*, *experiment=None*, *vert_which=None*, *flex_ts_type=True*, *prefer_longer=False*, *require_all_vars_avail=False*, ***kwargs*)

Read all variables that could be found

Reads all variables that are available (i.e. in *vars_filename*)

Parameters

- **vars_to_retrieve** (*list or str, optional*) – variables that are supposed to be read. If None, all variables that are available are read.
- **start** (*Timestamp or str, optional*) – start time of data import
- **stop** (*Timestamp or str, optional*) – stop time of data import
- **ts_type** (*str, optional*) – string specifying temporal resolution (choose from "hourly", "3hourly", "daily", "monthly"). If None, prioritised of the available resolutions is used
- **experiment** (*str*) – name of experiment (only relevant if this dataset contains more than one experiment)
- **vert_which** (*str or dict, optional*) – valid AeroCom vertical info string encoded in name (e.g. Column, ModelLevel) or dictionary containing var_name as key and vertical coded string as value, accordingly
- **flex_ts_type** (*bool*) – if True and if applicable, then another ts_type is used in case the input ts_type is not available for this variable
- **prefer_longer** (*bool*) – if True and applicable, the ts_type resulting in the longer time coverage will be preferred over other possible frequencies that match the query.
- **require_all_vars_avail** (*bool*) – if True, it is strictly required that all input variables are available.
- ****kwargs** – optional and support for deprecated input args

Returns
loaded data objects (type `GriddedData`)

Return type
tuple

Raises

- **IOError** – if input variable names is not list or string
- ***VarNotAvailableError*** –
  1. if `require_all_vars_avail=True` and one or more of the desired variables is not available in this class 2. if `require_all_vars_avail=True` and if none of the input variables is available in this object

read_var(*var_name*, *start=None*, *stop=None*, *ts_type=None*, *experiment=None*, *vert_which=None*, *flex_ts_type=True*, *prefer_longer=False*, *aux_vars=None*, *aux_fun=None*, *constraints=None*, *try_convert_units=True*, *rename_var=None*, ***kwargs*)

Read model data for a specific variable

This method searches all valid files for a given variable and for a provided temporal resolution (e.g. *daily, monthly*), optionally within a certain time window, that may be specified on class instantiation or using the corresponding input parameters provided in this method.

The individual NetCDF files for a given temporal period are loaded as instances of the `iris.Cube` object and appended to an instance of the `iris.cube.CubeList` object. The latter is then used to concatenate the individual cubes in time into a single instance of the `pyaerocom.GriddedData` class. In order to ensure that this works, several things need to be ensured, which are listed in the following and which may be controlled within the global settings for NetCDF import using the attribute `GRID_IO` (instance of `OnLoad`) in the default instance of the *pyaerocom.config.Config* object accessible via `pyaerocom.const`.

Parameters

- **var_name** (*str*) – variable that are supposed to be read

- **start** (*Timestamp or str, optional*) – start time of data import
- **stop** (*Timestamp or str, optional*) – stop time of data import
- **ts_type** (*str*) – string specifying temporal resolution (choose from "hourly", "3hourly", "daily", "monthly"). If None, prioritised of the available resolutions is used
- **experiment** (*str*) – name of experiment (only relevant if this dataset contains more than one experiment)
- **vert_which** (*str or dict, optional*) – valid AeroCom vertical info string encoded in name (e.g. Column, ModelLevel) or dictionary containing var_name as key and vertical coded string as value, accordingly
- **flex_ts_type** (*bool*) – if True and if applicable, then another ts_type is used in case the input ts_type is not available for this variable
- **prefer_longer** (*bool*) – if True and applicable, the ts_type resulting in the longer time coverage will be preferred over other possible frequencies that match the query.
- **aux_vars** (*list*) – only relevant if *var_name* is not available for reading but needs to be computed: list of variables that are required to compute *var_name*
- **aux_fun** (*callable*) – only relevant if *var_name* is not available for reading but needs to be computed: custom method for computation (cf. *add_aux_compute()* for details)
- **constraints** (*list, optional*) – list of reading constraints (dict type). See *check_constraint_valid()* and *apply_read_constraint()* for details related to format of the individual constraints.
- **try_convert_units** (*bool*) – if True, then the unit of the variable data is checked against AeroCom default unit for that variable and if it deviates, it is attempted to be converted to the AeroCom default unit. Default is True.
- **rename_var** (*str*) – if this is set, the *var_name* attribute of the output *GriddedData* object will be updated accordingly.
- ****kwargs** – additional keyword args parsed to `_load_var()`

Returns
loaded data object

Return type
GriddedData

Raises

- **AttributeError** – if none of the ts_types identified from file names is valid
- ***VarNotAvailableError*** – if specified ts_type is not supported

property registered_var_patterns

List of string patterns for computation of variables

The information is extracted from *AUX_REQUIRES*

Returns
list of variable patterns

Return type
list

reinit()

Reinit everything that is loaded specific to data_dir

search_all_files(*update_file_convention=True*)

Search all valid model files for this model

This method browses the data directory and finds all valid files, that is, file that are named according to one of the aerocom file naming conventions. The file list is stored in *files*.

Note: It is presumed, that naming conventions of files in the data directory are not mixed but all correspond to either of the conventions defined in

Parameters

update_file_convention (*bool*) – if True, the first file in *data_dir* is used to identify the file naming convention (cf. `FileConventionRead`)

Raises

DataCoverageError – if no valid files could be found

search_data_dir()

Search data directory based on model ID

Wrapper for method `search_data_dir_aerocom()`

Returns

data directory

Return type

str

Raises

IOError – if directory cannot be found

property start

First available year in the dataset (inferred from filenames)

Note: This is not variable or ts_type specific, so it is not necessarily given that data from this year is available for all variables in *vars* or all frequencies liste in *ts_types*

property stop

Last available year in the dataset (inferred from filenames)

Note: This is not variable or ts_type specific, so it is not necessarily given that data from this year is available for all variables in *vars* or all frequencies liste in *ts_types*

property ts_types

Available frequencies

update(***kwargs*)

Update one or more valid parameters

Parameters

****kwargs** – keyword args that will be used to update (overwrite) valid class attributes such as *data, data_dir, files*

property vars

property vars_filename

property vars_provided

Variables provided by this dataset

property years_avail: list

Years available in dataset

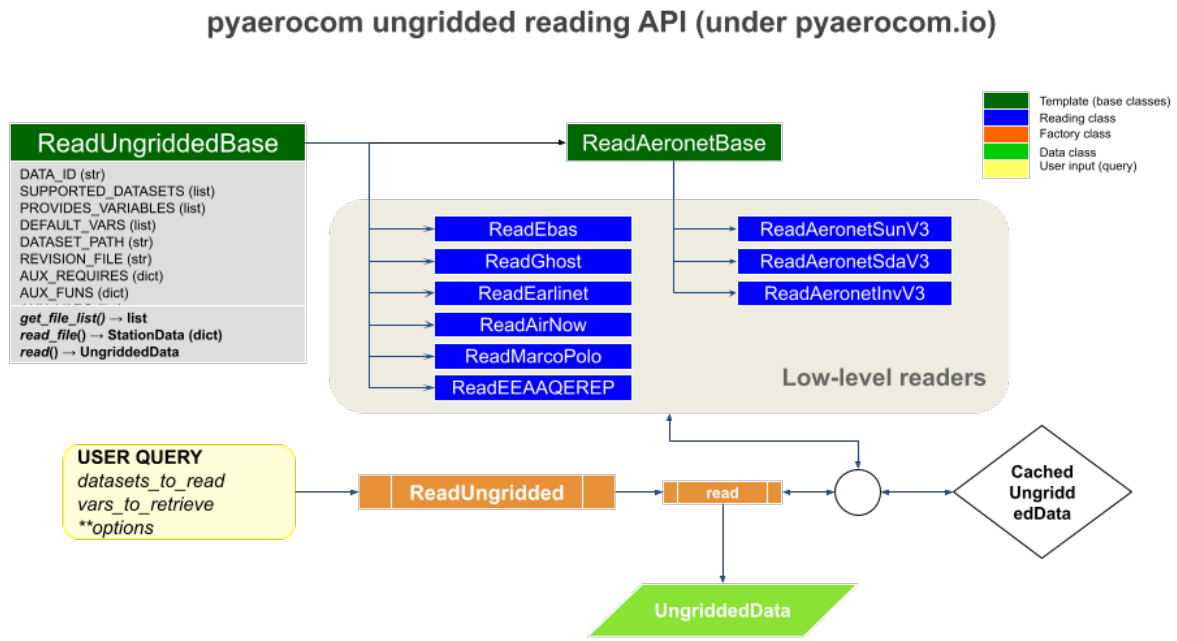
pyaerocom.io.readgridded.**is_3d**(*var_name*)

4.4.2 Gridded data using EMEP conventions

4.5 Reading of ungridded data

Other than gridded data, ungridded data represents data that is irregularly sampled in space and time, for instance, observations at different locations around the globe. Such data is represented in pyaerocom by *UngriddedData* which is essentially a point-cloud dataset. Reading of *UngriddedData* is typically specific for different observational data records, as they typically come in various data formats using various metadata conventions, which need to be harmonised, which is done during the data import.

The following flowchart illustrates the architecture of ungridded reading in pyaerocom. Below are information about the individual reading classes for each dataset (blue in flowchart), the abstract template base classes the reading classes are based on (dark green) and the factory class *ReadUngridded* (orange) which has registered all individual reading classes. The data classes that are returned by the reading class are indicated in light green.



4.5.1 ReadUngridded factory class

Factory class that has all reading class for the individual datasets registered.

class pyaerocom.io.readungridded.**ReadUngridded**(*data_ids=None*, *ignore_cache=False*, *data_dirs=None*, *configs: PyaroConfig | list[PyaroConfig] | None = None*)

Factory class for reading of ungridded data based on obsnetwork ID

This class also features reading functionality that goes beyond reading of inidividual observation datasets; including, reading of multiple datasets and post computation of new variables based on datasets that can be read.

Parameters
SOON (*COMING*)

DONOTCACHE_NAME = 'DONOTCACHE'

property INCLUDED_DATASETS

INCLUDED_READERS = [<class 'pyaerocom.io.read_aeronet_invv3.ReadAeronetInvV3'>, <class 'pyaerocom.io.read_aeronet_sdav3.ReadAeronetSdaV3'>, <class 'pyaerocom.io.read_aeronet_sunv3.ReadAeronetSunV3'>, <class 'pyaerocom.io.read_earlinet.ReadEarlinet'>, <class 'pyaerocom.io.read_ebas.ReadEbas'>, <class 'pyaerocom.io.read_aasetal.ReadAasEtal'>, <class 'pyaerocom.io.read_airnow.ReadAirNow'>, <class 'pyaerocom.io.read_eea_aqerep.ReadEEAAQEREP'>, <class 'pyaerocom.io.read_eea_aqerep_v2.ReadEEAAQEREP_V2'>, <class 'pyaerocom.io.cams2_83.read_obs.ReadCAMS2_83'>, <class 'pyaerocom.io.gaw.reader.ReadGAW'>, <class 'pyaerocom.io.ghost.reader.ReadGhost'>, <class 'pyaerocom.io.mep.reader.ReadMEP'>, <class 'pyaerocom.io.icos.reader.ReadICOS'>, <class 'pyaerocom.io.icpforests.reader.ReadICPForest'>]

property SUPPORTED_DATASETS

Returns list of strings containing all supported dataset names

SUPPORTED_READERS = [<class 'pyaerocom.io.read_aeronet_invv3.ReadAeronetInvV3'>, <class 'pyaerocom.io.read_aeronet_sdav3.ReadAeronetSdaV3'>, <class 'pyaerocom.io.read_aeronet_sunv3.ReadAeronetSunV3'>, <class 'pyaerocom.io.read_earlinet.ReadEarlinet'>, <class 'pyaerocom.io.read_ebas.ReadEbas'>, <class 'pyaerocom.io.read_aasetal.ReadAasEtal'>, <class 'pyaerocom.io.read_airnow.ReadAirNow'>, <class 'pyaerocom.io.read_eea_aqerep.ReadEEAAQEREP'>, <class 'pyaerocom.io.read_eea_aqerep_v2.ReadEEAAQEREP_V2'>, <class 'pyaerocom.io.cams2_83.read_obs.ReadCAMS2_83'>, <class 'pyaerocom.io.gaw.reader.ReadGAW'>, <class 'pyaerocom.io.ghost.reader.ReadGhost'>, <class 'pyaerocom.io.mep.reader.ReadMEP'>, <class 'pyaerocom.io.icos.reader.ReadICOS'>, <class 'pyaerocom.io.icpforests.reader.ReadICPForest'>, <class 'pyaerocom.io.pyaro.read_pyaro.ReadPyaro'>]

add_config(*config: PyaroConfig*) → None

Adds single PyaroConfig to self.configs

Parameters
config (*PyaroConfig*)

Raises
ValueError – If config is not PyaroConfig

add_pyaro_reader(*config: PyaroConfig*) → *ReadUngriddedBase*

property configs

List configs

property data_dirs

Data directory(ies) for dataset(s) to read (keys are data IDs)

Type
dict

property data_id

ID of dataset

Note: Only works if exactly one dataset is assigned to the reader, that is, length of *data_ids* is 1.

Raises
AttributeError – if number of items in *data_ids* is unequal one.

Returns
data ID

Return type
str

property data_ids

List of datasets supposed to be read

dataset_provides_variables(*data_id=None*)

List of variables provided by a certain dataset

get_lowlevel_reader(*data_id: str | None = None*) → *ReadUngriddedBase*

Helper method that returns initiated reader class for input ID

Parameters
data_id (*str*) – Name of dataset

Returns
instance of reading class (needs to be implementation of base class ReadUngriddedBase).

Return type
ReadUngriddedBase

get_reader(*data_id*)

get_vars_supported(*obs_id*, *vars_desired*)

Filter input list of variables by supported ones for a certain data ID

Parameters

- **obs_id** (*str*) – ID of observation network
- **vars_desired** (*list*) – List of variables that are desired

Returns
list of variables that can be read through the input network

Return type
list

property ignore_cache

Boolean specifying whether caching is active or not

property post_compute

Information about datasets that can be computed in post

read(*data_ids=None*, *vars_to_retrieve=None*, *only_cached=False*, *filter_post=None*, *configs: PyaroConfig | list[PyaroConfig] | None = None*, ***kwargs*)

Read observations

Iter over all datasets in `data_ids`, call `read_dataset()` and append to data object

Parameters

- **data_ids** (*str or list*) – data ID or list of all datasets to be imported
- **vars_to_retrieve** (*str or list*) – variable or list of variables to be imported
- **only_cached** (*bool*) – if True, then nothing is reloaded but only data is loaded that is available as cached objects (not recommended to use but may be used if working offline without connection to database)
- **filter_post** (*dict, optional*) – filters applied to *UngriddedData* object AFTER it is read into memory, via `UngriddedData.apply_filters()`. This option was introduced in pyaerocom version 0.10.0 and should be used preferably over **kwargs. There is a certain flexibility with respect to how these filters can be defined, for instance, sub dicts for each *data_id*. The most common way would be to provide directly the input needed for *UngriddedData.apply_filters*. If you want to read multiple variables from one or more datasets, and if you want to apply variable specific filters, it is recommended to read the data individually for each variable and corresponding set of filters and then merge the individual filtered *UngriddedData* objects afterwards, e.g. using *data_var1 & data_var2*.
- ****kwargs** – Additional input options for reading of data, which are applied WHILE the data is read. If any such additional options are provided that are applied during the reading, then automatic caching of the output *UngriddedData* object will be deactivated. Thus, it is recommended to handle data filtering via *filter_post* argument whenever possible, which will result in better performance as the unconstrained original data is read in and cached, and then the filtering is applied.

Example

```
>>> import pyaerocom.io.readungridded as pio
>>> from pyaerocom import const
>>> obj = pio.ReadUngridded(data_id=const.AERONET_SUN_V3L15_AOD_ALL_POINTS_NAME)
>>> obj.read()
>>> print(obj)
>>> print(obj.metadata[0.]['latitude'])
```

read_dataset(*data_id*, *vars_to_retrieve=None*, *only_cached=False*, *filter_post=None*, ***kwargs*)

Read dataset into an instance of `ReadUngridded`

Parameters

- **data_id** (*str*) – name of dataset

- **vars_to_retrieve** (*list*) – variable or list of variables to be imported
- **only_cached** (*bool*) – if True, then nothing is reloaded but only data is loaded that is available as cached objects (not recommended to use but may be used if working offline without connection to database)
- **filter_post** (*dict, optional*) – filters applied to *UngriddedData* object AFTER it is read into memory, via `UngriddedData.apply_filters()`. This option was introduced in pyaerocom version 0.10.0 and should be used preferably over **kwargs. There is a certain flexibility with respect to how these filters can be defined, for instance, sub dicts for each *data_id*. The most common way would be to provide directly the input needed for *UngriddedData.apply_filters*. If you want to read multiple variables from one or more datasets, and if you want to apply variable specific filters, it is recommended to read the data individually for each variable and corresponding set of filters and then merge the individual filtered *UngriddedData* objects afterwards, e.g. using *data_var1 & data_var2*.
- ****kwargs** – Additional input options for reading of data, which are applied WHILE the data is read. If any such additional options are provided that are applied during the reading, then automatic caching of the output *UngriddedData* object will be deactivated. Thus, it is recommended to handle data filtering via *filter_post* argument whenever possible, which will result in better performance as the unconstrained original data is read in and cached, and then the filtering is applied.

Returns
data object

Return type
UngriddedData

read_dataset_post(*data_id*, *vars_to_retrieve*, *only_cached=False*, *filter_post=None*, ***kwargs*)

Read dataset into an instance of *ReadUngridded*

Parameters

- **data_id** (*str*) – name of dataset
- **vars_to_retrieve** (*list*) – variable or list of variables to be imported
- **only_cached** (*bool*) – if True, then nothing is reloaded but only data is loaded that is available as cached objects (not recommended to use but may be used if working offline without connection to database)
- **filter_post** (*dict, optional*) – filters applied to *UngriddedData* object AFTER it is read into memory, via `UngriddedData.apply_filters()`. This option was introduced in pyaerocom version 0.10.0 and should be used preferably over **kwargs. There is a certain flexibility with respect to how these filters can be defined, for instance, sub dicts for each *data_id*. The most common way would be to provide directly the input needed for *UngriddedData.apply_filters*. If you want to read multiple variables from one or more datasets, and if you want to apply variable specific filters, it is recommended to read the data individually for each variable and corresponding set of filters and then merge the individual filtered *UngriddedData* objects afterwards, e.g. using *data_var1 & data_var2*.
- ****kwargs** – Additional input options for reading of data, which are applied WHILE the data is read. If any such additional options are provided that are applied during the reading, then automatic caching of the output *UngriddedData* object will be deactivated. Thus, it is recommended to handle data filtering via *filter_post* argument whenever possible, which will result in better performance as the unconstrained original data is read in and cached, and then the filtering is applied.

Returns
data object

Return type
UngriddedData

property supported_datasets

Wrapper for *SUPPORTED_DATASETS*

4.5.2 ReadUngriddedBase template class

All ungridded reading routines are based on this template class.

class pyaerocom.io.readungriddedbase.**ReadUngriddedBase**(*data_id: str | None = None*, *data_dir: str | None = None*)

TEMPLATE: Abstract base class template for reading of ungridded data

---

Note: The two dictionaries `AUX_REQUIRES` and `AUX_FUNS` can be filled with variables that are not contained in the original data files but are computed during the reading. The former specifies what additional variables are required to perform the computation and the latter specifies functions used to perform the computations of the auxiliary variables. See, for instance, the class `ReadAeronetSunV3`, which includes the computation of the AOD at 550nm and the Angstrom coefficient (in 440-870 nm range) from AODs measured at other wavelengths.

AUX_FUNS = {}

Functions that are used to compute additional variables (i.e. one for each variable defined in AUX_REQUIRES)

AUX_REQUIRES = {}

dictionary containing information about additionally required variables for each auxiliary variable (i.e. each variable that is not provided by the original data but computed on import)

property AUX_VARS

List of auxiliary variables (keys of attr. *AUX_REQUIRES*)

Auxiliary variables are those that are not included in original files but are computed from other variables during import

property DATASET_PATH

Wrapper for *data_dir*.

abstract property DATA_ID

Name of dataset (OBS_ID)

Note:

- May be implemented as global constant in header of derieved class
- May be multiple that can be specified on init (see example below)

---

abstract property DEFAULT_VARS

List containing default variables to read

IGNORE_META_KEYS = []

abstract property PROVIDES_VARIABLES

List of variables that are provided by this dataset

Note: May be implemented as global constant in header

---

property REVISION_FILE

Name of revision file located in data directory

abstract property SUPPORTED_DATASETS

List of all datasets supported by this interface

Note:

- best practice to specify in header of class definition
- needless to mention that *DATA_ID* needs to be in this list

---

abstract property TS_TYPE

Temporal resolution of dataset

This should be defined in the header of an implementation class if it can be globally defined for the corresponding obs-network or in other cases it should be initated as string `undefined` and then, if applicable, updated in the reading routine of a file.

The TS_TYPE information should ultimately be written into the meta-data of objects returned by the implementation of *read_file()* (e.g. instance of `StationData` or a normal dictionary) and the method *read()* (which should ALWAYS return an instance of the `UngriddedData` class).

Note:

- Please use `"undefined"` if the derived class is not sampled on a regular basis.
- If applicable please use Aerocom ts_type (i.e. hourly, 3hourly, daily, monthly, yearly)
- Note also, that the ts_type in a derived class may or may not be defined in a general case. For instance, in the EBAS database the resolution code can be found in the file header and may thus be intiated as `"undefined"` in the initiation of the reading class and then updated when the class is being read
- For derived implementation classes that support reading of multiple network versions, you may also assign

---

check_vars_to_retrieve(*vars_to_retrieve*)

Separate variables that are in file from those that are computed

Some of the provided variables by this interface are not included in the data files but are computed within this class during data import (e.g. od550aer, ang4487aer).

The latter may require additional parameters to be retrieved from the file, which is specified in the class header (cf. attribute `AUX_REQUIRES`).

This function checks the input list that specifies all required variables and separates them into two lists, one that includes all variables that can be read from the files and a second list that specifies all variables that are computed in this class.

Parameters

vars_to_retrieve (*list*) – all parameter names that are supposed to be loaded

Returns

2-element tuple, containing

- list: list containing all variables to be read
- list: list containing all variables to be computed

Return type
tuple

`compute_additional_vars`(*data*, *vars_to_compute*)

Compute all additional variables

The computations for each additional parameter are done using the specified methods in `AUX_FUNS`.

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_compute** (*list*) – list of variable names that are supposed to be computed. Variables that are required for the computation of the variables need to be specified in *AUX_VARS* and need to be available as data vectors in the provided data dictionary (key is the corresponding variable name of the required variable).

Returns
updated data object now containing also computed variables

Return type
dict

`property data_dir: str`

Location of the dataset

Note: This can be set explicitly when instantiating the class (e.g. if data is available on local machine). If unspecified, the data location is attempted to be inferred via `get_obsnetwork_dir()`

Raises
FileNotFoundError – if data directory does not exist or cannot be retrieved automatically

Type
str

`property data_id`

ID of dataset

`property data_revision`

Revision string from file Revision.txt in the main data directory

`find_in_file_list`(*pattern=None*)

Find all files that match a certain wildcard pattern

Parameters
pattern (str, optional) – wildcard pattern that may be used to narrow down the search (e.g. use `pattern=*Berlin*` to find only files that contain Berlin in their filename)

Returns
list containing all files in `files` that match pattern

Return type
list

Raises
IOError – if no matches can be found

get_file_list(*pattern=None*)

Search all files to be read

Uses _FILEMASK (+ optional input search pattern, e.g. station_name) to find valid files for query.

Parameters
pattern (*str, optional*) – file name pattern applied to search

Returns
list containing retrieved file locations

Return type
list

Raises
IOError – if no files can be found

logger

Class own instance of logger class

abstract read(*vars_to_retrieve=None*, *files=[]*, *first_file=None*, *last_file=None*)

Method that reads list of files as instance of UngriddedData

Parameters

- **vars_to_retrieve** (list or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded
- **files** (list, optional) – list of files to be read. If None, then the file list is used that is returned on *get_file_list()*.
- **first_file** (int, optional) – index of first file in file list to read. If None, the very first file in the list is used
- **last_file** (int, optional) – index of last file in list to read. If None, the very last file in the list is used

Returns
instance of ungridded data object containing data from all files.

Return type
UngriddedData

abstract read_file(*filename*, *vars_to_retrieve=None*)

Read single file

Parameters

- **filename** (*str*) – string specifying filename
- **vars_to_retrieve** (list or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded

Returns
imported data in a suitable format that can be handled by *read()* which is supposed to append the loaded results from this method (which reads one datafile) to an instance of UngriddedData for all files.

Return type
dict or StationData, or other...

read_first_file(***kwargs*)

Read first file returned from *get_file_list()*

Note: This method may be used for test purposes.

Parameters
****kwargs** – keyword args passed to *read_file()* (e.g. vars_to_retrieve)

Returns
dictionary or similar containing loaded results from first file

Return type
dict-like

read_station(*station_id_filename*, ***kwargs*)

Read data from a single station into UngriddedData

Find all files that contain the station ID in their filename and then call *read()*, providing the reduced filelist as input, in order to read all files from this station into data object.

Parameters

- **station_id_filename** (*str*) – name of station (MUST be encrypted in filename)
- ****kwargs** – additional keyword args passed to *read()* (e.g. vars_to_retrieve)

Returns
loaded data

Return type
UngriddedData

Raises
IOError – if no files can be found for this station ID

remove_outliers(*data*, *vars_to_retrieve*, ***valid_rng_vars*)

Remove outliers from data

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param vars_to_compute)
- **vars_to_retrieve** (*list*) – list of variable names for which outliers will be removed from data
- ****valid_rng_vars** – additional keyword args specifying variable name and corresponding min / max interval (list or tuple) that specifies valid range for the variable. For each variable that is not explicitely defined here, the default minimum / maximum value is used (accessed via pyaerocom.const.VARS[var_name])

var_supported(*var_name*)

Check if input variable is supported

Parameters
var_name (*str*) – AeroCom variable name or alias

Raises
VariableDefinitionError – if input variable is not supported by pyaerocom

Returns
True, if variable is supported by this interface, else False

Return type
bool

property verbosity_level

Current level of verbosity of logger

4.5.3 AERONET

Aerosol Robotic Network (AERONET)

AERONET base class

All AERONET reading classes are based on the template `ReadAeronetBase` class which, in turn inherits from *ReadUngriddedBase*.

class pyaerocom.io.readaeronetbase.**ReadAeronetBase**(*data_id=None*, *data_dir=None*)

Bases: *ReadUngriddedBase*

TEMPLATE: Abstract base class template for reading of Aeronet data

Extended abstract base class, derived from low-level base class `ReadUngriddedBase` that contains some more functionality.

ALT_VAR_NAMES_FILE = {}

dictionary specifying alternative column names for variables defined in *VAR_NAMES_FILE*

Type
OPTIONAL

AUX_FUNS = {}

Functions that are used to compute additional variables (i.e. one for each variable defined in AUX_REQUIRES)

AUX_REQUIRES = {}

dictionary containing information about additionally required variables for each auxiliary variable (i.e. each variable that is not provided by the original data but computed on import)

property AUX_VARS

List of auxiliary variables (keys of attr. *AUX_REQUIRES*)

Auxiliary variables are those that are not included in original files but are computed from other variables during import

COL_DELIM = ','

column delimiter in data block of files

property DATASET_PATH

Wrapper for *data_dir*.

abstract property DATA_ID

Name of dataset (OBS_ID)

Note:

- May be implemented as global constant in header of derieved class
- May be multiple that can be specified on init (see example below)

DEFAULT_UNIT = '1'

Default data unit that is assigned to all variables that are not specified in UNITS dictionary (cf. *UNITS*)

abstract property DEFAULT_VARS

List containing default variables to read

IGNORE_META_KEYS = ['date', 'time', 'day_of_year']

INSTRUMENT_NAME = 'sun_photometer'

name of measurement instrument

META_NAMES_FILE = {}

dictionary specifying the file column names (values) for each metadata key (cf. attributes of `StationData`, e.g. 'station_name', 'longitude', 'latitude', 'altitude')

META_NAMES_FILE_ALT = ({},)

abstract property PROVIDES_VARIABLES

List of variables that are provided by this dataset

Note: May be implemented as global constant in header

property REVISION_FILE

Name of revision file located in data directory

abstract property SUPPORTED_DATASETS

List of all datasets supported by this interface

Note:

- best practice to specify in header of class definition
- needless to mention that *DATA_ID* needs to be in this list

property TS_TYPE

Default implementation of string for temporal resolution

TS_TYPES = {}

dictionary assigning temporal resolution flags for supported datasets that are provided in a defined temporal resolution. Key is the name of the dataset and value is the corresponding ts_type

UNITS = {}

Variable specific units, only required for variables that deviate from *DEFAULT_UNIT* (is irrelevant for all variables that are so far supported by the implemented Aeronet products, i.e. all variables are dimensionless as specified in *DEFAULT_UNIT*)

VAR_NAMES_FILE = {}

dictionary specifying the file column names (values) for each Aerocom variable (keys)

VAR_PATTERNS_FILE = {}

Mappings for identifying variables in file (may be specified in addition to explicit variable names specified in VAR_NAMES_FILE)

check_vars_to_retrieve(*vars_to_retrieve*)

Separate variables that are in file from those that are computed

Some of the provided variables by this interface are not included in the data files but are computed within this class during data import (e.g. od550aer, ang4487aer).

The latter may require additional parameters to be retrieved from the file, which is specified in the class header (cf. attribute `AUX_REQUIRES`).

This function checks the input list that specifies all required variables and separates them into two lists, one that includes all variables that can be read from the files and a second list that specifies all variables that are computed in this class.

Parameters
vars_to_retrieve (*list*) – all parameter names that are supposed to be loaded

Returns

2-element tuple, containing

- list: list containing all variables to be read
- list: list containing all variables to be computed

Return type
tuple

property col_index

Dictionary that specifies the index for each data column

Note: Implementation depends on the data. For instance, if the variable information is provided in all files (of all stations) and always in the same column, then this can be set as a fixed dictionary in the __init__ function of the implementation (see e.g. class `ReadAeronetSunV2`). In other cases, it may not be ensured that each variable is available in all files or the column definition may differ between different stations. In the latter case you may automise the column index retrieval by providing the header names for each meta and data column you want to extract using the attribute dictionaries *META_NAMES_FILE* and *VAR_NAMES_FILE* by calling `_update_col_index()` in your implementation of *read_file()* when you reach the line that contains the header information.

---

compute_additional_vars(*data*, *vars_to_compute*)

Compute all additional variables

The computations for each additional parameter are done using the specified methods in `AUX_FUNS`.

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_compute** (*list*) – list of variable names that are supposed to be computed. Variables that are required for the computation of the variables need to be specified in *AUX_VARS* and need to be available as data vectors in the provided data dictionary (key is the corresponding variable name of the required variable).

Returns
updated data object now containing also computed variables

Return type
dict

property data_dir: str

Location of the dataset

Note: This can be set explicitly when instantiating the class (e.g. if data is available on local machine). If unspecified, the data location is attempted to be inferred via `get_obsnetwork_dir()`

Raises
FileNotFoundError – if data directory does not exist or cannot be retrieved automatically

Type
str

property data_id

ID of dataset

property data_revision

Revision string from file Revision.txt in the main data directory

find_in_file_list(*pattern=None*)

Find all files that match a certain wildcard pattern

Parameters
pattern (str, optional) – wildcard pattern that may be used to narrow down the search (e.g. use `pattern=*Berlin*` to find only files that contain Berlin in their filename)

Returns
list containing all files in `files` that match pattern

Return type
list

Raises
IOError – if no matches can be found

get_file_list(*pattern=None*)

Search all files to be read

Uses `_FILEMASK` (+ optional input search pattern, e.g. station_name) to find valid files for query.

Parameters
pattern (*str, optional*) – file name pattern applied to search

Returns
list containing retrieved file locations

Return type
list

Raises
IOError – if no files can be found

infer_wavelength_colname(*colname*, *low=250*, *high=2000*)

Get variable wavelength from column name

Parameters

- **colname** (*str*) – string of column name
- **low** (*int*) – lower limit of accepted value range
- **high** (*int*) – upper limit of accepted value range

Returns
wavelength in nm as floating str

Return type
str

Raises
ValueError – if None or more than one number is detected in variable string

logger

Class own instance of logger class

print_all_columns()

read(*vars_to_retrieve=None*, *files=None*, *first_file=None*, *last_file=None*, *file_pattern=None*, *common_meta=None*)

Method that reads list of files as instance of `UngriddedData`

Parameters

- **vars_to_retrieve** (`list` or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded
- **files** (`list`, optional) – list of files to be read. If None, then the file list is used that is returned on *get_file_list()*.
- **first_file** (`int`, optional) – index of first file in file list to read. If None, the very first file in the list is used. Note: is ignored if input parameter *file_pattern* is specified.
- **last_file** (`int`, optional) – index of last file in list to read. If None, the very last file in the list is used. Note: is ignored if input parameter *file_pattern* is specified.
- **file_pattern** (*str, optional*) – string pattern for file search (cf *get_file_list()*)
- **common_meta** (*dict, optional*) – dictionary that contains additional metadata shared for this network (assigned to each metadata block of the `UngriddedData` object that is returned)

Returns
data object

Return type
UngriddedData

abstract read_file(*filename*, *vars_to_retrieve=None*)

Read single file

Parameters

- **filename** (*str*) – string specifying filename
- **vars_to_retrieve** (`list` or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded

Returns
imported data in a suitable format that can be handled by *read()* which is supposed to append the loaded results from this method (which reads one datafile) to an instance of `UngriddedData` for all files.

Return type
`dict` or `StationData`, or other...

read_first_file(***kwargs*)

Read first file returned from *get_file_list()*

Note: This method may be used for test purposes.

Parameters
****kwargs** – keyword args passed to *read_file()* (e.g. vars_to_retrieve)

Returns
dictionary or similar containing loaded results from first file

Return type
dict-like

read_station(*station_id_filename*, ***kwargs*)

Read data from a single station into `UngriddedData`

Find all files that contain the station ID in their filename and then call *read()*, providing the reduced filelist as input, in order to read all files from this station into data object.

Parameters

- **station_id_filename** (*str*) – name of station (MUST be encrypted in filename)
- ****kwargs** – additional keyword args passed to *read()* (e.g. `vars_to_retrieve`)

Returns
loaded data

Return type
UngriddedData

Raises
IOError – if no files can be found for this station ID

remove_outliers(*data*, *vars_to_retrieve*, ***valid_rng_vars*)

Remove outliers from data

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_retrieve** (*list*) – list of variable names for which outliers will be removed from data
- ****valid_rng_vars** – additional keyword args specifying variable name and corresponding min / max interval (list or tuple) that specifies valid range for the variable. For each variable that is not explicitely defined here, the default minimum / maximum value is used (accessed via `pyaerocom.const.VARS[var_name]`)

var_supported(*var_name*)

Check if input variable is supported

Parameters
var_name (*str*) – AeroCom variable name or alias

Raises
VariableDefinitionError – if input variable is not supported by pyaerocom

Returns
True, if variable is supported by this interface, else False

Return type
bool

property verbosity_level

Current level of verbosity of logger

AERONET Sun (V3)

class pyaerocom.io.read_aeronet_sunv3.**ReadAeronetSunV3**(*data_id=None*, *data_dir=None*)

Bases: *ReadAeronetBase*

Interface for reading Aeronet direct sun version 3 Level 1.5 and 2.0 data

See also:

Base classes ReadAeronetBase and ReadUngriddedBase

ALT_VAR_NAMES_FILE = {}

dictionary specifying alternative column names for variables defined in *VAR_NAMES_FILE*

Type
OPTIONAL

AUX_FUNS = {'ang44&87aer': <function calc_ang4487aer>, 'od550aer': <function calc_od550aer>, 'od550lt1ang': <function calc_od550lt1ang>, 'proxyod550aerh2o': <function calc_od550aer>, 'proxyod550bc': <function calc_od550aer>, 'proxyod550dust': <function calc_od550aer>, 'proxyod550nh4': <function calc_od550aer>, 'proxyod550no3': <function calc_od550aer>, 'proxyod550oa': <function calc_od550aer>, 'proxyod550so4': <function calc_od550aer>, 'proxyod550ss': <function calc_od550aer>, 'proxyzaerosol': <function calc_od550aer>, 'proxyzdust': <function calc_od550aer>}

Functions that are used to compute additional variables (i.e. one for each variable defined in AUX_REQUIRES)

AUX_REQUIRES = {'ang44&87aer': ['od440aer', 'od870aer'], 'od550aer': ['od440aer', 'od500aer', 'ang4487aer'], 'od550lt1ang': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyod550aerh2o': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyod550bc': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyod550dust': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyod550nh4': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyod550no3': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyod550oa': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyod550so4': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyod550ss': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyzaerosol': ['od440aer', 'od500aer', 'ang4487aer'], 'proxyzdust': ['od440aer', 'od500aer', 'ang4487aer']}

dictionary containing information about additionally required variables for each auxiliary variable (i.e. each variable that is not provided by the original data but computed on import)

property AUX_VARS

List of auxiliary variables (keys of attr. *AUX_REQUIRES*)

Auxiliary variables are those that are not included in original files but are computed from other variables during import

COL_DELIM = ','

column delimiter in data block of files

property DATASET_PATH

Wrapper for *data_dir*.

DATA_ID = 'AeronetSunV3Lev2.daily'

Name of dataset (OBS_ID)

DEFAULT_UNIT = '1'

Default data unit that is assigned to all variables that are not specified in UNITS dictionary (cf. *UNITS*)

DEFAULT_VARS = ['od550aer', 'ang4487aer']

default variables for read method

IGNORE_META_KEYS = ['date', 'time', 'day_of_year']

INSTRUMENT_NAME = 'sun_photometer'

name of measurement instrument

META_NAMES_FILE = {'altitude': 'Site_Elevation(m)', 'data_quality_level': 'Data_Quality_Level', 'date': 'Date(dd:mm:yyyy)', 'day_of_year': 'Day_of_Year', 'instrument_number': 'AERONET_Instrument_Number', 'latitude': 'Site_Latitude(Degrees)', 'longitude': 'Site_Longitude(Degrees)', 'station_name': 'AERONET_Site', 'time': 'Time(hh:mm:ss)'}

dictionary specifying the file column names (values) for each metadata key (cf. attributes of `StationData`, e.g. 'station_name', 'longitude', 'latitude', 'altitude')

META_NAMES_FILE_ALT = {'AERONET_Site': ['AERONET_Site_Name']}

NAN_VAL = -999.0

PROVIDES_VARIABLES = ['od340aer', 'od440aer', 'od500aer', 'od870aer', 'ang4487aer']

List of variables that are provided by this dataset (will be extended by auxiliary variables on class init, for details see __init__ method of base class ReadUngriddedBase)

property REVISION_FILE

Name of revision file located in data directory

SUPPORTED_DATASETS = ['AeronetSunV3Lev1.5.daily', 'AeronetSunV3Lev1.5.AP', 'AeronetSunV3Lev2.daily', 'AeronetSunV3Lev2.AP']

List of all datasets supported by this interface

property TS_TYPE

Default implementation of string for temporal resolution

TS_TYPES = {'AeronetSunV3Lev1.5.daily': 'daily', 'AeronetSunV3Lev2.daily': 'daily'}

dictionary assigning temporal resolution flags for supported datasets that are provided in a defined temporal resolution

UNITS = {'proxyzaerosol': 'km', 'proxyzdust': 'km'}

Variable specific units, only required for variables that deviate from *DEFAULT_UNIT* (is irrelevant for all variables that are so far supported by the implemented Aeronet products, i.e. all variables are dimensionless as specified in *DEFAULT_UNIT*)

VAR_NAMES_FILE = {'ang4487aer': '440-870_Angstrom_Exponent', 'od340aer': 'AOD_340nm', 'od440aer': 'AOD_440nm', 'od500aer': 'AOD_500nm', 'od870aer': 'AOD_870nm'}

dictionary specifying the file column names (values) for each Aerocom variable (keys)

VAR_PATTERNS_FILE = {'AOD_([0-9]*)nm': 'od*aer'}

Mappings for identifying variables in file

check_vars_to_retrieve(*vars_to_retrieve*)

Separate variables that are in file from those that are computed

Some of the provided variables by this interface are not included in the data files but are computed within this class during data import (e.g. od550aer, ang4487aer).

The latter may require additional parameters to be retrieved from the file, which is specified in the class header (cf. attribute `AUX_REQUIRES`).

This function checks the input list that specifies all required variables and separates them into two lists, one that includes all variables that can be read from the files and a second list that specifies all variables that are computed in this class.

Parameters
vars_to_retrieve (*list*) – all parameter names that are supposed to be loaded

Returns

2-element tuple, containing

- list: list containing all variables to be read
- list: list containing all variables to be computed

Return type
tuple

property col_index

Dictionary that specifies the index for each data column

Note: Implementation depends on the data. For instance, if the variable information is provided in all files (of all stations) and always in the same column, then this can be set as a fixed dictionary in the __init__ function of the implementation (see e.g. class `ReadAeronetSunV2`). In other cases, it may not be ensured that each variable is available in all files or the column definition may differ between different stations. In the latter case you may automise the column index retrieval by providing the header names for each meta and data column you want to extract using the attribute dictionaries *META_NAMES_FILE* and *VAR_NAMES_FILE* by calling `_update_col_index()` in your implementation of *read_file()* when you reach the line that contains the header information.

---

compute_additional_vars(*data*, *vars_to_compute*)

Compute all additional variables

The computations for each additional parameter are done using the specified methods in `AUX_FUNS`.

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_compute** (*list*) – list of variable names that are supposed to be computed. Variables that are required for the computation of the variables need to be specified in *AUX_VARS* and need to be available as data vectors in the provided data dictionary (key is the corresponding variable name of the required variable).

Returns
updated data object now containing also computed variables

Return type
dict

property data_dir: str

Location of the dataset

Note: This can be set explicitly when instantiating the class (e.g. if data is available on local machine). If unspecified, the data location is attempted to be inferred via `get_obsnetwork_dir()`

Raises
FileNotFoundError – if data directory does not exist or cannot be retrieved automatically

Type
str

property data_id

ID of dataset

property data_revision

Revision string from file Revision.txt in the main data directory

find_in_file_list(*pattern=None*)

Find all files that match a certain wildcard pattern

Parameters
pattern (str, optional) – wildcard pattern that may be used to narrow down the search (e.g. use `pattern=*Berlin*` to find only files that contain Berlin in their filename)

Returns
list containing all files in `files` that match pattern

Return type
list

Raises
IOError – if no matches can be found

get_file_list(*pattern=None*)

Search all files to be read

Uses `_FILEMASK` (+ optional input search pattern, e.g. station_name) to find valid files for query.

Parameters
pattern (*str, optional*) – file name pattern applied to search

Returns
list containing retrieved file locations

Return type
list

Raises
IOError – if no files can be found

infer_wavelength_colname(*colname*, *low=250*, *high=2000*)

Get variable wavelength from column name

Parameters

- **colname** (*str*) – string of column name
- **low** (*int*) – lower limit of accepted value range
- **high** (*int*) – upper limit of accepted value range

Returns
wavelength in nm as floating str

Return type
str

Raises
ValueError – if None or more than one number is detected in variable string

logger

Class own instance of logger class

print_all_columns()

read(*vars_to_retrieve=None*, *files=None*, *first_file=None*, *last_file=None*, *file_pattern=None*, *common_meta=None*)

Method that reads list of files as instance of `UngriddedData`

Parameters

- **vars_to_retrieve** (`list` or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded
- **files** (`list`, optional) – list of files to be read. If None, then the file list is used that is returned on *get_file_list()*.
- **first_file** (`int`, optional) – index of first file in file list to read. If None, the very first file in the list is used. Note: is ignored if input parameter *file_pattern* is specified.
- **last_file** (`int`, optional) – index of last file in list to read. If None, the very last file in the list is used. Note: is ignored if input parameter *file_pattern* is specified.
- **file_pattern** (*str, optional*) – string pattern for file search (cf *get_file_list()*)
- **common_meta** (*dict, optional*) – dictionary that contains additional metadata shared for this network (assigned to each metadata block of the `UngriddedData` object that is returned)

Returns
data object

Return type
UngriddedData

read_file(*filename*, *vars_to_retrieve=None*, *vars_as_series=False*)

Read Aeronet Sun V3 level 1.5 or 2 file

Parameters

- **filename** (*str*) – absolute path to filename to read
- **vars_to_retrieve** (list, optional) – list of str with variable names to read. If None, use *DEFAULT_VARS*
- **vars_as_series** (*bool*) – if True, the data columns of all variables in the result dictionary are converted into pandas Series objects

Returns
dict-like object containing results

Return type
StationData

read_first_file(***kwargs*)

Read first file returned from *get_file_list()*

Note: This method may be used for test purposes.

Parameters
****kwargs** – keyword args passed to *read_file()* (e.g. vars_to_retrieve)

Returns
dictionary or similar containing loaded results from first file

Return type
dict-like

read_station(*station_id_filename*, ***kwargs*)

Read data from a single station into `UngriddedData`

Find all files that contain the station ID in their filename and then call *read()*, providing the reduced filelist as input, in order to read all files from this station into data object.

Parameters

- **station_id_filename** (*str*) – name of station (MUST be encrypted in filename)
- ****kwargs** – additional keyword args passed to *read()* (e.g. `vars_to_retrieve`)

Returns
loaded data

Return type
UngriddedData

Raises
IOError – if no files can be found for this station ID

remove_outliers(*data*, *vars_to_retrieve*, ***valid_rng_vars*)

Remove outliers from data

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)

- **vars_to_retrieve** (*list*) – list of variable names for which outliers will be removed from data
- ****valid_rng_vars** – additional keyword args specifying variable name and corresponding min / max interval (list or tuple) that specifies valid range for the variable. For each variable that is not explicitely defined here, the default minimum / maximum value is used (accessed via `pyaerocom.const.VARS[var_name]`)

var_supported(*var_name*)

Check if input variable is supported

Parameters
var_name (*str*) – AeroCom variable name or alias

Raises
VariableDefinitionError – if input variable is not supported by pyaerocom

Returns
True, if variable is supported by this interface, else False

Return type
bool

property verbosity_level

Current level of verbosity of logger

AERONET SDA (V3)

class pyaerocom.io.read_aeronet_sdav3.**ReadAeronetSdaV3**(*data_id=None*, *data_dir=None*)

Bases: *ReadAeronetBase*

Interface for reading Aeronet Sun SDA V3 Level 1.5 and 2.0 data

See also:

Base classes `ReadAeronetBase` and `ReadUngriddedBase`

ALT_VAR_NAMES_FILE = {}

dictionary specifying alternative column names for variables defined in *VAR_NAMES_FILE*

Type
OPTIONAL

AUX_FUNS = {'od550aer': <function calc_od550aer>, 'od550dust': <function calc_od550gt1aer>, 'od550gt1aer': <function calc_od550gt1aer>, 'od550lt1aer': <function calc_od550lt1aer>}

Functions that are used to compute additional variables (i.e. one for each variable defined in AUX_REQUIRES)

AUX_REQUIRES = {'od550aer': ['od500aer', 'ang4487aer'], 'od550dust': ['od500gt1aer', 'ang4487aer'], 'od550gt1aer': ['od500gt1aer', 'ang4487aer'], 'od550lt1aer': ['od500lt1aer', 'ang4487aer']}

dictionary containing information about additionally required variables for each auxiliary variable (i.e. each variable that is not provided by the original data but computed on import)

property AUX_VARS

List of auxiliary variables (keys of attr. *AUX_REQUIRES*)

Auxiliary variables are those that are not included in original files but are computed from other variables during import

COL_DELIM = ','

column delimiter in data block of files

property DATASET_PATH

Wrapper for *data_dir*.

DATA_ID = 'AeronetSDAV3Lev2.daily'

Name of dataset (OBS_ID)

DEFAULT_UNIT = '1'

Default data unit that is assigned to all variables that are not specified in UNITS dictionary (cf. *UNITS*)

DEFAULT_VARS = ['od550aer', 'od550gt1aer', 'od550lt1aer', 'od550dust']

default variables for read method

IGNORE_META_KEYS = ['date', 'time', 'day_of_year']

INSTRUMENT_NAME = 'sun_photometer'

name of measurement instrument

META_NAMES_FILE = {'altitude': 'Site_Elevation(m)', 'data_quality_level': 'Data_Quality_Level', 'date': 'Date_(dd:mm:yyyy)', 'day_of_year': 'Day_of_Year', 'instrument_number': 'AERONET_Instrument_Number', 'latitude': 'Site_Latitude(Degrees)', 'longitude': 'Site_Longitude(Degrees)', 'station_name': 'AERONET_Site', 'time': 'Time_(hh:mm:ss)'}

dictionary specifying the file column names (values) for each metadata key (cf. attributes of `StationData`, e.g. 'station_name', 'longitude', 'latitude', 'altitude')

META_NAMES_FILE_ALT = ({},)

NAN_VAL = -999.0

value corresponding to invalid measurement

PROVIDES_VARIABLES = ['od500gt1aer', 'od500lt1aer', 'od500aer', 'ang4487aer', 'od500dust']

List of variables that are provided by this dataset (will be extended by auxiliary variables on class init, for details see __init__ method of base class ReadUngriddedBase)

property REVISION_FILE

Name of revision file located in data directory

SUPPORTED_DATASETS = ['AeronetSDAV3Lev1.5.daily', 'AeronetSDAV3Lev2.daily']

List of all datasets supported by this interface

property TS_TYPE

Default implementation of string for temporal resolution

TS_TYPES = {'AeronetSDAV3Lev1.5.daily': 'daily', 'AeronetSDAV3Lev2.daily': 'daily'}

dictionary assigning temporal resolution flags for supported datasets that are provided in a defined temporal resolution

UNITS = {}

Variable specific units, only required for variables that deviate from *DEFAULT_UNIT* (is irrelevant for all variables that are so far supported by the implemented Aeronet products, i.e. all variables are dimensionless as specified in *DEFAULT_UNIT*)

VAR_NAMES_FILE = {'ang4487aer': 'Angstrom_Exponent(AE)-Total_500nm[alpha]', 'od500aer': 'Total_AOD_500nm[tau_a]', 'od500dust': 'Coarse_Mode_AOD_500nm[tau_c]', 'od500gt1aer': 'Coarse_Mode_AOD_500nm[tau_c]', 'od500lt1aer': 'Fine_Mode_AOD_500nm[tau_f]'}

dictionary specifying the file column names (values) for each Aerocom variable (keys)

VAR_PATTERNS_FILE = {}

Mappings for identifying variables in file (may be specified in addition to explicit variable names specified in VAR_NAMES_FILE)

check_vars_to_retrieve(*vars_to_retrieve*)

Separate variables that are in file from those that are computed

Some of the provided variables by this interface are not included in the data files but are computed within this class during data import (e.g. od550aer, ang4487aer).

The latter may require additional parameters to be retrieved from the file, which is specified in the class header (cf. attribute `AUX_REQUIRES`).

This function checks the input list that specifies all required variables and separates them into two lists, one that includes all variables that can be read from the files and a second list that specifies all variables that are computed in this class.

Parameters

vars_to_retrieve (*list*) – all parameter names that are supposed to be loaded

Returns

2-element tuple, containing

- list: list containing all variables to be read
- list: list containing all variables to be computed

Return type

tuple

property col_index

Dictionary that specifies the index for each data column

Note: Implementation depends on the data. For instance, if the variable information is provided in all files (of all stations) and always in the same column, then this can be set as a fixed dictionary in the __init__ function of the implementation (see e.g. class `ReadAeronetSunV2`). In other cases, it may not be ensured that each variable is available in all files or the column definition may differ between different stations. In the latter case you may automise the column index retrieval by providing the header names for each meta and data column you want to extract using the attribute dictionaries *META_NAMES_FILE* and *VAR_NAMES_FILE* by calling `_update_col_index()` in your implementation of *read_file()* when you reach the line that contains the header information.

---

compute_additional_vars(*data*, *vars_to_compute*)

Compute all additional variables

The computations for each additional parameter are done using the specified methods in `AUX_FUNS`.

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)

- **vars_to_compute** (*list*) – list of variable names that are supposed to be computed. Variables that are required for the computation of the variables need to be specified in *AUX_VARS* and need to be available as data vectors in the provided data dictionary (key is the corresponding variable name of the required variable).

Returns
updated data object now containing also computed variables

Return type
dict

property data_dir: str

Location of the dataset

Note: This can be set explicitly when instantiating the class (e.g. if data is available on local machine). If unspecified, the data location is attempted to be inferred via `get_obsnetwork_dir()`

Raises
FileNotFoundError – if data directory does not exist or cannot be retrieved automatically

Type
str

property data_id

ID of dataset

property data_revision

Revision string from file Revision.txt in the main data directory

find_in_file_list(*pattern=None*)

Find all files that match a certain wildcard pattern

Parameters
pattern (str, optional) – wildcard pattern that may be used to narrow down the search (e.g. use `pattern=*Berlin*` to find only files that contain Berlin in their filename)

Returns
list containing all files in `files` that match pattern

Return type
list

Raises
IOError – if no matches can be found

get_file_list(*pattern=None*)

Search all files to be read

Uses `_FILEMASK` (+ optional input search pattern, e.g. station_name) to find valid files for query.

Parameters
pattern (*str, optional*) – file name pattern applied to search

Returns
list containing retrieved file locations

Return type
list

Raises
IOError – if no files can be found

infer_wavelength_colname(*colname*, *low=250*, *high=2000*)

Get variable wavelength from column name

Parameters

- **colname** (*str*) – string of column name
- **low** (*int*) – lower limit of accepted value range
- **high** (*int*) – upper limit of accepted value range

Returns
wavelength in nm as floating str

Return type
str

Raises
ValueError – if None or more than one number is detected in variable string

logger

Class own instance of logger class

print_all_columns()

read(*vars_to_retrieve=None*, *files=None*, *first_file=None*, *last_file=None*, *file_pattern=None*, *common_meta=None*)

Method that reads list of files as instance of UngriddedData

Parameters

- **vars_to_retrieve** (list or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded
- **files** (list, optional) – list of files to be read. If None, then the file list is used that is returned on *get_file_list()*.
- **first_file** (int, optional) – index of first file in file list to read. If None, the very first file in the list is used. Note: is ignored if input parameter *file_pattern* is specified.
- **last_file** (int, optional) – index of last file in list to read. If None, the very last file in the list is used. Note: is ignored if input parameter *file_pattern* is specified.
- **file_pattern** (*str, optional*) – string pattern for file search (cf *get_file_list()*)
- **common_meta** (*dict, optional*) – dictionary that contains additional metadata shared for this network (assigned to each metadata block of the UngriddedData object that is returned)

Returns
data object

Return type
UngriddedData

read_file(*filename*, *vars_to_retrieve=None*, *vars_as_series=False*)

Read Aeronet SDA V3 file and return it in a dictionary

Parameters

- **filename** (*str*) – absolute path to filename to read

- **vars_to_retrieve** (*list*, optional) – list of str with variable names to read. If None, use *DEFAULT_VARS*
- **vars_as_series** (*bool*) – if True, the data columns of all variables in the result dictionary are converted into pandas Series objects

Returns
dict-like object containing results

Return type
StationData

read_first_file(***kwargs*)

Read first file returned from *get_file_list()*

Note: This method may be used for test purposes.

Parameters
****kwargs** – keyword args passed to *read_file()* (e.g. vars_to_retrieve)

Returns
dictionary or similar containing loaded results from first file

Return type
dict-like

read_station(*station_id_filename*, ***kwargs*)

Read data from a single station into `UngriddedData`

Find all files that contain the station ID in their filename and then call *read()*, providing the reduced filelist as input, in order to read all files from this station into data object.

Parameters

- **station_id_filename** (*str*) – name of station (MUST be encrypted in filename)
- ****kwargs** – additional keyword args passed to *read()* (e.g. `vars_to_retrieve`)

Returns
loaded data

Return type
UngriddedData

Raises
IOError – if no files can be found for this station ID

remove_outliers(*data*, *vars_to_retrieve*, ***valid_rng_vars*)

Remove outliers from data

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_retrieve** (*list*) – list of variable names for which outliers will be removed from data
- ****valid_rng_vars** – additional keyword args specifying variable name and corresponding min / max interval (list or tuple) that specifies valid range for the variable. For each

variable that is not explicitely defined here, the default minimum / maximum value is used (accessed via `pyaerocom.const.VARS[var_name]`)

var_supported(*var_name*)

Check if input variable is supported

Parameters
var_name (*str*) – AeroCom variable name or alias

Raises
VariableDefinitionError – if input variable is not supported by pyaerocom

Returns
True, if variable is supported by this interface, else False

Return type
bool

property verbosity_level

Current level of verbosity of logger

AERONET Inversion (V3)

class pyaerocom.io.read_aeronet_invv3.**ReadAeronetInvV3**(*data_id=None*, *data_dir=None*)

Bases: *ReadAeronetBase*

Interface for reading Aeronet inversion V3 Level 1.5 and 2.0 data

Parameters
data_id – string specifying either of the supported datasets that are defined in SUPPORTED_DATASETS

ALT_VAR_NAMES_FILE = {}

dictionary specifying alternative column names for variables defined in *VAR_NAMES_FILE*

Type
OPTIONAL

AUX_FUNS = {'abs550aer': <function calc_abs550aer>, 'od550aer': <function calc_od550aer>}

Functions that are used to compute additional variables (i.e. one for each variable defined in AUX_REQUIRES)

AUX_REQUIRES = {'abs550aer': ['abs440aer', 'angabs4487aer'], 'od550aer': ['od440aer', 'ang4487aer']}

dictionary containing information about additionally required variables for each auxiliary variable (i.e. each variable that is not provided by the original data but computed on import)

property AUX_VARS

List of auxiliary variables (keys of attr. *AUX_REQUIRES*)

Auxiliary variables are those that are not included in original files but are computed from other variables during import

COL_DELIM = ','

column delimiter in data block of files

property DATASET_PATH

Wrapper for *data_dir*.

DATA_ID = 'AeronetInvV3Lev2.daily'

Name of dataset (OBS_ID)

DEFAULT_UNIT = '1'

Default data unit that is assigned to all variables that are not specified in UNITS dictionary (cf. *UNITS*)

DEFAULT_VARS = ['abs550aer', 'od550aer']

default variables for read method

IGNORE_META_KEYS = ['date', 'time', 'day_of_year']

INSTRUMENT_NAME = 'sun_photometer'

name of measurement instrument

META_NAMES_FILE = {'altitude': 'Elevation(m)', 'date': 'Date(dd:mm:yyyy)', 'day_of_year': 'Day_of_Year(fraction)', 'latitude': 'Latitude(Degrees)', 'longitude': 'Longitude(Degrees)', 'station_name': 'AERONET_Site', 'time': 'Time(hh:mm:ss)'}

dictionary specifying the file column names (values) for each metadata key (cf. attributes of `StationData`, e.g. 'station_name', 'longitude', 'latitude', 'altitude')

META_NAMES_FILE_ALT = ({},)

NAN_VAL = -999.0

value corresponding to invalid measurement

PROVIDES_VARIABLES = ['abs440aer', 'angabs4487aer', 'od440aer', 'ang4487aer', 'ssa675aer', 'ssa670aer']

List of variables that are provided by this dataset (will be extended by auxiliary variables on class init, for details see __init__ method of base class ReadUngriddedBase)

property REVISION_FILE

Name of revision file located in data directory

SUPPORTED_DATASETS = ['AeronetInvV3Lev2.daily', 'AeronetInvV3Lev1.5.daily']

List of all datasets supported by this interface

property TS_TYPE

Default implementation of string for temporal resolution

TS_TYPES = {'AeronetInvV3Lev1.5.daily': 'daily', 'AeronetInvV3Lev2.daily': 'daily'}

dictionary assigning temporal resolution flags for supported datasets that are provided in a defined temporal resolution

UNITS = {}

Variable specific units, only required for variables that deviate from *DEFAULT_UNIT* (is irrelevant for all variables that are so far supported by the implemented Aeronet products, i.e. all variables are dimensionless as specified in *DEFAULT_UNIT*)

VAR_NAMES_FILE = {'abs440aer': 'Absorption_AOD[440nm]', 'ang4487aer': 'Extinction_Angstrom_Exponent_440-870nm-Total', 'angabs4487aer': 'Absorption_Angstrom_Exponent_440-870nm', 'od440aer': 'AOD_Extinction-Total[440nm]', 'ssa670aer': 'Single_Scattering_Albedo[675nm]', 'ssa675aer': 'Single_Scattering_Albedo[675nm]'}

dictionary specifying the file column names (values) for each Aerocom variable (keys)

VAR_PATTERNS_FILE = {}

Mappings for identifying variables in file (may be specified in addition to explicit variable names specified in VAR_NAMES_FILE)

check_vars_to_retrieve(*vars_to_retrieve*)

Separate variables that are in file from those that are computed

Some of the provided variables by this interface are not included in the data files but are computed within this class during data import (e.g. od550aer, ang4487aer).

The latter may require additional parameters to be retrieved from the file, which is specified in the class header (cf. attribute `AUX_REQUIRES`).

This function checks the input list that specifies all required variables and separates them into two lists, one that includes all variables that can be read from the files and a second list that specifies all variables that are computed in this class.

Parameters
vars_to_retrieve (*list*) – all parameter names that are supposed to be loaded

Returns

2-element tuple, containing

- list: list containing all variables to be read
- list: list containing all variables to be computed

Return type
tuple

property col_index

Dictionary that specifies the index for each data column

Note: Implementation depends on the data. For instance, if the variable information is provided in all files (of all stations) and always in the same column, then this can be set as a fixed dictionary in the __init__ function of the implementation (see e.g. class `ReadAeronetSunV2`). In other cases, it may not be ensured that each variable is available in all files or the column definition may differ between different stations. In the latter case you may automise the column index retrieval by providing the header names for each meta and data column you want to extract using the attribute dictionaries *META_NAMES_FILE* and *VAR_NAMES_FILE* by calling `_update_col_index()` in your implementation of *read_file()* when you reach the line that contains the header information.

---

compute_additional_vars(*data*, *vars_to_compute*)

Compute all additional variables

The computations for each additional parameter are done using the specified methods in `AUX_FUNS`.

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_compute** (*list*) – list of variable names that are supposed to be computed. Variables that are required for the computation of the variables need to be specified in *AUX_VARS* and need to be available as data vectors in the provided data dictionary (key is the corresponding variable name of the required variable).

Returns
updated data object now containing also computed variables

Return type
dict

property data_dir: str
Location of the dataset

Note: This can be set explicitly when instantiating the class (e.g. if data is available on local machine). If unspecified, the data location is attempted to be inferred via `get_obsnetwork_dir()`

Raises
FileNotFoundError – if data directory does not exist or cannot be retrieved automatically

Type
str

property data_id
ID of dataset

property data_revision
Revision string from file Revision.txt in the main data directory

find_in_file_list(*pattern=None*)
Find all files that match a certain wildcard pattern

Parameters
pattern (*str*, optional) – wildcard pattern that may be used to narrow down the search (e.g. use `pattern=*Berlin*` to find only files that contain Berlin in their filename)

Returns
list containing all files in `files` that match pattern

Return type
list

Raises
IOError – if no matches can be found

get_file_list(*pattern=None*)
Search all files to be read

Uses `_FILEMASK` (+ optional input search pattern, e.g. station_name) to find valid files for query.

Parameters
pattern (*str, optional*) – file name pattern applied to search

Returns
list containing retrieved file locations

Return type
list

Raises
IOError – if no files can be found

infer_wavelength_colname(*colname*, *low=250*, *high=2000*)
Get variable wavelength from column name

Parameters

- **colname** (*str*) – string of column name

- **low** (*int*) – lower limit of accepted value range
- **high** (*int*) – upper limit of accepted value range

Returns
wavelength in nm as floating str

Return type
str

Raises
ValueError – if None or more than one number is detected in variable string

logger
Class own instance of logger class

print_all_columns()

read(*vars_to_retrieve=None*, *files=None*, *first_file=None*, *last_file=None*, *file_pattern=None*, *common_meta=None*)
Method that reads list of files as instance of UngriddedData

Parameters

- **vars_to_retrieve** (list or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded
- **files** (list, optional) – list of files to be read. If None, then the file list is used that is returned on *get_file_list()*.
- **first_file** (int, optional) – index of first file in file list to read. If None, the very first file in the list is used. Note: is ignored if input parameter *file_pattern* is specified.
- **last_file** (int, optional) – index of last file in list to read. If None, the very last file in the list is used. Note: is ignored if input parameter *file_pattern* is specified.
- **file_pattern** (*str, optional*) – string pattern for file search (cf *get_file_list()*)
- **common_meta** (*dict, optional*) – dictionary that contains additional metadata shared for this network (assigned to each metadata block of the UngriddedData object that is returned)

Returns
data object

Return type
UngriddedData

read_file(*filename*, *vars_to_retrieve=None*, *vars_as_series=False*)
Read Aeronet file containing results from v2 inversion algorithm

Parameters

- **filename** (*str*) – absolute path to filename to read
- **vars_to_retrieve** (*list*) – list of str with variable names to read
- **vars_as_series** (*bool*) – if True, the data columns of all variables in the result dictionary are converted into pandas Series objects

Returns
dict-like object containing results

Return type
StationData

Example

```
>>> import pyaerocom.io as pio
>>> obj = pio.read_aeronet_invv2.ReadAeronetInvV2()
>>> files = obj.get_file_list()
>>> filedata = obj.read_file(files[0])
```

read_first_file(***kwargs*)

Read first file returned from *get_file_list()*

Note: This method may be used for test purposes.

Parameters
****kwargs** – keyword args passed to *read_file()* (e.g. vars_to_retrieve)

Returns
dictionary or similar containing loaded results from first file

Return type
dict-like

read_station(*station_id_filename*, ***kwargs*)

Read data from a single station into `UngriddedData`

Find all files that contain the station ID in their filename and then call *read()*, providing the reduced filelist as input, in order to read all files from this station into data object.

Parameters

- **station_id_filename** (*str*) – name of station (MUST be encrypted in filename)
- ****kwargs** – additional keyword args passed to *read()* (e.g. `vars_to_retrieve`)

Returns
loaded data

Return type
UngriddedData

Raises
IOError – if no files can be found for this station ID

remove_outliers(*data*, *vars_to_retrieve*, ***valid_rng_vars*)

Remove outliers from data

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_retrieve** (*list*) – list of variable names for which outliers will be removed from data
- ****valid_rng_vars** – additional keyword args specifying variable name and corresponding min / max interval (list or tuple) that specifies valid range for the variable. For each variable that is not explicitely defined here, the default minimum / maximum value is used (accessed via `pyaerocom.const.VARS[var_name]`)

var_supported(*var_name*)

Check if input variable is supported

Parameters
var_name (*str*) – AeroCom variable name or alias

Raises
VariableDefinitionError – if input variable is not supported by pyaerocom

Returns
True, if variable is supported by this interface, else False

Return type
bool

property verbosity_level

Current level of verbosity of logger

4.5.4 EARLINET

European Aerosol Research Lidar Network (EARLINET)

class pyaerocom.io.read_earlinet.**ReadEarlinet**(*data_id=None*, *data_dir=None*)

Bases: *ReadUngriddedBase*

Interface for reading of EARLINET data

ALTITUDE_ID = 'altitude'

variable name of altitude in files

AUX_FUNS = {}

Functions that are used to compute additional variables (i.e. one for each variable defined in AUX_REQUIRES)

AUX_REQUIRES = {}

dictionary containing information about additionally required variables for each auxiliary variable (i.e. each variable that is not provided by the original data but computed on import)

property AUX_VARS

List of auxiliary variables (keys of attr. *AUX_REQUIRES*)

Auxiliary variables are those that are not included in original files but are computed from other variables during import

property DATASET_PATH

Wrapper for *data_dir*.

DATA_ID = 'EARLINET'

Name of dataset (OBS_ID)

DEFAULT_VARS = ['bsc532aer', 'ec532aer']

default variables for read method

ERR_VARNAMES = {'ec355aer': 'error_extinction', 'ec532aer': 'error_extinction'}

Variable names of uncertainty data

EXCLUDE_CASES = ['cirrus.txt']

IGNORE_META_KEYS = []

KEEP_ADD_META = ['location', 'wavelength', 'zenith_angle', 'comment', 'shots', 'backscatter_evaluation_method']

Metadata keys from *META_NAMES_FILE* that are additional to standard keys defined in StationMetaData and that are supposed to be inserted into UngriddedData object created in *read()*

META_NAMES_FILE = {'PI': 'PI', 'altitude': 'altitude', 'comment': 'comment', 'dataset_name': 'title', 'instrument_name': 'system', 'location': 'location', 'start_utc': 'measurement_start_datetime', 'stop_utc': 'measurement_stop_datetime', 'wavelength_emis': 'wavelength', 'website': 'references'}

META_NEEDED = ['location', 'measurement_start_datetime', 'measurement_start_datetime']

metadata keys that are needed for reading (must be values in *META_NAMES_FILE*)

PROVIDES_VARIABLES = ['ec532aer', 'ec355aer', 'bsc532aer', 'bsc355aer', 'bsc1064aer']

READ_ERR = True

If true, the uncertainties are also read (where available, cf. ERR_VARNAMES)

property REVISION_FILE

Name of revision file located in data directory

SUPPORTED_DATASETS = ['EARLINET']

List of all datasets supported by this interface

TS_TYPE = 'hourly'

VAR_NAMES_FILE = {'bsc1064aer': 'backscatter', 'bsc355aer': 'backscatter', 'bsc532aer': 'backscatter', 'ec1064aer': 'extinction', 'ec355aer': 'extinction', 'ec532aer': 'extinction', 'zdust': 'DustLayerHeight'}

dictionary specifying the file column names (values) for each Aerocom variable (keys)

VAR_PATTERNS_FILE = {'bsc1064aer': '_Lev02_b1064', 'bsc355aer': '_Lev02_b0355', 'bsc532aer': '_Lev02_b0532', 'ec355aer': '_Lev02_e0355', 'ec532aer': '_Lev02_e0532'}

VAR_UNIT_NAMES = {'altitude': 'units', 'backscatter': ['units'], 'dustlayerheight': ['units'], 'extinction': ['units']}

Attribute access names for unit reading of variable data

check_vars_to_retrieve(*vars_to_retrieve*)

Separate variables that are in file from those that are computed

Some of the provided variables by this interface are not included in the data files but are computed within this class during data import (e.g. od550aer, ang4487aer).

The latter may require additional parameters to be retrieved from the file, which is specified in the class header (cf. attribute AUX_REQUIRES).

This function checks the input list that specifies all required variables and separates them into two lists, one that includes all variables that can be read from the files and a second list that specifies all variables that are computed in this class.

Parameters

vars_to_retrieve (*list*) – all parameter names that are supposed to be loaded

Returns

2-element tuple, containing

- list: list containing all variables to be read
- list: list containing all variables to be computed

Return type
tuple

`compute_additional_vars`(*data*, *vars_to_compute*)

Compute all additional variables

The computations for each additional parameter are done using the specified methods in `AUX_FUNS`.

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_compute** (*list*) – list of variable names that are supposed to be computed. Variables that are required for the computation of the variables need to be specified in *AUX_VARS* and need to be available as data vectors in the provided data dictionary (key is the corresponding variable name of the required variable).

Returns
updated data object now containing also computed variables

Return type
dict

`property data_dir: str`

Location of the dataset

Note: This can be set explicitly when instantiating the class (e.g. if data is available on local machine). If unspecified, the data location is attempted to be inferred via `get_obsnetwork_dir()`

Raises
FileNotFoundError – if data directory does not exist or cannot be retrieved automatically

Type
str

`property data_id`

ID of dataset

`property data_revision`

Revision string from file Revision.txt in the main data directory

`exclude_files`

files that are supposed to be excluded from reading

`excluded_files`

files that were actually excluded from reading

`find_in_file_list`(*pattern=None*)

Find all files that match a certain wildcard pattern

Parameters
pattern (str, optional) – wildcard pattern that may be used to narrow down the search (e.g. use pattern=*Berlin* to find only files that contain Berlin in their filename)

Returns
list containing all files in files that match pattern

Return type
list

Raises
IOError – if no matches can be found

get_file_list(*vars_to_retrieve=None*, *pattern=None*)

Perform recusive file search for all input variables

Note: Overloaded implementation of base class, since for Earlinet, the paths are variable dependent

Parameters
- **vars_to_retrieve** (*list*) – list of variables to retrieve
- **pattern** (*str, optional*) – file name pattern applied to search

Returns
list containing file paths

Return type
list

logger

Class own instance of logger class

read(*vars_to_retrieve=None*, *files=None*, *first_file=None*, *last_file=None*, *read_err=None*, *remove_outliers=True*, *pattern=None*)

Method that reads list of files as instance of UngriddedData

Parameters
- **vars_to_retrieve** (list or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded
- **files** (list, optional) – list of files to be read. If None, then the file list is used that is returned on *get_file_list()*.
- **first_file** (int, optional) – index of first file in file list to read. If None, the very first file in the list is used
- **last_file** (int, optional) – index of last file in list to read. If None, the very last file in the list is used
- **read_err** (*bool*) –

  if True, uncertainty data is also read (where available). If unspecified (None), then the default is used (cf. *READ_ERR*)

  pattern
  [str, optional] string pattern for file search (cf *get_file_list()*)

Returns
data object

Return type
UngriddedData

read_file(*filename*, *vars_to_retrieve=None*, *read_err=None*, *remove_outliers=True*)

Read EARLINET file and return it as instance of `StationData`

Parameters

- **filename** (*str*) – absolute path to filename to read
- **vars_to_retrieve** (list, optional) – list of str with variable names to read. If None, use *DEFAULT_VARS*
- **read_err** (*bool*) – if True, uncertainty data is also read (where available).
- **remove_outliers** (*bool*) – if True, outliers are removed for each variable using the *minimum* and *maximum* attributes for that variable (accessed via pyaerocom.const.VARS[var_name]).

Returns
dict-like object containing results

Return type
StationData

read_first_file(***kwargs*)

Read first file returned from *get_file_list()*

Note: This method may be used for test purposes.

Parameters
****kwargs** – keyword args passed to *read_file()* (e.g. vars_to_retrieve)

Returns
dictionary or similar containing loaded results from first file

Return type
dict-like

read_station(*station_id_filename*, ***kwargs*)

Read data from a single station into `UngriddedData`

Find all files that contain the station ID in their filename and then call *read()*, providing the reduced filelist as input, in order to read all files from this station into data object.

Parameters

- **station_id_filename** (*str*) – name of station (MUST be encrypted in filename)
- ****kwargs** – additional keyword args passed to *read()* (e.g. `vars_to_retrieve`)

Returns
loaded data

Return type
UngriddedData

Raises
IOError – if no files can be found for this station ID

remove_outliers(*data*, *vars_to_retrieve*, ***valid_rng_vars*)

Remove outliers from data

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_retrieve** (*list*) – list of variable names for which outliers will be removed from data
- ****valid_rng_vars** – additional keyword args specifying variable name and corresponding min / max interval (list or tuple) that specifies valid range for the variable. For each variable that is not explicitely defined here, the default minimum / maximum value is used (accessed via `pyaerocom.const.VARS[var_name]`)

var_supported(*var_name*)

Check if input variable is supported

Parameters
var_name (*str*) – AeroCom variable name or alias

Raises
VariableDefinitionError – if input variable is not supported by pyaerocom

Returns
True, if variable is supported by this interface, else False

Return type
bool

property verbosity_level

Current level of verbosity of logger

4.5.5 EBAS

EBAS is a database with atmospheric measurement data hosted by the Norwegian Institute for Air Research. Declaration of AEROCOM variables in EBAS and assocaited information such as acceptable minimum and maximum values occurs in `pyaerocom/data/variables.ini` .

class pyaerocom.io.read_ebas.**ReadEbas**(*data_id=None*, *data_dir=None*)

Bases: *ReadUngriddedBase*

Interface for reading EBAS data

Parameters

- **data_id** – string specifying either of the supported datasets that are defined in `SUPPORTED_DATASETS`
- **data_dir** (*str*) – directory where data is located (NOTE: needs to point to the directory that contains the "ebas_file_index.sqlite3" file and not to the underlying directory "data" which contains the actual NASA Ames files.)

ASSUME_AAE_SHIFT_WVL = 1.0

ASSUME_AE_SHIFT_WVL = 1

```
AUX_FUNS = {'ac550dryaer': <function compute_ac550dryaer>, 'ang4470dryaer':
<function compute_ang4470dryaer_from_dry_scat>, 'proxydryhno3': <function
compute_wetoxn_from_concprcpoxn>, 'proxydryhono': <function
compute_wetoxn_from_concprcpoxn>, 'proxydryn2o5': <function
compute_wetoxn_from_concprcpoxn>, 'proxydryna': <function
compute_wetna_from_concprcpna>, 'proxydrynh3': <function
compute_wetrdn_from_concprcprdn>, 'proxydrynh4': <function
compute_wetrdn_from_concprcprdn>, 'proxydryno2': <function
compute_wetoxn_from_concprcpoxn>, 'proxydryno2no2': <function
compute_wetoxn_from_concprcpoxn>, 'proxydryno3c': <function
compute_wetoxn_from_concprcpoxn>, 'proxydryno3f': <function
compute_wetoxn_from_concprcpoxn>, 'proxydryo3': <function
make_proxy_drydep_from_O3>, 'proxydryoxn': <function
compute_wetoxn_from_concprcpoxn>, 'proxydryoxs': <function
compute_wetoxs_from_concprcpoxs>, 'proxydrypm10': <function
compute_wetoxs_from_concprcpoxs>, 'proxydrypm25': <function
compute_wetoxs_from_concprcpoxs>, 'proxydryrdn': <function
compute_wetrdn_from_concprcprdn>, 'proxydryso2': <function
compute_wetoxs_from_concprcpoxs>, 'proxydryso4': <function
compute_wetoxs_from_concprcpoxs>, 'proxydryss': <function
compute_wetna_from_concprcpna>, 'proxywethno3': <function
compute_wetoxn_from_concprcpoxn>, 'proxywethono': <function
compute_wetoxn_from_concprcpoxn>, 'proxywetn2o5': <function
compute_wetoxn_from_concprcpoxn>, 'proxywetnh3': <function
compute_wetrdn_from_concprcprdn>, 'proxywetnh4': <function
compute_wetrdn_from_concprcprdn>, 'proxywetno2': <function
compute_wetoxn_from_concprcpoxn>, 'proxywetno2no2': <function
compute_wetoxn_from_concprcpoxn>, 'proxywetno3c': <function
compute_wetoxn_from_concprcpoxn>, 'proxywetno3f': <function
compute_wetoxn_from_concprcpoxn>, 'proxyweto3': <function
make_proxy_wetdep_from_O3>, 'proxywetoxn': <function
compute_wetoxn_from_concprcpoxn>, 'proxywetoxs': <function
compute_wetoxs_from_concprcpoxs>, 'proxywetpm10': <function
compute_wetoxs_from_concprcpoxs>, 'proxywetpm25': <function
compute_wetoxs_from_concprcpoxs>, 'proxywetrdn': <function
compute_wetrdn_from_concprcprdn>, 'proxywetso2': <function
compute_wetoxs_from_concprcpoxs>, 'proxywetso4': <function
compute_wetoxs_from_concprcpoxs>, 'sc440dryaer': <function compute_sc440dryaer>,
'sc550dryaer': <function compute_sc550dryaer>, 'sc700dryaer': <function
compute_sc700dryaer>, 'vmro3max': <function calc_vmro3max>, 'wetna': <function
compute_wetna_from_concprcpna>, 'wetnh4': <function
compute_wetnh4_from_concprcpnh4>, 'wetno3': <function
compute_wetno3_from_concprcpno3>, 'wetoxn': <function
compute_wetoxn_from_concprcpoxn>, 'wetoxs': <function
compute_wetoxs_from_concprcpoxs>, 'wetoxsc': <function
compute_wetoxs_from_concprcpoxsc>, 'wetoxst': <function
compute_wetoxs_from_concprcpoxst>, 'wetrdn': <function
compute_wetrdn_from_concprcprdn>, 'wetso4': <function
compute_wetso4_from_concprcpso4>}
```

Functions supposed to be used for computation of auxiliary variables

AUX_REQUIRES = {'ac550dryaer': ['ac550aer', 'acrh'], 'ang4470dryaer': ['sc440dryaer', 'sc700dryaer'], 'proxydryhno3': ['concprcpoxn', 'pr'], 'proxydryhono': ['concprcpoxn', 'pr'], 'proxydryn2o5': ['concprcpoxn', 'pr'], 'proxydryna': ['concprcpna', 'pr'], 'proxydrynh3': ['concprcprdn', 'pr'], 'proxydrynh4': ['concprcprdn', 'pr'], 'proxydryno2': ['concprcpoxn', 'pr'], 'proxydryno2no2': ['concprcpoxn', 'pr'], 'proxydryno3c': ['concprcpoxn', 'pr'], 'proxydryno3f': ['concprcpoxn', 'pr'], 'proxydryo3': ['vmro3'], 'proxydryoxn': ['concprcpoxn', 'pr'], 'proxydryoxs': ['concprcpoxs', 'pr'], 'proxydrypm10': ['concprcpoxs', 'pr'], 'proxydrypm25': ['concprcpoxs', 'pr'], 'proxydryrdn': ['concprcprdn', 'pr'], 'proxydryso2': ['concprcpoxs', 'pr'], 'proxydryso4': ['concprcpoxs', 'pr'], 'proxydryss': ['concprcpna', 'pr'], 'proxywethno3': ['concprcpoxn', 'pr'], 'proxywethono': ['concprcpoxn', 'pr'], 'proxywetn2o5': ['concprcpoxn', 'pr'], 'proxywetnh3': ['concprcprdn', 'pr'], 'proxywetnh4': ['concprcprdn', 'pr'], 'proxywetno2': ['concprcpoxn', 'pr'], 'proxywetno2no2': ['concprcpoxn', 'pr'], 'proxywetno3c': ['concprcpoxn', 'pr'], 'proxywetno3f': ['concprcpoxn', 'pr'], 'proxyweto3': ['vmro3'], 'proxywetoxn': ['concprcpoxn', 'pr'], 'proxywetoxs': ['concprcpoxs', 'pr'], 'proxywetpm10': ['concprcpoxs', 'pr'], 'proxywetpm25': ['concprcpoxs', 'pr'], 'proxywetrdn': ['concprcprdn', 'pr'], 'proxywetso2': ['concprcpoxs', 'pr'], 'proxywetso4': ['concprcpoxs', 'pr'], 'sc440dryaer': ['sc440aer', 'scrh'], 'sc550dryaer': ['sc550aer', 'scrh'], 'sc700dryaer': ['sc700aer', 'scrh'], 'vmro3max': ['vmro3'], 'wetna': ['concprcpna', 'pr'], 'wetnh4': ['concprcpnh4', 'pr'], 'wetno3': ['concprcpno3', 'pr'], 'wetoxn': ['concprcpoxn', 'pr'], 'wetoxs': ['concprcpoxs', 'pr'], 'wetoxsc': ['concprcpoxsc', 'pr'], 'wetoxst': ['concprcpoxst', 'pr'], 'wetrdn': ['concprcprdn', 'pr'], 'wetso4': ['concprcpso4', 'pr']}

variables required for computation of auxiliary variables

AUX_USE_META = {'ac550dryaer': 'ac550aer', 'sc440dryaer': 'sc440aer', 'sc550dryaer': 'sc550aer', 'sc700dryaer': 'sc700aer'}

Meta information supposed to be migrated to computed variables

property AUX_VARS

List of auxiliary variables (keys of attr. *AUX_REQUIRES*)

Auxiliary variables are those that are not included in original files but are computed from other variables during import

CACHE_SQLITE_FILE = ['EBASMC']

For the following data IDs, the sqlite database file will be cached if const.EBAS_DB_LOCAL_CACHE is True

property DATASET_PATH

Wrapper for *data_dir*.

DATA_ID = 'EBASMC'

Name of dataset (OBS_ID)

property DEFAULT_VARS

list of default variables to be read

Note: Currently a wrapper for *PROVIDES_VARIABLES*

Type
list

property FILE_REQUEST_OPTS

List of options for file retrieval

FILE_SUBDIR_NAME = 'data'

Name of subdirectory containing data files (relative to *data_dir*)

IGNORE_COLS_CONTAIN = ['fraction', 'artifact']

Ignore data columns in NASA Ames files that contain any of the listed attributes

**IGNORE_FILES = ['CA0420G.20100101000000.20190125102503.filter_absorption_photometer.aerosol_absorption_coefficient.aerosol.1y.1h.CA01L_Magee_AE31_ALT.CA01L_aethalometer.lev2.nas',
'DK0022R.20180101070000.20191014000000.bulk_sampler..precip.1y.15d.DK01L_bs_22.DK01L_IC.lev2.nas',
'DK0012R.20180101070000.20191014000000.bulk_sampler..precip.1y.15d.DK01L_bs_12.DK01L_IC.lev2.nas',
'DK0008R.20180101070000.20191014000000.bulk_sampler..precip.1y.15d.DK01L_bs_08.DK01L_IC.lev2.nas',
'DK0005R.20180101070000.20191014000000.bulk_sampler..precip.1y.15d.DK01L_bs_05.DK01L_IC.lev2.nas']**

list of EBAS data files that are flagged invalid and will not be imported

IGNORE_META_KEYS = []

MERGE_STATIONS = {'Birkenes': 'Birkenes II', 'Rörvik': 'Råö', 'Vavihill': 'Hallahus', 'Virolahti II': 'Virolahti III'}

property NAN_VAL

Irrelevant for implementation of EBAS I/O

property PROVIDES_VARIABLES

List of variables provided by the interface

property REVISION_FILE

Name of revision file located in data directory

SQL_DB_NAME = 'ebas_file_index.sqlite3'

Name of sqlite database file

SUPPORTED_DATASETS = ['EBASMC']

List of all datasets supported by this interface

TS_TYPE = 'undefined'

TS_TYPE_CODES = {'1d': 'daily', '1h': 'hourly', '1mn': 'minutely', '1mo': 'monthly', '1w': 'weekly', 'd': 'daily', 'h': 'hourly', 'mn': 'minutely', 'mo': 'monthly', 'w': 'weekly'}

Temporal resolution codes that (so far) can be understood by pyaerocom

VAR_READ_OPTS = {'pr': {'convert_units': False, 'freq_min_cov': 0.75}, 'prmm': {'freq_min_cov': 0.75}}

Custom reading options for individual variables. Keys need to be valid attributes of *ReadEbasOptions* and anything specified here (for a given variable) will be overwritten from the defaults specified in the options class.

property all_station_names

List of all available station names in EBAS database

check_vars_to_retrieve(*vars_to_retrieve*)

Separate variables that are in file from those that are computed

Some of the provided variables by this interface are not included in the data files but are computed within this class during data import (e.g. od550aer, ang4487aer).

The latter may require additional parameters to be retrieved from the file, which is specified in the class header (cf. attribute `AUX_REQUIRES`).

This function checks the input list that specifies all required variables and separates them into two lists, one that includes all variables that can be read from the files and a second list that specifies all variables that are computed in this class.

Parameters
vars_to_retrieve (*list*) – all parameter names that are supposed to be loaded

Returns

2-element tuple, containing

- list: list containing all variables to be read
- list: list containing all variables to be computed

Return type
tuple

compute_additional_vars(*data*, *vars_to_compute*)

Compute additional variables and put into station data

Note: Extended version of `ReadUngriddedBase.compute_additional_vars()`

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_compute** (*list*) – list of variable names that are supposed to be computed. Variables that are required for the computation of the variables need to be specified in *AUX_VARS* and need to be available as data vectors in the provided data dictionary (key is the corresponding variable name of the required variable).

Returns
updated data object now containing also computed variables

Return type
dict

property data_dir: str

Location of the dataset

Note: This can be set explicitly when instantiating the class (e.g. if data is available on local machine). If unspecified, the data location is attempted to be inferred via `get_obsnetwork_dir()`

Raises
FileNotFoundError – if data directory does not exist or cannot be retrieved automatically

Type
str

property data_id

ID of dataset

property data_revision

Revision string from file Revision.txt in the main data directory

property file_dir

Directory containing EBAS NASA Ames files

property file_index

SQlite file mapping metadata with filenames

files_contain

this is filled in method get_file_list and specifies variables to be read from each file

find_in_file_list(*pattern=None*)

Find all files that match a certain wildcard pattern

Parameters
pattern (str, optional) – wildcard pattern that may be used to narrow down the search (e.g. use `pattern=*Berlin*` to find only files that contain Berlin in their filename)

Returns
list containing all files in `files` that match pattern

Return type
list

Raises
IOError – if no matches can be found

find_var_cols(*vars_to_read*, *loaded_nasa_ames*)

Find best-match variable columns in loaded NASA Ames file

For each of the input variables, try to find one or more matches in the input NASA Ames file (loaded data object). If more than one match occurs, identify the best one (an example here is: user wants sc550aer and file contains scattering coefficients at 530 nm and 580 nm: in this case the 530 nm column will be used, cf. also accepted wavelength tolerance for reading of wavelength dependent variables `wavelength_tol_nm`).

Parameters

- **vars_to_read** (*list*) – list of variables that are supposed to be read
- **loaded_nasa_ames** (EbasNasaAmesFile) – loaded data object

Returns
dictionary specifying the best-match variable column for each of the input variables.

Return type
dict

get_ebas_var(*var_name*)

Get instance of `EbasVarInfo` for input AeroCom variable

get_file_list(*vars_to_retrieve*, ***constraints*)

Get list of files for all variables to retrieve

Parameters

- **vars_to_retrieve** (*list or str*) – list of variables that are supposed to be read
- ****constraints** – further EBAS request constraints deviating from default (default info for each AEROCOM variable can be found in `ebas_config.ini <https://github.com/metno/pyaerocom/blob/master/pyaerocom/data/ebas_config.ini>`__). For details on possible input parameters see `EbasSQLRequest` (or this tutorial)

Returns
unified list of file paths each containing either of the specified variables

Return type
list

get_read_opts(*var_name*)

Get reading options for input variable

Parameters
var_name (*str*) – name of variable

Returns
options

Return type
EbasReadOptions

logger

Class own instance of logger class

read(*vars_to_retrieve=None*, *first_file=None*, *last_file=None*, *files=None*, ***constraints*)

Method that reads list of files as instance of `UngriddedData`

Parameters

- **vars_to_retrieve** (`list` or similar, optional,) – list containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded
- **first_file** (`int`, optional) – index of first file in file list to read. If None, the very first file in the list is used
- **last_file** (`int`, optional) – index of last file in list to read. If None, the very last file in the list is used
- **files** (*list*) – list of files
- ****constraints** – further reading constraints deviating from default (default info for each AEROCOM variable can be found in `ebas_config.ini <https://github.com/metno/pyaerocom/blob/master/pyaerocom/data/ebas_config.ini>`__). For details on possible input parameters see `EbasSQLRequest` (or this tutorial)

Returns
data object

Return type
UngriddedData

read_file(*filename*, *vars_to_retrieve=None*, *_vars_to_read=None*, *_vars_to_compute=None*)

Read EBAS NASA Ames file

Parameters

- **filename** (*str*) – absolute path to filename to read

- **vars_to_retrieve** (*list*, optional) – list of str with variable names to read, if None (and if not both of the alternative possible parameters `_vars_to_read` and `_vars_to_compute` are specified explicitely) then the default settings are used

Returns
dict-like object containing results

Return type
StationData

read_first_file(***kwargs*)

Read first file returned from *get_file_list()*

Note: This method may be used for test purposes.

Parameters
****kwargs** – keyword args passed to *read_file()* (e.g. vars_to_retrieve)

Returns
dictionary or similar containing loaded results from first file

Return type
dict-like

read_station(*station_id_filename*, ***kwargs*)

Read data from a single station into `UngriddedData`

Find all files that contain the station ID in their filename and then call *read()*, providing the reduced filelist as input, in order to read all files from this station into data object.

Parameters

- **station_id_filename** (*str*) – name of station (MUST be encrypted in filename)
- ****kwargs** – additional keyword args passed to *read()* (e.g. `vars_to_retrieve`)

Returns
loaded data

Return type
UngriddedData

Raises
IOError – if no files can be found for this station ID

property readopts_default

Default reading options

These are applied to all variables if not defined explicitly for individual variables in :attr:`REA

remove_outliers(*data*, *vars_to_retrieve*, ***valid_rng_vars*)

Remove outliers from data

Parameters

- **data** (*dict-like*) – data object containing data vectors for variables that are required for computation (cf. input param `vars_to_compute`)
- **vars_to_retrieve** (*list*) – list of variable names for which outliers will be removed from data

- ****valid_rng_vars** – additional keyword args specifying variable name and corresponding min / max interval (list or tuple) that specifies valid range for the variable. For each variable that is not explicitely defined here, the default minimum / maximum value is used (accessed via `pyaerocom.const.VARS[var_name]`)

property sqlite_database_file

Path to EBAS SQL database

var_info(*var_name*)

Aerocom variable info for input var_name

var_supported(*var_name*)

Check if input variable is supported

Parameters
var_name (*str*) – AeroCom variable name or alias

Raises
VariableDefinitionError – if input variable is not supported by pyaerocom

Returns
True, if variable is supported by this interface, else False

Return type
bool

property verbosity_level

Current level of verbosity of logger

class pyaerocom.io.read_ebas.**ReadEbasOptions**(***args*)

Bases: *BrowseDict*

Options for EBAS reading routine

prefer_statistics

preferred order of data statistics. Some files may contain multiple columns for one variable, where each column corresponds to one of the here defined statistics that where applied to the data. This attribute is only considered for ebas variables, that have not explicitely defined what statistics to use (and in which preferred order, if applicable). Reading preferences for all Ebas variables are specified in the file ebas_config.ini in the data directory of pyaerocom.

Type
list

ignore_statistics

columns that have either of these statistics applied are ignored for variable data reading.

Type
list

wavelength_tol_nm

Wavelength tolerance in nm for reading of (wavelength dependent) variables. If multiple matches occur (e.g. query -> variable at 550nm but file contains 3 columns of that variable, e.g. at 520, 530 and 540 nm), then the closest wavelength to the queried wavelength is used within the specified tolerance level.

Type
int

shift_wavelengths

(only for wavelength dependent variables). If True, and a data columns candidate is valid within wavelength tolerance around desired wavelength, that column will be considered to be used for data import. Defaults to True.

Type
bool

assume_default_ae_if_unavail

assume an Angstrom Exponent for applying wavelength shifts of data. See *ReadEbas.ASSUME_AE_SHIFT_WVL* and *ReadEbas.ASSUME_AAE_SHIFT_WVL* for AE and AAE assumptions related to scattering and absorption coeffs. Defaults to True.

Type
bool

check_correct_MAAP_wrong_wvl

(BETA, do not use): set correct wavelength for certain absorption coeff measurements. Defaults to False.

Type
bool

eval_flags

If True, the flag columns in the NASA Ames files are read and decoded (using `EbasFlagCol.decode()`) and the (up to 3 flags for each measurement) are evaluated as valid / invalid using the information in the flags CSV file. The evaluated flags are stored in the data files returned by the reading methods *ReadEbas.read()* and *ReadEbas.read_file()*.

Type
bool

keep_aux_vars

if True, auxiliary variables required for computed variables will be written to the `UngriddedData` object created in *ReadEbas.read()* (e.g. if sc550dryaer is requested, this requires reading of sc550aer and scrh. The latter 2 will be written to the data object if this parameter evaluates to True)

Type
bool

convert_units

if True, variable units in EBAS files will be checked and attempted to be converted into AeroCom default unit for that variable. Defaults to True.

Type
bool

try_convert_vmr_conc

attempt to convert vmr data to conc if user requires conc (e.g. user wants conco3 but file only contains vmro3), and vice versa.

Type
bool

ensure_correct_freq

if True, the frequency set in NASA Ames files (provided via attr *resolution_code*) is checked using time differences inferred from start and stop time of each measurement. Measurements that are not in that resolution (within 5% tolerance level) will be flagged invalid.

Type
bool

freq_from_start_stop_meas

infer frequency from start / stop intervals of individual measurements.

Type
bool

freq_min_cov

defines minimum number of measurements that need to correspond to the detected sampling frequency in the file within the specified tolerance range. Only applies if *ensure_correct_freq* is True. E.g. if a file contains 100 measurements and the most common frequency (as inferred from stop-start of each measurement) is daily. Then, if *freq_min_cov* is 0.75, it will be ensured that at least 75 of the measurements are daily (within +/- 5% tolerance), otherwise this file is discarded. Defaults to 0.

Type
float

Parameters
****args** – key / value pairs specifying any of the supported settings.

ADD_GLOB = []

FORBIDDEN_KEYS = []

IGNORE_JSON = []

Keys to be ignored when converting to json

MAXLEN_KEYS = 100.0

SETTER_CONVERT = {}

clear() → None. Remove all items from D.

property filter_dict

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

import_from(*other*) → None

Import key value pairs from other object

Other than *update()* this method will silently ignore input keys that are not contained in this object.

Parameters
other (*dict or BrowseDict*) – other dict-like object containing content to be updated.

Raises
ValueError – If input is inalid type.

Return type
None

items() → a set-like object providing a view on D's items

json_repr() → dict

Convert object to serializable json dict

Returns
content of class

Return type
dict

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

pretty_str()

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

to_dict()

update([*E*,]**F) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

EBAS (low level)

Pyearocom module for reading and processing of EBAS NASA Ames files

For details on the file format see here

class pyaerocom.io.ebas_nasa_ames.**EbasColDef**(*name*, *is_var*, *is_flag*, *unit='1'*)
Dict-like object for EBAS NASA Ames column definitions

Note: The meta attribute name 'unit' can also be accessed using the CF attr name 'units'

name
column name

Type
str

unit
unit of data in column (if applicable)

Type
str

is_var
True if column corresponds to variable data, False if not

Type
bool

is_flag
True, if column corresponds to Flag column, False if not

Type
bool

flag_col

column number of flag column that corresponds to this data column (only relevant if *is_var* is True)

Type
int

Parameters

- **name** (*str*) – column name
- **is_var** (*bool*) – True if column corresponds to variable data, False if not
- **is_flag** (*bool*) – True, if column corresponds to Flag column, False if not
- **unit** (str, optional) – unit of data in column (if applicable)
- **flag_col** (str, optional) – name of flag column that corresponds to this data colum (only relevant if *is_var* is True)

get_wavelength_nm()

Try to access wavelength information in nm (as float)

to_dict(*ignore_keys=['is_var', 'is_flag', 'flag_col', 'wavelength_nm']*)

class pyaerocom.io.ebas_nasa_ames.**EbasFlagCol**(*raw_data*, *interpret_on_init=True*)

Simple helper class to decode and interpret EBAS flag columns

raw_data

raw flag column (containing X-digit floating point numbers)

Type
ndarray

property FLAG_INFO

Detailed information about EBAS flag definitions

decode()

Decode raw flag column

property decoded

Nx3 numpy array containing decoded flag columns

property valid

Boolean array specifying valid and invalid measurements

class pyaerocom.io.ebas_nasa_ames.**EbasNasaAmesFile**(*file=None*, *only_head=False*, *replace_invalid_nan=True*, *convert_timestamps=True*, *evaluate_flags=False*, *quality_check=True*, ***kwargs*)

EBAS NASA Ames file interface

Class interface for reading and processing of EBAS NASA Ames file

time_stamps

array containing datetime64 objects with timestamps

Type
ndarray

flags

dictionary containing *EbasFlagCol* objects for each column containing flags

Type
dict

Parameters

- **file** (str, optional) – EBAS NASA Ames file. if valid file path, then the file is read on init (please note following options for import)
- **only_head** (*bool*) – read only file header
- **replace_invalid_nan** (*bool*) – replace all invalid values in the table by NaNs. The invalid values for each dependent data column are identified based on the information in the file header.
- **convert_timestamps** (*bool*) – compute array of numpy datetime64 timestamps from numeric timestamps in data
- **evaluate_flags** (*bool*) – if True, all flags in all flag columns are decoded from floating point representation to 3 integers, e.g. 0.111222333 -> 111 222 333
- **quality_check** (*bool*) – perform quality check after import (for details see _quality_check())
- ****kwargs** – optional input args that are passed to init of *NasaAmesHeader* base class

ERR_HIGH_STATS = 'percentile:84.13'

ERR_LOW_STATS = 'percentile:15.87'

TIMEUNIT2SECFAC = {'Days': 86400, 'days': 86400}

all_cols_contain(*colnums*, *what*)

Check if all input columns contain input attr *what*

Parameters

- **colnums** (*list*) – list of column numbers
- **what** (*str*) – name of attribute (e.g. *matrix*, *statistics*, *tower_inlet_height*)

Returns
True if all input columns contain *what* attr., else False.

Return type
bool

assign_flagcols()

property base_date

Base date of data as numpy.datetime64[s]

property col_names

Column names of table

property col_names_vars

Names of all columns that are flagged as variables

property col_num

Number of columns in table

property col_nums_vars

Column index number of all variables

compute_time_stamps()

Compute time stamps from first two data columns

property data

2D numpy array containing data table

property data_header

get_dt_meas(*np_freq='s'*)

Get array with time between individual measurements

This is computed based on start and timestamps, e.g. *dt[0] = start[1] - start[0]*

Parameters
np_freq (*str*) – string specifying output frequency of gap values

Returns
array with time-differences as floating point number in specified input resolution

Return type
ndarray

get_time_gaps_meas(*np_freq='s'*)

Get array with time gaps between individual measurements

This is computed based on start and stop timestamps, e.g. *=dt[0] = start[1] - stop[0]*

Parameters
np_freq (*str*) – string specifying output frequency of gap values

Returns
array with time-differences as floating point number in specified input resolution

Return type
ndarray

init_flags(*evaluate=True*)

Decode flag columns and store info in *flags*

static numarr_to_datetime64(*basedate*, *num_arr*, *mulfac_to_sec*)

Convert array of numerical timestamps into datetime64 array

Parameters

- **basedate** (*datetime64*) – reference date
- **num_arr** (*ndarray*) – numerical time stamps relative to basedate
- **mulfac_to_sec** (*float*) – multiplicative factor to convert numerical values to unit of seconds

Returns
array containing timestamps as datetime64 objects

Return type
ndarray

print_col_info()

Print information about individual columns

read_file(*nasa_ames_file*, *only_head=False*, *replace_invalid_nan=True*, *convert_timestamps=True*, *evaluate_flags=False*, *quality_check=False*)

Read NASA Ames file

Parameters

- **nasa_ames_file** (*str*) – EBAS NASA Ames file
- **only_head** (*bool*) – read only file header
- **replace_invalid_nan** (*bool*) – replace all invalid values in the table by NaNs. The invalid values for each dependent data column are identified based on the information in the file header.
- **convert_timestamps** (*bool*) – compute array of numpy datetime64 timestamps from numeric timestamps in data
- **evaluate_flags** (*bool*) – if True, all data columns get assigned their corresponding flag column, the flags in all flag columns are decoded from floating point representation to 3 integers, e.g. 0.111222333 -> 111 222 333 and if input `replace_invalid_nan==True`, then the invalid measurements in each column are replaced with NaN's.
- **quality_check** (*bool*) – perform quality check after import (for details see _quality_check())

read_header(*nasa_ames_file*, *quality_check=True*)

property shape

Shape of data array

property time_unit

Time unit of data

class pyaerocom.io.ebas_nasa_ames.**NasaAmesHeader**(***kwargs*)

Header class for Ebas NASA Ames file

Note: Is used in *EbasNasaAmesFile* and should not be used directly.

CONV_FLOAT()

CONV_INT()

CONV_MULTIFLOAT()

CONV_MULTIINT()

CONV_PI()

CONV_STR()

property head_fix

Dictionary containing fixed header info (that is always available)

property meta

Meta data dictionary (specific for this file)

update(***kwargs*)

property var_defs

List containing column variable definitions

List index is column index in file and value is instance of *EbasColDef*

class pyaerocom.io.ebas_file_index.**EbasFileIndex**(*database=None*)

EBAS SQLite I/O interface

Takes care of connection to database and execution of requests

property ALL_INSTRUMENTS

List of all variables available

property ALL_MATRICES

List of all matrix values available

property ALL_STATION_CODES

List of all available station codes in database

Note: Not tested whether the order is the same as the order in `STATION_NAMES`, i.e. the lists should not be linked to each other

---

property ALL_STATION_NAMES

List of all available station names in database

property ALL_STATISTICS_PARAMS

List of all statistical parameters available

For more info see here

property ALL_VARIABLES

List of all variables available

property database

Path to ebas_file_index.sqlite3 file

execute_request(*request*, *file_request=False*)

Connect to database and retrieve data for input request

Parameters
request (*EbasSQLRequest* or `str`) – request specifications

Returns
list of tuples containing the retrieved results. The number of items in each tuple corresponds to the number of requested parameters (usually one, can be specified in `make_query_str()` using argument `what`)

Return type
list

get_file_names(*request*)

Get all files that match the request specifications

Parameters
request (*EbasSQLRequest* or `str`) – request specifications

Returns
list of file paths that match the request

Return type
list

`get_table_columns`(*table_name*)

Get all columns of a table in SQLite database file

`get_table_names`()

Get all table names in SQLite database file

class pyaerocom.io.ebas_file_index.**EbasSQLRequest**(*variables=None*, *start_date=None*, *stop_date=None*, *station_names=None*, *matrices=None*, *altitude_range=None*, *lon_range=None*, *lat_range=None*, *instrument_types=None*, *statistics=None*, *datalevel=None*)

Low level dictionary like object for EBAS sqlite queries

`variables`

tuple containing variable names to be extracted (e.g. `('aerosol_light_scattering_coefficient', 'aerosol_optical_depth')`). If None, all available is used

Type
tuple, optional

`start_date`

start date of data request (format YYYY-MM-DD). If None, all available is used

Type
str, optional

`stop_date`

stop date of data request (format YYYY-MM-DD). If None, all available is used

Type
str, optional

`station_names`

tuple containing station_names of request (e.g. `('Birkenes II', 'Asa')`).If None, all available is used

Type
tuple, optional

`matrices`

tuple containing station_names of request (e.g. `('pm1', 'pm10', 'pm25', 'aerosol')`) If None, all available is used

Type
tuple, optional

`altitude_range`

tuple specifying altitude range of station in m (e.g. `(0.0, 500.0)`). If None, all available is used

Type
tuple, optional

`lon_range`

tuple specifying longitude range of station in degrees (e.g. `(-20, 20)`). If None, all available is used

Type
tuple, optional

lat_range

tuple specifying latitude range of station in degrees (e.g. `(50, 80)`). If None, all available is used

Type
`tuple`, optional

instrument_type

string specifying instrument types (e.g. `("nephelometer")`)

Type
`str`, optional

statistics

string specifying statistics code (e.g. `("arithmetic mean")`)

Type
`tuple`, optional

Parameters
Attributes (*see*)

make_file_query_str(*distinct=True*, ***kwargs*)

Wrapper for base method *make_query_str()*

Parameters

- **distinct** (*bool*) – return unique files
- ****kwargs** – update request attributes (e.g. `lon_range=(30, 60)`)

Returns
SQL file request command for current specs

Return type
str

make_query_str(*what=None*, *distinct=True*, ***kwargs*)

Translate current class state into SQL query command string

Parameters

- **what** (*str or tuple, optional*) – what columns to retrieve (e.g. comp_name for all variables) from table specified. Defaults to None, in which case "filename" is used
- **distinct** (*bool*) – return unique files
- ****kwargs** – update request attributes (e.g. `lon_range=(30, 60)`)

Returns
SQL file request command for current specs

Return type
str

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

class pyaerocom.io.ebas_varinfo.**EbasVarInfo**(*var_name: str*, *init: bool = True*, ***kwargs*)

Interface for mapping between EBAS variable information and AeroCom

For more information about EBAS variable and data information see EBAS website.

var_name

AeroCom variable name

Type
str

component

list of EBAS variable / component names that are mapped to *var_name*

Type
list

matrix

list of EBAS matrix values that are accepted, default is None, i.e. all available matrices are used

Type
list, optional

instrument

list of all instruments that are accepted for this variable

Type
list, optional

requires

for variables that are computed and not directly available in EBAS. Provided as list of (AeroCom) variables that are required to compute *var_name* (e.g. for *sc550dryaer* this would be *[sc550aer,scrh]*).

Type
list, optional

scale_factor

multiplicative scale factor that is applied in order to convert EBAS variable into AeroCom variable (e.g. 1.4 for conversion of EBAS OC measurement to AeroCom concoa variable)

Type
float, optional

Parameters

- **var_name** (*str*) – AeroCom variable name
- **init** (*bool*) – if True, EBAS configuration for input variable is retrieved from data file ebas_config.ini (if possible)
- ****kwargs** – additional keyword arguments (currently not used)

static PROVIDES_VARIABLES() → list[str]

List specifying provided variables

instrument

list of instrument names (EBAS side, optional)

make_sql_request(***constraints*) → *EbasSQLRequest*

Create an SQL request for the specifications in this object

Parameters
constraints – request constraints deviating from default. For details on parameters see `EbasSQLRequest`

Returns
the SQL request object that can be used to retrieve corresponding file names using instance of `EbasFileIndex.get_file_names()`.

Return type
EbasSQLRequest

make_sql_requests(***constraints*) → list[*EbasSQLRequest*]

Create a list of SQL requests for the specifications in this object

Parameters

- **requests** (*dict, optional*) – other SQL requests linked to this one (e.g. if this variable requires)
- **constraints** – request constraints deviating from default. For details on parameters see `EbasSQLRequest`

Returns
list of `EbasSQLRequest` instances for this component and potential required components.

Return type
list

matrix

list of matrix names (EBAS side, optional)

static open_config()

Open ebas_config.ini file with *ConfigParser*

Return type
ConfigParser

parse_from_ini(*var_name: str*, *conf_reader: ConfigParser | None = None*)

Parse EBAS info for input AeroCom variable (works also for aliases)

Parameters

- **var_name** (*str*) – AeroCom variable name
- **conf_reader** (*ConfigParser*) – open config parser object

Raises
VarNotAvailableError – if variable is not supported

Returns
True, if default could be loaded, False if not

Return type
bool

requires

list of additional variable required for retrieval of this variable

scale_factor

scale factor for conversion to Aerocom units

statistics

list containing variable statistics info (EBAS side, optional)

to_dict() → dict

Convert into dictionary

property var_name_aerocom: str
Variable name in AeroCom convention

4.5.6 EEA data

EEA base reader

Reader for European air pollution data from EEA AqERep files. Interface for reading EEA AqERep files (formerly known as Airbase data).

class pyaerocom.io.read_eea_aqerep_base.**ReadEEAAQEREPBase**(*data_id=None*, *data_dir=None*)
Class for reading EEA AQErep data

Extended class derived from low-level base class ReadUngriddedBase that contains some more functionality.

Note: Currently only single variable reading into an UngriddedData object is supported.

ALTITUDENAME = 'altitude'
name of altitude variable in metadata file

AUX_FUNS = {'concNno': NotImplementedError(), 'concNno2': NotImplementedError(), 'concSso2': NotImplementedError(), 'vmrno2': NotImplementedError(), 'vmro3': NotImplementedError(), 'vmro3max': NotImplementedError()}
Functions that are used to compute additional variables (i.e. one for each variable defined in AUX_REQUIRES)

AUX_REQUIRES = {'concNno': ['concno'], 'concNno2': ['concno2'], 'concSso2': ['concso2'], 'vmrno2': ['concno2'], 'vmro3': ['conco3'], 'vmro3max': ['conco3']}
dictionary containing information about additionally required variables for each auxiliary variable (i.e. each variable that is not provided by the original data but computed on import)

CONV_FACTOR = {'concNno': 0.466788868521913, 'concNno2': 0.3044517868011477, 'concSso2': 0.50052292274792, 'vmrno2': 0.514, 'vmro3': 0.493, 'vmro3max': 0.493}

CONV_UNIT = {'concNno': 'µgN/m3', 'concNno2': 'µgN/m3', 'concSso2': 'µgS/m3', 'vmrno2': 'ppb', 'vmro3': 'ppb', 'vmro3max': 'ppb'}

property DATASET_NAME
Name of the dataset

DATA_ID = ''
Name of the dataset (OBS_ID)

DATA_PRODUCT = ''

DEFAULT_METADATA_FILE = 'metadata.csv'

property DEFAULT_VARS
List of default variables

END_TIME_NAME = 'datetimeend'
filed name of the end time of the measurement (in lower case)

FILE_COL_DELIM = ','
Column delimiter

FILE_MASKS = {'concNno': '/??_38_*_timeseries.csv*', 'concNno2': '**/??_8_*_timeseries.csv*', 'concSso2': '**/??_1_*_timeseries.csv*', 'concco': '**/??_10_*_timeseries.csv*', 'concno': '**/??_38_*_timeseries.csv*', 'concno2': '**/??_8_*_timeseries.csv*', 'conco3': '**/??_7_*_timeseries.csv*', 'concpm10': '**/??_5_*_timeseries.csv*', 'concpm25': '**/??_6001_*_timeseries.csv*', 'concso2': '**/??_1_*_timeseries.csv*', 'vmrno2': '**/??_8_*_timeseries.csv*', 'vmro3': '**/??_7_*_timeseries.csv*', 'vmro3max': '**/??_7_*_timeseries.csv*'}**

file masks for the data files

INSTRUMENT_NAME = 'unknown'

there's no general instrument name in the data

LATITUDENAME = 'latitude'

Name of latitude variable in metadata file

LONGITUDENAME = 'longitude'

name of longitude variable in metadata file

MAX_LINES_TO_READ = 8784

NAN_VAL = {}

Dictionary specifying values corresponding to invalid measurements there's no value for NaNs in this data set. It uses an empty string

PROVIDES_VARIABLES = ['concso2', 'conco3', 'concno2', 'concco', 'concno', 'concpm10', 'concpm25', 'vmro3', 'vmro3max', 'vmrno2', 'concSso2', 'concNno', 'concNno2']

List of variables that are provided by this dataset (will be extended by auxiliary variables on class init, for details see __init__ method of base class ReadUngriddedBase)

START_TIME_NAME = 'datetimebegin'

field name of the start time of the measurement (in lower case)

SUPPORTED_DATASETS = ['']

List of all datasets supported by this interface

TS_TYPE = 'variable'

There is no global ts_type but it is specified in the data files…

TS_TYPES_FILE = {'day': 'daily', 'hour': 'hourly'}

sampling frequencies found in data files

VAR_CODES = {'1': 'concso2', '10': 'concco', '38': 'concno', '5': 'concpm10', '6001': 'concpm25', '7': 'conco3', '8': 'concno2'}

dictionary that connects the EEA variable codes with aerocom variable names

VAR_CODE_NAME = 'airpollutantcode'

column name that holds the EEA variable code

VAR_NAMES_FILE = {'concNno': 'concentration', 'concNno2': 'concentration', 'concSso2': 'concentration', 'concco': 'concentration', 'concno': 'concentration', 'concno2': 'concentration', 'conco3': 'concentration', 'concpm10': 'concentration', 'concpm25': 'concentration', 'concso2': 'concentration', 'vmrno2': 'concentration', 'vmro3': 'concentration', 'vmro3max': 'concentration'}

VAR_UNITS_FILE = {'mg/m3': 'mg m-3', 'ppb': 'ppb', 'µg/m3': 'ug m-3', 'µgN/m3': 'ug N m-3', 'µgS/m3': 'ug S m-3'}

units of variables in files (needs to be defined for each variable supported)

WEBSITE = 'https://discomap.eea.europa.eu/map/fme/AirQualityExport.htm'

this class reads the European Environment Agency's Eionet data for details please read https://www.eea.europa.eu/about-us/countries-and-eionet

get_file_list(*pattern=None*)

Search all files to be read

Uses _FILEMASK (+ optional input search pattern, e.g. station_name) to find valid files for query.

Parameters
pattern (*str, optional*) – file name pattern applied to search

Returns
list containing retrieved file locations

Return type
list

Raises
IOError – if no files can be found

get_station_coords(*meta_key*)

get a station's coordinates

Parameters
meta_key (*str*) – string with the internal station key

read(*vars_to_retrieve=None*, *files=None*, *first_file=None*, *last_file=None*, *metadatafile=None*)

Method that reads list of files as instance of UngriddedData

Parameters

- **vars_to_retrieve** (list or similar, optional) – List containing variable IDs that are supposed to be read. If None, all variables in *PROVIDES_VARIABLES* are loaded.
- **files** (list, optional) – List of files to be read. If None, then the file list used is the returned from *get_file_list()*.
- **first_file** (int, optional) – Index of the first file in :obj:'file' to be read. If None, the very first file in the list is used.
- **last_file** (int, optional) – Index of the last file in :obj:'file' to be read. If None, the very last file in the list is used.
- **metadatafile** (*:obj:'str', optional*) – full qualified path to metadata file. If None, the default metadata file will be used

Returns
data object

Return type
UngriddedData

read_file(*filename*, *var_name*, *vars_as_series=False*)

Read a single EEA file

Note that there's only a single variable in the file

Parameters

- **filename** (*str*) – Absolute path to filename to read.
- **var_name** (*str*) – Name of variable in file.
- **vars_as_series** (*bool*) – If True, the data columns of all variables in the result dictionary are converted into pandas Series objects.

Returns
Dict-like object containing the results.

Return type
StationData

EEA E2a product (NRT)

Near realtime EEA data. Interface for reading EEA AqERep files (formerly known as Airbase data).

class pyaerocom.io.read_eea_aqerep.**ReadEEAAQEREP**(*data_id=None*, *data_dir=None*)

Class for reading EEA AQErep data

Extended class derived from low-level base class :class: ReadUngriddedBase that contains the main functionality.

DATA_ID = 'EEAAQeRep.NRT'

Name of the dataset (OBS_ID)

DATA_PRODUCT = 'E2a'

SUPPORTED_DATASETS = ['EEAAQeRep.NRT']

List of all datasets supported by this interface

EEA E1a product (QC)

Quality controlled EEA data. Interface for reading EEA AqERep files (formerly known as Airbase data).

class pyaerocom.io.read_eea_aqerep_v2.**ReadEEAAQEREP_V2**(*data_id=None*, *data_dir=None*)

Class for reading EEA AQErep data

Extended class derived from low-level base class :class: ReadUngriddedBase that contains the main functionality.

DATA_ID = 'EEAAQeRep.v2'

Name of the dataset (OBS_ID)

DATA_PRODUCT = 'E1a'

SUPPORTED_DATASETS = ['EEAAQeRep.v2']

List of all datasets supported by this interface

4.5.7 AirNow data

Reader for air quality measurements from North America.

class pyaerocom.io.read_airnow.**ReadAirNow**(*data_id=None*, *data_dir=None*)

Reading routine for North-American Air Now observations

BASEYEAR = 2000

DATA_ID = 'AirNow'

Name of dataset (OBS_ID)

DEFAULT_VARS = ['concbc', 'concpm10', 'concpm25', 'vmrco', 'vmrnh3', 'vmrno', 'vmrno2', 'vmrnox', 'vmrnoy', 'vmro3', 'vmrso2']

Default variables

FILE_COL_DELIM = '|'

Column delimiter

FILE_COL_NAMES = ['date', 'time', 'station_id', 'station_name', 'time_zone', 'variable', 'unit', 'value', 'institute']

Columns in data files

FILE_COL_ROW_NUMBER = 9

PROVIDES_VARIABLES = ['concbc', 'concpm10', 'concpm25', 'vmrco', 'vmrnh3', 'vmrno', 'vmrno2', 'vmrnox', 'vmrnoy', 'vmro3', 'vmrso2']

List of variables that are provided

REPLACE_STATNAME = {'&': 'and', "'": '', '.': ' ', '/': ' ', ':': ' '}

ROW_VAR_COL = 5

STATION_META_DTYPES = {'address': <class 'str'>, 'altitude': <class 'float'>, 'area_classification': <class 'str'>, 'city': <class 'str'>, 'comment': <class 'str'>, 'latitude': <class 'float'>, 'longitude': <class 'float'>, 'modificationdate': <class 'str'>, 'station_classification': <class 'str'>, 'station_id': <class 'str'>, 'station_name': <class 'str'>, 'timezone': <class 'str'>}

conversion functions for metadata dtypes

STATION_META_MAP = {'address': 'address', 'aqsid': 'station_id', 'city': 'city', 'comment': 'comment', 'elevation': 'altitude', 'environment': 'area_classification', 'lat': 'latitude', 'lon': 'longitude', 'modificationdate': 'modificationdate', 'name': 'station_name', 'populationclass': 'station_classification', 'timezone': 'timezone'}

Mapping of columns in station metadata file to pyaerocom standard

STAT_METADATA_FILENAME = 'allStations_20191224.csv'

file containing station metadata

SUPPORTED_DATASETS = ['AirNow']

List of all datasets supported by this interface

TS_TYPE = 'hourly'

Frequency of measurements

UNIT_MAP = {'C': 'celcius', 'M/S': 'm s-1', 'MILLIBAR': 'mbar', 'MM': 'mm', 'PERCENT': '%', 'PPB': 'ppb', 'PPM': 'ppm', 'UG/M3': 'ug m-3', 'WATTS/M2': 'W m-2'}

Units found in data files

VAR_MAP = {'concbc': 'BC', 'concpm10': 'PM10', 'concpm25': 'PM2.5', 'vmrco': 'CO', 'vmrnh3': 'NH3', 'vmrno': 'NO', 'vmrno2': 'NO2', 'vmrnox': 'NOX', 'vmrnoy': 'NOY', 'vmro3': 'OZONE', 'vmrso2': 'SO2'}

Variable names in data files

get_all_file_encodings(*filename*)

get_file_bom_encoding(*filename*)

get_file_encoding(*filename*)

get_file_list()

Retrieve list of data files

Return type
list

read(*vars_to_retrieve=None*, *first_file=None*, *last_file=None*)

Read variable data

Parameters

- **vars_to_retrieve** (*str or list, optional*) – List of variables to be retrieved. The default is None.
- **first_file** (*int, optional*) – Index of first file to be read. The default is None, in which case index 0 in file list is used.
- **last_file** (*int, optional*) – Index of last file to be read. The default is None, in which case last index in file list is used.

Returns
data – loaded data object.

Return type
UngriddedData

read_file(*filename*, *vars_to_retrieve=None*)

This method is returns just the raw content of a file as a dict

Parameters

- **filename** (*str*) – absolute path to filename to read
- **vars_to_retrieve** (list, optional) – list of str with variable names to read. If None, use *DEFAULT_VARS*
- **vars_as_series** (*bool*) – if True, the data columns of all variables in the result dictionary are converted into pandas Series objects

Returns
dict-like object containing results

Return type
StationData

Raises
NotImplementedError –

property station_metadata

Dictionary containing global metadata for each site

4.5.8 MarcoPolo data

Reader for air quality measurements for China from the EU-FP7 project MarcoPolo.

4.5.9 GHOST

GHOST (Globally Harmonised Observational Surface Treatment) project developed at the Earth Sciences Department of the Barcelona Supercomputing Center (see e.g., Petetin et al., 2020 for more information).

4.6 Further I/O features

Note: The *pyaerocom.io* package also includes all relevant data import and reading routines. These are introduced above, in Section *reading*.

4.6.1 AeroCom database browser

class pyaerocom.io.aerocom_browser.**AerocomBrowser**(**args*, ***kwargs*)

Interface for browsing all Aerocom data direcories

Note: Use `browse()` to find directories matching a certain search pattern. The class methods *find_matches()* and `find_dir()` both use `browse()`, the only difference is, that the *find_matches()* adds the search result (a list with strings) to

property dirs_found

All directories that were found

find_data_dir(*name_or_pattern*, *ignorecase=True*)

Find match of input name or pattern in Aerocom database

Parameters

- **name_or_pattern** (*str*) – name or pattern of data (can be model or obs data)
- **ignorecase** (*bool*) – if True, upper / lower case is ignored

Returns

data directory of match

Return type

str

Raises

DataSearchError – if no matches or no unique match can be found

find_matches(*name_or_pattern*, *ignorecase=True*)

Search all Aerocom data directories that match input name or pattern

Parameters

- **name_or_pattern** (*str*) – name or pattern of data (can be model or obs data)
- **ignorecase** (*bool*) – if True, upper / lower case is ignored

Returns

list of names that match the pattern (corresponding paths can be accessed from this class instance)

Return type

list

Raises

DataSearchError – if no matches can be found

property ids_found

All data IDs that were found

4.6.2 File naming conventions

class pyaerocom.io.fileconventions.**FileConventionRead**(*name='aerocom3'*, *file_sep='_'*, *year_pos=None*, *var_pos=None*, *ts_pos=None*, *vert_pos=None*, *data_id_pos=None*, *from_file=None*)

Class that represents a file naming convention for reading Aerocom files

name

name of this convention (e.g. "aerocom3")

Type

str

file_sep

filename delimiter for accessing different variables

Type

str

year_pos

position of year information in filename after splitting using delimiter *file_sep*

Type

int

var_pos

position of variable information in filename after splitting using delimiter *file_sep*

Type

int

ts_pos

position of information of temporal resolution in filename after splitting using delimiter *file_sep*

Type

int

vert_pos

position of information about vertical resolution of data

Type
int

data_id_pos

position of data ID

Type
int

AEROCOM3_VERT_INFO = {'2d': ['surface', 'column', 'modellevel', '2d'], '3d': ['modellevelatstations']}

check_validity(*file*)

Check if filename is valid

from_dict(*new_vals*)

Load info from dictionary

Parameters
new_vals (*dict*) – dictionary containing information

Return type
self

from_file(*file*)

Identify convention from a file

Currently only two conventions (aerocom2 and aerocom3) exist that are identified by the delimiter used.

Parameters
file (*str*) – file path or file name

Returns
this object (with updated convention)

Return type
FileConventionRead

Raises
FileConventionError – if convention cannot be identified

Example

```
>>> from pyaerocom.io import FileConventionRead
>>> filename = 'aerocom3_CAM5.3-Oslo_AP3-CTRL2016-PD_od550aer_Column_2010_
↪monthly.nc'
>>> print(FileConventionRead().from_file(filename))
pyaeorocom FileConventionRead
name: aerocom3
file_sep: _
year_pos: -2
var_pos: -4
ts_pos: -1
```

get_info_from_file(*file: str*) → dict

Identify convention from a file

Currently only two conventions (aerocom2 and aerocom3) exist that are identified by the delimiter used.

Parameters
file (*str*) – file path or file name

Returns
dictionary containing keys *year, var_name, ts_type* and corresponding variables, extracted from the filename

Return type
dict

Raises
FileConventionError – if convention cannot be identified

Example

```
>>> from pyaerocom.io import FileConventionRead
>>> filename = 'aerocom3_CAM5.3-Oslo_AP3-CTRL2016-PD_od550aer_Column_2010_
↪monthly.nc'
>>> conv = FileConventionRead("aerocom3")
>>> info = conv.get_info_from_file(filename)
>>> for item in info.items(): print(item)
('year', 2010)
('var_name', 'od550aer')
('ts_type', 'monthly')
```

import_default(*name: str*)

Checks and load default information from database

property info_init

Empty dictionary containing init values of infos to be extracted from filenames

string_mask(*data_id*, *var*, *year*, *ts_type*, *vert_which=None*)

Returns mask that can be used to identify files of this convention

Parameters

- **data_id** (*str*) – experiment ID (e.g. GISS-MATRIX.A2.CTRL)
- **var** (*str*) – variable string ID (e.g. "od550aer")
- **year** (*int*) – desired year of observation (e.g. 2012)
- **ts_type** (*str*) – string specifying temporal resolution (e.g. "daily")

Example

conf_aero2 = FileConventionRead(name="aerocom2") conf_aero3 = FileConvention-Read(name="aerocom2")

var = od550aer year = 2012 ts_type = "daily"

match_str_aero2 = conf_aero2.string_mask(var, year, ts_type)

match_str_aero3 = conf_aero3.string_mask(var, year, ts_type)

`to_dict()`

Convert this object to ordered dictionary

4.6.3 Iris helpers

Module containing helper functions related to iris I/O methods. These contain reading of Cubes, and some methods to perform quality checks of the data, e.g.

1. checking and correction of time definition
2. number and length of dimension coordinates must match data array
3. Longitude definition from -180 to 180 (corrected if defined on 0 -> 360 intervall)

`pyaerocom.io.iris_io.check_and_regrid_lons_cube(cube)`

Checks and corrects for if longitudes of `grid` are 0 -> 360

Note: This method checks if the maximum of the current longitudes array exceeds 180. Thus, it is not recommended to use this function after subsetting a cube, rather, it should be checked directly when the file is loaded (cf. `load_input()`)

Parameters
cube (*iris.cube.Cube*) – gridded data loaded as iris.Cube

Returns
True, if longitudes were on 0 -> 360 and have been rolled, else False

Return type
bool

`pyaerocom.io.iris_io.check_dim_coord_names_cube(cube)`

`pyaerocom.io.iris_io.check_dim_coords_cube(cube)`

Checks, and if necessary and applicable, updates coords names in Cube

Parameters
cube (*iris.cube.Cube*) – input cube

Returns
updated or unchanged cube

Return type
iris.cube.Cube

pyaerocom.io.iris_io.**check_time_coord**(*cube*, *ts_type*, *year*)

Method that checks the time coordinate of an iris Cube

This method checks if the time dimension of a cube is accessible and according to the standard (i.e. fully usable). It only checks, and does not correct. For the latter, please see *correct_time_coord()*.

Parameters

- **cube** (*Cube*) – cube containing data
- **ts_type** (*str*) – pyaerocom ts_type
- **year** – year of data

Returns
True, if time dimension is ok, False if not

Return type
bool

pyaerocom.io.iris_io.**concatenate_iris_cubes**(*cubes*, *error_on_mismatch=True*)

Concatenate list of `iris.Cube` instances cubes into single Cube

Helper method for concatenating list of cubes

This method is not supposed to be called directly but rather `concatenate_cubes()` (which ALWAYS returns instance of `Cube` or raises Exception) or `concatenate_possible_cubes()` (which ALWAYS returns instance of `CubeList` or raises Exception)

Parameters

- **cubes** (*CubeList or list(Cubes)*) – list of individual cubes
- **error_on_mismatch** – boolean specifying whether an Exception is supposed to be raised or not

Returns
result of concatenation

Return type
`Cube`

Raises
iris.exceptions.ConcatenateError – if `error_on_mismatch=True` and input cubes could not all concatenated into a single instance of `iris.Cube` class.

pyaerocom.io.iris_io.**correct_time_coord**(*cube*, *ts_type*, *year*)

Method that corrects the time coordinate of an iris Cube

Parameters

- **cube** (*Cube*) – cube containing data
- **ts_type** (*TsType or str*) – temporal resolution of data (e.g. "hourly", "daily"). This information is e.g. encoded in the filename of a NetCDF file and may be accessed using `pyaerocom.io.FileConventionRead`
- **year** (*int*) – integer specifying start year, e.g. 2017

Returns
the same instance of the input cube with corrected time dimension axis

Return type
Cube

pyaerocom.io.iris_io.**get_coord_names_cube**(*cube*)

pyaerocom.io.iris_io.**get_dim_names_cube**(*cube*)

pyaerocom.io.iris_io.**load_cube_custom**(*file*, *var_name=None*, *file_convention=None*, *perform_fmt_checks=None*)

Load netcdf file as iris.Cube

Parameters

- **file** (*str*) – netcdf file
- **var_name** (*str*) – name of variable to read
- **quality_check** (*bool*) – if True, then a quality check of data is performed against the information provided in the filename
- **file_convention** (FileConventionRead, optional) – Aerocom file convention. If provided, then the data content (e.g. dimension definitions) is tested against definition in file name
- **perform_fmt_checks** (*bool*) – if True, additional quality checks (and corrections) are (attempted to be) performed.

Returns

loaded data as Cube

Return type

iris.cube.Cube

pyaerocom.io.iris_io.**load_cubes_custom**(*files*, *var_name=None*, *file_convention=None*, *perform_fmt_checks=True*)

Load multiple NetCDF files into CubeList

Note: This function does not apply any concatenation or merging of the variable data in the individual files, it only loads the files into individual instances of `iris.cube.Cube`, which can be accessed via the returned list.

Parameters

- **files** (*list*) – list of netcdf file paths
- **var_name** (*str*) – name of variable to be imported from input files.
- **file_convention** (FileConventionRead, optional) – Aerocom file convention. If provided, then the data content (e.g. dimension definitions) is tested against definition in file name
- **perform_fmt_checks** (*bool*) – if True, additional quality checks (and corrections) are (attempted to be) performed.

Returns

- *list* – loaded cube instances.
- *list* – list containing all files from which the input variable could be successfully loaded.

pyaerocom.io.aux_read_cubes.**add_cubes**(*cube1*, *cube2*)

Method to add cubes from 2 gridded data objects

pyaerocom.io.aux_read_cubes.**apply_rh_thresh_cubes**(*cube*, *rh_cube*, *rh_max=None*)

Method that applies a low RH filter to input cube

pyaerocom.io.aux_read_cubes.**compute_angstrom_coeff_cubes**(*cube1*, *cube2*, *lambda1=None*, *lambda2=None*)

Compute Angstrom coefficient cube based on 2 optical densitiy cubes

Parameters

- **cube1** (*iris.cube.Cube*) – AOD at wavelength 1
- **cube2** (*iris.cube.Cube*) – AOD at wavelength 2
- **lambda1** (*float*) – wavelength 1
- **2** (*lambda*) – wavelength 2

Returns

Cube containing Angstrom exponent(s)

Return type

Cube

pyaerocom.io.aux_read_cubes.**conc_from_vmr**(*cube*, *ts*, *ps*)

pyaerocom.io.aux_read_cubes.**conc_from_vmr_STP**(*cube*)

pyaerocom.io.aux_read_cubes.**divide_cubes**(*cube1*, *cube2*)

Method to divide 2 cubes with each other

pyaerocom.io.aux_read_cubes.**lifetime_from_load_and_dep**(*load*, *wetdep*, *drydep*)

Compute lifetime from load and wet and dry deposition

pyaerocom.io.aux_read_cubes.**merge_meta_cubes**(*cube1*, *cube2*)

pyaerocom.io.aux_read_cubes.**mmr_from_vmr**(*cube*)

Convvert gas volume/mole mixing ratios into mass mixing ratios.

Parameters

cube (*iris.cube.Cube*) – A cube containing gas vmr data to be converted into mmr.

Returns

cube_out – Cube containing mmr data.

Return type

iris.cube.Cube

pyaerocom.io.aux_read_cubes.**mmr_to_vmr_cube**(*data*)

Convert cube containing MMR data to VMR

Parameters

data (*iris.Cube or GriddedData*) – input data object containing MMR data for a certain variable. Needs to have *var_name* attr. assigned and valid MMR AeroCom variable name (e.g. mmro3, mmrno2)

Raises

AttributeError – if attr. *var_name* of input data does not start with mmr

Returns

cube containing mixing ratios expressed as VMR in units of nmole mole-1

Return type

iris.Cube

pyaerocom.io.aux_read_cubes.**multiply_cubes**(*cube1*, *cube2*)

Method to multiply 2 cubes

pyaerocom.io.aux_read_cubes.**rho_from_ts_ps**(*ts*, *ps*)

pyaerocom.io.aux_read_cubes.**subtract_cubes**(*cube1*, *cube2*)

Method to subtract 1 cube from another

4.6.4 Handling of cached ungridded data objects

Caching class for reading and writing of ungridded data Cache objects

class pyaerocom.io.cachehandler_ungridded.**CacheHandlerUngridded**(*reader=None*, *cache_dir=None*, ***kwargs*)

Interface for reading and writing of cache files

Cache filename mask is

<data_id>_<var>.pkl

e.g. EBASMC_scatc550aer.pkl

reader

reading class for dataset

Type
ReadUngriddedBase

loaded_data

dictionary containing successfully loaded instances of single variable `UngriddedData` objects (keys are variable names)

Type
dict

CACHE_HEAD_KEYS = ['pyaerocom_version', 'newest_file_in_read_dir', 'newest_file_date_in_read_dir', 'data_revision', 'reader_version', 'ungridded_data_version', 'cacher_version']

Cache file header keys that are checked (and required unchanged) when reading a cache file

property cache_dir

Directory where cache data objects are stored

cache_meta_info()

Dictionary containing relevant caching meta-info

check_and_load(*var_or_file_name*, *force_use_outdated=False*, *cache_dir=None*)

Check if cache file exists and load

Note: If a cache file exists for this database, but cannot be loaded or is outdated against pyaerocom updates, then it will be removed (the latter only if `pyaerocom.const.RM_CACHE_OUTDATED` is True).

Parameters

- **var_or_file_name** (*str*) – name of output filename or variable that is supposed to be stored. Default usage is to provide variable and then *default_file_name()* is used. Can be None if input *data* contains only a single variable.ead

- **force_use_outdated** (*bool*) – if True, read existing cache file even if it is not up to date or pyaerocom version changed (not recommended to use)
- **cache_dir** (*str, optional*) – output directory (default is pyaerocom cache dir accessed via *cache_dir()*).

Returns
True, if cache file exists and could be successfully loaded, else False. Note: if import is successful, the corresponding data object (instance of `pyaerocom.UngriddedData` can be accessed via :attr:`loaded_data'

Return type
bool

Raises
TypeError – if cached file is not an instance of `pyaerocom.UngriddedData` class (which should not happen)

property data_id

Data ID of the associated dataset

default_file_name(*var_name*)

File name of cache file

Parameters
var_name (*str*) – name of variable to be cached.

Returns
file name of pickle file

Return type
str

file_path(*var_or_file_name*, *cache_dir=None*)

File path of cache file

Parameters

- **var_or_file_name** (*str*) – name of output filename or variable that is supposed to be stored. Default usage is to provide variable and then *default_file_name()* is used. Can be None if input *data* contains only a single variable.
- **cache_dir** (*str, optional*) – output directory (default is pyaerocom cache dir accessed via *cache_dir()*).

Returns
output file path

Return type
str

property reader

Instance of reader class

property src_data_dir

Data source directory of the associated dataset

Needed to check whether an existing cache file is outdated

write(*data*, *var_or_file_name=None*, *cache_dir=None*)

Write single-variable instance of UngriddedData to cache

Parameters

- **data** (UngriddedData) – object containing the data (possibly containing multiple variables)
- **var_or_file_name** (*str, optional*) – name of output filename or variable that is supposed to be stored. Default usage is to provide variable and then *default_file_name()* is used. Can be None if input *data* contains only a single variable.
- **cache_dir** (*str, optional*) – output directory (default is pyaerocom cache dir accessed via *cache_dir()*).

Returns

output file path

Return type

str

pyaerocom.io.cachehandler_ungridded.**list_cache_files**() → Iterator[Path]

List all pickled data objects in cache directory

If not set differently, the cache directory is the pyaerocom default, accessible via `pyaerocom.const.CACHEDIR`.

4.6.5 I/O utils

High level I/O utility methods for pyaerocom

pyaerocom.io.utils.**browse_database**(*model_or_obs*, *verbose=False*)

Browse Aerocom database using model or obs ID (or wildcard)

Searches database for matches and prints information about all matches found (e.g. available variables, years, etc.)

Parameters

- **model_or_obs** (*str*) – model or obs ID or search pattern
- **verbose** (*bool*) – if True, verbosity level will be set to debug, else to critical

Returns

list with data_ids of all matches

Return type

list

Example

```
>>> import pyaerocom as pya
>>> pya.io.browse_database('AATSR*ORAC*v4*')
Pyaerocom ReadGridded
---------------------
Model ID: AATSR_ORAC_v4.01
Data directory: /lustre/storeA/project/aerocom/aerocom-users-database/CCI-Aerosol/
↪CCI_AEROSOL_Phase2/AATSR_ORAC_v4.01/renamed
Available variables: ['abs550aer', 'ang4487aer', 'clt', 'landseamask', 'od550aer',
```

(continues on next page)

(continued from previous page)

```
↪'od550dust', 'od550gt1aer', 'od550lt1aer', 'pixelcount']
Available years: [2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012]
Available time resolutions ['daily']
```

pyaerocom.io.utils.**get_ungridded_reader**(*obs_id*)

4.6.6 I/O helpers

I/O helper methods of the pyaerocom package

pyaerocom.io.helpers.**COUNTRY_CODE_FILE = 'country_codes.json'**

country code file name will be prepended with the path later on

pyaerocom.io.helpers.**add_file_to_log**(*filepath*, *err_msg*)

Add input file path to error logdir

The logdir location can be accessed via `pyaerocom.const.LOGFILESDIR`

Parameters

- **filepath** (*str or Path*) – path of file that has an error
- **err_msg** (*str*) – Problem associated with input file

pyaerocom.io.helpers.**aerocom_savename**(*data_id*, *var_name*, *vert_code*, *year*, *ts_type*)

Generate filename in AeroCom conventions

ToDo: complete docstring

pyaerocom.io.helpers.**get_all_supported_ids_ungridded**()

Get list of datasets that are supported by `ReadUngridded`

Returns

list with supported network names

Return type

list

pyaerocom.io.helpers.**get_country_name_from_iso**(*iso_code: str | None = None*, *filename: str | Path | None = None*, *return_as_dict: bool = False*)

get the country name from the 2 digit iso country code

the underlaying json file was taken from this github repository https://github.com/lukes/ISO-3166-Countries-with-Regional-Codes

Parameters

- **iso_code** (*str*) – string containing the 2 character iso code of the country (e.g. no for Norway)
- **filename** (*str* , optional) – optional string with the json file to read
- **return_as_dict** (*bool*, optional) – flag to get the entire list of countries as a dictionary with the country codes as keys and the country names as value Useful if you have to get the names for a lot of country codes

Returns

- *string with country name or dictionary with iso codes as keys and the country names as values*

- *empty string if the country code was not found*

Raises
ValueError – if the country code ins invalid

pyaerocom.io.helpers.**get_metadata_from_filename**(*filename*)

Try access metadata information from filename

pyaerocom.io.helpers.**get_obsnetwork_dir**(*obs_id*)

Returns data path for obsnetwork ID

Parameters
obs_id (*str*) – ID of obsnetwork (e.g. AeronetSunV2Lev2.daily)

Returns
corresponding directory from `pyaerocom.const`

Return type
str

Raises

- **ValueError** – if obs_id is invalid
- **IOError** – if directory does not exist

pyaerocom.io.helpers.**get_standard_name**(*var_name*)

Get standard name of aerocom variable

Parameters
var_name (*str*) – HTAP2 variable name

Returns
corresponding standard name

Return type
str

Raises

- ***VarNotAvailableError*** – if input variable is not defined in *variables.ini* file
- ***VariableDefinitionError*** – if standarad name is not set for variable in *variables.ini* file

pyaerocom.io.helpers.**read_ebas_flags_file**(*ebas_flags_csv*)

Reads file ebas_flags.csv

Parameters
ebas_flags_csv (*str*) – file containing flag info

Returns
dict with loaded flag info

Return type
dict

pyaerocom.io.helpers.**search_data_dir_aerocom**(*name_or_pattern*, *ignorecase=True*)

Search Aerocom data directory based on model / data ID

4.7 Metadata and vocabulary standards

class pyaerocom.metastandards.**AerocomDataID**(*data_id=None*, ***meta_info*)

Class representing a model data ID following AeroCom PhaseIII conventions

The ID must contain 4 substrings with meta parameters:

<ModelName>-<MeteoConfigSpecifier>_<ExperimentName>-<PerturbationName>

E.g.

NorESM2-met2010_CTRL-AP3

For more information see AeroCom diagnostics spreadsheet

This interface can be used to make sure a provided data ID is following this convention and to extract the corresponding meta parameters as dictionary (*to_dict()*) or to create an data_id from the corresponding meta parameters *from_dict()*.

DELIM = '_'

KEYS = ['model_name', 'meteo', 'experiment', 'perturbation']

SUBDELIM = '-'

property data_id

str AeroCom data ID

static from_dict(*meta*)

Create instance of AerocomDataID from input meta dictionary

Parameters
meta (*dict*) – dictionary containing required keys (cf. *KEYS*) and corresponding values to create an data_id

Raises
KeyError – if not all information required is provided

Return type
AerocomDataID

static from_values(*values*)

Create data_id from list of values

Note: The values have to be in the right order, cf. *KEYS*

Parameters
values (*list*) – list containing values for each key in *KEYS*

Raises
ValueError – if length of input list mismatches length of *KEYS*

Returns
generated data_id

Return type
str

to_dict()

Convert data_id to dictionary

Returns
dictionary with metadata information

Return type
dict

property values

class pyaerocom.metastandards.**DataSource**(***info*)

Dict-like object defining a data source

data_id

name (or ID) of dataset (e.g. AeronetSunV3Lev2.daily)

dataset_name

name of dataset (e.g. AERONET)

data_product

data product (e.g. SDA, Inv, Sun for Aeronet)

data_version

version of data (e.g. 3)

data_level

level of data (e.g. 2)

framework

ID of framework to which data is associated (e.g. ACTRIS, GAW)

Type
str

instr_vert_loc

Vertical location of measuring instrument(s).

Type
str

revision_date

last revision date of dataset

ts_type_src

sampling frequency as defined in data files (use None if undefined)

stat_merge_pref_attr

optional, a metadata attribute that is available in data and that is used to order the individual stations by relevance in case overlaps occur. The associated values of this attribute need to be sortable (e.g. revision_date). This is only relevant in case overlaps occur.

Type
str

SUPPORTED_VERT_LOCS = ['ground', 'space', 'airborne']

property data_dir

Directory containing data files

dataset_str()

load_dataset_info()

Wrapper for _parse_source_info_from_ini()

class pyaerocom.metastandards.**StationMetaData**(**info*)

This object defines a standard for station metadata in pyaerocom

Variable names associated with meta data can vary significantly between different conventions (e.g. conventions in modellers community vs. observations community).

Note:

- This object is a dictionary and can be easily expanded
- In many cases, only some of the attributes are relevant

---

filename

name of file (may be full path or only filename)

Type

str

station_id

Code or unique ID of station

Type

str

station_name

name or ID of a station. Note, that the concept of a station in pyaerocom is not necessarily related to a fixed coordinate. A station can also be a satellite, ship, or a human walking around and measuring something

Type

str

instrument_name

name (or ID) of instrument

Type

str

PI

principal investigator

Type

str

country

string specifying country (or country ID)

Type

str

ts_type

frequency of data (e.g. monthly). Note the difference between ts_type_src of *DataSource*, which specifies the freq. of the original files.

Type

str

latitude

latitude coordinate

Type
float

longitude

longitude coordinate

Type
float

altitude

altitude coordinate

Type
float

4.8 Variables

4.8.1 Variable collection

class pyaerocom.variable.**Variable**(*var_name=None*, *init=True*, *cfg=None*, ***kwargs*)

Interface that specifies default settings for a variable

See variables.ini file for an overview of currently available default variables.

Parameters

- **var_name** (*str*) – string ID of variable (see file variables.ini for valid IDs)
- **init** (*bool*) – if True, input variable name is attempted to be read from config file
- **cfg** (*ConfigParser*) – open config parser that holds the information in config file available (i.e. `ConfigParser.read()` has been called with config file as input)
- ****kwargs** – any valid class attribute (e.g. map_vmin, map_vmax, ...)

var_name

input variable name

Type
str

var_name_aerocom

AEROCOM variable name (see e.g. AEROCOM protocol for a list of available variables)

Type
str

is_3d

flag that indicates if variable is 3D

Type
bool

is_dry

flag that is set based on filename that indicates if variable data corresponds to dry conditions.

Type
bool

units

unit of variable (None if no unit)

Type
str

default_vert_code

default vertical code to be loaded (i.e. Column, ModelLevel, Surface). Only relevant during reading and in case conflicts occur (e.g. abs550aer, 2010, Column and Surface files)

Type
str, optional

aliases

list of alternative names for this variable

Type
list

minimum

lower limit of allowed value range

Type
float

upper_limit

upper limit of allowed value range

Type
float

obs_wavelength_tol_nm

wavelength tolerance (+/-) for reading of obsdata. Default is 10, i.e. if this variable is defined at 550 nm and obsdata contains measured values of this quantity within interval of 540 - 560, then these data is used

Type
float

scat_xlim

x-range for scatter plot

Type
float

scat_ylim

y-range for scatter plot

Type
float

scat_loglog

scatter plot on loglog scale

Type
bool

scat_scale_factor
scale factor for scatter plot

Type
float

map_cmap
name of default colormap (matplotlib) of this variable.

Type
str

map_vmin
data value corresponding to lower end of colormap in map plots of this quantity

Type
float

map_vmax
data value corresponding to upper end of colormap in map plots of this quantity

Type
float

map_c_under
color used for values below *map_vmin* in map plots of this quantity

Type
str

map_c_over
color used for values exceeding *map_vmax* in map plots of this quantity

Type
str

map_cbar_levels
levels of colorbar

Type
list, optional

map_cbar_ticks
colorbar ticks

Type
list, optional

ALT_NAMES = {'unit': 'units'}

VMAX_DEFAULT = inf

VMIN_DEFAULT = -inf

property aliases
Alias variable names that are frequently found or used

Returns
list containing valid aliases

Return type
list

get_cmap()

Get cmap str for var

Return type
str

get_cmap_bins(*infer_if_missing=True*)

Get cmap discretisation bins

Parameters
infer_if_missing (*bool*) – if True and *map_cbar_levels* is not defined, try to infer using _cmap_bins_from_vmin_vmax().

Raises
AttributeError – if unavailable

Returns
levels

Return type
list

get_default_vert_code()

Get default vertical code for variable name

property has_unit

Boolean specifying whether variable has unit

property is_3d

True if str '3d' is contained in *var_name_input*

property is_alias

property is_at_dry_conditions

Indicate whether variable denotes dry conditions

property is_deposition

Indicates whether input variables is a deposition rate

Note: This funtion only identifies wet and dry deposition based on the variable names, there might be other variables that are deposition variables but cannot be identified by this function.

Parameters
var_name (*str*) – Name of variable to be checked

Returns
If True, then variable name denotes a deposition variables

Return type
bool

property is_emission

Indicates whether input variables is an emission rate

Note: This funtion only identifies wet and dry deposition based on the variable names, there might be other variables that are deposition variables but cannot be identified by this function.

Parameters
var_name (*str*) – Name of variable to be checked

Returns
If True, then variable name denotes a deposition variables

Return type
bool

property is_rate

Indicates whether variable name is a rate

Rates include e.g. deposition or emission rate variables but also precipitation

Returns
True if variable is rate, else False

Return type
bool

property is_wavelength_dependent

Indicates whether this variable is wavelength dependent

keys()

literal_eval_list()

property long_name

Wrapper for `description`

property lower_limit

Old attribute name for *minimum* (following HTAP2 defs)

parse_from_ini(*var_name=None*, *cfg=None*)

Import information about default region

Parameters

- **var_name** (*str*) – variable name
- **var_name_alt** (*str*) – alternative variable name that is used if variable name is not available
- **cfg** (*ConfigParser*) – open config parser object

Returns
True, if default could be loaded, False if not

Return type
bool

property plot_info

Dictionary containing plot information

plot_info_keys = ['scat_xlim', 'scat_ylim', 'scat_loglog', 'scat_scale_factor', 'map_vmin', 'map_vmax', 'map_cmap', 'map_c_under', 'map_c_over', 'map_cbar_levels', 'map_cbar_ticks']

static read_config()

str2bool()

str2list()

property unit

Unit of variable (old name, deprecated)

property unit_str

string representation of unit

update(***kwargs*)

property upper_limit

Old attribute name for `maximum` (following HTAP2 defs)

property var_name_aerocom

AeroCom variable name of the input variable

property var_name_info

property var_name_input

Input variable

4.8.2 Variable class

class pyaerocom.variable.**Variable**(*var_name=None*, *init=True*, *cfg=None*, ***kwargs*)

Interface that specifies default settings for a variable

See variables.ini file for an overview of currently available default variables.

Parameters

- **var_name** (*str*) – string ID of variable (see file variables.ini for valid IDs)
- **init** (*bool*) – if True, input variable name is attempted to be read from config file
- **cfg** (*ConfigParser*) – open config parser that holds the information in config file available (i.e. `ConfigParser.read()` has been called with config file as input)
- ****kwargs** – any valid class attribute (e.g. map_vmin, map_vmax, ...)

var_name

input variable name

Type
str

var_name_aerocom

AEROCOM variable name (see e.g. AEROCOM protocol for a list of available variables)

Type
str

is_3d

flag that indicates if variable is 3D

Type
bool

is_dry

flag that is set based on filename that indicates if variable data corresponds to dry conditions.

Type
bool

units

unit of variable (None if no unit)

Type
str

default_vert_code

default vertical code to be loaded (i.e. Column, ModelLevel, Surface). Only relevant during reading and in case conflicts occur (e.g. abs550aer, 2010, Column and Surface files)

Type
str, optional

aliases

list of alternative names for this variable

Type
list

minimum

lower limit of allowed value range

Type
float

upper_limit

upper limit of allowed value range

Type
float

obs_wavelength_tol_nm

wavelength tolerance (+/-) for reading of obsdata. Default is 10, i.e. if this variable is defined at 550 nm and obsdata contains measured values of this quantity within interval of 540 - 560, then these data is used

Type
float

scat_xlim

x-range for scatter plot

Type
float

scat_ylim

y-range for scatter plot

Type
float

scat_loglog

scatter plot on loglog scale

Type
bool

scat_scale_factor

scale factor for scatter plot

Type
float

map_cmap

name of default colormap (matplotlib) of this variable.

Type
str

map_vmin

data value corresponding to lower end of colormap in map plots of this quantity

Type
float

map_vmax

data value corresponding to upper end of colormap in map plots of this quantity

Type
float

map_c_under

color used for values below *map_vmin* in map plots of this quantity

Type
str

map_c_over

color used for values exceeding *map_vmax* in map plots of this quantity

Type
str

map_cbar_levels

levels of colorbar

Type
list, optional

map_cbar_ticks

colorbar ticks

Type
list, optional

ALT_NAMES = {'unit': 'units'}

VMAX_DEFAULT = inf

VMIN_DEFAULT = -inf

property aliases

Alias variable names that are frequently found or used

Returns
list containing valid aliases

Return type
list

`get_cmap()`

Get cmap str for var

Return type
str

`get_cmap_bins(infer_if_missing=True)`

Get cmap discretisation bins

Parameters
infer_if_missing (*bool*) – if True and *map_cbar_levels* is not defined, try to infer using `_cmap_bins_from_vmin_vmax()`.

Raises
AttributeError – if unavailable

Returns
levels

Return type
list

`get_default_vert_code()`

Get default vertical code for variable name

`property has_unit`

Boolean specifying whether variable has unit

`property is_3d`

True if str '3d' is contained in *var_name_input*

`property is_alias`

`property is_at_dry_conditions`

Indicate whether variable denotes dry conditions

`property is_deposition`

Indicates whether input variables is a deposition rate

Note: This funtion only identifies wet and dry deposition based on the variable names, there might be other variables that are deposition variables but cannot be identified by this function.

Parameters
var_name (*str*) – Name of variable to be checked

Returns
If True, then variable name denotes a deposition variables

Return type
bool

`property is_emission`

Indicates whether input variables is an emission rate

Note: This funtion only identifies wet and dry deposition based on the variable names, there might be other variables that are deposition variables but cannot be identified by this function.

Parameters
var_name (*str*) – Name of variable to be checked

Returns
If True, then variable name denotes a deposition variables

Return type
bool

property is_rate

Indicates whether variable name is a rate

Rates include e.g. deposition or emission rate variables but also precipitation

Returns
True if variable is rate, else False

Return type
bool

property is_wavelength_dependent

Indicates whether this variable is wavelength dependent

keys()

literal_eval_list()

property long_name

Wrapper for `description`

property lower_limit

Old attribute name for *minimum* (following HTAP2 defs)

parse_from_ini(*var_name=None*, *cfg=None*)

Import information about default region

Parameters

- **var_name** (*str*) – variable name
- **var_name_alt** (*str*) – alternative variable name that is used if variable name is not available
- **cfg** (*ConfigParser*) – open config parser object

Returns
True, if default could be loaded, False if not

Return type
bool

property plot_info

Dictionary containing plot information

plot_info_keys = ['scat_xlim', 'scat_ylim', 'scat_loglog', 'scat_scale_factor', 'map_vmin', 'map_vmax', 'map_cmap', 'map_c_under', 'map_c_over', 'map_cbar_levels', 'map_cbar_ticks']

static read_config()

str2bool()

str2list()

property unit

Unit of variable (old name, deprecated)

property unit_str

string representation of unit

update(***kwargs*)

property upper_limit

Old attribute name for `maximum` (following HTAP2 defs)

property var_name_aerocom

AeroCom variable name of the input variable

property var_name_info

property var_name_input

Input variable

4.8.3 Variable helpers

pyaerocom.variable_helpers.**get_aliases**(*var_name: str*, *parser: ConfigParser | None = None*)

Get aliases for a certain variable

pyaerocom.variable_helpers.**get_variable**(*var_name: str*)

Get a certain variable

Parameters
var_name (*str*) – variable name

Return type
Variable

pyaerocom.variable_helpers.**parse_aliases_ini**()

Returns instance of ConfigParser to access information

pyaerocom.variable_helpers.**parse_variables_ini**(*fpath: str | Path | None = None*)

Returns instance of ConfigParser to access information

4.8.4 Variable name info

class pyaerocom.varnameinfo.**VarNameInfo**(*var_name*)

This class can be used to retrieve information from variable names

DEFAULT_VERT_CODE_PATTERNS = {'abs*': 'Column', 'ang*': 'Column', 'dry*': 'Surface', 'emi*': 'Surface', 'load*': 'Column', 'od*': 'Column', 'wet*': 'Surface'}

PATTERNS = {'od': 'od\\d+aer'}

property contains_numbers

Boolean specifying whether this variable name contains numbers

property contains_wavelength_nm

Boolean specifying whether this variable contains a certain wavelength

get_default_vert_code()

Get default vertical code for variable name

in_wavelength_range(*low*, *high*)

Boolean specifying whether variable is within wavelength range

Parameters

- **low** (*float*) – lower end of wavelength range to be tested
- **high** (*float*) – upper end of wavelength range to be tested

Returns

True, if this variable is wavelength dependent and if the wavelength that is inferred from the filename is within the specified input range

Return type

bool

property is_wavelength_dependent

Boolean specifying whether this variable name is wavelength dependent

translate_to_wavelength(*to_wavelength*)

Create new variable name at a different wavelength

Parameters

to_wavelength (*float*) – new wavelength in nm

Returns

new variable name

Return type

VarNameInfo

property wavelength_nm

Wavelength in nm (if appliable)

4.9 Helpers for auxiliary variables

pyaerocom.aux_var_helpers.**calc_abs550aer**(*data*)

Compute AOD at 550 nm using Angstrom coefficient and 500 nm AOD

Parameters

data (*dict-like*) – data object containing imported results

Returns

AOD(s) at shifted wavelength

Return type

float or ndarray

pyaerocom.aux_var_helpers.**calc_ang4487aer**(*data*)

Compute Angstrom coefficient (440-870nm) from 440 and 870 nm AODs

Parameters

data (*dict-like*) – data object containing imported results

Note: Requires the following two variables to be available in provided data object:

1. od440aer
2. od870aer

Raises
AttributError – if either 'od440aer' or 'od870aer' are not available in data object

Returns
array containing computed angstrom coefficients

Return type
ndarray

pyaerocom.aux_var_helpers.**calc_od550aer**(*data*)

Compute AOD at 550 nm using Angstrom coefficient and 500 nm AOD

Parameters
data (*dict-like*) – data object containing imported results

Returns
AOD(s) at shifted wavelength

Return type
float or ndarray

pyaerocom.aux_var_helpers.**calc_od550gt1aer**(*data*)

Compute coarse mode AOD at 550 nm using Angstrom coeff. and 500 nm AOD

Parameters
data (*dict-like*) – data object containing imported results

Returns
AOD(s) at shifted wavelength

Return type
float or ndarray

pyaerocom.aux_var_helpers.**calc_od550lt1aer**(*data*)

Compute fine mode AOD at 550 nm using Angstrom coeff. and 500 nm AOD

Parameters
data (*dict-like*) – data object containing imported results

Returns
AOD(s) at shifted wavelength

Return type
float or ndarray

pyaerocom.aux_var_helpers.**calc_od550lt1ang**(*data*)

Compute AOD at 550 nm using Angstrom coeff. and 500 nm AOD,
that is filtered for angstrom coeff < 1 to get AOD representative of coarse particles.

Parameters
data (*dict-like*) – data object containing imported results

Returns
AOD(s) at shifted wavelength

Return type
float or ndarray

pyaerocom.aux_var_helpers.**calc_vmro3max**(*data*)

pyaerocom.aux_var_helpers.**compute_ac550dryaer**(*data*)

Compute aerosol dry absorption coefficent applying RH threshold

Cf. `_compute_dry_helper()`

Parameters
dict – data object containing scattering and RH data

Returns
modified data object containing new column sc550dryaer

Return type
dict

pyaerocom.aux_var_helpers.**compute_ang4470dryaer_from_dry_scat**(*data*)

Compute angstrom exponent between 440 and 700 nm

Parameters
dict (*StationData or*) – data containing dry scattering coefficients at 440 and 700 nm (i.e. keys sc440dryaer and sc700dryaer)

Returns
extended data object containing angstrom exponent

Return type
StationData or dict

pyaerocom.aux_var_helpers.**compute_angstrom_coeff**(*od1*, *od2*, *lambda1*, *lambda2*)

Compute Angstrom coefficient based on 2 optical densities

Parameters

- **od1** (float or ndarray) – AOD at wavelength 1
- **od2** (float or ndarray) – AOD at wavelength 2
- **lambda1** (float or ndarray) – wavelength 1
- **2** (*lambda*) – wavelength 2

Returns
Angstrom exponent(s)

Return type
float or ndarray

pyaerocom.aux_var_helpers.**compute_od_from_angstromexp**(*to_lambda*, *od_ref*, *lambda_ref*, *angstrom_coeff*)

Compute AOD at specified wavelength

Uses Angstrom coefficient and reference AOD to compute the corresponding wavelength shifted AOD

Parameters

- **to_lambda** (float or ndarray) – wavelength for which AOD is calculated

- **od_ref** (float or ndarray) – reference AOD
- **lambda_ref** (float or ndarray) – wavelength corresponding to reference AOD
- **angstrom_coeff** (float or ndarray) – Angstrom coefficient

Returns
AOD(s) at shifted wavelength

Return type
float or ndarray

pyaerocom.aux_var_helpers.**compute_sc440dryaer**(*data*)

Compute dry scattering coefficent applying RH threshold

Cf. _compute_dry_helper()

Parameters
dict – data object containing scattering and RH data

Returns
modified data object containing new column sc550dryaer

Return type
dict

pyaerocom.aux_var_helpers.**compute_sc550dryaer**(*data*)

Compute dry scattering coefficent applying RH threshold

Cf. _compute_dry_helper()

Parameters
dict – data object containing scattering and RH data

Returns
modified data object containing new column sc550dryaer

Return type
dict

pyaerocom.aux_var_helpers.**compute_sc700dryaer**(*data*)

Compute dry scattering coefficent applying RH threshold

Cf. _compute_dry_helper()

Parameters
dict – data object containing scattering and RH data

Returns
modified data object containing new column sc550dryaer

Return type
dict

pyaerocom.aux_var_helpers.**compute_wetna_from_concprcpna**(*data*)

pyaerocom.aux_var_helpers.**compute_wetnh4_from_concprcpnh4**(*data*)

pyaerocom.aux_var_helpers.**compute_wetno3_from_concprcpno3**(*data*)

pyaerocom.aux_var_helpers.**compute_wetoxn_from_concprcpoxn**(*data*)

Compute wdep from conc in precip and precip data

Note: In addition to the returned numpy array, the input instance of `StationData` is modified by additional metadata and flags for the new variable. See also `_compute_wdep_from_concprcp_helper()`.

Parameters
StationData – data object containing concprcp and precip data

Returns
array with wet deposition values

Return type
numpy.ndarray

pyaerocom.aux_var_helpers.**compute_wetoxs_from_concprcpoxs**(*data*)

Compute wdep from conc in precip and precip data

Note: In addition to the returned numpy array, the input instance of `StationData` is modified by additional metadata and flags for the new variable. See also `_compute_wdep_from_concprcp_helper()`.

Parameters
StationData – data object containing concprcp and precip data

Returns
array with wet deposition values

Return type
numpy.ndarray

pyaerocom.aux_var_helpers.**compute_wetoxs_from_concprcpoxsc**(*data*)

Compute wdep from conc in precip and precip data

Note: In addition to the returned numpy array, the input instance of `StationData` is modified by additional metadata and flags for the new variable. See also `_compute_wdep_from_concprcp_helper()`.

Parameters
StationData – data object containing concprcp and precip data

Returns
array with wet deposition values

Return type
numpy.ndarray

pyaerocom.aux_var_helpers.**compute_wetoxs_from_concprcpoxst**(*data*)

Compute wdep from conc in precip and precip data

Note: In addition to the returned numpy array, the input instance of `StationData` is modified by additional metadata and flags for the new variable. See also `_compute_wdep_from_concprcp_helper()`.

Parameters
StationData – data object containing concprcp and precip data

Returns
array with wet deposition values

Return type
numpy.ndarray

pyaerocom.aux_var_helpers.**compute_wetrdn_from_concprcprdn**(*data*)

Compute wdep from conc in precip and precip data

Note: In addition to the returned numpy array, the input instance of `StationData` is modified by additional metadata and flags for the new variable. See also `_compute_wdep_from_concprcp_helper()`.

Parameters
StationData – data object containing concprcp and precip data

Returns
array with wet deposition values

Return type
numpy.ndarray

pyaerocom.aux_var_helpers.**compute_wetso4_from_concprcpso4**(*data*)

pyaerocom.aux_var_helpers.**concx_to_vmrx**(*data*, *p_pascal*, *T_kelvin*, *conc_unit*, *mmol_var*, *mmol_air=None*, *to_unit=None*)

Convert mass concentration to volume mixing ratio (vmr)

Parameters

- **data** (*float or ndarray*) – array containing vmr values
- **p_pascal** (*float*) – pressure in Pa of input data
- **T_kelvin** (*float*) – temperature in K of input data
- **vmr_unit** (*str*) – unit of input data
- **mmol_var** (*float*) – molar mass of variable represented by input data
- **mmol_air** (*float, optional*) – Molar mass of air. Uses average density of dry air if None. The default is None.
- **to_unit** (*str, optional*) – Unit to which output data is converted. If None, output unit is kg m-3. The default is None.

Returns
input data converted to volume mixing ratio

Return type
float or ndarray

pyaerocom.aux_var_helpers.**identity**(*data*)

pyaerocom.aux_var_helpers.**make_proxy_drydep_from_O3**(*data*)

pyaerocom.aux_var_helpers.**make_proxy_wetdep_from_O3**(*data*)

pyaerocom.aux_var_helpers.**vmrx_to_concx**(*data*, *p_pascal*, *T_kelvin*, *vmr_unit*, *mmol_var*, *mmol_air=None*, *to_unit=None*)

Convert volume mixing ratio (vmr) to mass concentration

Parameters

- **data** (*float or ndarray*) – array containing vmr values
- **p_pascal** (*float*) – pressure in Pa of input data
- **T_kelvin** (*float*) – temperature in K of input data
- **vmr_unit** (*str*) – unit of input data
- **mmol_var** (*float*) – molar mass of variable represented by input data
- **mmol_air** (*float, optional*) – Molar mass of air. Uses average density of dry air if None. The default is None.
- **to_unit** (*str, optional*) – Unit to which output data is converted. If None, output unit is kg m-3. The default is None.

Returns

input data converted to mass concentration

Return type

float or ndarray

4.10 Variable categorisations

Variable categorisation groups

These are needed in some cases to infer, e.g. units associated with variable names. Used in *pyaerocom.variable.Variable* to identify certain groups.

Note: The below definitions are far from complete

pyaerocom.var_groups.**dep_add_vars = []**

additional deposition rate variables (that do not start with wet* or dry*)

pyaerocom.var_groups.**drydep_startswith = 'dry'**

start string of dry deposition variables

pyaerocom.var_groups.**emi_add_vars = []**

additional emission rate variables (that do not start with emi*)

pyaerocom.var_groups.**emi_startswith = 'emi'**

start string of emission variables

pyaerocom.var_groups.**totdep_startswith = 'dep'**

start string of total deposition variables

pyaerocom.var_groups.**wetdep_startswith = 'wet'**

start string of wet deposition variables

4.11 Regions and data filtering

4.11.1 Region class and helper functions

This module contains functionality related to regions in pyaerocom

class pyaerocom.region.**Region**(*region_id=None*, ***kwargs*)

Class specifying a region

region_id

ID of region (e.g. EUROPE)

Type
str

name

name of region (e.g. Europe) used e.g. in plotting.

Type
str

lon_range

longitude range (min, max) covered by region

Type
list

lat_range

latitude range (min, max) covered by region

Type
list

lon_range_plot

longitude range (min, max) used for plotting region.

Type
list

lat_range_plot

latitude range (min, max) used for plotting region.

Type
list

lon_ticks

list of longitude ticks used for plotting

Type
list

lat_ticks

list of latitude ticks used for plotting

Type
list

Parameters

- **region_id** (*str*) – ID of region (e.g. "EUROPE"). If the input region ID is registered as a default region in *pyaerocom.region_defs*, then the default information is automatically imported on class instantiation.
- ****kwargs** – additional class attributes (see above for available default attributes). Note, any attr. values provided by kwargs are preferred over potentially defined default attrs. that are imported automatically.

property center_coordinate

Center coordinate of this region

contains_coordinate(*lat*, *lon*)

Check if input lat/lon coordinate is contained in region

Parameters

- **lat** (*float*) – latitude of coordinate
- **lon** (*float*) – longitude of coordinate

Returns

True if coordinate is contained in this region, False if not

Return type

bool

distance_to_center(*lat*, *lon*)

Compute distance of input coordinate to center of this region

Parameters

- **lat** (*float*) – latitude of coordinate
- **lon** (*float*) – longitude of coordinate

Returns

distance in km

Return type

float

get_mask_data()

import_default(*region_id*)

Import region definition

Parameters

region_id (*str*) – ID of region

Raises

KeyError – if no region is registered for the input ID

is_htap()

Boolean specifying whether region is an HTAP binary region

mask_available()

plot(*ax=None*)

Plot this region

Draws a rectangle of the outer bounds of the region and if a binary mask is available for this region, it will be plotted as well.

Parameters
ax (*GeoAxes, optional*) – axes instance to be used for plotting. Defaults to None in which case a new instance is created.

Returns
axes instance used for plotting

Return type
GeoAxes

plot_borders(*ax*, *color*, *lw=2*)

plot_mask(*ax*, *color*, *alpha=0.2*)

pyaerocom.region.**all**()

Wrapper for *get_all_default_region_ids()*

pyaerocom.region.**find_closest_region_coord**(*lat: float*, *lon: float*, *regions: dict | None = None*, ***kwargs*) → list[str]

Finds list of regions sorted by their center closest to input coordinate

Parameters

- **lat** (*float*) – latitude of coordinate
- **lon** (*float*) – longitude of coordinate
- **regions** (*dict, optional*) – dictionary containing instances of *Region* as values, which are considered. If None, then all default regions are used.

Returns
sorted list of region IDs of identified regions

Return type
list[str]

pyaerocom.region.**get_all_default_region_ids**()

Get list containing IDs of all default regions

Returns
IDs of all predefined default regions

Return type
list

pyaerocom.region.**get_all_default_regions**()

Get dictionary containing all default regions from region.ini file

Returns
dictionary containing all default regions; keys are region ID's, values are instances of *Region*.

Return type
dict

pyaerocom.region.**get_htap_regions**()

Load dictionary with HTAP regions

Returns
keys are region ID's, values are instances of *Region*

Return type
dict

pyaerocom.region.**get_old_aerocom_default_regions**()

Load dictionary with default AeroCom regions

Returns
keys are region ID's, values are instances of *Region*

Return type
dict

pyaerocom.region.**get_regions_coord**(*lat*, *lon*, *regions=None*)

Get the region that contains an input coordinate

Note: This does not yet include HTAP, since this causes troules in automated AeroCom processing

Parameters

- **lat** (*float*) – latitude of coordinate
- **lon** (*float*) – longitude of coordinate
- **regions** (*dict, optional*) – dictionary containing instances of *Region* as values, which are considered. If None, then all default regions are used.

Returns
list of regions that contain this coordinate

Return type
list

4.11.2 Region definitions

Definitions of rectangular regions used in pyaerocom

NOTE: replaces former regions.ini in pyaerocom/data dir

pyaerocom.region_defs.**ALL_REGION_NAME: Final = 'ALL'**

Name of region containing absolute all valid data points (WORLD in old aerocom notation)

4.11.3 Region filter

class pyaerocom.filter.**Filter**(*name=None*, *region=None*, *altitude_filter=None*, *land_ocn=None*, ***kwargs*)

Class that can be used to filter gridded and ungridded data objects

Note:

- BETA version (currently being tested)
- Can only filter spatially
- Might be renamed to RegionFilter at some point in the future

ALTITUDE_FILTERS = {'noMOUNTAINS': [-1000000.0, 1000.0], 'wMOUNTAINS': None}

dictionary specifying altitude filters

LAND_OCN_FILTERS = ['LAND', 'OCN']

NO_ALTITUDE_FILTER_NAME = 'wMOUNTAINS'

NO_REGION_FILTER_NAME = 'ALL'

property alt_range

Altitude range of filter

apply(*data_obj*)

Apply filter to data object

Parameters
data_obj (UngriddedData, GriddedData) – input data object that is supposed to be filtered

Returns
filtered data object

Return type
UngriddedData, GriddedData

Raises
IOError – if input is invalid

from_list(*lst*)

Set filter name based on input list

property land_ocn

property lat_range

Latitude range of region

property lon_range

Longitude range of region

property name

Name of filter

String containing up to 3 substrings (delimited using dash -) containing: <region_id>-<altitude_filter>-<land_or_sea_only_info>

property region

Region associated with this filter (instance of Region)

property region_name

Name of region

property spl

to_dict()

Convert filter to dictionary

property valid_alt_filter_codes

Valid codes for altitude filters

property valid_land_sea_filter_codes

Codes specifying land/sea filters

property valid_regions

Names of valid regions (AeroCom regions and HTAP regions)

4.11.4 Land / Sea masks

Helper methods for access of and working with land/sea masks. pyaerocom provides automatic access to HTAP land sea masks from this URL:

https://pyaerocom.met.no/pyaerocom-suppl

Filtering by these masks is implemented in `Filter` and all relevant data classes (i.e. `GriddedData`, `UngriddedData`, `ColocatedData`).

pyaerocom.helpers_landsea_masks.**available_htap_masks**()

List of HTAP mask names

Returns
Returns a list of available htap region masks.

Return type
list

pyaerocom.helpers_landsea_masks.**check_all_htap_available**()

Check for missing HTAP masks on local computer and download

pyaerocom.helpers_landsea_masks.**download_htap_masks**(*regions_to_download=None*)

Download HTAP mask

URL: https://pyaerocom.met.no/pyaerocom-suppl.

Parameters
regions_to_download (*list*) – List containing the regions to download.

Returns
List of file paths that point to the mask files that were successfully downloaded

Return type
list

Raises

- **ValueError** – if one of the input regions does not exist
- ***DataRetrievalError*** – if download fails for one of the input regions

pyaerocom.helpers_landsea_masks.**get_htap_mask_files**(**region_ids*)

Get file paths to input HTAP regions

Parameters
***region_ids** – ID's of regions for which mask files are supposed to be retrieved

Returns
list of file paths for each input region

Return type
list

Raises

- **FileNotFoundError** – if default local directory for storage of HTAP masks does not exist
- **NameError** – if multiple mask files are found for the same region

pyaerocom.helpers_landsea_masks.**get_lat_lon_range_mask_region**(*mask*, *latdim_name=None*, *londim_name=None*)

Get outer lat/lon rectangle of a binary mask

Parameters

- **mask** (*xr.DataArray*) – binary mask
- **latdim_name** (*str, optional*) – Name of latitude dimension. The default is None, in which case lat is assumed.
- **londim_name** (*str, optional*) – Name of longitude dimension. The default is None, in which case long is assumed.

Returns
dictionary containing lat and lon ranges of the mask.

Return type
dict

pyaerocom.helpers_landsea_masks.**get_mask_value**(*lat*, *lon*, *mask*)

Get value of mask at input lat / lon position

Parameters

- **lat** (*float*) – latitute
- **lon** (*float*) – longitude
- **mask** (*xarray.DataArray*) – data array

Returns
neirest neigbhour mask value to input lat lon

Return type
float

pyaerocom.helpers_landsea_masks.**load_region_mask_iris**(**regions*)

Loads regional mask to iris.

Parameters
region_id (*str*) – Chosen region.

Returns
cube representing merged mask from input regions

Return type
iris.cube.Cube

pyaerocom.helpers_landsea_masks.**load_region_mask_xr**(**regions*)

Load boolean mask for input regions (as xarray.DataArray)

Parameters
***regions** – regions that are supposed to be loaded and merged (just use string, no list or similar)

Returns
boolean mask for input region(s)

Return type
xarray.DataArray

4.12 Time and frequencies

4.12.1 Handling of time frequencies

General helper methods for the pyaerocom library.

class pyaerocom.tstype.**TsType**(*val*)

FROM_PANDAS = {'AS': 'yearly', 'D': 'daily', 'H': 'hourly', 'MS': 'monthly', 'Q': 'season', 'T': 'minutely', 'W-MON': 'weekly'}

TOL_SECS_PERCENT = 5

TO_NUMPY = {'daily': 'D', 'hourly': 'h', 'minutely': 'm', 'monthly': 'M', 'weekly': 'W', 'yearly': 'Y'}

TO_PANDAS = {'daily': 'D', 'hourly': 'H', 'minutely': 'T', 'monthly': 'MS', 'season': 'Q', 'weekly': 'W-MON', 'yearly': 'AS'}

TO_SI = {'daily': 'd', 'hourly': 'h', 'minutely': 'min', 'monthly': 'month', 'weekly': 'week', 'yearly': 'yr'}

TSTR_TO_CF = {'daily': 'days', 'hourly': 'hours', 'monthly': 'days'}

TS_MAX_VALS = {'daily': 180, 'hourly': 168, 'minutely': 360, 'monthly': 120, 'weekly': 104}

VALID = ['minutely', 'hourly', 'daily', 'weekly', 'monthly', 'yearly', 'native']

VALID_ITER = ['minutely', 'hourly', 'daily', 'weekly', 'monthly', 'yearly']

property base

Base string (without multiplication factor, cf *mulfac*)

property cf_base_unit

Convert ts_type str to CF convention time unit

check_match_total_seconds(*total_seconds*)

Check if this object matches with input interval length in seconds

Parameters
total_seconds (*int or float*) – interval length in units of seconds (e.g. 86400 for daily)

Return type
bool

property datetime64_str

Convert ts_type str to datetime64 unit string

static from_total_seconds(*total_seconds*)

Try to infer TsType based on interval length

Parameters
total_seconds (*int or float*) – total number of seconds

Raises
TemporalResolutionError – If no TsType can be inferred for input number of seconds

Return type
TsType

get_min_num_obs(*to_ts_type:* TsType, *min_num_obs: dict*) → int

property mulfac

Multiplication factor of frequency

property next_higher

Next lower resolution code

property next_lower

Next lower resolution code

This will go to the next lower base resolution, that is if current is 3daily, it will return weekly, however, if current exceeds next lower base, it will iterate that base, that is, if current is 8daily, next lower will be 2weekly (and not 9daily).

property num_secs

Number of seconds in one period

Note: Be aware that for monthly frequency the number of seconds is not well defined!

property timedelta64_str

Convert ts_type str to datetime64 unit string

to_numpy_freq()

to_pandas_freq()

Convert ts_type to pandas frequency string

to_si()

Convert to SI conform string (e.g. used for unit conversion)

to_timedelta64()

Convert frequency to timedelta64 object

Can be used, e.g. as tolerance when reindexing pandas Series

Return type
timedelta64

property tol_secs

Tolerance in seconds for current TsType

property val

Value of frequency (string type), e.g. 3daily

static valid(*val*)

4.12.2 Temporal resampling

Module containing time resampling functionality

class pyaerocom.time_resampler.**TimeResampler**(*input_data=None*)

Object that can be use to resample timeseries data

It supports hierarchical resampling of `xarray.DataArray` objects and `pandas.Series` objects.

Hierarchical means, that resampling constraints can be applied for each level, that is, if hourly data is to be resampled to monthly, it may be specified to first required minimum number of hours per day, and minimum days per month, to create the output data.

AGGRS_UNIT_PRESERVE = ('mean', 'median', 'std', 'max', 'min')

DEFAULT_HOW = 'mean'

property fun

Resamplig method (depends on input data type)

property input_data

Input data object that is to be resampled

property last_units_preserved

Boolean indicating if last resampling operation preserves units

resample(*to_ts_type*, *input_data=None*, *from_ts_type=None*, *how=None*, *min_num_obs=None*, ***kwargs*)

Resample input data

Parameters

- **to_ts_type** (*str or TsType*) – output resolution
- **input_data** (*pandas.Series or xarray.DataArray*) – data to be resampled
- **from_ts_type** (*str or TsType, optional*) – current temporal resolution of data
- **how** (*str*) – string specifying how the data is to be aggregated, default is mean
- **min_num_obs** (*dict or int, optinal*) – integer or nested dictionary specifying minimum number of observations required to resample from higher to lower frequency. For instance, if *input_data* is hourly and *to_ts_type* is monthly, you may specify something like:

```
min_num_obs =
    {'monthly'  :   {'daily'  : 7},
     'daily'    :   {'hourly' : 6}}
```

  to require at least 6 hours per day and 7 days per month.

- ****kwargs** – additional input arguments passed to resampling method

Returns

resampled data object

Return type

pandas.Series or xarray.DataArray

4.12.3 Global constants

Definitions and helpers related to time conversion

4.13 Vertical coordinate support

Note: BETA: most functionality of this module is currently not implemented in any of the pyaerocom standard API.

Methods to convert different standards of vertical coordinates

For details see here:

http://cfconventions.org/Data/cf-conventions/cf-conventions-1.0/build/apd.html

Note: UNDER DEVELOPMENT -> NOT READY YET

class pyaerocom.vert_coords.**AltitudeAccess**(*gridded_data*)

ADD_FILE_OPT = {'pres': ['temp']}

ADD_FILE_REQ = {'deltaz3d': ['ps']}

Additional variables that are required to compute altitude levels

ADD_FILE_VARS = ['z', 'z3d', 'pres', 'deltaz3d']

Additional variable names (in AEROCOM convention) that are used to search for additional files that can be used to access or compute the altitude levels at each grid point

check_altitude_access(***coord_info*)

Checks if altitude levels can be accessed

Parameters
****coord_info** – test coordinate specifications for extraction of 1D data object. Passed to *extract_1D_subset_from_data()*.

Returns
True, if altitude access is provided, else False

Return type
bool

property coord_list

List of AeroCom coordinate names for altitude access

extract_1D_subset_from_data(***coord_info*)

Extract 1D subset containing only vertical coordinate dimension

Note: So far this Works only for 4D or 3D data that contains latitude and longitude dimension and a vertical coordinate, optionally also a time dimension.

The subset is extracted for a test coordinate (latitude, longitude) that may be specified optinally via `coord_info`.

Parameters
****coord_info** – optional test coordinate specifications for other than vertical dimension. For all dimensions that are not specified explicitely, the first available coordinate in `data_obj` is used.

get_altitude(*latitude*, *longitude*)

property has_access

Boolean specifying whether altitudes can be accessed

Note: Performs access check using *check_altitude_access()* if access flag is False

property reader

Instance of `ReadGridded`

search_aux_coords(*coord_list*)

Search and assign coordinates provided by input list

All coordinates that are found are assigned to this object and can be accessed via `self[coord_name]`.

Parameters
coord_list (*list*) – list containing AeroCom coordinate names

Returns
True if all coordinates can be accessed, else False

Return type
bool

Raises
CoordinateNameError – if one of the input coordinate names is not supported by pyaerocom. See coords.ini file of pyaerocom for available coordinates.

class pyaerocom.vert_coords.**VerticalCoordinate**(*name=None*)

CONVERSION_METHODS = {'ahspc': <function atmosphere_hybrid_sigma_pressure_coordinate_to_pressure>, 'asc': <function atmosphere_sigma_coordinate_to_pressure>, 'gph': <function geopotentialheight2altitude>}

CONVERSION_REQUIRES = {'ahspc': ['a', 'b', 'ps', 'p0'], 'asc': ['sigma', 'ps', 'ptop'], 'gph': []}

FUNS_YIELD = {'ahspc': 'air_pressure', 'asc': 'air_pressure', 'gph': 'altitude'}

NAMES_NOT_SUPPORTED = ['model_level_number']

NAMES_SUPPORTED = {'air_pressure': 'pres', 'altitude': 'z', 'atmosphere_hybrid_sigma_pressure_coordinate': 'ahspc', 'atmosphere_sigma_coordinate': 'asc', 'geopotential_height': 'gph'}

REGISTERED = ['altitude', 'air_pressure', 'geopotential_height', 'atmosphere_sigma_coordinate', 'atmosphere_hybrid_sigma_pressure_coordinate', 'model_level_number']

registered names

STANDARD_NAMES = {'ahspc': 'atmosphere_hybrid_sigma_pressure_coordinate', 'asc': 'atmosphere_sigma_coordinate', 'gph': 'geopotential_height', 'pres': 'air_pressure', 'z': 'altitude'}

calc_pressure(*lev*, ***kwargs*)

Compute pressure levels for input vertical coordinate

Parameters

- **vals** – level values that are supposed to be converted into pressure
- ****kwargs** – additional keyword args required for computation of pressure levels (cf. *CONVERSION_METHODS* and corresponding inputs for method available)

Returns
pressure levels in Pa

Return type
ndarray

property conversion_requires

Valid argument names for *fun()*

property conversion_supported

Boolean specifying whether a conversion scheme is provided

property fun

Function used to convert levels into pressure

property lev_increases_with_alt

Boolean specifying whether coordinate levels increase with altitude

Return type
True

pressure2altitude(*p*, ***kwargs*)

Convert pressure to altitude

Wrapper for method

property vars_supported_str

pyaerocom.vert_coords.**atmosphere_hybrid_sigma_pressure_coordinate_to_pressure**(*a*, *b*, *ps*, *p0=None*)

Convert atmosphere_hybrid_sigma_pressure_coordinate to pressure in Pa

Formula:

Either

$$p(k) = a(k) \cdot p_0 + b(k) \cdot p_{surface}$$

or

$$p(k) = ap(k) + b(k) \cdot p_{surface}$$

Parameters

- **a** (*ndarray*) – sigma level values (a(k) in formula 1, and ap(k) in formula 2)
- **b** (*ndarray*) – dimensionless fraction per level (must be same length as a)

- **ps** (*float*) – surface pressure
- **p0** – reference pressure (only relevant for alternative formula 1)

Returns
computed pressure levels in Pa (standard_name=air_pressure)

Return type
ndarray

pyaerocom.vert_coords.**atmosphere_sigma_coordinate_to_pressure**(*sigma*, *ps*, *ptop*)

Convert atmosphere sigma coordinate to pressure in Pa

Note: This formula only works at one lon lat coordinate and at one instant in time.

Formula:

$$p(k) = p_{top} + \sigma(k) \cdot (p_{surface} - p_{top})$$

Parameters

- **sigma** (*ndarray or float*) – sigma coordinate (1D) array
- **ps** (*float*) – surface pressure
- **ptop** (*float*) – ToA pressure

Returns
computed pressure levels in Pa (standard_name=air_pressure)

Return type
ndarray or float

pyaerocom.vert_coords.**geopotentialheight2altitude**(*geopotential_height*)

Convert geopotential height in m to altitude in m

Note: This is a dummy function that returns the input, as the conversion is not yet implemented.

Parameters
geopotential_height – input geopotential height values in m

Return type
Computed altitude levels

pyaerocom.vert_coords.**is_supported**(*standard_name*)

Checks if input coordinate standard name is supported by pyaerocom

Parameters
standard_name (*str*) – standard name of vertical coordinate

Returns
True, if this coordinate is supported, else False

Return type
bool

pyaerocom.vert_coords.**pressure2altitude**(*p*, **args*, ***kwargs*)

General formula to convert atm. pressure to altitude

Wrapper method for `geonum.atmosphere.pressure2altitude()`

Formula:

$$h = h_{ref} + \frac{T_{ref}}{L}\left(\exp\left[-\frac{\ln\left(\frac{p}{p_{ref}}\right)}{\beta}\right] - 1\right) \quad [m]$$

where:

- h_{ref} is a reference altitude
- T_{ref} is a reference temperature
- L is the atmospheric lapse-rate (cf. `L_STD_ATM`, `L_DRY_AIR`)
- p is the pressure (cf. `pressure()`)
- p_{ref} is a reference pressure
- β is computed using `beta_exp()`

Parameters

- **p** – pressure in Pa
- ***args** – additional non-keyword args passed to `geonum.atmosphere.pressure2altitude()`
- ****kwargs** – additional keyword args passed to `geonum.atmosphere.pressure2altitude()`

Return type

altitudes in m corresponding to input pressure levels in defined atmosphere

4.14 Trends computation

4.14.1 Trends engine

class pyaerocom.trends_engine.**TrendsEngine**

Trend computation engine (does not need to be instantiated)

static compute_trend(*data*, *ts_type*, *start_year*, *stop_year*, *min_num_yrs*, *season=None*, *slope_confidence=None*)

Compute trend

Parameters

- **data** (*pd.Series*) – input timeseries data
- **ts_type** (*str*) – frequency of input data (must be monthly or yearly)
- **start_year** (*int or str*) – start of period for trend
- **stop_year** (*int or str*) – end of period for trend
- **min_num_yrs** (*int*) – minimum number of years for trend computation
- **season** (*str, optional*) – which season to use, defaults to whole year (no season)

- **slope_confidence** (*float, optional*) – confidence of slope, between 0 and 1, defaults to 0.68.

Returns
trends results for input data

Return type
dict

4.14.2 Helper methods

Helper methods for computation of trends

Note: Most methods here are private and not to be used directly. Please use `TrendsEngine` instead.

pyaerocom.trends_helpers.**_compute_trend_error**(*m*, *m_err*, *v0*, *v0_err*)

Computes error of trend estimate using gaussian error propagation

The (normalised) trend is computed as T = m / v0

where m denotes the slope of a regression line and v0 denotes the normalistation value. This method computes the uncertainty of T (delta_T) using gaussian error propagation of uncertainties accompanying m and v0.

Parameters

- **m** (*float*) – slope in units of <U> yr-1 (where <U> denotes the unit of the data). (m -> "montant").
- **m_err** (*float*) – slope error (same unit as *m*)
- **v0** (*float*) – normalisation value in units of <U>
- **v0_err** (*float*) – error of *v0* (same units as *v0*)

Returns
error of T in computed using gaussian error propagation of trend formula in units of %/yr

Return type
float

pyaerocom.trends_helpers.**_end_season**(*seas*, *yr*)

pyaerocom.trends_helpers.**_find_area**(*lat*, *lon*, *regions_dict=None*)

Find area corresponding to input lat/lon coordinate

Parameters

- **lat** (*float*) – latitude
- **lon** (*float*) – longitude

Returns
name of region

Return type
str

pyaerocom.trends_helpers.**_get_season**(*mon*)

pyaerocom.trends_helpers.**_get_season_from_months**(*months: str*) → str

pyaerocom.trends_helpers.**_get_unique_seasons**(*idx*)

pyaerocom.trends_helpers.**_get_yearly**(*data*, *seas*, *start_yr*)

pyaerocom.trends_helpers.**_init_period_dates**(*start_year*, *stop_year*, *season*)

pyaerocom.trends_helpers.**_init_trends_result_dict**(*start_yr*)

pyaerocom.trends_helpers.**_mid_season**(*seas*, *yr*)

pyaerocom.trends_helpers.**_seas_slice**(*yr*, *season*)

pyaerocom.trends_helpers.**_start_season**(*seas*, *yr*)

pyaerocom.trends_helpers.**_start_stop_period**(*period*)

Convert period str to start / stop dates

Parameters

period (*str*) – period str, e.g. '1990-2010'

Returns

- *date* – start datetime
- *date* – stop datetime

pyaerocom.trends_helpers.**_years_from_periodstr**(*period*)

Convert period str to start / stop years

Parameters

period (*str*) – period str, e.g. '1990-2010'

Returns

- *int* – start year
- *int* – stop year

4.15 Utility functions

pyaerocom.utils.**create_varinfo_table**(*model_ids*, *vars_or_var_patterns*, *read_data=False*, *sort_by_cols=['Var', 'Model']*)

Create an info table for model list based on variables

The method iterates over all models in `model_list` and creates an instance of `ReadGridded`. Variable matches are searched based on input list `vars_or_var_patterns` (you may also use wildcards to specify a family of variables) and for each match the information below is collected. The search also includes variables that are not directly available in the model data but can be computed from other available variables. That is, all variables that are defined in `ReadGridded.AUX_REQUIRES`.

The output table (DataFrame) then consists of the following columns:

- Var: variable name
- Model: model name
- Years: available years
- Freq: frequency
- Vertical: information about vertical dimension (inferred from Aerocom file name)

- At stations: data is at stations (inferred from filename)
- AUX vars: Auxiliary variable required to compute Var (col 1). Only relevant for variables that are computed by the interface
- Dim: number of dimensions (only retrieved if *read_data* is True)
- Dim names: names of dimension coordinates (only retrieved if *read_data* is True)
- Shape: Shape of data (only retrieved if *read_data* is True)
- Read ok: reading was successful (only retrieved if *read_data* is True)

Parameters

- **model_ids** (*list*) – list of model ids to be analysed (can also be string -> single model)
- **vars_or_var_patterns** (*list*) – list of variables or variable patterns to be analysed (can also be string -> single variable or variable family)
- **read_data** (*bool*) – if True, more information about the imported data will be available in the table (e.g. no. of dimensions, names of dimension coords) but the routine will run longer since the data is imported
- **sort_by_cols** (*list*) – column sort order (use header names in listing above). Defaults to *['Var', 'Model']*

Returns

dataframe including result table (ready to be saved as csv or other tabular format or to be displayed in a jupyter notebook)

Return type

pandas.DataFrame

Example

```
>>> from pyaerocom import create_varinfo_table
>>> models = ['INCA-BCext_CTRL2016-PD',
              'GEOS5-freegcm_CTRL2016-PD']
>>> vars = ['ang4487aer', 'od550aer', 'ec*']
>>> df = create_varinfo_table(models, vars)
>>> print(df)
```

pyaerocom.utils.**print_file**(*path: Path | str*)

4.16 Helpers

General helper methods for the pyaerocom library.

pyaerocom.helpers.**calc_climatology**(*s*, *start*, *stop*, *min_count=None*, *set_year=None*, *resample_how='mean'*)

Compute climatological timeseries from pandas.Series

Parameters

- **s** (*pandas.Series*) – time series data

- **start** (*numpy.datetime64 or similar*) – start time of data used to compute climatology
- **stop** (*numpy.datetime64 or similar*) – start time of data used to compute climatology
- **mincount_month** (*int, optional*) – minimum number of observations required per aggregated month in climatological interval. Months not meeting this requirement will be set to NaN.
- **set_year** (*int, optional*) – if specified, the output data will be assigned the input year. Else the middle year of the climatological interval is used.
- **resample_how** (*str*) – string specifying how the climatological timeseries is to be aggregated

Returns
dataframe containing climatological timeseries as well as columns std and count

Return type
DataFrame

pyaerocom.helpers.**cftime_to_datetime64**(*times*, *cfunit=None*, *calendar=None*)

Convert numerical timestamps with epoch to numpy datetime64

This method was designed to enhance the performance of datetime conversions and is based on the corresponding information provided in the cftime package (see here). Particularly, this object does, what the `num2date()` therein does, but faster, in case the time stamps are not defined on a non standard calendar.

Parameters

- **times** (`list` or `ndarray` or `iris.coords.DimCoord`) – array containing numerical time stamps (relative to basedate of `cfunit`). Can also be a single number.
- **cfunit** (`str` or `Unit`, optional) – CF unit string (e.g. day since 2018-01-01 00:00:00.00000000 UTC) or unit. Required if *times* is not an instance of `iris.coords.DimCoord`
- **calendar** (`str`, optional) – string specifying calendar (only required if `cfunit` is of type `str`).

Returns
numpy array containing timestamps as datetime64 objects

Return type
ndarray

Raises
`ValueError` – if cfunit is `str` and calendar is not provided or invalid, or if the cfunit string is invalid

Example

```
>>> cfunit_str = 'day since 2018-01-01 00:00:00.00000000 UTC'
>>> cftime_to_datetime64(10, cfunit_str, "gregorian")
array(['2018-01-11T00:00:00.000000'], dtype='datetime64[us]')
```

pyaerocom.helpers.**check_coord_circular**(*coord_vals*, *modulus*, *rtol=1e-05*)

Check circularity of coordinate

Parameters

- **coord_vals** (*list or ndarray*) – values of coordinate to be tested
- **modulus** (*float or int*) – modulus of coordinate (e.g. 360 for longitude)
- **rtol** (*float*) – relative tolerance

Returns
True if circularity is given, else False

Return type
bool

Raises
ValueError – if circularity is given and results in overlap (right end of input array is mapped to a value larger than the first one at the left end of the array)

pyaerocom.helpers.**copy_coords_cube**(*to_cube*, *from_cube*, *inplace=True*)

Copy all coordinates from one cube to another

Requires the underlying data to be the same shape.

Warning: This operation will delete all existing coordinates and auxiliary coordinates and will then copy the ones from the input data object. No checks of any kind will be performed

Parameters

- **to_cube**
- **other** (*GriddedData or Cube*) – other data object (needs to be same shape as this object)

Returns
data object containing coordinates from other object

Return type
GriddedData

pyaerocom.helpers.**datetime2str**(*time*, *ts_type=None*)

pyaerocom.helpers.**delete_all_coords_cube**(*cube*, *inplace=True*)

Delete all coordinates of an iris cube

Parameters

- **cube** (*iris.cube.Cube*) – input cube that is supposed to be cleared of coordinates
- **inplace** (*bool*) – if True, then the coordinates are deleted in the input object, else in a copy of it

Returns
input cube without coordinates

Return type
iris.cube.Cube

pyaerocom.helpers.**extract_latlon_dataarray**(*arr*, *lat*, *lon*, *lat_dimname=None*, *lon_dimname=None*, *method='nearest'*, *new_index_name=None*, *check_domain=True*)

Extract individual lat / lon coordinates from *DataArray*

Parameters

- **arr** (*DataArray*) – data (must contain lat and lon dimensions)
- **lat** (*array or similar*) – 1D array containing latitude coordinates
- **lon** (*array or similar*) – 1D array containing longitude coordinates
- **lat_dimname** (*str, optional*) – name of latitude dimension in input data (if None, it assumes standard name)
- **lon_dimname** (*str, optional*) – name of longitude dimension in input data (if None, it assumes standard name)
- **method** (*str*) – how to interpolate to input coordinates (defaults to nearest neighbour)
- **new_index_name** (*str, optional*) – name of flattend latlon dimension (defaults to latlon)
- **check_domain** (*bool*) – if True, lat/lon domain of datarray is checked and all input coordinates that are outside of the domain are ignored.

Returns
data at input coordinates

Return type
DataArray

pyaerocom.helpers.**get_constraint**(*lon_range=None*, *lat_range=None*, *time_range=None*, *meridian_centre=True*)

Function that creates an `iris.Constraint` based on input

Note: Please be aware of the definition of the longitudes in your data when cropping within the longitude dimension. The longitudes in your data may be defined either from **-180 <= lon <= 180** (pyaerocom standard) or from **0 <= lon <= 360**. In the former case (-180 -> 180) you can leave the additional input parameter `meridian_centre=True` (default).

Parameters

- **lon_range** (`tuple`, optional) – 2-element tuple containing longitude range for cropping Example input to crop around meridian: *lon_range=(-30, 30)*
- **lat_range** (`tuple`, optional) – 2-element tuple containing latitude range for cropping.
- **time_range** (`tuple`, optional) – 2-element tuple containing time range for cropping. Allowed data types for specifying the times are
  1. a combination of 2 `pandas.Timestamp` instances or
  2. a combination of two strings that can be directly converted into `pandas.Timestamp` instances (e.g. *time_range=("2010-1-1", "2012-1-1")*) or
  3. directly a combination of indices (`int`).
- **meridian_centre** (*bool*) – specifies the coordinate definition range of longitude array. If True, then -180 -> 180 is assumed, else 0 -> 360

Returns
the combined constraint from all valid input parameters

Return type
iris.Constraint

pyaerocom.helpers.**get_highest_resolution**(*ts_type*, **ts_types*)

Get the highest resolution from several ts_type codes

Parameters

- **ts_type** (*str*) – first ts_type
- ***ts_types** – one or more additional ts_type codes

Returns
the ts_type that corresponds to the highest resolution

Return type
str

Raises
ValueError – if one of the input ts_type codes is not supported

pyaerocom.helpers.**get_lat_rng_constraint**(*low*, *high*)

Create latitude constraint based on input range

Parameters

- **low** (*float or int*) – lower latitude coordinate
- **high** (*float or int*) – upper latitude coordinate

Returns
the corresponding iris.Constraint instance

Return type
iris.Constraint

pyaerocom.helpers.**get_lon_rng_constraint**(*low*, *high*, *meridian_centre=True*)

Create longitude constraint based on input range

Parameters

- **low** (*float or int*) – left longitude coordinate
- **high** (*float or int*) – right longitude coordinate
- **meridian_centre** (*bool*) – specifies the coordinate definition range of longitude array of the data to be cropped. If True, then -180 -> 180 is assumed, else 0 -> 360

Returns
the corresponding iris.Constraint instance

Return type
iris.Constraint

Raises

- **ValueError** – if first coordinate in lon_range equals or exceeds second
- ***LongitudeConstraintError*** – if the input implies cropping over border of longitude array (e.g. 160 -> - 160 if -180 <= lon <= 180).

pyaerocom.helpers.**get_lowest_resolution**(*ts_type*, **ts_types*)

Get the lowest resolution from several ts_type codes

Parameters

- **ts_type** (*str*) – first ts_type
- ***ts_types** – one or more additional ts_type codes

Returns
the ts_type that corresponds to the lowest resolution

Return type
str

Raises
ValueError – if one of the input ts_type codes is not supported

pyaerocom.helpers.**get_max_period_range**(*periods*)

pyaerocom.helpers.**get_standard_name**(*var_name*)

Converts AeroCom variable name to CF standard name

Also handles alias names for variables, etc. or strings corresponding to older conventions (e.g. names containing 3D).

Parameters
var_name (*str*) – AeroCom variable name

Returns
corresponding standard name

Return type
str

pyaerocom.helpers.**get_standard_unit**(*var_name*)

Gets standard unit of AeroCom variable

Also handles alias names for variables, etc. or strings corresponding to older conventions (e.g. names containing 3D).

Parameters
var_name (*str*) – AeroCom variable name

Returns
corresponding standard unit

Return type
str

pyaerocom.helpers.**get_time_rng_constraint**(*start*, *stop*)

Create iris.Constraint for data extraction along time axis

Parameters

- **start** (Timestamp or :obj:` str`) – start time of desired subset. If string, it must be convertible into pandas.Timestamp (e.g. "2012-1-1")
- **stop** (Timestamp or :obj:` str`) – start time of desired subset. If string, it must be convertible into pandas.Timestamp (e.g. "2012-1-1")

Returns
iris Constraint instance that can, e.g., be used as input for *pyaerocom.griddeddata.GriddedData.extract()*

Return type
iris.Constraint

pyaerocom.helpers.**get_tot_number_of_seconds**(*ts_type*, *dtime=None*)

Get total no. of seconds for a given frequency

Parameters

- **ts_type** (*str or TsType*) – frequency for which number of seconds is supposed to be retrieved
- **dtime** (*TYPE, optional*) – DESCRIPTION. The default is None.

Raises
AttributeError – DESCRIPTION.

Returns
DESCRIPTION.

Return type
TYPE

pyaerocom.helpers.**infer_time_resolution**(*time_stamps*, *dt_tol_percent=5*, *minfrac_most_common=0.8*)

Infer time resolution based on input time-stamps

Calculates time difference *dt* between consecutive timestamps provided via input array or list. Then it counts the most common *dt* (e.g. 86400 s for daily). Before inferring the frequency it then checks all other *dts* occurring in the input array to see if they are within a certain interval around the most common one (e.g. +/- 5% as default, via arg *dt_tol_percent*), that is, 86390 would be included if most common dt is 86400 s but not 80000s. Then it checks if the number of *dts* that are within that tolerance level around the most common *dt* exceed a certain fraction (arg *minfrac_most_common*) of the total number of *dts* that occur in the input array (default is 80%). If that is the case, the most common frequency is attempted to be derived using `TsType.from_total_seconds()` based on the most common *dt* (in this example that would be *daily*).

Parameters

- **time_stamps** (*pandas.DatetimeIndex, or similar*) – list of time stamps
- **dt_tol_percent** (*int*) – tolerance in percent of accepted range of time diffs with respect to most common time difference.
- **minfrac_most_common** (*float*) – minimum required fraction of time diffs that have to be equal to, or within tolerance range, the most common time difference.

Raises
TemporalResolutionError – if frequency cannot be derived.

Returns
inferred frequency

Return type
str

pyaerocom.helpers.**is_year**(*val*)

Check if input is / may be year

Parameters
val – input that is supposed to be checked

Returns
True if input is a number between -2000 and 10000, else False

Return type
bool

pyaerocom.helpers.**isnumeric**(*val*)

Check if input value is numeric

Parameters
val – input value to be checked

Returns
True, if input value corresponds to a range, else False.

Return type
bool

pyaerocom.helpers.**isrange**(*val*)

Check if input value corresponds to a range

Checks if input is list, or array or tuple with 2 entries, or alternatively a slice that has defined start and stop and has set step to None.

Note: No check is performed, whether first entry is smaller than second entry if all requirements for a range are fulfilled.

Parameters
val – input value to be checked

Returns
True, if input value corresponds to a range, else False.

Return type
bool

pyaerocom.helpers.**lists_to_tuple_list**(**lists*)

Convert input lists (of same length) into list of tuples

e.g. input 2 lists of latitude and longitude coords, output one list with tuple coordinates at each index

pyaerocom.helpers.**make_datetime_index**(*start*, *stop*, *freq*)

Make pandas.DatetimeIndex for input specs

Note: If input frequency is specified in *PANDAS_RESAMPLE_OFFSETS*, an offset will be added (e.g. 15 days for monthly data).

Parameters

- **start** – start time. Preferably as pandas.Timestamp, else it will be attempted to be converted.
- **stop** – stop time. Preferably as pandas.Timestamp, else it will be attempted to be converted.
- **freq** – frequency of datetime index.

Return type
DatetimeIndex

pyaerocom.helpers.**make_datetimeindex_from_year**(*freq*, *year*)

Create pandas datetime index

Parameters

- **freq** (*str*) – pandas frequency str
- **year** (*int*) – year

Returns
index object

Return type
pandas.DatetimeIndex

pyaerocom.helpers.**make_dummy_cube**(*var_name: str*, *start_yr: int = 2000*, *stop_yr: int = 2020*, *freq: str = 'daily'*, *dtype=<class 'float'>*) → Cube

pyaerocom.helpers.**make_dummy_cube_latlon**(*lat_res_deg: float = 2*, *lon_res_deg: float = 3*, *lat_range: list[float] | tuple[float, float] = (-90, 90)*, *lon_range: list[float] | tuple[float, float] = (-180, 180)*)

Make an empty Cube with given latitude and longitude resolution

Dimensions will be lat, lon

Parameters

- **lat_res_deg** (*float or int*) – latitude resolution of grid
- **lon_res_deg** (*float or int*) – longitude resolution of grid
- **lat_range** (*tuple or list*) – 2-element list containing latitude range. If *None*, then *(-90, 90)* is used.
- **lon_range** (*tuple or list*) – 2-element list containing longitude range. If *None*, then *(-180, 180)* is used.

Returns
dummy cube in input resolution

Return type
Cube

pyaerocom.helpers.**merge_station_data**(*stats*, *var_name*, *pref_attr=None*, *sort_by_largest=True*, *fill_missing_nan=True*, *add_meta_keys=None*, *resample_how=None*, *min_num_obs=None*)

Merge multiple StationData objects (from one station) into one instance

---

Note: all input `StationData` objects need to have same attributes `station_name`, `latitude`, `longitude` and `altitude`

Parameters

- **stats** (*list*) – list containing `StationData` objects (note: all of these objects must contain variable data for the specified input variable)
- **var_name** (*str*) – data variable name that is to be merged
- **pref_attr** – optional argument that may be used to specify a metadata attribute that is available in all input `StationData` objects and that is used to order the input stations by relevance. The associated values of this attribute need to be sortable (e.g. revision_date). This is only relevant in case overlaps occur. If unspecified the relevance of the stations is sorted based on the length of the associated data arrays.
- **sort_by_largest** (*bool*) – if True, the result from the sorting is inverted. E.g. if `pref_attr` is unspecified, then the stations will be sorted based on the length of the data vectors, starting with the shortest, ending with the longest. This sorting result will then be inverted, if `sort_by_largest=True`, so that the longest time series get's highest importance.

If, e.g. `pref_attr='revision_date'`, then the stations are sorted by the associated revision date value, starting with the earliest, ending with the latest (which will also be inverted if this argument is set to True)

- **fill_missing_nan** (*bool*) – if True, the resulting time series is filled with NaNs. NOTE: this requires that information about the temporal resolution (ts_type) of the data is available in each of the StationData objects.
- **add_meta_keys** (*str or list, optional*) – additional non-standard metadata keys that are supposed to be considered for merging.
- **resample_how** (*str or dict, optional*) – in case input stations come in different frequencies they are merged to the lowest common freq. This parameter can be used to control, which aggregator(s) are to be used (e.g. mean, median).
- **min_num_obs** (*str or dict, optional*) – in case input stations come in different frequencies they are merged to the lowest common freq. This parameter can be used to control minimum number of observation constraints for the downsampling.

Returns
merged data

Return type
StationData

pyaerocom.helpers.**numpy_to_cube**(*data*, *dims=None*, *var_name=None*, *units=None*, ***attrs*)

Make a cube from a numpy array

Parameters

- **data** (*ndarray*) – input data
- **dims** (*list, optional*) – list of `iris.coord.DimCoord` instances in order of dimensions of input data array (length of list and shapes of each of the coordinates must match dimensions of input data)
- **var_name** (*str, optional*) – name of variable
- **units** (*str*) – unit of variable
- ****attrs** – additional attributes to be added to metadata

Return type
iris.cube.Cube

Raises
DataDimensionError – if input *dims* is specified and results in conflict

pyaerocom.helpers.**resample_time_dataarray**(*arr*, *freq*, *how=None*, *min_num_obs=None*)

Resample the time dimension of a `xarray.DataArray`

Note: The dataarray must have a dimension coordinate named "time"

Parameters

- **arr** (*DataArray*) – data array to be resampled
- **freq** (*str*) – new temporal resolution (can be pandas freq. string, or pyaerocom ts_type)
- **how** (*str*) – how to aggregate (e.g. mean, median)

- **min_num_obs** (*int, optional*) – minimum number of observations required per period (when downsampling). E.g. if input is in daily resolution and freq is monthly and min_num_obs is 10, then all months that have less than 10 days of data are set to nan.

Returns
resampled data array object

Return type
DataArray

Raises

- **IOError** – if data input *arr* is not an instance of `DataArray`
- ***DataDimensionError*** – if time dimension is not available in dataset

`pyaerocom.helpers.`**resample_timeseries**(*ts*, *freq*, *how=None*, *min_num_obs=None*)

Resample a timeseries (pandas.Series)

Parameters

- **ts** (*Series*) – time series instance
- **freq** (*str*) – new temporal resolution (can be pandas freq. string, or pyaerocom ts_type)
- **how** – aggregator to be used, accepts everything that is accepted by `pandas.core.resample.Resampler.agg()` and in addition, percentiles may be provided as str using e.g. 75percentile as input for the 75% percentile.
- **min_num_obs** (*int, optional*) – minimum number of observations required per period (when downsampling). E.g. if input is in daily resolution and freq is monthly and min_num_obs is 10, then all months that have less than 10 days of data are set to nan.

Returns
resampled time series object

Return type
Series

`pyaerocom.helpers.`**same_meta_dict**(*meta1*, *meta2*, *ignore_keys=['PI']*, *num_keys=['longitude', 'latitude', 'altitude']*, *num_rtol=0.01*)

Compare meta dictionaries

Parameters

- **meta1** (*dict*) – meta dictionary that is to be compared with `meta2`
- **meta2** (*dict*) – meta dictionary that is to be compared with `meta1`
- **ignore_keys** (*list*) – list containing meta keys that are supposed to be ignored
- **num_keys** (*keys that contain numerical values*)
- **num_rtol** (*float*) – relative tolerance level for comparison of numerical values

Returns
True, if dictionaries are the same, else False

Return type
bool

`pyaerocom.helpers.`**seconds_in_periods**(*timestamps*, *ts_type*)

Calculates the number of seconds for each period in timestamps.

Parameters

- **timestamps** (*numpy.datetime64 or numpy.ndarray*) – Either a single datetime or an array of datetimes.
- **ts_type** (*str*) – Frequency of timestamps.

Returns
Array with same length as timestamps containing number of seconds for each period.

Return type
np.array

pyaerocom.helpers.**sort_ts_types**(*ts_types*)

Sort a list of ts_types

Parameters
ts_types (*list*) – list of strings (or instance of TsType) to be sorted

Returns
list of strings with sorted frequencies

Return type
list

Raises
TemporalResolutionError – if one of the input ts_types is not supported

pyaerocom.helpers.**start_stop**(*start*, *stop=None*, *stop_sub_sec=True*)

Create pandas timestamps from input start / stop values

Note: If input suggests climatological data in AeroCom format (i.e. year=9999) then the year is converted to 2222 instead since pandas cannot handle year 9999.

Parameters

- **start** – start time (any format that can be converted to pandas.Timestamp)
- **stop** – stop time (any format that can be converted to pandas.Timestamp)
- **stop_sub_sec** (*bool*) – if True and if input for stop is a year (e.g. 2015) then one second is subtracted from stop timestamp (e.g. if input stop is 2015 and denotes "until 2015", then for the returned stop timestamp one second will be subtracted, so it would be 31.12.2014 23:59:59).

Returns

- *pandas.Timestamp* – start timestamp
- *pandas.Timestamp* – stop timestamp

Raises
ValueError – if input cannot be converted to pandas timestamps

pyaerocom.helpers.**start_stop_from_year**(*year*)

Create start / stop timestamp from year

Parameters
year (*int*) – the year for which start / stop is to be instantiated

Returns

- *numpy.datetime64* – start datetime

- *numpy.datetime64* – stop datetime

pyaerocom.helpers.**start_stop_str**(*start*, *stop=None*, *ts_type=None*)

pyaerocom.helpers.**str_to_iris**(*key*, ***kwargs*)

Mapping function that converts strings into iris analysis objects

Please see dictionary STR_TO_IRIS in this module for valid definitions

Parameters
key (*str*) – key of STR_TO_IRIS dictionary

Returns
corresponding iris analysis object (e.g. Aggregator, method)

Return type
obj

pyaerocom.helpers.**to_datestring_YYYYMMDD**(*value*)

Convert input time to string with format YYYYMMDD

Parameters
value – input time, may be string, datetime, numpy.datetime64 or pandas.Timestamp

Returns
input formatted to string YYYYMMDD

Return type
str

Raises
ValueError – if input is not supported

pyaerocom.helpers.**to_datetime64**(*value*)

Convert input value to numpy.datetime64

Parameters
value – input value that is supposed to be converted, needs to be either str, datetime.datetime, pandas.Timestamp or an integer specifying the desired year.

Returns
input timestamp converted to datetime64

Return type
datetime64

pyaerocom.helpers.**to_pandas_timestamp**(*value*)

Convert input to instance of pandas.Timestamp

Parameters
value – input value that is supposed to be converted to time stamp

Return type
pandas.Timestamp

pyaerocom.helpers.**tuple_list_to_lists**(*tuple_list*)

Convert list with tuples (e.g. (lat, lon)) into multiple lists

pyaerocom.helpers.**varlist_aerocom**(*varlist*)

4.17 Mathematical helpers

Mathematical low level utility methods of pyaerocom

pyaerocom.mathutils.**closest_index**(*num_array*, *value*)

Returns index in number array that is closest to input value

pyaerocom.mathutils.**corr**(*ref_data*, *data*, *weights=None*)

Compute correlation coefficient

Parameters

- **data_ref** (*ndarray*) – x data
- **data** (*ndarray*) – y data
- **weights** (*ndarray, optional*) – array containing weights for each point in *data*

Returns

correlation coefficient

Return type

float

pyaerocom.mathutils.**estimate_value_range**(*vmin*, *vmax*, *extend_percent=0*)

Round and extend input range to estimate lower and upper bounds of range

Parameters

- **vmin** (*float*) – lower value of range
- **vmax** (*float*) – upper value of range
- **extend_percent** (*int*) – percentage specifying to which extent the input range is supposed to be extended.

Returns

- *float* – estimated lower end of range
- *float* – estimated upper end of range

pyaerocom.mathutils.**exponent**(*num*)

Get exponent of input number

Parameters

num (float or iterable) – input number

Returns

exponent of input number(s)

Return type

int or ndarray containing ints

Example

```
>>> from pyaerocom.mathutils import exponent
>>> exponent(2340)
3
```

pyaerocom.mathutils.**in_range**(*x*, *low*, *high*)

pyaerocom.mathutils.**is_strictly_monotonic**(*iter1d*) → bool

Check if 1D iterble is strictly monotonic

Parameters
iter1d – 1D iterable object to be tested

Return type
bool

pyaerocom.mathutils.**make_binlist**(*vmin: float*, *vmax: float*, *num: int | None = None*) → list

pyaerocom.mathutils.**numbers_in_str**(*input_string*)

This method finds all numbers in a string

Note:

- Beta version, please use with care
- Detects only integer numbers, dots are ignored

Parameters
input_string (*str*) – string containing numbers

Returns
list of strings specifying all numbers detected in string

Return type
list

Example

```
>>> numbers_in_str('Bla42Blub100')
[42, 100]
```

pyaerocom.mathutils.**range_magnitude**(*low*, *high*)

Returns magnitude of value range

Parameters

- **low** (*float*) – lower end of range
- **high** (*float*) – upper end of range

Returns
magnitudes spanned by input numbers

Return type
int

Example

```
>>> range_magnitude(0.1, 100)
3
>>> range_magnitude(100, 0.1)
-3
>>> range_magnitude(1e-3, 1e6)
9
```

pyaerocom.mathutils.**sum**(*data*, *weights=None*)

Summing operation with option to perform weighted sum

Parameters

- **data** (*ndarray*) – data array that is supposed to be summed up
- **weights** (*ndarray, optional*) – array containing weights for each point in *data*

Returns
sum of values in input array

Return type
float or int

pyaerocom.mathutils.**weighted_corr**(*ref_data*, *data*, *weights*)

Compute weighted correlation

Parameters

- **data_ref** (*ndarray*) – x data
- **data** (*ndarray*) – y data
- **weights** (*ndarray*) – array containing weights for each point in *data*

Returns
weighted correlation coefficient

Return type
float

pyaerocom.mathutils.**weighted_cov**(*ref_data*, *data*, *weights*)

Compute weighted covariance

Parameters

- **data_ref** (*ndarray*) – x data
- **data** (*ndarray*) – y data
- **weights** (*ndarray*) – array containing weights for each point in *data*

Returns
covariance

Return type
float

pyaerocom.mathutils.**weighted_mean**(*data*, *weights*)

Compute weighted mean

Parameters

- **data** (*ndarray*) – data array that is supposed to be averaged

- **weights** (*ndarray*) – array containing weights for each point in *data*

Returns
weighted mean of data array

Return type
float or int

pyaerocom.mathutils.**weighted_sum**(*data*, *weights*)

Compute weighted sum using numpy dot product

Parameters

- **data** (*ndarray*) – data array that is supposed to be summed up
- **weights** (*ndarray*) – array containing weights for each point in *data*

Returns
weighted sum of values in input array

Return type
float

4.18 Geodesic calculations and topography

Module for geographical calculations

This module contains low-level methods to perform geographical calculations, (e.g. distance between two coordinates)

pyaerocom.geodesy.**calc_distance**(*lat0*, *lon0*, *lat1*, *lon1*, *alt0=None*, *alt1=None*, *auto_altitude_srtm=False*)

Calculate distance between two coordinates

Parameters

- **lat0** (*float*) – latitude of first point in decimal degrees
- **lon0** (*float*) – longitude of first point in decimal degrees
- **lat1** (*float*) – latitude of secondpoint in decimal degrees
- **lon1** (*float*) – longitude of second point in decimal degrees
- **alt0** (float, optional) – altitude of first point in m
- **alt1** (float, optional) – altitude of second point in m
- **auto_altitude_srtm** (*bool*) – if True, then all altitudes that are unspecified are set to the corresponding topographic altitude of that coordinate, using SRTM (only works for coordinates where SRTM topographic data is accessible).

Returns
distance between points in km

Return type
float

pyaerocom.geodesy.**calc_latlon_dists**(*latref*, *lonref*, *latlons*)

Calculate distances of (lat, lon) coords to input lat, lon coordinate

Parameters

- **latref** (*float*) – latitude of reference coordinate

- **lonref** (*float*) – longitude of reference coordinate
- **latlons** (*list*) – list of (lat, lon) tuples for which distances to (latref, lonref) are computed

Returns
list of computed geographic distances to input reference coordinate for all (lat, lon) coords in *latlons*

Return type
list

pyaerocom.geodesy.**find_coord_indices_within_distance**(*latref*, *lonref*, *latlons*, *radius=1*)

Find indices of coordinates that match input coordinate

Parameters

- **latref** (*float*) – latitude of reference coordinate
- **lonref** (*float*) – longitude of reference coordinate
- **latlons** (*list*) – list of (lat, lon) tuples for which distances to (latref, lonref) are computed
- **radius** (*float or int, optional*) – Maximum allowed distance to input coordinate. The default is 1.

Returns
Indices of latlon coordinates in **:param:`latlons`** that are within the specified radius around (*latref*, *lonref*). The indices are sorted by distance to the input coordinate, starting with the closest

Return type
ndarray

pyaerocom.geodesy.**get_country_info_coords**(*coords*)

Get country information for input lat/lon coordinates

Parameters
coords (*list or tuple*) – list of coord tuples (lat, lon) or single coord tuple

Raises
ValueError – if input format is incorrect

Returns
list of dictionaries containing country information for each input coordinate

Return type
list

pyaerocom.geodesy.**get_topo_altitude**(*lat*, *lon*, *topo_dataset='srtm'*, *topodata_loc=None*, *try_etopo1=True*)

Retrieve topographic altitude for a certain location

Supports topography datasets supported by geonum. These are currently (20 Feb. 19) srtm (SRTM dataset, default, automatic access if online) and etopo1 (ETOPO1 dataset, lower resolution, must be available on local machine or server).

Parameters

- **lat** (*float*) – latitude of coordinate
- **lon** (*float*) – longitude of coordinate
- **topo_dataset** (*str*) – name of topography dataset
- **topodata_loc** (*str*) – filepath or directory containing supported topographic datasets

- **try_etopo1** (*bool*) – if True and if access fails via input arg *topo_dataset*, then try to access altitude using ETOPO1 dataset.

Returns
dictionary containing input latitude, longitude, altitude and topographic dataset name used to retrieve the altitude.

Return type
dict

Raises
ValueError – if altitude data cannot be accessed

pyaerocom.geodesy.**get_topo_data**(*lat0*, *lon0*, *lat1=None*, *lon1=None*, *topo_dataset='srtm'*, *topodata_loc=None*, *try_etopo1=False*)

Retrieve topographic altitude for a certain location

Supports topography datasets supported by geonum. These are currently (20 Feb. 19) srtm (SRTM dataset, default, automatic access if online) and etopo1 (ETOPO1 dataset, lower resolution, must be available on local machine or server).

Parameters

- **lat0** (*float*) – start longitude for data extraction
- **lon0** (*float*) – start latitude for data extraction
- **lat1** (*float*) – stop longitude for data extraction (default: None). If None only data around lon0, lat0 will be extracted.
- **lon1** (*float*) – stop latitude for data extraction (default: None). If None only data around lon0, lat0 will be extracted
- **topo_dataset** (*str*) – name of topography dataset
- **topodata_loc** (*str*) – filepath or directory containing supported topographic datasets
- **try_etopo1** (*bool*) – if True and if access fails via input arg *topo_dataset*, then try to access altitude using ETOPO1 dataset.

Returns
data object containing topography data in specified range

Return type
geonum.TopoData

Raises
ValueError – if altitude data cannot be accessed

pyaerocom.geodesy.**haversine**(*lat0*, *lon0*, *lat1*, *lon1*, *earth_radius=6371.0*)

Haversine formula

Approximate horizontal distance between 2 points assuming a spherical earth using haversine formula.

Note: This code was copied from geonum library (date 12/11/2018, J. Gliss)

Parameters

- **lat0** (*float*) – latitude of first point in decimal degrees
- **lon0** (*float*) – longitude of first point in decimal degrees

- **lat1** (*float*) – latitude of second point in decimal degrees
- **lon1** (*float*) – longitude of second point in decimal degrees
- **earth_radius** (*float*) – average earth radius in km, defaults to 6371.0

Returns
horizontal distance between input coordinates in km

Return type
float

pyaerocom.geodesy.**is_within_radius_km**(*lat0*, *lon0*, *lat1*, *lon1*, *maxdist_km*, *alt0=0*, *alt1=0*, ***kwargs*)

Checks if two lon/lat coordinates are within a certain distance to each other

Parameters

- **lat0** (*float*) – latitude of first point in decimal degrees
- **lon0** (*float*) – longitude of first point in decimal degrees
- **lat1** (*float*) – latitude of second point in decimal degrees
- **lon1** (*float*) – longitude of second point in decimal degrees
- **maxdist_km** (*float*) – maximum distance between two points in km
- **alt0** (*float*) – altitude of first point in m
- **alt1** (*float*) – altitude of second point in m

Returns
True, if coordinates are within specified distance to each other, else False

Return type
bool

4.19 Units and unit conversion

4.19.1 Units helpers in base package

pyaerocom.units_helpers.**RATES_FREQ_DEFAULT = 'd'**

default frequency for rates variables (e.g. deposition, precip)

pyaerocom.units_helpers.**UCONV_MUL_FACS = to fac var_name from concso2 ug S/m3 ug m-3 1.997910 concbc ug C/m3 ug m-3 1.000000 concoa ug C/m3 ug m-3 1.000000 concoc ug C/m3 ug m-3 1.000000 conctc ug C/m3 ug m-3 1.000000 concpm25 ug m-3 1 1.000000 concpm10 ug m-3 1 1.000000 concno2 ug N/m3 ug m-3 3.284478 concnh3 ug N/m3 ug m-3 1.215862 wetso4 kg S/ha kg m-2 0.000300 concso4pr mg S/L g m-3 2.995821**

Custom unit conversion factors for certain variables columns: variable -> from unit -> to_unit -> conversion factor

pyaerocom.units_helpers.**convert_unit**(*data*, *from_unit*, *to_unit*, *var_name=None*, *ts_type=None*)

Convert unit of data

Parameters

- **data** (*np.ndarray or similar*) – input data
- **from_unit** (*cf_units.Unit or str*) – current unit of input data

- **to_unit** (*cf_units.Unit or str*) – new unit of input data
- **var_name** (*str, optional*) – name of variable. If provided, and standard conversion with `cf_units` fails, then custom unit conversion is attempted.
- **ts_type** (*str, optional*) – frequency of data. May be needed for conversion of rate variables such as precip, deposition, etc, that may be defined implictly without proper frequency specification in the unit string.

Returns
data in new unit

Return type
data

pyaerocom.units_helpers.**get_unit_conversion_fac**(*from_unit*, *to_unit*, *var_name=None*, *ts_type=None*)

pyaerocom.units_helpers.**rate_unit_implicit**(*unit*)

Check whether input rate unit is implicit

Implicit rate units do not contain frequency string, e.g. "mg m-2" instead of "mg m-2 d-1". Such units are, e.g. used in EMEP output where the frequency corresponds to the output frequency, e.g. "mg m-2" per day if output is daily.

Note: For now, this is just a wrapper for `_check_unit_endswith_freq()`, but there may be more sophisticated options in the future, which may be added to this function.

Parameters
unit (*str*) – unit to be tested

Returns
True if input unit appears to be implicit, else False.

Return type
bool

4.19.2 Units helpers in *io* sub-package

pyaerocom.io.helpers_units.**mass_to_nr_molecules**(*mass*, *mm*)

Calculating the number of molecules form mass and molarmass.

Mass, Molar mass need to be in the same unit, either both g and g/mol or kg and kg/mol.

Parameters

- **mass** (*float*) – mass of all compounds.
- **mm** (*float*) – molar mass of compounds.

Returns
nr_molecules – number of molecules

Return type
float

pyaerocom.io.helpers_units.**nr_molecules_to_mass**(*nr_molecules*, *mm*)

Calculates the mass from the number of molecules and molar mass.

Parameters

- **nr_molecules** (*int*) – Number of molecules
- **mm** (*float*) – Molar mass [g/mol]

Returns
mass – mass in grams

Return type
float

pyaerocom.io.helpers_units.**unitconv_sfc_conc**(*data*, *nr_of_O=2*)

Unitconverting: ugS/m3 to ugSOx/m3

Parameters

- **data** (*array_like*) – Contains the data in units of ugS/m3.
- **nr_of_O** (*int*) – The number of O's in you desired SOx compound.

Returns
data – data in units of ug SOx/m3

Return type
ndarray

pyaerocom.io.helpers_units.**unitconv_sfc_conc_bck**(*data*, *x=2*)

Converting: ugSOx/m3 to ugS/ m3.

Parameters

- **data** (*ndarray*) – Contains the data in units of ugSoX/m3.
- **x** (*int*) – The number of oxygen atoms, O in you desired SOx compound.

Returns
data – in units of ugS/ m3.

Return type
ndarray

Notes

micro grams to kilos is 10**6

pyaerocom.io.helpers_units.**unitconv_wet_depo**(*data*, *time*, *ts_type='monthly'*)

Unitconversion kg S/ha to kg SOx m-2 s-1.

Adding mass of oksygen.

Parameters

- **data** (*ndarray*) – data in unit kg S/ha = kg S/(1000 m2)
- **time** (*pd.Seires[numpy.datetime64]*) – Array of datetime64 timesteps.
- **ts_type** (*str*) – The timeseries type. Default "monthly".

Returns
data – data in units of ugSOx/m3

Return type
ndarray

pyaerocom.io.helpers_units.**unitconv_wet_depo_bck**(*data*, *time*, *ts_type='monthly'*)

The unitconversion kg SO4 m-2 s-1 to kgS/ha.

Removing the weight of oxygen.

Parameters

- **data** (*ndarray*) – Sulphur data you wish to convert.
- **time** (*pd.Seires[numpy.datetime64]*) – Array of datetime64 timesteps.
- **ts_type** (*str*) – The timeseries type. Default monthly.

Returns
data – Sulphur data in units of ugSOx m-3 s-1.

Return type
ndarray

pyaerocom.io.helpers_units.**unitconv_wet_depo_from_emep**(*data*, *time*, *ts_type='monthly'*)

Unitconversion mgS m-2 to kg SO4 m-2 s-1.

Milligram to kilos is 10-6.

Adding mass of oksygen.

Parameters

- **data** (*ndarray*) – data in unit mg S m-2.
- **time** (*pd.Seires[numpy.datetime64]*) – Array of datetime64 timesteps.
- **ts_type** (*str*) – The timeseries type. Default "monthly".

Returns
data – data in units of ugSOx/m3

Return type
ndarray

4.20 Plotting / visualisation (sub package *plot*)

The pyaerocom.plot package contains algorithms related to data visualisation and plotting.

4.20.1 Plotting of maps

pyaerocom.plot.mapping.**get_cmap_maps_aerocom**(*color_theme=None*, *vmin=None*, *vmax=None*)

Get colormap using pyAeroCom color scheme

Parameters

- **color_theme** (*:ColorTheme, optional*) – instance of pyaerocom color theme. If None, the default schemes is used
- **vmin** (*float, optional*) – lower end of value range (only considered for diverging color maps with non-symmetric mapping)
- **vmax** (*float, optional*) – upper end of value range only considered for diverging color maps with non-symmetric mapping)

Return type
colormap

pyaerocom.plot.mapping.**init_map**(*xlim=(-180, 180)*, *ylim=(-90, 90)*, *figh=8*, *fix_aspect=False*, *xticks=None*, *yticks=None*, *color_theme=light*, *projection=None*, *title=None*, *gridlines=False*, *fig=None*, *ax=None*, *draw_coastlines=True*, *contains_cbar=False*)

Initalise a map plot

Parameters

- **xlim** (*tuple*) – 2-element tuple specifying plotted longitude range
- **ylim** (*tuple*) – 2-element tuple specifying plotted latitude range
- **figh** (*int*) – height of figure in inches
- **fix_aspect** (*bool, optional*) – if True, the aspect of the GeoAxes instance is kept fix using the default aspect MAP_AXES_ASPECT defined in *pyaerocom.plot.config*
- **xticks** (*iterable, optional*) – ticks of x-axis (longitudes)
- **yticks** (*iterable, optional*) – ticks of y-axis (latitudes)
- **color_theme** (ColorTheme) – pyaerocom color theme.
- **projection** – projection instance from cartopy.crs module (e.g. PlateCaree). May also be string.
- **title** (*str, optional*) – title that is supposed to be inserted
- **gridlines** (*bool*) – whether or not to add gridlines to the map
- **fig** (*matplotlib.figure.Figure, optional*) – instance of matplotlib Figure class. If specified, the former to input args (figh and fix_aspect) are ignored. Note that the Figure is wiped clean before plotting, so any plotted content will be lost
- **ax** (*GeoAxes, optional*) – axes in which the map is plotted
- **draw_coastlines** (*bool*) – whether or not to draw coastlines
- **contains_cbar** (*bool*) – whether or not a colorbar is intended to be added to the figure (impacts the aspect ratio of the figure).

Returns
ax – axes instance

Return type
cartopy.mpl.geoaxes.GeoAxes

pyaerocom.plot.mapping.**plot_griddeddata_on_map**(*data*, *lons=None*, *lats=None*, *var_name=None*, *unit=None*, *xlim=(-180, 180)*, *ylim=(-90, 90)*, *vmin=None*, *vmax=None*, *add_zero=False*, *c_under=None*, *c_over=None*, *log_scale=True*, *discrete_norm=True*, *cbar_levels=None*, *cbar_ticks=None*, *add_cbar=True*, *cmap=None*, *cbar_ticks_sci=False*, *color_theme=None*, *ax=None*, *ax_cbar=None*, ***kwargs*)

Make a plot of gridded data onto a map

Parameters

- **data** (*ndarray*) – 2D data array

- **lons** (*ndarray*) – longitudes of data
- **lats** (*ndarray*) – latitudes of data
- **var_name** (str, optional) – name of variable that is plotted
- **xlim** (*tuple*) – 2-element tuple specifying plotted longitude range
- **ylim** (*tuple*) – 2-element tuple specifying plotted latitude range
- **vmin** (float, optional) – lower value of colorbar range
- **vmax** (float, optional) – upper value of colorbar range
- **add_zero** (*bool*) – if True and vmin is not 0, then, the colorbar is extended down to 0. This may be used, e.g. for logarithmic scales that should include 0.
- **c_under** (float, optional) – colour of data values smaller than `vmin`
- **c_over** (float, optional) – colour of data values exceeding `vmax`
- **log_scale** (*bool*) – if True, the value to color mapping is done in a pseudo log scale (see `get_cmap_levels_auto()` for implementation)
- **discrete_norm** (*bool*) – if True, color mapping will be subdivided into discrete intervals
- **cbar_levels** (*iterable, optional*) – discrete colorbar levels. Will be computed automatically, if None (and applicable)
- **cbar_ticks** (*iterable, optional*) – ticks of colorbar levels. Will be computed automatically, if None (and applicable)

Returns

matplotlib figure instance containing plot result. Use `fig.axes[0]` to access the map axes instance (e.g. to modify the title or lon / lat range, etc.)

Return type

fig

pyaerocom.plot.mapping.**plot_map_aerocom**(*data*, *region*, ***kwargs*)

High level map plotting function for Aerocom default plotting

Note: This function does not iterate over a cube in time, but uses the first available time index in the data.

Parameters

- **data** (GriddedData) – input data from one timestamp (if data contains more than one time stamp, the first index is used)
- **region** (*str or* Region) – valid region ID or region

pyaerocom.plot.mapping.**plot_nmb_map_colocateddata**(*coldata*, *in_percent=True*, *vmin=-100*, *vmax=100*, *cmap=None*, *s=80*, *marker=None*, *step_bounds=None*, *add_cbar=True*, *norm=None*, *cbar_extend=None*, *add_mean_edgecolor=True*, *ax=None*, *ax_cbar=None*, *cbar_outline_visible=False*, *cbar_orientation=None*, *ref_label=None*, *stats_area_weighted=False*, ***kwargs*)

Plot map of normalised mean bias from instance of ColocatedData

Parameters

- **coldata** (ColocatedData) – data object
- **in_percent** (*bool*) – plot bias in percent
- **vmin** (*int*) – minimum value of colormapping
- **vmax** (*int*) – maximum value of colormapping
- **cmap** (*str or cmap*) – colormap used, defaults to bwr
- **s** (*int*) – size of marker
- **marker** (*str*) – marker used
- **step_bounds** (*int, optional*) – step used for discrete colormapping (if None, continuous is used)
- **cbar_extend** (*str*) – extend colorbar
- **ax** (*GeoAxes, optional*) – axes into which the bias is supposed to be plotted
- **ax_cbar** (*plt.Axes, optional*) – axes for colorbar
- **cbar_outline_visible** (*bool*) – if False, borders of colorbar are removed
- **cbar_orientation** (*str*) – e.g. 'vertical', defaults to 'vertical'
- ****kwargs** – keyword args passed to *init_map()*

Return type
GeoAxes

pyaerocom.plot.mapping.**set_map_ticks**(*ax*, *xticks=None*, *yticks=None*)

Set or update ticks in instance of GeoAxes object (cartopy)

Parameters

- **ax** (*cartopy.GeoAxes*) – map axes instance
- **xticks** (*iterable, optional*) – ticks of x-axis (longitudes)
- **yticks** (*iterable, optional*) – ticks of y-axis (latitudes)

Returns
modified axes instance

Return type
cartopy.GeoAxes

4.20.2 Plotting coordinates on maps

pyaerocom.plot.plotcoordinates.**plot_coordinates**(*lons*, *lats*, *xlim=None*, *ylim=None*, *label=None*, *legend=True*, *color=None*, *marker=None*, *markersize=8*, *ax=None*, ***kwargs*)

Plot input coordinates on a map

lons
[ndarray] array of longitude coordinates (can also be list or tuple)

lats
[ndarray] array of latitude coordinates (can also be list or tuple)

xlim
[tuple] longitude range

ylim
[tuple] latitude range

label
[str, optional] label of data

legend
[bool] whether or not to display a legend, defaults to True.

color
[str, optional] color of markers, defaults to red

marker
[str, optional] marker shape, defaults to 'o'

markersize
[int] size of markers

ax
[GeoAxes] axes instance to be plotted into

****kwargs**
additional keyword args passed on to `init_map()`

Return type
GeoAxes

4.20.3 Scatter plots

This module contains scatter plot routines for Aerocom data.

pyaerocom.plot.plotscatter.**plot_scatter**(*x_vals*, *y_vals*, ***kwargs*)

Scatter plot

Currently a wrapper for high-level method plot_scatter_aerocom (same module, see there for details)

pyaerocom.plot.plotscatter.**plot_scatter_aerocom**(*x_vals*, *y_vals*, *var_name=None*, *var_name_ref=None*, *x_name=None*, *y_name=None*, *start=None*, *stop=None*, *ts_type=None*, *unit=None*, *stations_ok=None*, *filter_name=None*, *lowlim_stats=None*, *highlim_stats=None*, *loglog=None*, *ax=None*, *figsize=None*, *fontsize_base=11*, *fontsize_annot=None*, *marker=None*, *color=None*, *alpha=0.5*, ***kwargs*)

Method that performs a scatter plot of data in AEROCOM format

Parameters

- **y_vals** (*ndarray*) – 1D array (or list) of model data points (y-axis)
- **x_vals** (*ndarray*) – 1D array (or list) of observation data points (x-axis)
- **var_name** (str, optional) – name of variable that is plotted
- **var_name_ref** (str, optional) – name of variable of reference data
- **x_name** (str, optional) – Name of observation network
- **y_name** (str, optional) – Name / ID of model

- **start** (str or :obj`datetime` or similar) – start time of data
- **stop** (str or :obj`datetime` or similar) – stop time of data
- **ts_type** (*str*) – frequency of data
- **unit** (*str, optional*) – unit of data
- **stations_ok** (*int, optional*) – number of stations from which data were generated
- **filter_name** (*str, optional*) – name of filter
- **lowlim_stats** (*float, optional*) – lower value considered for statistical parameters
- **highlim_stats** (*float, optional*) – upper value considered for statistical parameters
- **loglog** (*bool, optional*) – plot log log scale, if None, pyaerocom default is used
- **ax** (*Axes*) – axes into which the data are to be plotted
- **figsize** (*tuple*) – size of figure (if new figure is created, ie ax is None)
- **fontsize_base** (*int*) – basic fontsize, defaults to 11
- **fontsize_annot** (*int, optional*) – fontsize used for annotations
- **marker** (*str, optional*) – marker used for data, if None, '+' is used
- **color** (*str, optional*) – color of markers, default to 'k'
- **alpha** (*float, optional*) – transparency of markers (does not apply to all marker types), defaults to 0.5.
- ****kwargs** – additional keyword args passed to `ax.plot()`

Returns
plot axes

Return type
matplotlib.axes.Axes

4.20.4 Heatmap plots

pyaerocom.plot.heatmaps.**df_to_heatmap**(*df*, *cmap=None*, *center=None*, *low=0.3*, *high=0.3*, *vmin=None*, *vmax=None*, *color_rowwise=False*, *normalise_rows=False*, *normalise_rows_how=None*, *normalise_rows_col=None*, *norm_ref=None*, *sub_norm_before_div=True*, *annot=True*, *num_digits=None*, *ax=None*, *figsize=(12, 12)*, *cbar=False*, *cbar_label=None*, *cbar_labelsize=None*, *xticklabels=None*, *xtick_rot=45*, *yticklabels=None*, *ytick_rot=45*, *xlabel=None*, *ylabel=None*, *title=None*, *labelsize=12*, *annot_fontsize=None*, *annot_fmt_rowwise=False*, *annot_fmt_exceed=None*, *annot_fmt_rows=None*, *cbar_ax=None*, *cbar_kws=None*, ***kwargs*)

Plot a pandas dataframe as heatmap

Parameters

- **df** (*DataFrame*) – table data
- **cmap** (*str, optional*) – string specifying colormap to be used
- **center** (*float, optional*) – value that is mapped to center colour of colormap (e.g. 0)

- **low** (*float, optional*) – Extends lower range of the table values so that when mapped to the colormap, it's entire range isn't used. E.g. 0.3 roughly corresponds to colormap crop of 30% at the lower end.
- **high** (*float, optional*) – Extends upper range of the table values so that when mapped to the colormap, it's entire range isn't used. E.g. 0.3 roughly corresponds to colormap crop of 30% at the upper end.
- **vmin** (*float, optional*) – lower end of value range to be plotted. If specified, input arg *low* will be ignored.
- **vmax** (*float, optional*) – upper end of value range to be plotted. If specified, input arg *low* will be ignored.
- **color_rowwise** (*bool, optional*) – if True, the color mapping is applied row by row, else, for the whole table. Defaults to False.
- **normalise_rows** (*bool, optional*) – if True, the table is normalised in a rowwise manner either using the mean value in each row (if argument `normalise_rows_col` is unspecified) or using the value in a specified column. Defaults to False.
- **normalise_rows_how** (*str, optional*) – aggregation string for row normalisation. Choose from mean or median. Only relevant if input arg `normalise_rows==True`.
- **normalise_rows_col** (*int, optional*) – if provided and if arg. `normalise_rows==True`, then the corresponding table column is used for normalisation rather than the mean value of each row.
- **norm_ref** (*float or ndarray, optional*) – reference value(s) used for rowwise normalisation. Only relevant if normalise_rows is True. If specified, normalise_rows_how and normalise_rows_col will be ignored.
- **sub_norm_before_div** (*bool, optional*) – if True, the rowwise normilisation is applied by subtracting the normalisation value for each row before dividing by it. This can be useful to visualise positive or negative departures from the mean or median.
- **annot** (*bool or list or ndarray, optional*) – if True, the table values are printed into the heatmap. Defaults to True, in which case the values are computed based on the table content. If list or ndarray, the shape needs to be the same as input table shape (no of rows and cols), in which case the values of that 2D frame are used.
- **num_digits** (*int, optional*) – number of digits printed in heatmap annotation.
- **ax** (*axes, optional*) – matplotlib axes instance used for plotting, if None, an axes will be created
- **figsize** (*tuple, optional*) – size of figure for plot
- **cbar** (*bool, optional*) – if True, a colorbar is included
- **cbar_label** (*str, optional*) – label of colorbar (if colorbar is included, see cbar option)
- **cbar_labelsize** (*int, optional*) – size of colorbar label
- **xticklabels** (*list, optional*) – List of x axis labels.
- **xtick_rot** (*int, optional*) – rotation of x axis labels, defaults to 45 degrees.
- **yticklabels** (*list, optional*) – List of string labels.
- **ytick_rot** (*int, optional*) – rotation of y axis labels, defaults to 45 degrees.
- **xlabel** (*str, optional*) – x axis label
- **ylabel** (*str, optional*) – y axis label

- **title** (*str, optional*) – title of heatmap
- **labelsize** (*int, optional*) – fontsize of labels, default to 12
- **annot_fontsize** (*int, optional*) – fontsize of annotated text.
- **annot_fmt_rowwise** (*bool*) – rowwise formatting of annotation values, based on row value ranges. Defaults to False.
- **annot_fmt_exceed** (*list, optional*) – how to format annotated values that exceed a certain threshold. The list contains 2 entries, 1. the threshold values, 2. how values exceeding this thrshold should be formatted. This parameter is only considered if *annot_fmt_rowwise* is True. See also `_format_annot_heatmap()`.
- **annot_fmt_rows** (*list*) – annotation formatting strings for each row of the input table. This parameter is only considered if *annot_fmt_rowwise* is True. See also `_format_annot_heatmap()`.
- **cbar_ax** (*Axes, optional*) – axes instance for colorbar, parsed to `seaborn.heatmap()`.
- **cbar_kws** (*dict, optional*) – keywords for colorbar formatting, , parsed to `seaborn.heatmap()`.
- ****kwargs** – further keyword args parsed to `seaborn.heatmap()`

Raises

ValueError – if input *annot* is list or ndarray and has a different shape than the input *df*.

Returns

- *Axes* – plot axes instance
- *list or None* – annotation information for rows

4.20.5 Colors schemes

class pyaerocom.plot.config.**ColorTheme**(*name='dark'*, *cmap_map=None*, *color_coastline=None*, *cmap_map_div=None*, *cmap_map_div_shifted=True*)

Pyaerocom class specifiying plotting color theme

name

name of color theme (e.g. “light” or “dark”)

Type

str

cmap_map

name of colormap or colormap for map plotting

Type

str

color_coastline

coastline color for map plotting

Type

str

cmap_map_div

name of diverging colormap (used in map plots when plotted value range crosses 0)

Type
str

cmap_map_div_shifted

boolean specifying whether center of diverging colormaps for map plots is supposed to be shifted to 0

Type
bool

Example

Load default color theme >>> theme = ColorTheme(name="dark") >>> print(theme) pyaerocom ColorTheme name : dark cmap_map : viridis color_coastline : #e6e6e6

from_dict(*info_dict*)

Import theme information from dictionary

info_dict
[dict] dictionary containing theme settings

load_default(*theme_name='dark'*)

Load default color theme

Parameters
theme_name (*str*) – name of default theme

Raises
ValueError – if theme_name is not a valid default theme

to_dict()

Convert this object into dictionary

Returns
dictionary representation of this object

Return type
dict

pyaerocom.plot.config.**get_color_theme**(*theme_name='dark'*)

4.20.6 Plot helper functions

pyaerocom.plot.helpers.**calc_figsize**(*lon_range*, *lat_range*, *figh=8*)

Calculate figure size based on data

The required figure width is computed based on the input height and the aspect ratio of the longitude and latitude arrays

Parameters

- **lon_range** (*tuple*) – 2-element tuple specifying longitude range (may also be list or array)
- **lat_range** (*tuple*) – 2-element tuple specifying latitude range (may also be list or array)
- **figh** (*int*) – figure height in inches
- **add_cbar** (*bool*) – if True, the width is adapted accordingly

Returns
2-element tuple containing figure width and height

Return type
tuple

pyaerocom.plot.helpers.**calc_pseudolog_cmaplevels**(*vmin*, *vmax*, *add_zero=False*)

Initiate pseudo-log discrete colormap levels

Parameters

- **vmin** (*float*) – lower end of colormap (e.g. minimum value of data)
- **vmax** (*float*) – upper value of colormap (e.g. maximum value of data)
- **add_zero** (*bool*) – if True, the lower bound is set to 0 (irrelevant if vmin is 0).

Returns
list containing boundary array for discrete colormap (e.g. using BoundaryNorm)

Return type
list

Example

```
>>> vmin, vmax = 0.02, 0.75
>>> vals = calc_pseudolog_cmaplevels(vmin, vmax, num_per_mag=10, add_zero=True)
>>> for val in vals: print("%.4f" %val)
0.0000
0.0100
0.0126
0.0158
0.0200
0.0251
0.0316
0.0398
0.0501
0.0631
0.0794
0.1000
```

pyaerocom.plot.helpers.**custom_mpl**(*mpl_rcparams=None*, *default_large=True*, ***kwargs*)

Custom matplotlib settings

pyaerocom.plot.helpers.**get_cmap_levels_auto**(*vmin*, *vmax*, *num_per_mag=10*)

Initiate pseudo-log discrete colormap levels

Note: This is a beta version and aims to

Parameters

- **vmin** (*float*) – lower end of colormap (e.g. minimum value of data)
- **vmax** (*float*) – upper value of colormap (e.g. maximum value of data)

pyaerocom.plot.helpers.**get_cmap_ticks_auto**(*lvls*, *num_per_mag=3*)

Compute cmap ticks based on cmap levels

The cmap levels may be computed automatically using *get_cmap_levels_auto()*.

Parameters

- **lvls** (*list*) – list containing colormap levels
- **num_per_mag** (*int*) – desired number of ticks per magnitude

pyaerocom.plot.helpers.**projection_from_str**(*projection_str='PlateCarree'*)

Return instance of cartopy projection class based on string ID

4.21 Configuration and global constants

4.21.1 Basic configuration class

Will be initiated on input and is accessible via *pyaerocom.const*.

class pyaerocom.config.**Config**(*config_file=None*, *try_infer_environment=True*)

Class containing relevant paths for read and write routines

A loaded instance of this class is created on import of pyaerocom and can be accessed via *pyaerocom.const*.

TODO: provide more information

AEOLUS_NAME = 'AeolusL2A'

AERONET_INV_V2L15_ALL_POINTS_NAME = 'AeronetInvV2Lev1.5.AP'

AERONET_INV_V2L15_DAILY_NAME = 'AeronetInvV2Lev1.5.daily'

AERONET_INV_V2L2_ALL_POINTS_NAME = 'AeronetInvV2Lev2.AP'

AERONET_INV_V2L2_DAILY_NAME = 'AeronetInvV2Lev2.daily'

AERONET_INV_V3L15_DAILY_NAME = 'AeronetInvV3Lev1.5.daily'

Aeronet V3 inversions

AERONET_INV_V3L2_DAILY_NAME = 'AeronetInvV3Lev2.daily'

AERONET_SUN_V2L15_AOD_ALL_POINTS_NAME = 'AeronetSun_2.0_NRT'

AERONET_SUN_V2L15_AOD_DAILY_NAME = 'AeronetSunV2Lev1.5.daily'

Aeronet Sun V2 access names

AERONET_SUN_V2L2_AOD_ALL_POINTS_NAME = 'AeronetSunV2Lev2.AP'

AERONET_SUN_V2L2_AOD_DAILY_NAME = 'AeronetSunV2Lev2.daily'

AERONET_SUN_V2L2_SDA_ALL_POINTS_NAME = 'AeronetSDAV2Lev2.AP'

AERONET_SUN_V2L2_SDA_DAILY_NAME = 'AeronetSDAV2Lev2.daily'

Aeronet SDA V2 access names

AERONET_SUN_V3L15_AOD_ALL_POINTS_NAME = 'AeronetSunV3Lev1.5.AP'

AERONET_SUN_V3L15_AOD_DAILY_NAME = 'AeronetSunV3Lev1.5.daily'

Aeronet Sun V3 access names

AERONET_SUN_V3L15_SDA_ALL_POINTS_NAME = 'AeronetSDAV3Lev1.5.AP'

AERONET_SUN_V3L15_SDA_DAILY_NAME = 'AeronetSDAV3Lev1.5.daily'

Aeronet SDA V3 access names

AERONET_SUN_V3L2_AOD_ALL_POINTS_NAME = 'AeronetSunV3Lev2.AP'

AERONET_SUN_V3L2_AOD_DAILY_NAME = 'AeronetSunV3Lev2.daily'

AERONET_SUN_V3L2_SDA_ALL_POINTS_NAME = 'AeronetSDAV3Lev2.AP'

AERONET_SUN_V3L2_SDA_DAILY_NAME = 'AeronetSDAV3Lev2.daily'

property ALL_DATABASE_IDS

ID's of available database configurations

property CACHEDIR

Cache directory for UngriddedData objects

property CACHING

Activate writing of and reading from cache files

CAMS2_83_NRT_NAME = 'CAMS2_83.NRT'

CLIM_FREQ = 'daily'

CLIM_MIN_COUNT = {'daily': 30, 'monthly': 5}

CLIM_RESAMPLE_HOW = 'mean'

CLIM_START = 2005

CLIM_STOP = 2015

property COLOCATEDDATADIR

Directory for accessing and saving colocated data objects

property COORDINFO

Instance of `VarCollection` containing coordinate info

property DATA_SEARCH_DIRS

Directories which pyaerocom will consider for data access

Note: This corresponds to directories considered for searching gridded data (e.g. models and level 3 satellite products). Please see `OBSLOCS_UNGRIDDED` for available data directories for reading of ungridded data.

Returns

list of directories

Return type

list

DEFAULT_REG_FILTER = 'ALL-wMOUNTAINS'

DEFAULT_VERT_GRID_DEF = {'lower': 0, 'step': 250, 'upper': 15000}

Information specifying default vertical grid for post processing of profile data. The values are in units of m.

DMS_AMS_CVO_NAME = 'DMS_AMS_CVO'

DMS

DONOTCACHEFILE = None

property DOWNLOAD_DATADIR

Directory where data is downloaded into

EARLINET_NAME = 'EARLINET'

Earlinet access name;

EBAS_DB_LOCAL_CACHE = True

boolean specifying wheter EBAS DB is copied to local cache for faster access, defaults to True

property EBAS_FLAGS_FILE

Location of CSV file specifying meaning of EBAS flags

EBAS_MULTICOLUMN_NAME = 'EBASMC'

EBAS name

EEA_NAME = 'EEAAQeRep'

EEA nmea

EEA_NRT_NAME = 'EEAAQeRep.NRT'

EEA.NRT name

EEA_V2_NAME = 'EEAAQeRep.v2'

EEAV2 name

property ERA5_SURFTEMP_FILE

ERA5_SURFTEMP_FILENAME = 'era5.msl.t2m.201001-201012.nc'

property ETOPO1_AVAILABLE

Boolean specifying if access to ETOPO1 dataset is provided

Return type
bool

property FILTERMASKKDIR

GAWTADSUBSETAASETAL_NAME = 'GAWTADsubsetAasEtAl'

GAW TAD subset aas et al paper

GRID_IO

Settings for reading and writing of gridded data

property HOMEDIR

Home directory of user

HTAP_REGIONS = ['PAN', 'EAS', 'NAF', 'MDE', 'LAND', 'SAS', 'SPO', 'OCN', 'SEA', 'RBU', 'EEUROPE', 'NAM', 'WEUROPE', 'SAF', 'USA', 'SAM', 'EUR', 'NPO', 'MCA']

ICOS_NAME = 'ICOS'

ICOS name

ICPFORESTS_NAME = 'ICPFORESTS'

ICP Forests

property LOCAL_TMP_DIR

Local TEMP directory

property LOGFILESDIR

Directory where logfiles are stored

MAX_YEAR = 20000

Highest possible year in data

MEP_NAME = 'MEP'

MEP name

MIN_YEAR = 0

Lowest possible year in data

OBS_ALLOW_ALT_WAVELENGTHS = True

This boolean can be used to enable / disable the former (i.e. use available wavelengths of variable in a certain range around variable wavelength).

property OBS_IDS_UNGRIDDED

List of all data IDs of supported ungridded observations

OBS_MIN_NUM_RESAMPLE = {'daily': {'hourly': 6}, 'hourly': {'minutely': 15}, 'monthly': {'daily': 7}, 'yearly': {'monthly': 3}}

Time resample strategies for certain cominations, first level refers to TO, second to FROM and values are minimum number of observations

OBS_WAVELENGTH_TOL_NM = 10.0

Wavelength tolerance for observations imports

OLD_AEROCOM_REGIONS = ['ALL', 'ASIA', 'AUSTRALIA', 'CHINA', 'EUROPE', 'INDIA', 'NAFRICA', 'SAFRICA', 'SAMERICA', 'NAMERICA']

property OUTPUTDIR

Default output directory

REVISION_FILE = 'Revision.txt'

Name of the file containing the revision string of an obs data network

RH_MAX_PERCENT_DRY = 40

maximum allowed RH to be considered dry

RM_CACHE_OUTDATED = True

property ROOTDIR

Local root directory

SENTINEL5P_NAME = 'Sentinel5P'

SERVER_CHECK_TIMEOUT = 1

timeout to check if one of the supported server locations can be accessed

STANDARD_COORD_NAMES = ['latitude', 'longitude', 'altitude']

standard names for coordinates

URL_HTAP_MASKS = 'https://pyaerocom.met.no/pyaerocom-suppl/htap_masks/'

property VARS

Instance of class VarCollection (for default variable information)

property VAR_PARAM

Deprecated name, please use *VARS* instead

add_data_search_dir(**dirs*)

Add data search directories for database browsing

add_ungridded_obs(*obs_id*, *data_dir*, *reader=None*, *check_read=False*)

Add a network to the data search structure

Parameters

- **obs_id** (*str*) – name of network. E.g. MY_OBS or EBASMC
- **data_dir** (*str*) – directory where data files are stored
- **reader** (*pyaerocom.io.ReadUngriddedBase, optional*) – reading class used to import these data. If *obs_id* is known (e.g. EBASMC) this is not needed.

Raises

- **AttributeError** – if the network name is already reserved in OBSLOCS_UNGRIDDED
- **ValueError** – if the data directory does not exist

add_ungridded_post_dataset(*obs_id*, *obs_vars*, *obs_aux_requires*, *obs_merge_how*, *obs_aux_funs=None*, *obs_aux_units=None*, ***kwargs*)

Register new ungridded dataset

Other than *add_ungridded_obs()*, this method adds required logic for a "virtual" ungridded observation datasets, that is, a dataset that can only be computed from other ungridded datasets but not read from disk.

If all input parameters are okay, the new dataset will be registered in OBS_UNGRIDDED_POST and will then be accessible for import in ungridded reading factory class pyaerocom.io.ReadUngridded.

Parameters

- **obs_id** (*str*) – Name of new dataset.
- **obs_vars** (*str or list*) – variables supported by this dataset.
- **obs_aux_requires** (*dict*) – dicionary specifying required datasets and variables for each variable supported by the auxiliary dataset.
- **obs_merge_how** (*str or dict*) – info on how to derive each of the supported coordinates (e.g. eval, combine). For valid input args see *pyaerocom.combine_vardata_ungridded*. If value is string, then the same method is used for all variables.
- **obs_aux_funs** (*dict, optional*) – dictionary specifying computation methods for auxiliary variables that are supposed to be retrieved via *obs_merge_how='eval'*. Keys are variable names, values are respective computation methods (which need to be strings as they will be evaluated via

  pandas.DataFrame.eval() in *pyaerocom.combine_vardata_ungridded*). This input is

  optional, but mandatory if any of the *obs_vars* is supposed to be retrieved via *merge_how='eval'*.
- **obs_aux_units** (*dict, optional*) – output units of auxiliary variables (only needed for varibales that are derived via *merge_how='eval'*)

- ****kwargs** – additional keyword arguments (unused, but serves the purpose to allow for parsing info from dictionaries and classes that contain additional attributes than the ones needed here).

Raises
ValueError – if input obs_id is already reserved

Return type
None.

property cache_basedir

Base directory for caching

The actual files are cached in user subdirectory, cf *CACHEDIR*

property ebas_flag_info

Information about EBAS flags

Note: Is loaded upon request -> cf. *pyaerocom.io.ebas_nasa_ames.EbasFlagCol.FLAG_INFO*

Dictionary containing 3 dictionaries (keys: `valid, values, info`) that contain information about validity of each flag (`valid`), their actual values (`values`, e.g. V, M, I)

---

property has_access_lustre

Boolean specifying whether MetNO AeroCom server is accessible

property has_access_users_database

infer_basedir_and_config()

Boolean specifying whether the lustre database can be accessed

make_default_vert_grid()

Makes default vertical grid for resampling of profile data

path = PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/pyaerocom/checkouts/latest/pyaerocom/data/coords.ini')

read_config(*config_file*, *basedir=None*, *init_obslocs_ungridded=False*, *init_data_search_dirs=False*)

Import paths from one of the config ini files

Parameters

- **config_file** (*str*) – file location of config ini file
- **basedir** (*str, optional*) – Base directory to be used for relative model and obs dirs specified via BASEDIR in config file. If None, then the BASEDIR value in the config file is used. The default is None.
- **init_obslocs_ungridded** (*bool, optional*) – If True, OBSLOCS_UNGRIDDED will be re-instantiated (i.e. all currently set obs locations will be deleted). The default is False.
- **init_data_search_dirs** (*bool, optional*) – If True, *DATA_SEARCH_DIRS* will be re-instantiated (i.e. all currently set data search directories will be deleted). The default is False.

Raises
FileNotFoundError – If input config file is not a file or does not exist.

Return type
None.

reload(*keep_basedirs=True*)

Reload config file (for details see *read_config()*)

short_str()

Deprecated method

property user

User ID

4.21.2 Config defaults related to gridded data

class pyaerocom.grid_io.**GridIO**(***kwargs*)

Global I/O settings for gridded data

This class includes options related to the import of gridded data. This includes both options related to file search as well as preprocessing options.

FILE_TYPE

file type of data files. Defaults to .nc

Type
str

TS_TYPES

list of strings specifying temporal resolution options encrypted in file names.

Type
list

PERFORM_FMT_CHECKS

perform formatting checks when reading netcdf data, using metadata encoded in filenames (requires that NetCDF file follows a registered naming convention)

Type
bool

DEL_TIME_BOUNDS

if True, preexisting bounds on time are deleted when grid data is loaded. Else, nothing is done. Aerocom default is True

Type
bool

SHIFT_LONS

if True, longitudes are shifted to -180 <= lon <= 180 when data is loaded (in case they are defined 0 <= lon <= 360. Aerocom default is True.

Type
bool

CHECK_TIME_FILENAME

the times stored in NetCDF files may be wrong or not stored according to the CF conventions. If True, the times are checked and if *CORRECT_TIME_FILENAME*, corrected for on data import based what is encrypted in the file name. In case of Aerocom models, it is ensured that the filename contains both the year and the temporal resolution in the filenames (for details see `pyaerocom.io.FileConventionRead`). Aerocom default is True

Type
bool

CORRECT_TIME_FILENAME

if True and time dimension in data is found to be different from filename, it is attempted to be corrected

Type
bool

EQUALISE_METADATA

if True (and if metadata varies between different NetCDF files that are supposed to be merged in time), the metadata in all loaded objects is unified based on the metadata of the first grid (otherwise, concatenating them in time might not work using the Iris interface). This might need to be reviewed and should be used with care if specific metadata aspects of individual files need to be accessed. Aerocom default is True

Type
bool

USE_FILECONVENTION

if True, file names are strictly required to follow one of the file naming conventions that can be specified in the file file_conventions.ini. Aerocom default is True.

Type
bool

INCLUDE_SUBDIRS

if True, search for files is expanded to all subdirecories included in data directory. Aerocom default is False.

Type
bool

INFER_SURFACE_LEVEL

if True then surface level for 4D gridded data is inferred automatically when necessary (e.g. when extracting surface time series from 4D gridded data object that does not contain sufficient information about vertical dimension)

Type
bool

UNITS_ALIASES = {'/m': 'm-1'}

from_dict(*dictionary=None*, ***settings*)

Import settings from dictionary

load_aerocom_default()

load_default()

to_dict()

Convert object to dictionary

Returns
settings dictionary

Return type
dict

4.21.3 Config details related to observations

Settings and helper methods / classes for I/O of obervation data

Note: Some settings like paths etc can be found in `pyaerocom.config.py`

class pyaerocom.obs_io.**AuxInfoUngridded**(*data_id*, *vars_supported*, *aux_requires*, *aux_merge_how*, *aux_funs=None*, *aux_units=None*)

MAX_VARS_PER_METHOD = 2

check_status()

Check if specifications are correct and consistent

Raises

- **ValueError** – If one of the class attributes is invalid
- **NotImplementedError** – If computation method contains more than 2 variables / datasets

to_dict()

Dictionary representation of this object

Ignores any potential private attributes.

pyaerocom.obs_io.**OBS_ALLOW_ALT_WAVELENGTHS = True**

This boolean can be used to enable / disable the former (i.e. use available wavelengths of variable in a certain range around variable wavelength).

pyaerocom.obs_io.**OBS_WAVELENGTH_TOL_NM = 10.0**

Wavelength tolerance for observations if data for required wavelength is not available

class pyaerocom.obs_io.**ObsVarCombi**(*obs_id*, *var_name*)

4.21.4 Molar masses and related helpers

exception pyaerocom.molmasses.**UnkownSpeciesError**

pyaerocom.molmasses.**get_mmr_to_vmr_fac**(*var_name*)

Get conversion factor for MMR -> VMR conversion for input variable

Note: Assumes dry air molar mass

Parameters
var_name (*str*) – Name of variable to be converted

Returns
multiplication factor to convert MMR -> VMR

Return type
float

pyaerocom.molmasses.**get_molmass**(*var_name*)

Get molar mass for input variable

Parameters
var_name (*str*) – pyaerocom variable name (cf. variables.ini) or name of species

Returns
molar mass of species in units of g/mol

Return type
float

pyaerocom.molmasses.**get_species**(*var_name*)

Get species name from variable name

Parameters
var_name (*str*) – pyaerocom variable name (cf. variables.ini)

Raises
UnkownSpeciesError – if species cannot be inferred

Returns
name of species

Return type
str

4.22 Access to minimal test dataset

4.23 Low-level helper classes and functions

Small helper utility functions for pyaerocom

class pyaerocom._lowlevel_helpers.**AsciiFileLoc**(*default=None*, *assert_exists=False*, *auto_create=False*, *tooltip=None*, *logger=None*)

create(*value*)

class pyaerocom._lowlevel_helpers.**BrowseDict**(**args*, ***kwargs*)

Dictionary-like object with getattr and setattr options

Extended dictionary that supports dynamic value generation (i.e. if an assigned value is callable, it will be executed on demand).

ADD_GLOB = []

FORBIDDEN_KEYS = []

IGNORE_JSON = []

Keys to be ignored when converting to json

MAXLEN_KEYS = 100.0

SETTER_CONVERT = {}

import_from(*other*) → None

Import key value pairs from other object

Other than `update()` this method will silently ignore input keys that are not contained in this object.

Parameters
other (*dict or BrowseDict*) – other dict-like object containing content to be updated.

Raises
ValueError – If input is inalid type.

Return type
None

items() → a set-like object providing a view on D's items

json_repr() → dict

Convert object to serializable json dict

Returns
content of class

Return type
dict

keys() → a set-like object providing a view on D's keys

pretty_str()

to_dict()

values() → an object providing a view on D's values

class pyaerocom._lowlevel_helpers.**ConstrainedContainer**(**args*, ***kwargs*)

Restrictive dict-like class with fixed keys

This class enables to create dict-like objects that have a fixed set of keys and value types (once assigned). Optional values may be instantiated as None, in which case the first time instantiation definecs its type.

Note: The limitations for assignments are only restricted to setitem operations and attr assignment via "." works like in every other class.

Example

class MyContainer(ConstrainedContainer):

def __init__(self):
self.val1 = 1 self.val2 = 2 self.option = None

```
>>> mc = MyContainer()
>>> mc['option'] = 42
```

CRASH_ON_INVALID = True

class pyaerocom._lowlevel_helpers.**DictStrKeysListVals**

validate(*val: dict*)

class pyaerocom._lowlevel_helpers.**DictType**

validate(*val*)

class pyaerocom._lowlevel_helpers.**DirLoc**(*default=None*, *assert_exists=False*, *auto_create=False*, *tooltip=None*, *logger=None*)

create(*value*)

class pyaerocom._lowlevel_helpers.**EitherOf**(*allowed: list*)

validate(*val*)

class pyaerocom._lowlevel_helpers.**FlexList**

list that can be instantated via input str, tuple or list or None

validate(*val*)

class pyaerocom._lowlevel_helpers.**ListOfStrings**

validate(*val*)

class pyaerocom._lowlevel_helpers.**Loc**(*default=None*, *assert_exists=False*, *auto_create=False*, *tooltip=None*, *logger=None*)

Abstract descriptor representing a path location

Descriptor??? See here: https://docs.python.org/3/howto/descriptor.html#complete-practical-example

---

Note:

- Child classes need to implement *create()*
- value is allowed to be *None* in which case no checks are performed

---

abstract create(*value*)

validate(*value*)

class pyaerocom._lowlevel_helpers.**NestedContainer**(**args*, ***kwargs*)

keys_unnested() → list

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

class pyaerocom._lowlevel_helpers.**StrType**

validate(*val*)

class pyaerocom._lowlevel_helpers.**StrWithDefault**(*default: str*)

validate(*val*)

class pyaerocom._lowlevel_helpers.**TypeValidator**(*type*)

validate(*val*)

class pyaerocom._lowlevel_helpers.**Validator**

abstract validate(*val*)

pyaerocom._lowlevel_helpers.**check_dir_access**(*path*)

Uses multiprocessing approach to check if location can be accessed

Parameters
loc (*str*) – path that is supposed to be checked

Returns
True, if location is accessible, else False

Return type
bool

pyaerocom._lowlevel_helpers.**check_dirs_exist**(**dirs*, ***add_dirs*)

pyaerocom._lowlevel_helpers.**check_write_access**(*path*)

Check if input location provides write access

Parameters
path (*str*) – directory to be tested

pyaerocom._lowlevel_helpers.**chk_make_subdir**(*base*, *name*)

Check if sub-directory exists in parent directory

pyaerocom._lowlevel_helpers.**dict_to_str**(*dictionary*, *indent=0*, *ignore_null=False*)

Custom function to convert dictionary into string (e.g. for print)

Parameters

- **dictionary** (*dict*) – the dictionary
- **indent** (*int*) – indent of dictionary content
- **ignore_null** (*bool*) – if True, None entries in dictionary are ignored

Returns
the modified input string

Return type
str

pyaerocom._lowlevel_helpers.**invalid_input_err_str**(*argname*, *argval*, *argopts*)

Just a small helper to format an input error string for functions

Parameters

- **argname** (*str*) – name of input argument
- **argval** – (invalid) value of input argument
- **argopts** – possible input args for arg

Returns
formatted string that can be parsed to an Exception

Return type
str

pyaerocom._lowlevel_helpers.**list_to_shortstr**(*lst*, *indent=0*)

Custom function to convert a list into a short string representation

pyaerocom._lowlevel_helpers.**merge_dicts**(*dict1*, *dict2*, *discard_failing=True*)

Merge two dictionaries

Parameters

- **dict1** (*dict*) – first dictionary
- **dict2** (*dict*) – second dictionary
- **discard_failing** (*bool*) – if True, any key, value pair that cannot be merged from the 2nd into the first will be skipped, which means, the value of the output dict for that key will be the one of the first input dict. All keys that could not be merged can be accessed via key 'merge_failed' in output dict. If False, any Exceptions that may occur will be raised.

Returns
merged dictionary

Return type
dict

pyaerocom._lowlevel_helpers.**sort_dict_by_name**(*d*, *pref_list: list | None = None*) → dict

Sort entries of input dictionary by their names and return ordered

Parameters

- **d** (*dict*) – input dictionary
- **pref_list** (*list, optional*) – preferred order of items (may be subset of keys in input dict)

Returns
sorted and ordered dictionary

Return type
dict

pyaerocom._lowlevel_helpers.**str_underline**(*title: str*, *indent: int = 0*)

Create underlined string

4.24 Custom exceptions

Module containing pyaerocom custom exceptions

exception pyaerocom.exceptions.**AeronetReadError**

exception pyaerocom.exceptions.**CacheReadError**

exception pyaerocom.exceptions.**CacheWriteError**

exception pyaerocom.exceptions.**CachingError**

exception pyaerocom.exceptions.**ColocationError**

exception pyaerocom.exceptions.**ColocationSetupError**

exception pyaerocom.exceptions.**CoordinateError**

exception pyaerocom.exceptions.**CoordinateNameError**

exception pyaerocom.exceptions.**DataCoverageError**

exception pyaerocom.exceptions.**DataDimensionError**

exception pyaerocom.exceptions.**DataExtractionError**

exception pyaerocom.exceptions.**DataIdError**

exception pyaerocom.exceptions.**DataQueryError**

exception pyaerocom.exceptions.**DataRetrievalError**

exception pyaerocom.exceptions.**DataSearchError**

exception pyaerocom.exceptions.**DataSourceError**

exception pyaerocom.exceptions.**DataUnitError**

exception pyaerocom.exceptions.**DeprecationError**

exception pyaerocom.exceptions.**DimensionOrderError**

exception pyaerocom.exceptions.**EEAv2FileError**

exception pyaerocom.exceptions.**EbasFileError**

exception pyaerocom.exceptions.**EntryNotAvailable**

exception pyaerocom.exceptions.**EvalEntryNameError**

exception pyaerocom.exceptions.**FileConventionError**

exception pyaerocom.exceptions.**InitialisationError**

exception pyaerocom.exceptions.**LongitudeConstraintError**

exception pyaerocom.exceptions.**MetaDataError**

exception pyaerocom.exceptions.**ModelVarNotAvailable**

exception pyaerocom.exceptions.**NasaAmesReadError**

exception pyaerocom.exceptions.**NetcdfError**

exception pyaerocom.exceptions.**NetworkNotImplemented**

exception pyaerocom.exceptions.**NetworkNotSupported**

exception pyaerocom.exceptions.**NotInFileError**

exception pyaerocom.exceptions.**ResamplingError**

exception pyaerocom.exceptions.**StationCoordinateError**

exception pyaerocom.exceptions.**StationNotFoundError**

exception pyaerocom.exceptions.**TemporalResolutionError**

exception pyaerocom.exceptions.**TemporalSamplingError**

exception pyaerocom.exceptions.**TimeMatchError**

exception pyaerocom.exceptions.**TimeZoneError**

exception pyaerocom.exceptions.**UnitConversionError**

exception pyaerocom.exceptions.**UnknownRegion**

exception pyaerocom.exceptions.**UnresolvableTimeDefinitionError**

Is raised if time definition in NetCDF file is wrong and cannot be corrected

exception pyaerocom.exceptions.**VarNotAvailableError**

exception pyaerocom.exceptions.**VariableDefinitionError**

exception pyaerocom.exceptions.**VariableNotFoundError**

CHAPTER

FIVE

API OF AEROVAL TOOLS

Documentation of the pyaerocom AeroVal API, for high level web processing tools.

5.1 Tools for AeroVal experiment setup

5.1.1 High level analysis setup for AeroVal experiment

class pyaerocom.aeroval.setupclasses.**CAMS2_83Setup**(*, *use_cams2_83: bool = False*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'use_cams2_83': FieldInfo(annotation=bool, required=False, default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class pyaerocom.aeroval.setupclasses.**EvalRunOptions**(*, *clear_existing_json: bool = True*, *only_json: bool = False*, *only_colocation: bool = False*, *only_model_maps: bool = False*, *obs_only: bool = False*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'clear_existing_json': FieldInfo(annotation=bool, required=False, default=True), 'obs_only': FieldInfo(annotation=bool, required=False, default=False), 'only_colocation': FieldInfo(annotation=bool, required=False, default=False), 'only_json': FieldInfo(annotation=bool, required=False, default=False), 'only_model_maps': FieldInfo(annotation=bool, required=False, default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

only_model_maps: bool

If True, process only maps (skip obs evaluation)

class pyaerocom.aeroval.setupclasses.**EvalSetup**(**, IGNORE_JSON: list[str] = ['_aux_funs'], ADD_GLOB: list[str] = ['io_aux_file'], io_aux_file: ~pathlib.Path | str = '', proj_id: str, exp_id: str, var_web_info: dict = {}, obs_cfg: ~pyaerocom.aeroval.collections.ObsCollection | dict = ObsCollection: {}, model_cfg: ~pyaerocom.aeroval.collections.ModelCollection | dict = ModelCollection: {}, **extra_data: ~typing.Any*)

Composite class representing a whole analysis setup

This represents the level at which json I/O happens for configuration setup files.

static from_json(*filepath: str*) → Self

Load configuration from json config file

get_model_entry(*model_name*) → dict

Get model entry configuration

Since the configuration files for experiments are in json format, they do not allow the storage of executable custom methods for model data reading. Instead, these can be specified in a python module that may be specified via `add_methods_file` and that contains a dictionary *FUNS* that maps the method names with the callable methods.

As a result, this means that, by default, custom read methods for individual models in *model_config* do not contain the callable methods but only the names. This method will take care of handling this and will return a dictionary where potential custom method strings have been converted to the corresponding callable methods.

Parameters
model_name (*str*) – name of model

Returns
Dictionary that specifies the model setup ready for the analysis

Return type
dict

property json_filename: str

cfg_<proj_id>_<exp_id>.json

Type
str

Type
Savename of config file

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {'cams2_83_cfg': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.CAMS2_83Setup'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'colocation_opts': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.colocation_auto.ColocationSetup'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'exp_info': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.ExperimentInfo'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'modelmaps_opts': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.ModelMapsSetup'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'path_manager': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.OutputPaths'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'processing_opts': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.EvalRunOptions'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'proj_info': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.ProjectInfo'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'statistics_opts': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.StatisticsSetup'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'time_cfg': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.TimeSetup'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True), 'webdisp_opts': ComputedFieldInfo(wrapped_property=<functools.cached_property object>, return_type=<class 'pyaerocom.aeroval.setupclasses.WebDisplaySetup'>, alias=None, alias_priority=None, title=None, description=None, deprecated=None, examples=None, json_schema_extra=None, repr=True)}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'allow', 'protected_namespaces': ()}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'ADD_GLOB': FieldInfo(annotation=list[str], required=False, default=['io_aux_file']), 'IGNORE_JSON': FieldInfo(annotation=list[str], required=False, default=['_aux_funs']), 'exp_id': FieldInfo(annotation=str, required=True), 'io_aux_file': FieldInfo(annotation=Union[Path, str], required=False, default='', metadata=['.py file containing additional read methods for modeldata']), 'model_cfg': FieldInfo(annotation=Union[ModelCollection, dict], required=False, default=ModelCollection: {}), 'obs_cfg': FieldInfo(annotation=Union[ObsCollection, dict], required=False, default=ObsCollection: {}), 'proj_id': FieldInfo(annotation=str, required=True), 'var_web_info': FieldInfo(annotation=dict, required=False, default={})}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

to_json(*outdir: str*, *ignore_nan: bool = True*, *indent: int = 3*) → None

Save configuration as JSON file

Parameters

- **outdir** (*str*) – directory where the config json file is supposed to be stored
- **ignore_nan** (*bool*) – set NaNs to Null when writing
- **indent** (*int*) – json indentation

class pyaerocom.aeroval.setupclasses.**ExperimentInfo**(*, *exp_id: str*, *exp_name: str = ''*, *exp_descr: str = ''*, *public: bool = False*, *exp_pi: str = 'docs'*, *pyaerocom_version: str = '0.18.dev0'*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'exp_descr': FieldInfo(annotation=str, required=False, default=''), 'exp_id': FieldInfo(annotation=str, required=True), 'exp_name': FieldInfo(annotation=str, required=False, default=''), 'exp_pi': FieldInfo(annotation=str, required=False, default='docs'), 'public': FieldInfo(annotation=bool, required=False, default=False), 'pyaerocom_version': FieldInfo(annotation=str, required=False, default='0.18.dev0')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class pyaerocom.aeroval.setupclasses.**ModelMapsSetup**(*, *maps_freq: Literal['monthly', 'yearly'] = 'monthly'*, *maps_res_deg: int = 5*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'maps_freq': FieldInfo(annotation=Literal['monthly', 'yearly'], required=False, default='monthly'), 'maps_res_deg': FieldInfo(annotation=int, required=False, default=5, metadata=[Gt(gt=0)])}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class pyaerocom.aeroval.setupclasses.**OutputPaths**(*, *JSON_SUBDIRS: list[str] = ['map', 'ts', 'ts/diurnal', 'scat', 'hm', 'hm/ts', 'contour', 'profiles']*, *json_basedir: Path | str = '/home/docs/MyPyaerocom/aeroval/data'*, *coldata_basedir: Path | str = '/home/docs/MyPyaerocom/aeroval/coldata'*, *ADD_GLOB: list[str] = ['coldata_basedir', 'json_basedir']*, *proj_id: str*, *exp_id: str*)

Setup class for output paths of json files and co-located data

This interface generates all paths required for an experiment.

proj_id

project ID

Type
str

exp_id

experiment ID

Type
str

json_basedir

Type
str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'ADD_GLOB': FieldInfo(annotation=list[str], required=False, default=['coldata_basedir', 'json_basedir']), 'JSON_SUBDIRS': FieldInfo(annotation=list[str], required=False, default=['map', 'ts', 'ts/diurnal', 'scat', 'hm', 'hm/ts', 'contour', 'profiles']), 'coldata_basedir': FieldInfo(annotation=Union[Path, str], required=False, default='/home/docs/MyPyaerocom/aeroval/coldata', validate_default=True), 'exp_id': FieldInfo(annotation=str, required=True), 'json_basedir': FieldInfo(annotation=Union[Path, str], required=False, default='/home/docs/MyPyaerocom/aeroval/data', validate_default=True), 'proj_id': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class pyaerocom.aeroval.setupclasses.**ProjectInfo**(*, *proj_id: str*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'proj_id': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class pyaerocom.aeroval.setupclasses.**StatisticsSetup**(*, *MIN_NUM: int = 1*, *weighted_stats: bool = True*, *annual_stats_constrained: bool = False*, *add_trends: bool = False*, *trends_min_yrs: int = 7*, *stats_tseries_base_freq: str | None = None*, *forecast_evaluation: bool = False*, *forecast_days: int = 4*, *use_fairmode: bool = False*, *use_diurnal: bool = False*, *obs_only_stats: bool = False*, *model_only_stats: bool = False*, *drop_stats: tuple[str, ...] = ()*, *stats_decimals: int | None = None*, *round_floats_precision: int | None = None*, ***extra_data: Any*)

Setup options for statistical calculations

weighted_stats

if True, statistics are calculated using area weights, this is only relevant for gridded / gridded evaluations.

Type
bool

annual_stats_constrained

if True, then only sites are considered that satisfy a potentially specified annual resampling constraint (see *pyaerocom.colocation_auto.ColocationSetup.min_num_obs*). E.g.

lets say you want to calculate statistics (bias, correlation, etc.) for monthly model / obs data for a given site and year. Lets further say, that there are only 8 valid months of data, and 4 months are missing, so

statistics will be calculated for that year based on 8 vs. 8 values. Now if *pyaerocom.colocation_auto.ColocationSetup.min_num_obs* is specified in way that requires e.g. at least 9 valid months to represent the whole year, then this station will not be considered in case *annual_stats_constrained* is True, else it will. Defaults to False.

Type
bool

stats_tseries_base_freq

The statistics Time Series display in AeroVal (under Overall Evaluation) is computed in intervals of a certain frequency, which is specified via `TimeSetup.main_freq` (defaults to monthly). That is, monthly colocated data is used as a basis to compute the statistics for each month (e.g. if you have 10 sites, then statistics will be computed based on 10 monthly values for each month of the timeseries, 1 value for each site). *stats_tseries_base_freq* may be specified in case a higher resolution is supposed to be used as a basis to compute the timeseries in the resolution specified by `TimeSetup.main_freq` (e.g. if daily is specified here, then for the above example 310 values would be used - 31 for each site - to compute the statistics for a given month (in this case, a month with 31 days, obviously).

Type
str, optional

drop_stats

tuple of strings with names of statistics (as determined by keys in aeroval.glob_defaults.py's statistics_defaults) to not compute. For example, setting drop_stats = ("mb", "mab"), results in json files in hm/ts with entries which do not contain the mean bias and mean absolute bias, but the other statistics are preserved.

Type
tuple, optional

stats_decimals

If provided, overwrites the decimals key in glod_defaults for the statistics, which has a deault of 3. Setting this higher of lower changes the number of decimals shown on the Aeroval webpage.

Type
int, optional

round_floats_precision

Sets the precision argument for the function *pyaerocom.aaeroval.json_utils:set_float_serialization_precision*

Type
int, optional

Parameters
kwargs – any of the supported attributes, e.g. *StatisticsSetup(annual_stats_constrained=True)*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'extra': 'allow', 'protected_namespaces': ()}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'MIN_NUM': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'add_trends': FieldInfo(annotation=bool, required=False, default=False), 'annual_stats_constrained': FieldInfo(annotation=bool, required=False, default=False), 'drop_stats': FieldInfo(annotation=tuple[str, ...], required=False, default=()), 'forecast_days': FieldInfo(annotation=int, required=False, default=4, metadata=[Gt(gt=0)]), 'forecast_evaluation': FieldInfo(annotation=bool, required=False, default=False), 'model_only_stats': FieldInfo(annotation=bool, required=False, default=False), 'obs_only_stats': FieldInfo(annotation=bool, required=False, default=False), 'round_floats_precision': FieldInfo(annotation=Union[int, NoneType], required=False, default=None), 'stats_decimals': FieldInfo(annotation=Union[int, NoneType], required=False, default=None), 'stats_tseries_base_freq': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'trends_min_yrs': FieldInfo(annotation=int, required=False, default=7, metadata=[Gt(gt=0)]), 'use_diurnal': FieldInfo(annotation=bool, required=False, default=False), 'use_fairmode': FieldInfo(annotation=bool, required=False, default=False), 'weighted_stats': FieldInfo(annotation=bool, required=False, default=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class pyaerocom.aeroval.setupclasses.**TimeSetup**(*, *DEFAULT_FREQS: Literal['monthly', 'yearly'] = 'monthly'*, *SEASONS: list[str] = ['all', 'DJF', 'MAM', 'JJA', 'SON']*, *main_freq: str = 'monthly'*, *freqs: list[str] = ['monthly', 'yearly']*, *periods: list[str] = None*, *add_seasons: bool = True*)

get_seasons()

Get list of seasons to be analysed

Returns SEASONS if add_seasons it True, else *['all']* (only whole year).

Returns
list of season strings for analysis

Return type
list

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'DEFAULT_FREQS': FieldInfo(annotation=Literal['monthly', 'yearly'], required=False, default='monthly'), 'SEASONS': FieldInfo(annotation=list[str], required=False, default=['all', 'DJF', 'MAM', 'JJA', 'SON']), 'add_seasons': FieldInfo(annotation=bool, required=False, default=True), 'freqs': FieldInfo(annotation=list[str], required=False, default=['monthly', 'yearly']), 'main_freq': FieldInfo(annotation=str, required=False, default='monthly'), 'periods': FieldInfo(annotation=list[str], required=False, default_factory=list)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class pyaerocom.aeroval.setupclasses.**WebDisplaySetup**(*, *map_zoom: str = 'World'*, *regions_how: Literal['default', 'aerocom', 'htap', 'country'] = 'default'*, *add_model_maps: bool = False*, *modelorder_from_config: bool = True*, *obsorder_from_config: bool = True*, *var_order_menu: tuple[str, ...] = ()*, *obs_order_menu: tuple[str, ...] = ()*, *model_order_menu: tuple[str, ...] = ()*, *hide_charts: tuple[str, ...] = ()*, *hide_pages: tuple[str, ...] = ()*, *ts_annotations: dict[str, str] = None*, *add_pages: tuple[str, ...] = ()*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'protected_namespaces': ()}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'add_model_maps': FieldInfo(annotation=bool, required=False, default=False), 'add_pages': FieldInfo(annotation=tuple[str, ...], required=False, default=()), 'hide_charts': FieldInfo(annotation=tuple[str, ...], required=False, default=()), 'hide_pages': FieldInfo(annotation=tuple[str, ...], required=False, default=()), 'map_zoom': FieldInfo(annotation=str, required=False, default='World'), 'model_order_menu': FieldInfo(annotation=tuple[str, ...], required=False, default=()), 'modelorder_from_config': FieldInfo(annotation=bool, required=False, default=True), 'obs_order_menu': FieldInfo(annotation=tuple[str, ...], required=False, default=()), 'obsorder_from_config': FieldInfo(annotation=bool, required=False, default=True), 'regions_how': FieldInfo(annotation=Literal['default', 'aerocom', 'htap', 'country'], required=False, default='default'), 'ts_annotations': FieldInfo(annotation=dict[str, str], required=False, default_factory=dict), 'var_order_menu': FieldInfo(annotation=tuple[str, ...], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

5.1.2 Specification of observation datasets

class pyaerocom.aeroval.obsentry.**ObsEntry**(***kwargs*)

Observation configuration for evaluation (dictionary)

Note: Only *obs_id* and *obs_vars* are mandatory, the rest is optional.

obs_id

ID of observation network in AeroCom database (e.g. 'AeronetSunV3Lev2.daily')

Type
str

obs_vars

list of pyaerocom variable names that are supposed to be analysed (e.g. ['od550aer', 'ang4487aer'])

Type
list

obs_ts_type_read

may be specified to explicitly define the reading frequency of the observation data (so far, this does only apply to gridded obsdata such as satellites). For ungridded reading, the frequency may be specified via *obs_id*, where applicable (e.g. AeronetSunV3Lev2.daily). Can be specified variable specific in form of dictionary.

Type
str or dict, optional

obs_vert_type

Aerocom vertical code encoded in the model filenames (only AeroCom 3 and later).

Type
str, optional

obs_aux_requires

information about required datasets / variables for auxiliary variables.

Type
dict, optional

instr_vert_loc

vertical location code of observation instrument. This is used in the aeroval interface for separating different categories of measurements such as "ground", "space" or "airborne".

Type
str, optional

is_superobs

if True, this observation is a combination of several others which all have to have their own obs config entry.

Type
bool

only_superobs

this indicates whether this configuration is only to be used as part of a superobs network, and not individually.

Type
bool

read_opts_ungridded

dictionary that specifies reading constraints for ungridded reading (c.g. pyaerocom.io.ReadUngridded).

Type
dict, optional

only_json

Only to be set if the obs entry already has colocated data files which were preprocessed outside of pyaerocom. Setting to True will skip the colcoation and just create the JSON output.

Type
bool

coldata_dir

Only to be set if the obs entry already has colocated data files which were preprocessed outside of pyaerocom. This is the directory in which the colocated data files are located.

Type
str

check_add_obs()

Check if this dataset is an auxiliary post dataset

check_cfg()

Check that minimum required attributes are set and okay

get_all_vars() → list

Get list of all variables associated with this entry

Returns
DESCRIPTION.

Return type
list

get_vert_code(*var*)

Get vertical code name for obs / var combination

has_var(*var_name*)

Check if input variable is defined in entry

Returns
True if entry has variable available, else False

Return type
bool

5.1.3 Specification of model datasets

class pyaerocom.aeroval.modelentry.**ModelEntry**(*model_id*, ***kwargs*)

Modeln configuration for evaluation (dictionary)

Note: Only *model_id* is mandatory, the rest is optional.

model_id

ID of model run in AeroCom database (e.g. 'ECMWF_CAMS_REAN')

Type
str

model_ts_type_read

may be specified to explicitly define the reading frequency of the model data. Not to be confused with `ts_type`, which specifies the frequency used for colocation. Can be specified variable specific by providing a dictionary.

Type
str or dict, optional

model_use_vars

dictionary that specifies mapping of model variables. Keys are observation variables, values are strings specifying the corresponding model variable to be used (e.g. model_use_vars=dict(od550aer='od550csaer'))

Type
dict

model_add_vars

dictionary that specifies additional model variables. Keys are observation variables, values are lists of strings specifying the corresponding model variables to be used (e.g. model_use_vars=dict(od550aer=['od550csaer', 'od550so4']))

Type
dict

model_rename_vars

key / value pairs specifying new variable names for model variables in the output json files (is applied after co-location).

Type
dict

model_read_aux

may be used to specify additional computation methods of variables from models. Keys are obs variables, values are dictionaries with keys *vars_required* (list of required variables for computation of var and *fun* (method that takes list of read data objects and computes and returns var)

Type
dict

property aux_funs_required

Boolean specifying whether this entry requires auxiliary variables

get_vars_to_process(*obs_vars: list*) → tuple

Get lists of obs / mod variables to be processed

Parameters
obs_vars (*list*) – list of observation variables

Returns

- *list* – list of observation variables (potentially extended from input list)
- *list* – corresponding model variables which are mapped based on content of *model_add_vars* and *model_use_vars*.

5.1.4 Containers for model and observation setup

Collection classes to specify a number of model entries and a number of observation entries for a given AeroVal experiment.

class pyaerocom.aeroval.collections.**BaseCollection**(**args*, ***kwargs*)

FORBIDDEN_CHARS_KEYS = ['_']

Invalid chars in entry names

MAXLEN_KEYS = 25

maximum length of entry names

abstract get_entry(*key*) → object

Getter for eval entries

Raises
KeyError – if input name is not in this collection

keylist(*name_or_pattern: str | None = None*) → list

Find model names that match input search pattern(s)

Parameters
name_or_pattern (*str, optional*) – Name or pattern specifying search string.

Returns
list of keys in collection that match input requirements. If *name_or_pattern* is None, all keys will be returned.

Return type
list

Raises
KeyError – if no matches can be found

abstract property web_iface_names: list

List of webinterface names for

class pyaerocom.aeroval.collections.**ModelCollection**(**args*, ***kwargs*)

Dict-like object that represents a collection of model entries

Keys are model names, values are instances of `ModelEntry`. Values can also be assigned as dict and will automatically be converted into instances of `ModelEntry`.

Note: Entries must not necessarily be only models but may also be observations. Entries provided in this collection refer to the x-axis in the AeroVal heatmap display and must fulfill the protocol defined by `ModelEntry`.

---

get_entry(*key*) → object

Get model entry configuration

Since the configuration files for experiments are in json format, they do not allow the storage of executable custom methods for model data reading. Instead, these can be specified in a python module that may be specified via `add_methods_file` and that contains a dictionary *FUNS* that maps the method names with the callable methods.

As a result, this means that, by default, custom read methods for individual models in `model_config` do not contain the callable methods but only the names. This method will take care of handling this and will return a dictionary where potential custom method strings have been converted to the corresponding callable methods.

Parameters
model_name (*str*) – name of model

Returns
Dictionary that specifies the model setup ready for the analysis

Return type
dict

property web_iface_names: list

List of web interface names for each obs entry

Return type
list

class pyaerocom.aeroval.collections.**ObsCollection**(**args*, ***kwargs*)

Dict-like object that represents a collection of obs entries

Keys are obs names, values are instances of `ObsEntry`. Values can also be assigned as dict and will automatically be converted into instances of `ObsEntry`.

Note: Entries must not necessarily be only observations but may also be models. Entries provided in this collection refer to the y-axis in the AeroVal heatmap display and must fulfill the protocol defined by `ObsEntry`.

---

property all_vert_types

List of unique vertical types specified in this collection

get_all_vars() → list

Get unique list of all obs variables from all entries

Returns
list of variables specified in obs collection

Return type
list

get_entry(*key*) → object

Getter for obs entries

Raises
KeyError – if input name is not in this collection

get_web_iface_name(*key*)

Get webinterface name for entry

Note: Normally this is the key of the obsentry in `obs_config`, however, it might be specified explicitly via key *web_interface_name* in the corresponding value.

Parameters
key (*str*) – key of entry.

Returns
corresponding name

Return type
str

property web_iface_names: list

List of web interface names for each obs entry

Return type
list

5.2 Processing tools

5.2.1 Experiment processing engine

class pyaerocom.aeroval.experiment_processor.**ExperimentProcessor**(*cfg:* EvalSetup)

Processing engine for AeroVal experiment

By default, this class processes one configuration file, represented by `EvalSetup`. As such, an instance of `EvalSetup` represents an AeroVal experiment, comprising a list of models, a list of observations (and variables).

For each possible (or defined) model / obs / variable combination, the processing engine will perform spatial and temporal co-location and will store on co-located NetCDF file (e.g. if there are 2 models, 2 observation networks and 2 variables there will be 4 co-located NetCDF files). The co-location is done using *pyaerocom.colocation_auto.Colocator*.

run(*model_name=None*, *obs_name=None*, *var_list=None*, *update_interface=True*)

Create colocated data and json files for model / obs combination

Parameters

- **model_name** (*str or list, optional*) – Name or pattern specifying model that is supposed to be analysed. Can also be a list of names or patterns to specify multiple models. If None (default), then all models are run that are part of this experiment.
- **obs_name** (str, or list, optional) – Like `model_name`, but for specification(s) of observations that are supposed to be used. If None (default) all observations are used.
- **var_list** (*list, optional*) – list variables supposed to be analysed. If None, then all variables available are used. Defaults to None. Can also be *str* type.
- **update_interface** (*bool*) – if true, relevant json files that determine what is displayed online are updated after the run, including the the menu.json file and also, the model info table (minfo.json) file is created and saved in `exp_dir`.

Returns

list containing all colocated data objects that have been converted to json files.

Return type

list

update_interface()

Update aeroval interface

Things done here:

- Update menu file
- Make aeroval info table json (tab informations in interface)
- update and order heatmap file

5.2.2 Model maps processing

class pyaerocom.aeroval.modelmaps_engine.**ModelMapsEngine**(*cfg:* EvalSetup)

Engine for processing of model maps

run(***kwargs*)

Method that runs the processing based on settings in `cfg`

Parameters

- ***args** – positional arguments.
- ****kwargs** – Keyword arguments.

Returns
list of output file paths generated by the engine.

Return type
list

5.2.3 Processing of super-observation entries

Super observations refer to merged observation datasets to increase the number of stations.

class pyaerocom.aeroval.superobs_engine.**SuperObsEngine**(*cfg:* EvalSetup)

Class to handle the processing of combined obs datasets

run(*model_name*, *obs_name*, *var_list*, *try_colocate_if_missing=True*)

Method that runs the processing based on settings in `cfg`

Parameters

- ***args** – positional arguments.
- ****kwargs** – Keyword arguments.

Returns
list of output file paths generated by the engine.

Return type
list

5.2.4 Low-level base classes for processing engines

class pyaerocom.aeroval._processing_base.**DataImporter**(*cfg:* EvalSetup)

Class that supports reading of model and obs data based on an eval config.

Depending on a `EvalSetup`, reading of model and obs data may have certain constraints (e.g. freq, years, alias variable names, etc.), which are / can be specified flexibly for each model and obs entry in an analysis setup (`EvalSetup`). Proper handling of these reading constraints and data import settings are handled in the *pyaerocom.colocation_auto.Colocator* engine, therefore the reading in this class is done via the `Colocator` engine.

read_model_data(*model_name*, *var_name*)

Import model data

Parameters

- **model_name** (*str*) – Name of model in `cfg`,

- **var_name** (*str*) – Name of variable to be read.

Returns
data – loaded model data.

Return type
GriddedData

read_ungridded_obsdata(*obs_name*, *var_name*)

Import ungridded observation data

Parameters

- **obs_name** (*str*) – Name of observation network in `cfg`
- **var_name** (*str*) – Name of variable to be read.

Returns
data – loaded obs data.

Return type
UngriddedData

class pyaerocom.aeroval._processing_base.**HasColocator**(*cfg:* EvalSetup)

Config class that also has the ability to co-locate

get_colocator(*model_name: str | None = None*, *obs_name: str | None = None*) → *Colocator*

Instantiate colocation engine

Parameters

- **model_name** (*str, optional*) – name of model. The default is None.
- **obs_name** (*str, optional*) – name of obs. The default is None.

Return type
Colocator

class pyaerocom.aeroval._processing_base.**HasConfig**(*cfg:* EvalSetup)

Base class that ensures that evaluation configuration is available

cfg

AeroVal experiment setup

Type
EvalSetup

exp_output

Manages output for an AeroVal experiment (e.g. path locations).

Type
ExperimentOutput

class pyaerocom.aeroval._processing_base.**ProcessingEngine**(*cfg:* EvalSetup)

Abstract base for classes supposed to do one or more processing tasks

Requirement for a processing class is to inherit attrs from *HasConfig* and, in addition to that, to have implemented a method **:fun:`run`** which is running the corresponding processing task and storing all the associated output files, that are read by the frontend.

One example of an implementation is the *pyaerocom.aeroval.modelmaps_engine.ModelMapsEngine*.

abstract run(**args*, ***kwargs*) → list

Method that runs the processing based on settings in `cfg`

Parameters

- ***args** – positional arguments.
- ****kwargs** – Keyword arguments.

Returns
list of output file paths generated by the engine.

Return type
list

5.2.5 Helpers for processing of auxiliary variables

class pyaerocom.aeroval.aux_io_helpers.**ReadAuxHandler**(*aux_file: str*)

Helper class for import of auxiliary function objects

aux_file

path to python module containing function definitions (note: function definitions in module need to be stored in a dictionary called *FUNS* in the file, where keys are names of the functions and values are callable objects.)

Type
str

Parameters
aux_file (*str*) – input file containing auxiliary functions (details see Attributes section).

import_all()

Import all callable functions in module with their names

Currently, these are expected to be stored in a dictionary called *FUNS* which should be defined in the python module.

Returns
function definitions.

Return type
dict

import_module()

Import *aux_file* as python module

Uses `importlib.import_module()` for import. :returns: imported module. :rtype: module

pyaerocom.aeroval.aux_io_helpers.**check_aux_info**(*fun*, *vars_required*, *funcs*)

Make sure information is correct for computation of auxiliary variables

Parameters

- **fun** (*str or callable*) – name of function or function used to compute auxiliary variable. If str, then arg *funcs* needs to be provided.
- **vars_required** (*list*) – list of required variables for computation of auxiliary variable.
- **funcs** (*dict*) – Dictionary with possible functions (values) and names (keys)

Returns
dict containing callable function object and list of variables required.

Return type
dict

5.2.6 Conversion of co-located data to json output

class pyaerocom.aeroval.coldatatojson_engine.**ColdataToJsonEngine**(*cfg:* EvalSetup)

process_coldata(*coldata:* ColocatedData)

Creates all json files for one ColocatedData object

Parameters
coldata (ColocatedData) – colocated data to be processed.

Raises

- **NotImplementedError** – DESCRIPTION.
- **ValueError** – DESCRIPTION.
- **ConfigError** – DESCRIPTION.

Return type
None.

run(*files*)

Convert colocated data files to json

Parameters
files (*list*) – list of file paths pointing to colocated data objects to be processed.

Returns
list of files that have been converted.

Return type
list

5.3 Output management

class pyaerocom.aeroval.experiment_output.**ExperimentOutput**(*cfg*)

JSON output for experiment

clean_json_files() → list

Checks all existing json files and removes outdated data

This may be relevant when updating a model name or similar.

delete_experiment_data(*also_coldata=True*) → None

Delete all data associated with a certain experiment

Note: This simply deletes the experiment directory with all the json files and, if *also_coldata* is True, also the associated co-located data objects.

Parameters

- **base_dir** (*str, optional*) – basic output direcory (containing subdirs of all projects)
- **proj_name** (*str, optional*) – name of project, if None, then this project is used
- **exp_name** (*str, optional*) – name experiment, if None, then this project is used
- **also_coldata** (*bool*) – if True and if output directory for colocated data is default and specific for input experiment ID, then also all associated colocated NetCDF files are deleted. Defaults to True.

property exp_dir: str

Experiment directory

property exp_id: str

Experiment ID

get_model_order_menu() → list

Order of models in menu

Note: Returns empty list if no specific order is to be used in which case the models will be alphabetically ordered

---

get_obs_order_menu() → list

Order of observation entries in menu

property menu_file: str

json file containing region specifications

property out_dirs_json: dict

json output directories (*dict*)

property regions_file: str

json file containing region specifications

reorder_experiments(*exp_order=None*) → None

Reorder experiment order in evaluation interface

Puts experiment list into order as specified by *exp_order*, all remaining experiments are sorted alphabetically.

Parameters

exp_order (*list, optional*) – desired experiment order, if None, then alphabetical order is used.

property results_available: bool

True if results are available for this experiment, else False

Type

bool

property statistics_file: str

json file containing region specifications

update_interface() → None

Update web interface

Steps:

1. Check if results are available, and if so:

2. Add entry for this experiment in experiments.json
3. Create/update ranges.json file in experiment directory
4. Update menu.json against available output and evaluation setup
5. Synchronise content of heatmap json files with menu
6. Create/update file statistics.json in experiment directory
7. Copy json version of EvalSetup into experiment directory

Return type
None

update_menu() → None

Update menu

The menu.json file is created based on the available json map files in the map directory of an experiment.

Parameters

- **menu_file** (*str*) – path to json menu file
- **delete_mode** (*bool*) – if True, then no attempts are being made to find json files for the experiment specified in *config*.

property var_ranges_file: str

json file containing region specifications

class pyaerocom.aeroval.experiment_output.**ProjectOutput**(*proj_id: str*, *json_basedir: str*)

JSON output for project

property available_experiments: list

List of available experiments

property experiments_file: str

json file containing region specifications

property proj_dir: str

Project directory

5.4 Global settings

5.4.1 Global defaults

pyaerocom.aeroval.glob_defaults.**statistics_defaults** = {'R': {'colmap': 'RdYlGn', 'decimals': 2, 'forecast': True, 'longname': 'Correlation Coefficient', 'name': 'R', 'scale': [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1], 'unit': '1'}, 'R_spearman': {'colmap': 'RdYlGn', 'decimals': 2, 'longname': 'R Spearman Correlation', 'name': 'R Spearman', 'scale': [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1], 'unit': '1'}, 'data_mean': {'colmap': 'coolwarm', 'decimals': 2, 'longname': 'Model Mean', 'name': 'Mean-Mod', 'scale': None, 'unit': '1'}, 'fge': {'colmap': 'reverseColmap(RdYlGn)', 'decimals': 2, 'forecast': True, 'longname': 'Fractional Gross Error', 'name': 'FGE', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2], 'unit': '1'}, 'mab': {'colmap': 'bwr', 'decimals': 1, 'longname': 'Mean Absolute Bias', 'name': 'MAB', 'scale': [0, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15], 'unit': 'var'}, 'mb': {'colmap': 'bwr', 'decimals': 1, 'longname': 'Mean Bias', 'name': 'MB', 'scale': [-0.15, -0.1, -0.05, 0, 0.05, 0.1, 0.15], 'unit': 'var'}, 'mnmb': {'colmap': 'bwr', 'decimals': 1, 'forecast': True, 'longname': 'Modified Normalized Mean Bias', 'name': 'MNMB', 'scale': [-100, -75, -50, -25, 0, 25, 50, 75, 100], 'unit': '%'}, 'nmb': {'colmap': 'bwr', 'decimals': 1, 'forecast': True, 'longname': 'Normalized Mean Bias', 'name': 'NMB', 'scale': [-100, -75, -50, -25, 0, 25, 50, 75, 100], 'unit': '%'}, 'nrms': {'colmap': 'Reds', 'decimals': 1, 'longname': 'Normalized Root Mean Square Error', 'name': 'NRMSE', 'scale': [0, 25, 50, 75, 100, 125, 150, 175, 200], 'unit': '%'}, 'num_coords_with_data': {'colmap': None, 'decimals': 0, 'longname': 'Number of Stations with data', 'name': 'Nb. Stations', 'overall_only': True, 'scale': None, 'unit': '1'}, 'num_valid': {'colmap': None, 'decimals': 0, 'longname': 'Number of Valid Observations', 'name': 'Nb. Obs', 'overall_only': True, 'scale': None, 'unit': '1'}, 'refdata_mean': {'colmap': 'coolwarm', 'decimals': 2, 'longname': 'Observation Mean', 'name': 'Mean-Obs', 'scale': None, 'unit': '1'}, 'rms': {'colmap': 'coolwarm', 'decimals': 2, 'forecast': True, 'longname': 'Root Mean Square Error', 'name': 'RMSE', 'scale': None, 'unit': '1'}}

Default information for statistical parameters

pyaerocom.aeroval.glob_defaults.**statistics_trend** = {'mod_trend': {'colmap': 'bwr', 'decimals': 1, 'longname': 'Modelled Trends', 'name': 'Mod-Trends', 'scale': [-10, -7.5, -5.0, -2.5, 0, 2.5, 5.0, 7.5, 10.0], 'unit': '%/yr'}, 'obs/mod_trend': {'colmap': 'bwr', 'decimals': 1, 'longname': 'Trends', 'name': 'Obs/Mod-Trends', 'scale': [-10.0, -7.5, -5.0, -2.5, 0, 2.5, 5.0, 7.5, 10.0], 'unit': '%/yr'}, 'obs_trend': {'colmap': 'bwr', 'decimals': 1, 'longname': 'Observed Trends', 'name': 'Obs-Trends', 'scale': [-10.0, -7.5, -5.0, -2.5, 0, 2.5, 5.0, 7.5, 10.0], 'unit': '%/yr'}}

Default information about trend display

pyaerocom.aeroval.glob_defaults.var_ranges_defaults = {'abs550aer': {'colmap': 'coolwarm', 'scale': [0, 0.0125, 0.025, 0.0375, 0.05, 0.0625, 0.075, 0.0875, 0.1]}, 'absc550aer': {'colmap': 'coolwarm', 'scale': [0, 12.5, 25, 37.5, 50, 62.5, 75, 87.5, 100]}, 'ang4487aer': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]}, 'backscatter': {'colmap': 'coolwarm', 'scale': [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1]}, 'bsc532aer': {'colmap': 'coolwarm', 'scale': [0.0, 0.0005, 0.001, 0.0015, 0.002, 0.0025, 0.003, 0.0035, 0.004]}, 'concCec25': {'colmap': 'coolwarm', 'scale': [0, 1.25, 2.5, 3.75, 5, 6.25, 7.5, 8.75, 10]}, 'concCecpm25': {'colmap': 'coolwarm', 'scale': [0, 1.25, 2.5, 3.75, 5, 6.25, 7.5, 8.75, 10]}, 'concCoc25': {'colmap': 'coolwarm', 'scale': [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 10]}, 'concCocpm25': {'colmap': 'coolwarm', 'scale': [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 10]}, 'concNhno3': {'colmap': 'coolwarm', 'scale': [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 1]}, 'concNnh3': {'colmap': 'coolwarm', 'scale': [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 7.5, 10.0, 20]}, 'concNnh4': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0]}, 'concNno3pm10': {'colmap': 'coolwarm', 'scale': [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1, 1.5, 2, 5, 10]}, 'concNno3pm25': {'colmap': 'coolwarm', 'scale': [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1, 1.5, 2, 5, 10]}, 'concNtnh': {'colmap': 'coolwarm', 'scale': [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 7.5, 10.0, 20, 50]}, 'concNtno3': {'colmap': 'coolwarm', 'scale': [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0, 1.5, 2, 5]}, 'concco': {'colmap': 'coolwarm', 'scale': [100.0, 125.0, 150.0, 175.0, 200.0, 225.0, 250.0, 275.0, 300.0]}, 'concdust': {'colmap': 'coolwarm', 'scale': [0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]}, 'concno2': {'colmap': 'coolwarm', 'scale': [0, 10, 20, 30, 40, 50, 60, 70, 80]}, 'conco3': {'colmap': 'coolwarm', 'scale': [0, 15, 30, 45, 60, 75, 90, 105, 120]}, 'concom25': {'colmap': 'coolwarm', 'scale': [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]}, 'concpm10': {'colmap': 'coolwarm', 'scale': [0, 10, 20, 30, 40, 50, 60, 70, 80]}, 'concpm25': {'colmap': 'coolwarm', 'scale': [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]}, 'concso2': {'colmap': 'coolwarm', 'scale': [0, 0.75, 1.5, 2.25, 3.0, 3.75, 4.5, 5.25, 6.0, 6.75, 7.5, 8.25]}, 'concso4': {'colmap': 'coolwarm', 'scale': [0, 0.75, 1.5, 2.25, 3.0, 3.75, 4.5, 5.25, 6.0, 6.75, 7.5, 8.25]}, 'concsspm10': {'colmap': 'coolwarm', 'scale': [0, 0.75, 1.5, 2.25, 3.0, 3.75, 4.5, 5.25, 6.0, 6.75, 7.5, 8.25, 10, 15, 20, 50]}, 'concsspm25': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 5, 10]}, 'default': {'colmap': 'coolwarm', 'scale': [0, 1.25, 2.5, 3.75, 5, 6.25, 7.5, 8.75, 10]}, 'depdust': {'colmap': 'coolwarm', 'scale': [0.01, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 50.0, 100.0, 200.0, 500.0, 1000.0]}, 'depna': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20, 50, 100, 200]}, 'depnaf': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20]}, 'depoxn': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20]}, 'depoxnf': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20]}, 'depoxs': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20]}, 'depoxsf': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20]}, 'deprdn': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20, 50, 100, 200]}, 'deprdnf': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20, 50, 100, 200]}, 'drydust': {'colmap': 'coolwarm', 'scale': [0.0, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 50.0, 100.0, 200.0, 500.0, 1000.0]}, 'dryhno3': {'colmap': 'coolwarm', 'scale': [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.5, 1.0, 2.0, 5.0]}, 'dryhono': {'colmap': 'coolwarm', 'scale': [0.0, 0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.008, 0.01, 0.02]}, 'dryn2o5': {'colmap': 'coolwarm', 'scale': [0.0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.1, 0.2]}, 'dryna': {'colmap': 'coolwarm', 'scale': [0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 5, 10, 20, 50, 100, 200]}, 'drynh3': {'colmap': 'coolwarm', 'scale': [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1, 2, 5]}, 'drynh4': {'colmap': 'coolwarm', 'scale': [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.5, 1.0]}, 'dryno2': {'colmap': 'coolwarm', 'scale': [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.5, 1.0, 2]}, 'dryno3c': {'colmap': 'coolwarm', 'scale': [0.0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.2, 0.5]}, 'dryno3f': {'colmap': 'coolwarm', 'scale': [0.0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.2, 0.5]}, 'dryo3': {'colmap': 'coolwarm', 'scale': [0, 0.5, 1, 5, 10, 15, 20, 25, 30, 40, 50]}, 'dryoxn': {'colmap': 'coolwarm', 'scale': [0, 0.1, 0.2, 0.5, 1, 2.0, 5, 10, 20, 50]}, 'dryoxs': {'colmap': 'coolwarm', 'scale': [0,

Default variable ranges for web display

5.4.2 Frontend variable naming conventions

class pyaerocom.aeroval.varinfo_web.**VarinfoWeb**(*var_name: str*, *cmap: str | None = None*, *cmap_bins: list | None = None*, *vmin: float | None = None*, *vmax: float | None = None*)

Additional variable information relevant for AeroVal web output

var_name

Name of variable (AeroCom name, not web display name)

Type
str

cmap_bins

Value bins for web display

Type
list

cmap

name of colormap for web display

Type
str

Parameters

- **var_name** (*str*) – Name of variable (AeroCom name, not web display name)
- **cmap** (*str, optional*) – name of colormap for web display. If None, the colormap associated with the input variable is used (via *pyaerocom.variable.Variable.get_cmap()*). Defaults to None.
- **cmap_bins** (*list, optional*) – Value bins for web display. If None, then they are inferred from input vmin and vmax, or, if the latter are also None, from attrs *pyaerocom.variable.Variable.minimum* and `pyaerocom.variable.Variable.maximum`. If the latter are not defined an AttributeError will be thrown on initialisation.
- **vmin** (*float, optional*) – lower end of range
- **vmax** (*float, optional*) – upper end of range

autofill_missing(*vmin: float | None = None*, *vmax: float | None = None*) → None

Autofill missing attributes related to cmap bins and cmap

Parameters

- **vmin** (*float, optional*) – lower end of range
- **vmax** (*float, optional*) – upper end of range

Return type
None

static from_dict(*dict*)

Instantiate from dictionary

Parameters
dict (*dict*) – settings

Returns
instantiated instance

Return type
VarinfoWeb

to_dict()

Convert to dictionary

Return type
dict

property vmax: float

Upper end of range

property vmin: float

Lower end of range

5.5 High-level utility functions

pyaerocom.aeroval.utils.**compute_model_average_and_diversity**(*cfg*, *var_name*, *model_names=None*, *ts_type=None*, *lat_res_deg=2*, *lon_res_deg=3*, *data_id=None*, *avg_how=None*, *extract_surface=True*, *ignore_models=None*, *comment=None*, *model_use_vars=None*)

Compute median or mean model based on input models

Note:

- BETA version that will likely undergo revisions.
- Time selection currently not properly handled

Parameters

- **cfg** (*AerocomEvaluation*) – analysis instance
- **var_name** (*str*) – name of variable
- **model_names** (*list, optional*) – list of model names. If None, all entries in input engine are used.
- **ts_type** (*str, optional*) – output freq. Defaults to monthly.
- **lat_res_deg** (*int, optional*) – output latitude resolution, defaults to 2 degrees.
- **lon_res_deg** (*int, optional*) – output longitude resolution, defaults to 3 degrees.

- **data_id** (*str, optional*) – output data_id of ensemble model.
- **avg_how** (*str, optional*) – how to compute averages (choose from mean or median), defaults to "median".
- **extract_surface** (*bool*) – if True (and if data contains model levels), surface level is extracted
- **ignore_models** (*list, optional*) – list of models to be ignored
- **comment** (*str, optional*) – comment string added to metadata of output data objects.
- **model_use_vars** (*dict, optional*) – model variables to be used.

Returns

- *GriddedData* – ensemble model for input variable computed averaged using median or mean (input avg_how). Default is median.
- *GriddedData* – corresponding diversity field, if avg_how is "mean", then computed using definition from Textor et al., 2006 (ACP) DOI: 10.5194/acp-6-1777-2006. If avg_how is "median" then interquartile range is used (Q3-Q1)/Q2
- *GriddedData or None* – Q1 field (only output if avg_how is median)
- *GriddedData or None* – Q3 field (only output if avg_how is median)
- *GriddedData or None* – standard deviation field (only output if avg_how is mean)

pyaerocom.aeroval.utils.**make_config_template**(*proj_id: str*, *exp_id: str*) → *EvalSetup*

Make a template for an AeroVal evaluation setup

Parameters

- **proj_id** (*str*) – ID of project.
- **exp_id** (*str*) – ID of experiment

Returns

template evaluation setup (all defaults are set) that can be used to add model and obs entries, and modified to meet the purposes.

Return type

EvalSetup

5.6 Helper modules

5.6.1 General helper functions

pyaerocom.aeroval.helpers.**check_if_year**(*periods: list[str]*) → bool

Checks if the periods in the periods list are years or dates

pyaerocom.aeroval.helpers.**check_var_ranges_avail**(*model_data*, *var_name*)

Check if lower and upper variable ranges are available for input variable

Parameters

- **model_data** (*GriddedData*) – modeldata containing variable data
- **var_name** (*str*) – variable name to be checked (must be the same as model data AeroCom variable name).

Raises
ValueError – if ranges for input variable are not defined and if input model data corresponds to a different variable than the input variable name.

Return type
None

5.6.2 Helpers for coldat2json conversion

Helpers for conversion of ColocatedData to JSON files for web interface.

pyaerocom.aeroval.coldatatojson_helpers.**add_profile_entry_json**(*profile_file: str*, *data:* ColocatedData, *profile_viz: dict*, *periods: list[str]*, *seasons: list[str]*)

Analogous to _add_heatmap_entry_json for profile data. Every time this function is called it checks to see if the profile_file exists. If so, it reads it, if not it makes a new one. This is because one can not add to json files and so everytime we want to add entries for profile layers we must read in the old file, add the entries, and write a new file.

Parameters

- **profile_file** (*str*) – Name of profile_file
- **data** (ColocatedData) – For this vertical layer
- **profile_viz** (*dict*) – Output of process_profile_data()
- **periods** (*list[str]*) – periods to compute over (years)
- **seasons** (*list[str]*) – seasons to compute over (e.g., All, DJF, etc.)

pyaerocom.aeroval.coldatatojson_helpers.**get_json_mapname**(*obs_name*, *var_name_web*, *model_name*, *model_var*, *vert_code*, *period*)

Get base name of json file

pyaerocom.aeroval.coldatatojson_helpers.**get_stationfile_name**(*station_name*, *obs_name*, *var_name_web*, *vert_code*)

Get name of station timeseries file

pyaerocom.aeroval.coldatatojson_helpers.**process_profile_data_for_regions**(*data:* ColocatedData, *region_id: str*, *use_country: bool*, *periods: list[str]*, *seasons: list[str]*) → dict

This method populates the json files in data/profiles which are use for visualization. Analogous to _process_map_and_scat for profile data. Each json file corresponds to a region or station, obs network, and variable. Inside the json, it is broken up by model. Each model has a key for "z" (the vertical dimension), "obs", and "mod" Each "obs" and "mod" is broken up by period.

Parameters

- **data** (ColocatedData) – ColocatedData object for this layer
- **region_id** (*str*) – Spatial subset to compute the mean profiles over
- **station_name** (*str*) – Station to compute mean profiles over for period
- **use_country** (*boolean*) – Passed to filter_region().

- **periods** (*str*) – Year part of the temporal range to average over
- **seasons** (*str*) – Sesonal part of the temporal range to average over

Returns
Dictionary to write to json

Return type
output (dict)

pyaerocom.aeroval.coldatatojson_helpers.**process_profile_data_for_stations**(*data: ColocatedData*, *station_name: str*, *use_country: bool*, *periods: list[str]*, *seasons: list[str]*) → dict

This method populates the json files in data/profiles which are use for visualization. Analogous to _process_map_and_scat for profile data. Each json file corresponds to a region, obs network, and variable. Inside the json, it is broken up by model. Each model has a key for "z" (the vertical dimension), "obs", and "mod" Each "obs" and "mod" is broken up by period.

Parameters

- **data** (ColocatedData) – ColocatedData object for this layer
- **region_id** (*str*) – Spatial subset to compute the mean profiles over
- **station_name** (*str*) – Station to compute mean profiles over for period
- **use_country** (*boolean*) – Passed to filter_region().
- **periods** (*str*) – Year part of the temporal range to average over
- **seasons** (*str*) – Sesonal part of the temporal range to average over

Returns
Dictionary to write to json

Return type
output (dict)

pyaerocom.aeroval.coldatatojson_helpers.**update_regions_json**(*region_defs*, *regions_json*)

Check current regions.json for experiment and update if needed

Parameters

- **region_defs** (*dict*) – keys are names of region (not IDs!) values define rectangular borders
- **regions_json** (*str*) – regions.json file (if it does not exist it will be created).

Returns
current content of updated regions.json

Return type
dict

5.6.3 Model maps helper functions

pyaerocom.aeroval.modelmaps_helpers.**calc_contour_json**(*data*, *cmap*, *cmap_bins*)

Convert gridded data into contours for json output

Parameters

- **data** (GriddedData) – input data
- **cmap** (*str*) – colormap of output
- **cmap_bins** (*list*) – list containing the bins to which the values are mapped.

Returns

dictionary containing contour data

Return type

dict

pyaerocom.aeroval.modelmaps_helpers.**griddeddata_to_jsondict**(*data*, *lat_res_deg=5*, *lon_res_deg=5*)

Convert gridded data to json dictionary

Parameters

- **data** (GriddedData) – input data to be converted
- **lat_res_deg** (*int*) – output latitude resolution in decimal degrees
- **lon_res_deg** (*int*) – output longitude resolution in decimal degrees

Returns

data dictionary for json output (keys are metadata and data).

Return type

dict

CHAPTER

SIX

EXAMPLE CONFIGURATION FILES FOR AEROVAL

This section provides some example setup files for AeroVal evaluations with detailed explanations of the setup parameters. A configuration could be run as the following:

```
python cfg_example1.py
```

The code blocks below are the Python configuruation files *cfg_examples_example1.py* and *sample_gridded_io_aux.py*.

6.1 Example 1

NorESM2 and CAMS reanalysis vs AERONET and merged satellite AOD dataset.

```
"""
Example AeroVal configuration file
==================================

Author: Jonas Gliß
Date: 15.10.2021
Minimum pyaerocom version: 0.12.2

IMPORTANT NOTE
--------------
This script requires access to the PPI infrastructure of MET
Norway and cannot be run by external users.
If you work at MET, you can check whether pyaerocom has access to PPI by
running (from the command line):

pya --ppiaccess

What this is about?
-------------------

This example script will show how to create a basic AeroVal evaluation setup
and run the evaluation. The output of this example is available at:

https://aeroval.met.no/evaluation.php?project=examples&exp_name=example1

General evaluation procedure:
-----------------------------
```

(continues on next page)

(continued from previous page)

```
This configuration defines a set of models to be evaluated against a certain
set of observations, within what is called an "Experiment" in AeroVal. The
observations defined specify which variables are supposed to be evaluated
from each of the models. For instance, one "Experiment" could comprise 2
observation data sources (OBS1 and OBS2), each of which measures 2 variables (
OBS1: var1, var2 and OBS2: var1 and var3).
Now if that "Experiment" comprises 2 models (MOD1 and MOD2, which all have
output for var1, var2 and var3) then, this would result in an evaluation
matrix of 4x2 entries (4 OBS / var combinations and 2 models). Each
"pixel" of that matrix corresponds to one evaluation entry (e.g. var1 from
OBS1 is co-located with var1 from MOD1, and so on). Based on the co-located
data objects (pyaerocom.ColocatedData) that are created for each of these
"pixels" of the evaluation matrix, a set of statistical parameters (such as
bias, correlation coefficients, RMS, etc) are computed within certain time
periods that can be defined flexibly by the user (e.g. 2000-2010 and 2010-2020
or a single year, e.g. 2010).

In AeroVal, each of these "pixels" of that matrix is evaluated completely
independently, doing the following steps:

- Create co-located NetCDF file for that model / obs / var entry:
    - Read model data
    - Read obs data
    - Co-locate model with obs data (in space and time) within the time
    period range specified (e.g. 2000-2020 if user specifies output periods
    2000-2010 and 2010-2020).
    - Save co-located data as NetCDF file.
- Convert co-located data to json output files read by the AeroVal frontend

The script below is grouped into the following sections:
-------------------------------------------------------

- Section 1: Global setup for AeroVal output
- Section 2: Default co-location settings
- Section 3: Configuration of observations
- Section 4: Configuration of models
- Section 5: Main script (run analysis)

"""

"""
Section 1: Global setup for AeroVal output
------------------------------------------

The global setup defines:

- Project and experiment ID
- Information about the purpose of the experiment and PI
- Information about the output paths
- Time periods to be evaluated
- Desired output frequencies
- Additional options such as:
```

(continues on next page)

(continued from previous page)

```
    - Whether to compute trends
    - Whether to compute model maps
    - Whether to add seasonal statistics
    - Which regions to use for regional statistics
    - Web display options (e.g. default map zoom)

"""
import os  # needed for specification of path locations below

GLOB_CFG = dict(
    # PI of experiment
    exp_pi="Jonas Gliß",
    # ID of project (will define URL, see below)
    proj_id="examples",
    # ID of experiment (will define URL, see below)
    exp_id="example1",
    # These 2 IDs define where the output data is stored relative to the 2
    # output directories below, and they also define the URL link for this
    # project, e.g. for this example, the web URL for this experiment would be:
    #
    # https://aeroval.met.no/evaluation.php?project=examples&exp_name=example1
    #
    # Base directory for json output files
    json_basedir=os.path.abspath("../../data"),
    # Base directory for co-located data output NetCDF files
    coldata_basedir=os.path.abspath("../../coldata"),
    # E.g. for this experiment, the co-located data files can be found in:
    #
    # ../../coldata/examples/example1
    #
    # And the output json files in:
    #
    # ../../data/examples/example1
    #
    # relative to the location of this script file. Note: if
    #
    # ../../data/examples
    #
    # happens to be a Gitlab (or Github) repository (or Github),
    # then pyaerocom will write all required json files into that
    # repository, which makes it easier to share the results by pushing them
    # to the remote repository. You can then contact e.g. Augustin Mortier (
    # augustinm@met.no) at MET Norway to clone that repository on the
    # AeroVal web server, so it becomes visible at
    #
    # https://aeroval.met.no/evaluation.php?project=examples&exp_name=example1
    # Optional: location a python file containing function definitions that
    # combine some iris.Cube instances to compute new variables. This
    # enables flexible computation of new model variables that are not in
    # the model output. The file has to have a FUNS attribute which is a
    # dictionary mapping names of the functions with the callable objects.
    # Below in Section 4, this is used to compute clear-sky Angstrom
```

(continues on next page)

(continued from previous page)

```
    # Exponent for NorESM2 model from 440nm and 870nm AOD, which are output
    # by the model.
    io_aux_file=os.path.abspath("../eval_py/gridded_io_aux.py"),
    # Frequencies for which statistical parameters are computed
    freqs=["daily", "monthly", "yearly"],
    # Main output frequency for AeroVal (some of the AeroVal processing
    # steps are only done for this resolution, since they would create too
    # much output otherwise, such as statistics timeseries or scatter plot in
    # "Overall Evaluation" tab on AeroVal).
    # Note that this frequency needs to be included in previous setting "freqs".
    main_freq="monthly",
    # Time periods for which statistical parameters are computed
    periods=["2010"],
    # Whether or not to add seasonal statistics
    add_seasons=True,
    # Whether or not to add trends output to the analysis. Trends analysis
    # needs at least 7 years of data, so this is skipped here for this
    # single year experiment
    add_trends=False,
    # Whether or not to re-colocate existing co-located data files from the
    # input data or not.
    reanalyse_existing=True,
    # Name of experiment
    exp_name="AeroVal example 1",
    # Description of experiment
    exp_descr=(
        "A simple setup evaluating AOD and Angstrom Exponent of 2 "
        "models (NorESM2 from AeroCom phase 3 control experiment and "
        "CAMS reanalysis dateset) for the year 2010, using AERONET "
        "version 3 sun photometer data as well as data from a merged "
        "satellite product"
    ),
    public=True,
    # Whether or not maps of the model fields are supposed to be computed
    add_model_maps=True,
)

"""
Section 2: Default co-location settings
---------------------------------------

The following specifies some settings to be used for co-location. Note
that this does note cover all possible settings, but only the ones that
deviate from the default co-location settings.
For all co-location options (and their defaults), see here:
https://pyaerocom.readthedocs.io/en/latest/api.html?highlight=ColocationSetup#pyaerocom.
↪colocation_auto.ColocationSetup

Note that all of the available co-location settings can also be specified
for model and obs entries individually below.
"""
DEFAULT_COLOCATION_SETUP = dict(
```

(continues on next page)

(continued from previous page)

```
    # Default frequency of co-located data objects.
    # NOTE: the output resolution of the co-located data files needs to be at
    # least as high as the highest output resolution for the statistical
    # results specified in parameter "freqs" above, since all statistical
    # results (stored in the output json files) are computed based on the
    # co-located NetCDF files.
    ts_type="daily",
    # Time resample setup: the following setup will perform resampling
    # from daily to monthly in 2 steps, 1. daily to weekly (requiring at
    # least 5 valid daily values per week) and 2. weekly to monthly,
    # requiring at least 3 weekly values per week. This corresponds to a
    # conservative resampling approach, requiring ca 75% coverage of
    # measurements. Note that min_num_obs may also be a simple integer,
    # specifying the minimum number of observations to resample the input
    # data to the co-location frequency, disregarding the frequency of the
    # input data (e.g. if min_num_obs is=5 and ts_type="monthly" and input
    # data is in hourly resolution, then this translated to "at least 5
    # hourly values per month", however, if input data is in daily resolution
    # then this translated to "at least 5 daily values per month").
    min_num_obs=dict(weekly=dict(daily=5), monthly=dict(weekly=3)),
    # How to aggregate the data when resampling (linked with min_num_obs and
    # can be setup in a similar way). This setting defaults to "mean",
    # however, here we use a sligthly more complicated setup that does
    # "median" and is synched with the stepwise min_num_obs regime.
    resample_how=dict(weekly=dict(daily="median"), monthly=dict(weekly="median")),
)

"""
Section 3: Configuration of observations
----------------------------------------

Each obs entry needs at least an obs ID specified, a list of variables
that are supposed to be analysed and a specification of the vertical type
of the data (e.g. Column, Surface).
You may search for observational data via the pyaerocom CLI. E.g. to search
for observations containing the strings "Aeronet" and "Sun" and "V3" you may
search from the command line via:

pya --browse *Aeronet*Sun*V3*

This will list all data IDs matching this request and will also show
variables that are provided by each of the datasets that match the search.
"""

# Define a filter for the AERONET obs dataset that excludes data from
# stations that start with DRAGON and use only sites that are located
# between 0 m and 1000 m a.s.l.
AERONET_SITE_FILTER = dict(station_name="DRAGON*", negate="station_name", altitude=[0,␣
↪1000])

# Define a python dictionary containing observation datasets to be used and
# the associated variables.
```

(continues on next page)

(continued from previous page)

```
OBS_CFG = {
    # This entry is for the AERONET Sun version 3, daily dataset and will
    # provides total AOD (od550aer) and Angstrom Exponent (ang4487aer) to be
    # compared with the models.
    # AERONET is a ground based measurement network of sun photometer
    # measurements, for more info see here:
    # https://aeronet.gsfc.nasa.gov/
    #
    # This dataset will be read by pyaerocom as "UngriddedData" since the
    # measurements are performed at AERONET site locations, and the input
    # data are provided as CSV files "per site location".
    #
    # This dataset will appear as AERONET in the AeroVal interface.
    "AERONET": dict(
        # obs_id tells pyaerocom where to find the data
        obs_id="AeronetSunV3Lev2.daily",
        # Which variables to use from AERONET, here we use total AOD (
        # od550aer) and Angstrom Exponent (ang4487aer).
        obs_vars=["od550aer", "ang4487aer"],
        # What vertical type do the variables represent (columnar)
        obs_vert_type="Column",
        # Metadata filters applied to the observations AFTER reading them
        # (as UngriddedData) and BEFORE co-locating them with the models.
        obs_filters={**AERONET_SITE_FILTER},
    ),
    # 2nd obs entry is a merged satellite AOD dataset, for more info see here:
    # https://acp.copernicus.org/articles/20/2031/2020/
    #
    # This dataset will be read by pyaerocom as "GriddedData" as it is
    # provided as gridded NetCDF file (near global coverage).
    #
    "MERGED-SAT": dict(
        # ID is linked with data location
        obs_id="MODIS6.1terra",
        # Which variables to use
        obs_vars=["od550aer"],
        # Which vertical code
        obs_vert_type="Column",
        # Since these data are gridded observations with near global
        # coverage, the underlying co-location routine will be
        # gridded OBS / gridded MOD. The following parameter defines to
        # which lat / lon resolution both MOD and OBS should be regridded
        # for co-location (5 degrees is a good compromise for a global
        # dataset, between spatial resolution and required storage of the
        # output files).
        regrid_res_deg=5,
    ),
}


"""
Section 4: Configuration of models
```

(continues on next page)

(continued from previous page)

```
---------------------------------

Each model entry needs at least a model ID specified. If the model data is
available in the AeroCom database (in AeroCom file format) you may search
for data via the pyaerocom CLI. E.g. to search for model IDs containing
the strings "ECMWF" and "CAMS" you may search from the command line via:

pya --browse *ECMWF*CAMS*

"""
MODEL_CFG = {
    # The key "CAMS-REAN" is how this model will be named in AeroVal
    "CAMS-REAN": dict(
        # Data ID of model data (tells pyaerocom where the data is located)
        model_id="ECMWF_CAMS_REAN"
    ),
    # The key "NorESM2" is how this model will be named in AeroVal
    "NorESM2": dict(
        # Data ID of model run (tells pyaerocom where the data is located)
        model_id="NorESM2-met2010_AP3-CTRL",
        # if model
        model_use_vars={
            # if obs var is od550aer, use od550csaer from the model
            # (od550csaer refers to clear-sky AOD)
            "od550aer": "od550csaer",
            # if obs var is ang4487aer, use ang4487csaer from the model
            "ang4487aer": "ang4487csaer",
        },
        model_read_aux={
            # Clear-sky AE is actually not available from the model so,
            # it is computed from clear-sky AOD at 440nm and 870 nm.
            # The function "calc_ae" is defined in "io_aux_file" (see
            # global setup above).
            "ang4487csaer": dict(vars_required=["od440csaer", "od870csaer"], fun="calc_ae
↪")
        },
    ),
}


"""
Section 5: Main script (run analysis)
-------------------------------------
"""
# Combine all settings into one dictionary called CFG
CFG = {**GLOB_CFG, **DEFAULT_COLOCATION_SETUP}

# Add obs config
CFG["obs_cfg"] = OBS_CFG
# Add model config
CFG["model_cfg"] = MODEL_CFG
```

(continues on next page)

(continued from previous page)

```
def main():
    """
    Create EvalSetup and pass to ExperimentProcessor and run analysis
    """
    from pyaerocom.aeroval import EvalSetup, ExperimentProcessor

    CFG["raise_exceptions"] = True
    stp = EvalSetup(**CFG)
    ana = ExperimentProcessor(stp)

    # The following command is optional:
    # it checks deletes all output files associated with this experiment,
    # that is, for this experiment, everything under:
    # ../../coldata/examples/example1
    # ../../data/examples/example1
    ana.exp_output.delete_experiment_data()

    # The following command is optional:
    # it checks all output json files associated with
    # this proj_id and exp_id for outdated data. This is useful if e.g.
    # model or observation entries are removed or renamed and the experiment
    # is rerun. It can also be run independently of the actual processing.
    # It is especially useless to call this one if
    # ana.exp_output.delete_experiment_data() is called before =)
    # Anyways, good to know that it exists.
    ana.exp_output.clean_json_files()

    # Run the experiment
    # This co-locates all "pixels" of the evaluation matrix and creates all
    # json files needed for AeroVal. It should end with printing something
    # like:
    # Processing finished.
    ana.run()


if __name__ == "__main__":
    main()
```

6.2 Example IO aux file for model reading

```
"""
Sample gridded IO AUX file for computation of new model variables

This file can be registered in an AeroVal configuration script and the
callable functions in FUNS can be used to compute new model variables.

In this example only 1 function is registered that is already implemented in
pyaerocom.
```

(continues on next page)

(continued from previous page)

```
The function is documented here:

https://pyaerocom.readthedocs.io/en/latest/api.html?highlight=compute_angstrom_coeff_
↪cubes#pyaerocom.io.aux_read_cubes.compute_angstrom_coeff_cubes

Similar functions (that take iris.cube.Cube instances as input and return a
new "Cube" can be defined in this file directly and then registered with a
name (key) in FUNS and then used in configuration files to compute auxiliary
model variables.
"""
from pyaerocom.io.aux_read_cubes import compute_angstrom_coeff_cubes

FUNS = {
    "calc_ae": compute_angstrom_coeff_cubes,
}
```

CHAPTER

SEVEN

CLI

Documentation of pyaerocom command line interface.

--help

-V, --version: Version of pyaerocom

Commands:

browse: Browse database. e.g., browse <DATABASE>

clearcache: Delete cached data objects

ppiaccess: Check if MetNO PPI can be accessed

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

NINE

ISSUES?

Please submit an issue if you would like to see a feature or experience a bug.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z