pyresample Documentation

Release 1.16.0+0.gef31c3c.dirty

Esben S. Nielsen

Jun 10, 2020

Contents

Pyresample is a python package for resampling geospatial image data. It is the primary method for resampling in the SatPy library, but can also be used as a standalone library. Resampling or reprojection is the process of mapping input geolocated data points to a new target geographic projection and area.

Pyresample can operate on both fixed grids of data and geolocated swath data. To describe these data Pyresample uses various "geometry" objects including the *AreaDefinition* and *SwathDefinition* classes.

Pyresample offers multiple resampling algorithms including:

- Nearest Neighbor
- Elliptical Weighted Average (EWA)
- Bilinear
- Bucket resampling (count hits per bin, averaging, ratios)

For nearest neighbor and bilinear interpolation pyresample uses a kd-tree approach by using the fast KDTree implementation provided by the pykdtree library. Pyresample works with numpy arrays and numpy masked arrays. Interfaces to XArray objects (including dask array support) are provided in separate Resampler class interfaces and are in active development. Utility functions are available to easily plot data using Cartopy.

Changed in version 1.15: Dropped Python 2 and Python <3.4 support.

CHAPTER 1

Documentation

1.1 Installing Pyresample

Pyresample depends on pyproj, numpy(>= 1.10), pyyaml, configobj, and pykdtree (>= 1.1.1).

In order to use the pyresample plotting functionality Cartopy and matplotlib (>= 1.0) must be installed. These packages are not a prerequisite for using any other pyresample functionality.

Optionally, for dask and xarray support these libraries must also be installed. Some utilities like converting from rasterio objects to pyresample objects will require rasterio or other libraries to be installed. The older multiprocessing interfaces (Proj_MP) use the `scipy` package's KDTree implementation. These multiprocessing interfaces are used when the `nprocs` keyword argument in the various pyresample interfaces is greater than 1. Newer xarray/dask interfaces are recommended when possible.

1.1.1 Package test

Testing pyresample requires all optional packages to be installed including rasterio, dask, xarray, cartopy, pillow, and matplotlib. Without all of these dependencies some tests may fail. To run tests from a source tarball:

```
tar -zxf pyresample-<version>.tar.gz
cd pyresample-<version>
python setup.py test
```

If all the tests passes the functionality of all pyresample functions on the system has been verified.

1.1.2 Package installation

Pyresample is available from PyPI and can be installed with pip:

```
pip install pyresample
```

Pyresample can also be installed with conda via the conda-forge channel:

```
conda install -c conda-forge pyresample
```

Or directly from a source tarball:

```
tar -zxvf pyresample-<version>.tar.gz
cd pyresample-<version>
pip install .
```

To install in a "development" mode where source file changes are immediately reflected in your python environment run the following instead of the above pip command:

```
pip install -e .
```

1.1.3 pykdtree and numexpr

Pyresample uses the `pykdtree` package which can be built with multi-threaded support. If it is built with this support the environment variable `OMP_NUM_THREADS` can be used to control the number of threads. Please refer to the pykdtree repository for more information.

As of pyresample v1.0.0 numexpr will be used for minor bottleneck optimization if available.

1.2 Geometry definitions

The *pyresample.geometry* module contains classes for describing different geographic areas using a mesh of points or pixels. Some classes represent geographic areas made of of evenly spaced/sized pixels, others handle the cases where the region is described by non-uniform pixels. The best object for describing a region depends on the use case and the information known about it. The different classes available in pyresample are described below.

Note that all longitudes and latitudes provided to *pyresample.geometry* classes must be in degrees. Additionally, longitudes must be in the [-180;+180[validity range.

Changed in version 1.8.0: Geometry objects no longer check the validity of the provided longitude and latitude coordinates to improve performance. Longitude arrays are expected to be between -180 and 180 degrees, latitude -90 to 90 degrees. This also applies to all geometry definitions that are provided longitude and latitude arrays on initialization. Use *check_and_wrap()* to preprocess your arrays.

1.2.1 AreaDefinition

An *AreaDefinition*, or `area`, is the primary way of specifying a uniformly spaced geographic region in pyresample. It is also one of the only geometry objects that understands geographic projections. Areas use the PROJ.4 method for describing projected coordinate reference systems (CRS). If the projection for an area is not described by longitude/latitude coordinates then it is typically described in X/Y coordinates in meters. See the PROJ.4 documentation for more information on projections and coordinate reference systems.

The following arguments are needed to initialize an area:

- **area_id**: ID of area
- **description**: Description
- **proj_id**: ID of projection (being deprecated)
- **projection**: Proj4 parameters as a dict or string
- **width**: Number of grid columns

- **height**: Number of grid rows
- **area_extent**: (lower_left_x, lower_left_y, upper_right_x, upper_right_y)

where

- **lower_left_x**: projection x coordinate of lower left corner of lower left pixel
- **lower_left_y**: projection y coordinate of lower left corner of lower left pixel
- **upper_right_x**: projection x coordinate of upper right corner of upper right pixel
- **upper_right_y**: projection y coordinate of upper right corner of upper right pixel

Example:

```
>>> from pyresample.geometry import AreaDefinition
>>> area_id = 'ease_sh'
>>> description = 'Antarctic EASE grid'
>>> proj_id = 'ease_sh'
>>> projection = {'proj': 'laea', 'lat_0': -90, 'lon_0': 0, 'a': 6371228.0, 'units':
↪'m'}
>>> width = 425
>>> height = 425
>>> area_extent = (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
>>> area_def
Area ID: ease_sh
Description: Antarctic EASE grid
Projection ID: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

You can also specify the projection using a PROJ.4 string

```
>>> projection = '+proj=laea +lat_0=-90 +lon_0=0 +a=6371228.0 +units=m'
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
```

or an EPSG code:

```
>>> projection = '+init=EPSG:3409'  # Use 'EPSG:3409' with pyproj 2.0+
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
```

Note: With pyproj 2.0+ please use the new `'EPSG:XXXX'` syntax as the old `'+init=EPSG:XXXX'` is no longer supported.

Creating an `AreaDefinition` can be complex if you don't know everything about the region being described. Pyresample provides multiple utilities for creating areas as well as storing them on disk for repeated use. See the *Geometry Utilities* documentation for more information.

1.2.2 GridDefinition

If the longitude and latitude values for an area are known, the complexity of an `AreaDefinition` can be skipped by using a *`GridDefinition`* object instead. Note that although grid definitions are simpler to define they come at the cost of much higher memory and CPU usage for almost all operations. The longitude and latitude arrays passed to `GridDefinition` are expected to be evenly spaced. If they are not then a `SwathDefinition` should be used (see below).

```
>>> import numpy as np
>>> from pyresample.geometry import GridDefinition
>>> lons = np.ones((100, 100))
>>> lats = np.ones((100, 100))
>>> grid_def = GridDefinition(lons=lons, lats=lats)
```

1.2.3 SwathDefinition

A swath is defined by the longitude and latitude coordinates for the pixels it represents. The coordinates represent the center point of each pixel. Swaths make no assumptions about the uniformity of pixel size and spacing. This means that operations using then may take longer, but are also accurately represented.

```
>>> import numpy as np
>>> from pyresample.geometry import SwathDefinition
>>> lons = np.ones((500, 20))
>>> lats = np.ones((500, 20))
>>> swath_def = SwathDefinition(lons=lons, lats=lats)
```

Two swaths can be concatenated if their column count matches

```
>>> lons1 = np.ones((500, 20))
>>> lats1 = np.ones((500, 20))
>>> swath_def1 = SwathDefinition(lons=lons1, lats=lats1)
>>> lons2 = np.ones((300, 20))
>>> lats2 = np.ones((300, 20))
>>> swath_def2 = SwathDefinition(lons=lons2, lats=lats2)
>>> swath_def3 = swath_def1.concatenate(swath_def2)
```

1.2.4 Geographic coordinates and boundaries

All geometry definition objects provide access to longitude and latitude coordinates. The `get_lonlats()` method can be used to get this data and will perform any additional calculations needed to get the coordinates.

`AreaDefinition` exposes the full set of projection coordinates as **projection_x_coords** and **projection_y_coords** properties. Note that for lon/lat projections (*+proj=latlong*) these coordinates will be in longitude/latitude degrees, where **projection_x_coords** will be longitude and **projection_y_coords** will be latitude.

Changed in version 1.5.1: Renamed *proj_x_coords* to *projection_x_coords* and *proj_y_coords* to *projection_y_coords*.

Get longitude and latitude arrays:

```
>>> area_id = 'ease_sh'
>>> description = 'Antarctic EASE grid'
>>> proj_id = 'ease_sh'
>>> projection = '+proj=laea +lat_0=-90 +lon_0=0 +a=6371228.0 +units=m'
>>> width = 425
>>> height = 425
```

(continues on next page)

(continued from previous page)

```
>>> area_extent = (-5326849.0625,-5326849.0625,5326849.0625,5326849.0625)
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
>>> lons, lats = area_def.get_lonlats()
```

Get geocentric X, Y, Z coordinates:

```
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
>>> cart_subset = area_def.get_cartesian_coords()[100:200, 350:]
```

If only the 1D range of a projection coordinate is required it can be extracted using the **projection_x_coord** or **projection_y_coords** property of a geographic coordinate

```
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
>>> proj_x_range = area_def.projection_x_coords
```

1.2.5 Spherical geometry operations

Some basic spherical operations are available for geometry definition objects. The spherical geometry operations are calculated based on the corners of a GeometryDefinition (*GridDefinition*, *AreaDefinition*, or a 2D *SwathDefinition*) assuming the edges are great circle arcs.

Geometries can be checked for overlap:

```
>>> import numpy as np
>>> area_id = 'ease_sh'
>>> description = 'Antarctic EASE grid'
>>> proj_id = 'ease_sh'
>>> projection = '+proj=laea +lat_0=-90 +lon_0=0 +a=6371228.0 +units=m'
>>> width = 425
>>> height = 425
>>> area_extent = (-5326849.0625,-5326849.0625,5326849.0625,5326849.0625)
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
>>> lons = np.array([[-40, -11.1], [9.5, 19.4], [65.5, 47.5], [90.3, 72.3]])
>>> lats = np.array([[-70.1, -58.3], [-78.8, -63.4], [-73, -57.6], [-59.5, -50]])
>>> swath_def = SwathDefinition(lons, lats)
>>> print(swath_def.overlaps(area_def))
True
```

The fraction of overlap can be calculated

```
>>> overlap_fraction = swath_def.overlap_rate(area_def)
>>> overlap_fraction = round(overlap_fraction, 10)
>>> print(overlap_fraction)
0.0584395313
```

And the polygon defining the (great circle) boundaries over the overlapping area can be calculated

```
>>> overlap_polygon = swath_def.intersection(area_def)
>>> print(overlap_polygon)
[(-40.0, -70.1), (-11.1, -58.3), (72.3, -50.0), (90.3, -59.5)]
```

It can be tested if a (lon, lat) point is inside a GeometryDefinition

```
>>> print((0, -90) in area_def)
True
```

1.3 Geometry Utilities

Pyresample provides convenience functions for constructing area definitions. This includes functions for loading `AreaDefinition` from on-disk files, and netCDF/CF files. Some of these utility functions are described below.

1.3.1 AreaDefinition Creation

The main utility function for creating *`AreaDefinition`* objects is the *`create_area_def()`* function. This function will take whatever information can be provided to describe a geographic region and create a valid `AreaDefinition` object if possible. If it can't make a fully specified `AreaDefinition` then it will provide a *`DynamicAreaDefinition`* instead. The function can handle unit conversions and will perform the coordinate calculations necessary to get an area's `shape` and `area_extent`.

The `create_area_def` function has the following required arguments:

- **area_id**: ID of area
- **projection**: Projection parameters as a dictionary or string of PROJ parameters.

and optional arguments:

- **description**: Human-readable description. If not provided, defaults to **area_id**
- **proj_id**: ID of projection (deprecated)
- **units: Units that provided arguments should be interpreted as. This can be** one of 'deg', 'degrees', 'meters', 'metres', and any parameter supported by the cs2cs -lu command. Units are determined in the following priority:
  1. units expressed with each variable through a DataArray's attrs attribute.
  2. units passed to `units`
  3. units used in `projection`
  4. meters
- **shape**: Number of pixels in the y and x direction following row-column format (height, width)
- **area_extent**: Area extent as a tuple (lower_left_x, lower_left_y, upper_right_x, upper_right_y)
- **upper_left_extent**: x and y coordinates of the upper left corner of the upper left pixel (x, y)
- **center**: x and y coordinate of the center of projection (x, y)
- **resolution**: Size of pixels in the x and y direction (dx, dy)
- **radius**: Length from the center to the left/right and top/bottom outer edges (dx, dy)

```
>>> from pyresample import create_area_def
>>> area_id = 'ease_sh'
>>> proj_dict = {'proj': 'laea', 'lat_0': -90, 'lon_0': 0, 'a': 6371228.0, 'units': 'm
↪'}
>>> center = (0, 0)
>>> radius = (5326849.0625, 5326849.0625)
>>> resolution = (25067.525, 25067.525)
```

(continues on next page)

(continued from previous page)

```
>>> area_def = create_area_def(area_id, proj_dict, center=center, radius=radius,␣
↪resolution=resolution)
>>> print(area_def)
Area ID: ease_sh
Description: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

Note: Projection (CRS) information is stored internally using the pyproj library's `CRS` object. To meet certain standards for representing CRS information, pyproj may rename parameters or use completely different parameters from what you provide.

The `create_area_def` function accepts some parameters in multiple forms to make it as easy as possible. For example, the **resolution** and **radius** keyword arguments can be specified with one value if `dx == dy`:

```
>>> proj_string = '+proj=laea +lat_0=-90 +lon_0=0 +a=6371228.0 +units=m'
>>> area_def = create_area_def(area_id, proj_string, center=center,
...                            radius=5326849.0625, resolution=25067.525)
>>> print(area_def)
Area ID: ease_sh
Description: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

You can also specify parameters in degrees even if the projection space is defined in meters. For example the below code creates an area in the mercator projection with radius and resolution defined in degrees.

```
>>> proj_dict = {'proj': 'merc', 'lat_0': 0, 'lon_0': 0, 'a': 6371228.0, 'units': 'm'}
>>> area_def = create_area_def(area_id, proj_dict, center=(0, 0),
...                            radius=(47.90379019311, 43.1355420077),
...                            resolution=(0.22542960090875294, 0.
↪22542901929487608),
...                            units='degrees', description='Antarctic EASE grid')
>>> print(area_def)
Area ID: ease_sh
Description: Antarctic EASE grid
Projection: {'R': '6371228', 'k': '1', 'lon_0': '0', 'no_defs': 'None', 'proj': 'merc
↪', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

The area definition corresponding to a given lat-lon grid (defined by area extent and resolution) can be obtained as follows:

```
>>> area_def = create_area_def('my_area',
...                            {'proj': 'longlat', 'datum': 'WGS84'},
...                            area_extent=[-180, -90, 180, 90],
```

(continues on next page)

(continued from previous page)

```
...                             resolution=1,
...                             units='degrees',
...                             description='Global 1x1 degree lat-lon grid')
>>> print(area_def)
Area ID: my_area
Description: Global 1x1 degree lat-lon grid
Projection: {'datum': 'WGS84', 'no_defs': 'None', 'proj': 'longlat', 'type': 'crs'}
Number of columns: 360
Number of rows: 180
Area extent: (-180.0, -90.0, 180.0, 90.0)
```

If only one of **area_extent** or **shape** can be computed from the information provided by the user, a *DynamicAreaDefinition* object is returned:

```
>>> area_def = create_area_def(area_id, proj_string, radius=radius,␣
↪resolution=resolution)
>>> print(type(area_def))
<class 'pyresample.geometry.DynamicAreaDefinition'>
```

Note: **radius** and **resolution** are distances, **NOT** coordinates. When expressed as angles, they represent the degrees of longitude/latitude away from the center that they should span. Hence in these cases **center or area_extent must be provided**.

1.3.2 AreaDefinition Class Methods

There are four class methods available on the *AreaDefinition* class utilizing *create_area_def()* providing a simpler interface to the functionality described in the previous section. Hence each argument used below is the same as the `create_area_def` arguments described above and can be used in the same way (i.e. units). The following functions require **area_id** and **projection** along with a few other arguments:

from_extent

from_extent()

```
>>> from pyresample.geometry import AreaDefinition
>>> area_id = 'ease_sh'
>>> proj_string = '+proj=laea +lat_0=-90 +lon_0=0 +a=6371228.0 +units=m'
>>> area_extent = (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
>>> shape = (425, 425)
>>> area_def = AreaDefinition.from_extent(area_id, proj_string, shape, area_extent)
>>> print(area_def)
Area ID: ease_sh
Description: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

from_circle

from_circle()

```
>>> proj_dict = {'proj': 'laea', 'lat_0': -90, 'lon_0': 0, 'a': 6371228.0, 'units': 'm
↪'}
>>> center = (0, 0)
>>> radius = 5326849.0625
>>> area_def = AreaDefinition.from_circle(area_id, proj_dict, center, radius,␣
↪shape=shape)
>>> print(area_def)
Area ID: ease_sh
Description: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

```
>>> resolution = 25067.525
>>> area_def = AreaDefinition.from_circle(area_id, proj_string, center, radius,␣
↪resolution=resolution)
>>> print(area_def)
Area ID: ease_sh
Description: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

from_area_of_interest

from_area_of_interest()

```
>>> area_def = AreaDefinition.from_area_of_interest(area_id, proj_dict, shape, center,
↪ resolution)
>>> print(area_def)
Area ID: ease_sh
Description: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

from_ul_corner

from_ul_corner()

```
>>> upper_left_extent = (-5326849.0625, 5326849.0625)
>>> area_def = AreaDefinition.from_ul_corner(area_id, proj_string, shape, upper_left_
↪extent, resolution)
>>> print(area_def)
```

(continues on next page)

(continued from previous page)

```
Area ID: ease_sh
Description: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

1.3.3 Loading from disk

The *load_area()* function can be used to parse area definitions from a configuration file by giving it the area file name and regions you wish to load. *load_area()* takes advantage of *create_area_def()* and hence allows for the same arguments in the on-disk file. Pyresample uses the YAML file format to store on-disk area definitions. Below is an example YAML configuration file showing the various ways an area might be specified.

```
boundary:
  area_id: ease_sh
  description: Example of making an area definition using shape and area_extent
  projection:
    proj: laea
    lat_0: -90
    lon_0: 0
    a: 6371228.0
    units: m
  shape: [425, 425]
  area_extent: [-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625]

boundary_2:
  description: Another example of making an area definition using shape and area_
↪extent
  units: degrees
  projection:
    proj: laea
    lat_0: -90
    lon_0: 0
    a: 6371228.0
    units: m
  shape:
    height: 425
    width: 425
  area_extent:
    lower_left_xy: [-135.0, -17.516001139327766]
    upper_right_xy: [45.0, -17.516001139327766]

corner:
  description: Example of making an area definition using shape, upper_left_extent,␣
↪and resolution
  projection:
    proj: laea
    lat_0: -90
    lon_0: 0
    a: 6371228.0
    units: m
  shape: [425, 425]
  upper_left_extent: [-5326849.0625, 5326849.0625]
```

(continues on next page)

(continued from previous page)

```
  resolution: 25067.525

corner_2:
  area_id: ease_sh
  description: Another example of making an area definition using shape, upper_left_
↪extent, and resolution
  units:  degrees
  projection:
    proj: laea
    lat_0: -90
    lon_0: 0
    a: 6371228.0
    units: m
  shape: [425, 425]
  upper_left_extent:
    x: -45.0
    y: -17.516001139327766
  resolution:
    dx: 25067.525
    dy: 25067.525
    units: meters

circle:
  description: Example of making an area definition using center, resolution, and␣
↪radius
  projection:
    proj: laea
    lat_0: -90
    lon_0: 0
    a: 6371228.0
    units: m
  center: [0, 0]
  resolution: [25067.525, 25067.525]
  radius: 5326849.0625

circle_2:
  area_id: ease_sh
  description: Another example of making an area definition using center, resolution,␣
↪and radius
  projection:
    proj: laea
    lat_0: -90
    lon_0: 0
    a: 6371228.0
    units: m
  center:
    x: 0
    y: -90
    units: degrees
  shape:
    width: 425
    height: 425
  radius:
    dx: 49.4217406986
    dy: 49.4217406986
    units: degrees
```

(continues on next page)

(continued from previous page)

```
area_of_interest:
  description: Example of making an area definition using shape, center, and␣
↪resolution
  projection:
    proj: laea
    lat_0: -90
    lon_0: 0
    a: 6371228.0
    units: m
  shape: [425, 425]
  center: [0, 0]
  resolution: [25067.525, 25067.525]

area_of_interest_2:
  area_id: ease_sh
  description: Another example of making an area definition using shape, center, and␣
↪resolution
  projection:
    proj: laea
    lat_0: -90
    lon_0: 0
    a: 6371228.0
    units: m
  shape: [425, 425]
  center:
    center: [0, -90]
    units: deg
  resolution:
    resolution: 0.22542974631297721
    units: deg

epsg:
  area_id: ease_sh
  description: Example of making an area definition using EPSG codes
  projection:
    init: EPSG:3410
  shape: [425, 425]
  area_extent: [-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625]
```

Note: The *lower_left_xy* and *upper_right_xy* items give the coordinates of the outer edges of the corner pixels on the x and y axis respectively. When the projection coordinates are longitudes and latitudes, it is expected to provide the extent in *longitude, latitude* order.

Note: When using pyproj 2.0+, please use the new `'EPSG: XXXX'` syntax as the old `'init: EPSG:XXXX'` is no longer supported.

If we assume the YAML content is stored in an `areas.yaml` file, we can read a single `AreaDefinition` named `corner` by doing:

```
>>> from pyresample import load_area
>>> import yaml
>>> area_def = load_area('areas.yaml', 'corner')
>>> print(area_def)
```

(continues on next page)

(continued from previous page)

```
Area ID: corner
Description: Example of making an area definition using shape, upper_left_extent, and␣
↪resolution
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

Several area definitions can be read at once using the region names as a series of arguments:

```
>>> corner, boundary = load_area('areas.yaml', 'corner', 'boundary')
>>> print(boundary)
Area ID: ease_sh
Description: Example of making an area definition using shape and area_extent
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

1.3.4 Loading from disk (legacy)

For backwards compatibility, we still support the legacy area file format. Assuming the file **areas.cfg** exists with the following content

```
REGION: ease_sh {
   NAME:           Antarctic EASE grid
   PCS_ID:         ease_sh
       PCS_DEF:        proj=laea, lat_0=-90, lon_0=0, a=6371228.0, units=m
       XSIZE:          425
       YSIZE:          425
       AREA_EXTENT:    (-5326849.0625,-5326849.0625,5326849.0625,5326849.0625)
};

REGION: ease_nh {
       NAME:           Arctic EASE grid
       PCS_ID:         ease_nh
       PCS_DEF:        proj=laea, lat_0=90, lon_0=0, a=6371228.0, units=m
       XSIZE:          425
       YSIZE:          425
       AREA_EXTENT:    (-5326849.0625,-5326849.0625,5326849.0625,5326849.0625)
};
```

An area definition dict can be read using

```
>>> from pyresample import load_area
>>> area = load_area('areas.cfg', 'ease_nh')
>>> print(area)
Area ID: ease_nh
Description: Arctic EASE grid
Projection ID: ease_nh
Projection: {'R': '6371228', 'lat_0': '90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
```

(continues on next page)

(continued from previous page)

```
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

Note: In the configuration file **REGION** maps to **area_id** and **PCS_ID** maps to **proj_id**.

Several area definitions can be read at once using the region names in an argument list:

```
>>> nh_def, sh_def = load_area('areas.cfg', 'ease_nh', 'ease_sh')
>>> print(sh_def)
Area ID: ease_sh
Description: Antarctic EASE grid
Projection ID: ease_sh
Projection: {'R': '6371228', 'lat_0': '-90', 'lon_0': '0', 'no_defs': 'None', 'proj':
↪'laea', 'type': 'crs', 'units': 'm', 'x_0': '0', 'y_0': '0'}
Number of columns: 425
Number of rows: 425
Area extent: (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
```

1.3.5 Loading from netCDF/CF

`AreaDefinition` objects can be loaded from netCDF CF files with function *`load_cf_area()`*.

```
>>> from pyresample.utils import load_cf_area
```

The *`load_cf_area()`* routine offers three call forms:

- Load the `AreaDefinition` from a specific CF *grid_mapping* object: with all three of `variable=`, `x=`, and `y=` ;
- Load the `AreaDefinition` sustaining a CF variable: only `variable=` ;
- Find and load the valid `AreaDefinition` in a CF file: no parameter ;

Consider the following netCDF/CF file:

```
netcdf cf_nh10km {
dimensions:
     xc = 760 ;
     yc = 1120 ;
variables:
     int Polar_Stereographic_Grid ;
             Polar_Stereographic_Grid:grid_mapping_name = "polar_stereographic" ;
             Polar_Stereographic_Grid:false_easting = 0. ;
             Polar_Stereographic_Grid:false_northing = 0. ;
             Polar_Stereographic_Grid:semi_major_axis = 6378273. ;
             Polar_Stereographic_Grid:semi_minor_axis = 6356889.44891 ;
             Polar_Stereographic_Grid:straight_vertical_longitude_from_pole = -45. ;
             Polar_Stereographic_Grid:latitude_of_projection_origin = 90. ;
             Polar_Stereographic_Grid:standard_parallel = 70. ;
     double xc(xc) ;
             xc:axis = "X" ;
             xc:units = "km" ;
             xc:long_name = "x coordinate in Cartesian system" ;
             xc:standard_name = "projection_x_coordinate" ;
     double yc(yc) ;
             yc:axis = "Y" ;
```

(continues on next page)

(continued from previous page)

```
            yc:units = "km" ;
            yc:long_name = "y coordinate in Cartesian system" ;
            yc:standard_name = "projection_y_coordinate" ;
    float lat(yc, xc) ;
            lat:long_name = "latitude coordinate" ;
            lat:standard_name = "latitude" ;
            lat:units = "degrees_north" ;
    float lon(yc, xc) ;
            lon:long_name = "longitude coordinate" ;
            lon:standard_name = "longitude" ;
            lon:units = "degrees_east" ;
    short ice_conc(yc, xc) ;
            ice_conc:_FillValue = -999s ;
            ice_conc:grid_mapping = "Polar_Stereographic_Grid" ;
            ice_conc:coordinates = "lat lon" ;
            ice_conc:standard_name = "sea_ice_area_fraction" ;
            ice_conc:units = "%" ;
            ice_conc:scale_factor = 0.01f ;
            ice_conc:add_offset = 0.f ;
            ice_conc:valid_min = 0 ;
            ice_conc:valid_max = 10000 ;
// global attributes:
            :Conventions = "CF-1.7"

}
```

The three call forms are:

1st call form:

```
>>> area_def, cf_info = load_cf_area('/path/to/cf_nh10km.nc', variable='Polar_
↪Stereographic_Grid', x='xc', y='yc')
```

This will directly create the AreaDefinition `area_def` from the content of the *grid_mapping* variable 'Polar_Stereographic_Grid', and the area extent from the 'xc' and 'yc'.

2nd call form:

```
>>> area_def, cf_info = load_cf_area('/path/to/cf_nh10km.nc', variable='ice_conc')
```

This will search which *grid_mapping*, *x* and *y* axes sustain the 'ice_conc' variable, and create the `AreaDefinition` from this information.

3rd call form:

```
>>> area_def, cf_info = load_cf_area('/path/to/cf_nh10km.nc')
```

This will look through the whole netCDF/CF file, and guess all information needed to load a `AreaDefinition` object.

Note: The CF convention allows that a single file defines several different *grid_mappings*. At present, the 3rd call form of `load_cf_area()` will raise a `ValueError` exception when this happens.

If you have several *grid_mappings* in your CF file, be specific which one you want to access with the 1st or 2nd call form.

Although a recommended practice, it cannot be trusted that the 'y' and 'x' axes are in the last two dimensions of a CF variable. This is because the CF convention does not impose the order of the dimensions of a variable. `load_cf_area()` will effectively look for the variables holding the *x* and *y* coordinates of the Earth mapping projection, not based on the order of the dimensions of the CF variable.

Access to additional info from the CF file:

Not all relevant information can be stored in the `AreaDefinition` object. For example, it can be useful to know what were the names of the variables holding the coordinate variables ('xc' and 'yc' in the example above), or that the *latitude* and *longitude* associated to the *grid_mapping* are stored in variables 'lat' and 'lon'. Such information can be useful for writing additional variables to the CF file, or to create a new file that looks similar to the one we just read.

This information is in a second return value `cf_info`:

```
>>> area_def, cf_info = load_cf_area('/path/to/cf_nh10km.nc', with_cf_info=True)
```

The `cf_info` is a `dict()` holding additional information about the way the *grid_mapping* information was coded in the CF file. It may not contain the same amount of information in all three call forms. For example, the 1st call form does not allow to find the name of the *latitude* or *longitude* variables, since the 1st call form only gives access to the *grid_mapping* variable and its coordinate axes.

1.4 Geographic filtering

The module **pyresample.geo_filter** contains classes to filter geo data

1.4.1 GridFilter

Allows for filtering of data based on a geographic mask. The filtering uses a bucket sampling approach.

The following example shows how to select data falling in the upper left and lower right quadrant of a full globe Plate Carrée projection using an 8x8 filter mask

```
>>> import numpy as np
>>> from pyresample import geometry, geo_filter
>>> lons = np.array([-170, -30, 30, 170])
>>> lats = np.array([20, -40, 50, -80])
>>> swath_def = geometry.SwathDefinition(lons, lats)
>>> data = np.array([1, 2, 3, 4])
>>> filter_area = geometry.AreaDefinition('test', 'test', 'test',
...         {'proj' : 'eqc', 'lon_0' : 0.0, 'lat_0' : 0.0},
...           8, 8,
...          (-20037508.34, -10018754.17, 20037508.34, 10018754.17)
...             )
>>> filter = np.array([[1, 1, 1, 1, 0, 0, 0, 0],
...         [1, 1, 1, 1, 0, 0, 0, 0],
...         [1, 1, 1, 1, 0, 0, 0, 0],
...         [1, 1, 1, 1, 0, 0, 0, 0],
...         [0, 0, 0, 0, 1, 1, 1, 1],
...         [0, 0, 0, 0, 1, 1, 1, 1],
...         [0, 0, 0, 0, 1, 1, 1, 1],
...         [0, 0, 0, 0, 1, 1, 1, 1],
...         ])
>>> grid_filter = geo_filter.GridFilter(filter_area, filter)
>>> swath_def_filtered, data_filtered = grid_filter.filter(swath_def, data)
```

Input swath_def and data must match as described in *Resampling of swath data*.

The returned data will always have a 1D geometry_def and if multiple channels are present the filtered data will have the shape (number_of_points, channels).

1.5 Resampling of gridded data

Pyresample can be used to resample from an existing grid to another. Nearest neighbour resampling is used.

1.5.1 pyresample.image

A grid can be stored in an object of type **ImageContainer** along with its area definition. An object of type **ImageContainer** allows for calculating resampling using preprocessed arrays using the method **get_array_from_linesample**

Resampling can be done using descendants of **ImageContainer** and calling their **resample** method.

An **ImageContainerQuick** object allows for the grid to be resampled to a new area defintion using an approximate (but fast) nearest neighbour method. Resampling an object of type **ImageContainerQuick** returns a new object of type **ImageContainerQuick**.

An **ImageContainerNearest** object allows for the grid to be resampled to a new area defintion (or swath definition) using an accurate kd-tree method. Resampling an object of type **ImageContainerNearest** returns a new object of type **ImageContainerNearest**.

```
>>> import numpy as np
>>> from pyresample import image, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> msg_area = geometry.AreaDefinition('msg_full', 'Full globe MSG image 0 degrees',
...                                    'msg_full',
...                                    {'a': '6378169.0', 'b': '6356584.0',
...                                     'h': '35785831.0', 'lon_0': '0',
...                                     'proj': 'geos'},
...                                    3712, 3712,
...                                    [-5568742.4, -5568742.4,
...                                     5568742.4, 5568742.4])
>>> data = np.ones((3712, 3712))
>>> msg_con_quick = image.ImageContainerQuick(data, msg_area)
>>> area_con_quick = msg_con_quick.resample(area_def)
>>> result_data_quick = area_con_quick.image_data
>>> msg_con_nn = image.ImageContainerNearest(data, msg_area, radius_of_
↪influence=50000)
>>> area_con_nn = msg_con_nn.resample(area_def)
>>> result_data_nn = area_con_nn.image_data
```

Data is assumed to be a numpy array of shape (rows, cols) or (rows, cols, channels).

Masked arrays can be used as data input. In order to have undefined pixels masked out instead of assigned a fill value set **fill_value=None** when calling **resample_area_***.

Using **ImageContainerQuick** the risk of image artifacts increases as the distance from source projection center increases.

The constructor argument **radius_of_influence** to **ImageContainerNearest** specifices the maximum distance to search for a neighbour for each point in the target grid. The unit is meters.

The constructor arguments of an ImageContainer object can be changed as attributes later

```
>>> import numpy as np
>>> from pyresample import image, geometry
>>> msg_area = geometry.AreaDefinition('msg_full', 'Full globe MSG image 0 degrees',
...                                    'msg_full',
...                                    {'a': '6378169.0', 'b': '6356584.0',
...                                     'h': '35785831.0', 'lon_0': '0',
...                                     'proj': 'geos'},
...                                    3712, 3712,
...                                    [-5568742.4, -5568742.4,
...                                     5568742.4, 5568742.4])
>>> data = np.ones((3712, 3712))
>>> msg_con_nn = image.ImageContainerNearest(data, msg_area, radius_of_
↪influence=50000)
>>> msg_con_nn.radius_of_influence = 45000
>>> msg_con_nn.fill_value = -99
```

Multi channel images

If the dataset has several channels the last index of the data array specifies the channels

```
>>> import numpy as np
>>> from pyresample import image, geometry
>>> msg_area = geometry.AreaDefinition('msg_full', 'Full globe MSG image 0 degrees',
...                                    'msg_full',
...                                    {'a': '6378169.0', 'b': '6356584.0',
...                                     'h': '35785831.0', 'lon_0': '0',
...                                     'proj': 'geos'},
...                                    3712, 3712,
...                                    [-5568742.4, -5568742.4,
...                                     5568742.4, 5568742.4])
>>> channel1 = np.ones((3712, 3712))
>>> channel2 = np.ones((3712, 3712)) * 2
>>> channel3 = np.ones((3712, 3712)) * 3
>>> data = np.dstack((channel1, channel2, channel3))
>>> msg_con_nn = image.ImageContainerNearest(data, msg_area, radius_of_
↪influence=50000)
```

Segmented resampling

Pyresample calculates the result in segments in order to reduce memory footprint. This is controlled by the **segments** contructor keyword argument. If no **segments** argument is given pyresample will estimate the number of segments to use.

Forcing quick resampling to use 4 resampling segments:

```
>>> import numpy as np
>>> from pyresample import image, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
```

(continues on next page)

(continued from previous page)

```
...                                  'lon_0': '8.00', 'proj': 'stere'},
...                                 800, 800,
...                                 [-1370912.72, -909968.64,
...                                  1029087.28, 1490031.36])
>>> msg_area = geometry.AreaDefinition('msg_full', 'Full globe MSG image 0 degrees',
...                                 'msg_full',
...                                 {'a': '6378169.0', 'b': '6356584.0',
...                                  'h': '35785831.0', 'lon_0': '0',
...                                  'proj': 'geos'},
...                                 3712, 3712,
...                                 [-5568742.4, -5568742.4,
...                                  5568742.4, 5568742.4])
>>> data = np.ones((3712, 3712))
>>> msg_con_quick = image.ImageContainerQuick(data, msg_area, segments=4)
>>> area_con_quick = msg_con_quick.resample(area_def)
```

Constructor arguments

The full list of constructor arguments:

ImageContainerQuick:

- image_data : Dataset. Masked arrays can be used.
- geo_def : Geometry definition.
- fill_value (optional) : Fill value for undefined pixels. Defaults to 0. If set to **None** they will be masked out.
- nprocs (optional) : Number of processor cores to use. Defaults to 1.
- segments (optional) : Number of segments to split resampling in. Defaults to auto estimation.

ImageContainerNearest:

- image_data : Dataset. Masked arrays can be used.
- geo_def : Geometry definition.
- radius_of_influence : Cut off radius in meters when considering neighbour pixels.
- epsilon (optional) : The distance to a found value is guaranteed to be no further than (1 + eps) times the distance to the correct neighbour.
- fill_value (optional) : Fill value for undefined pixels. Defaults to 0. If set to **None** they will be masked out.
- reduce_data (optional) : Apply geographic reduction of dataset before resampling. Defaults to True
- nprocs (optional) : Number of processor cores to use. Defaults to 1.
- segments (optional) : Number of segments to split resampling in. Defaults to auto estimation.

Preprocessing of grid resampling

For preprocessing of grid resampling see *Preprocessing of grids*

1.6 Resampling of swath data

Pyresample can be used to resample a swath dataset to a grid, a grid to a swath or a swath to another swath. Resampling can be done using nearest neighbour method, Guassian weighting, weighting with an arbitrary radial function.

Changed in version 1.8.0: *SwathDefinition* no longer checks the validity of the provided longitude and latitude coordinates to improve performance. Longitude arrays are expected to be between -180 and 180 degrees, latitude -90 to 90 degrees. This also applies to all geometry definitions that are provided longitude and latitude arrays on initialization. Use *check_and_wrap()* to preprocess your arrays.

1.6.1 pyresample.image

The ImageContainerNearest and ImageContanerBilinear classes can be used for resampling of swaths as well as grids. Below is an example using nearest neighbour resampling.

```
>>> import numpy as np
>>> from pyresample import image, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> swath_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> swath_con = image.ImageContainerNearest(data, swath_def, radius_of_influence=5000)
>>> area_con = swath_con.resample(area_def)
>>> result = area_con.image_data
```

For other resampling types or splitting the process in two steps use e.g. the functions in **pyresample.kd_tree** described below.

1.6.2 pyresample.kd_tree

This module contains several functions for resampling swath data.

Note distance calculation is approximated with cartesian distance.

Masked arrays can be used as data input. In order to have undefined pixels masked out instead of assigned a fill value set **fill_value=None** when calling the **resample_*** function.

resample_nearest

Function for resampling using nearest neighbour method.

Example showing how to resample a generated swath dataset to a grid using nearest neighbour method:

```
>>> import numpy as np
>>> from pyresample import kd_tree, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
```

(continues on next page)

(continued from previous page)

```
...                                  'lat_0': '50.00', 'lat_ts': '50.00',
...                                  'lon_0': '8.00', 'proj': 'stere'},
...                                 800, 800,
...                                 [-1370912.72, -909968.64,
...                                  1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> swath_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> result = kd_tree.resample_nearest(swath_def, data,
... area_def, radius_of_influence=50000, epsilon=0.5)
```

If the arguments **swath_def** and **area_def** where switched (and **data** matched the dimensions of **area_def**) the grid of **area_def** would be resampled to the swath defined by **swath_def**.

Note the keyword arguments:

- **radius_of_influence**: The radius around each grid pixel in meters to search for neighbours in the swath.
- **epsilon**: The distance to a found value is guaranteed to be no further than (1 + eps) times the distance to the correct neighbour. Allowing for uncertanty decreases execution time.

If **data** is a masked array the mask will follow the neighbour pixel assignment.

If there are multiple channels in the dataset the **data** argument should be of the shape of the lons and lat arrays with the channels along the last axis e.g. (rows, cols, channels). Note: the convention of pyresample < 0.7.4 is to pass **data** in the form of (number_of_data_points, channels) is still accepted.

```
>>> import numpy as np
>>> from pyresample import kd_tree, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> channel1 = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> channel2 = np.fromfunction(lambda y, x: y*x, (50, 10)) * 2
>>> channel3 = np.fromfunction(lambda y, x: y*x, (50, 10)) * 3
>>> data = np.dstack((channel1, channel2, channel3))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> swath_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> result = kd_tree.resample_nearest(swath_def, data,
... area_def, radius_of_influence=50000)
```

For nearest neighbour resampling the class **image.ImageContainerNearest** can be used as well as **kd_tree.resample_nearest**

resample_gauss

Function for resampling using nearest Gussian weighting. The Gauss weigh function is defined as exp(-dist^2/sigma^2). Note the pyresample sigma is **not** the standard deviation of the gaussian. Example showing how to resample a generated swath dataset to a grid using Gaussian weighting:

```
>>> import numpy as np
>>> from pyresample import kd_tree, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> swath_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> result = kd_tree.resample_gauss(swath_def, data,
... area_def, radius_of_influence=50000, sigmas=25000)
```

If more channels are present in **data** the keyword argument **sigmas** must be a list containing a sigma for each channel.

If **data** is a masked array any pixel in the result data that has been "contaminated" by weighting of a masked pixel is masked.

Using the function **utils.fwhm2sigma** the sigma argument to the gauss resampling can be calculated from 3 dB FOV levels.

resample_custom

Function for resampling using arbitrary radial weight functions.

Example showing how to resample a generated swath dataset to a grid using an arbitrary radial weight function:

```
>>> import numpy as np
>>> from pyresample import kd_tree, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> swath_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> wf = lambda r: 1 - r/100000.0
>>> result  = kd_tree.resample_custom(swath_def, data,
...  area_def, radius_of_influence=50000, weight_funcs=wf)
```

If more channels are present in **data** the keyword argument **weight_funcs** must be a list containing a radial function for each channel.

If **data** is a masked array any pixel in the result data that has been "contaminated" by weighting of a masked pixel is masked.

Uncertainty estimates

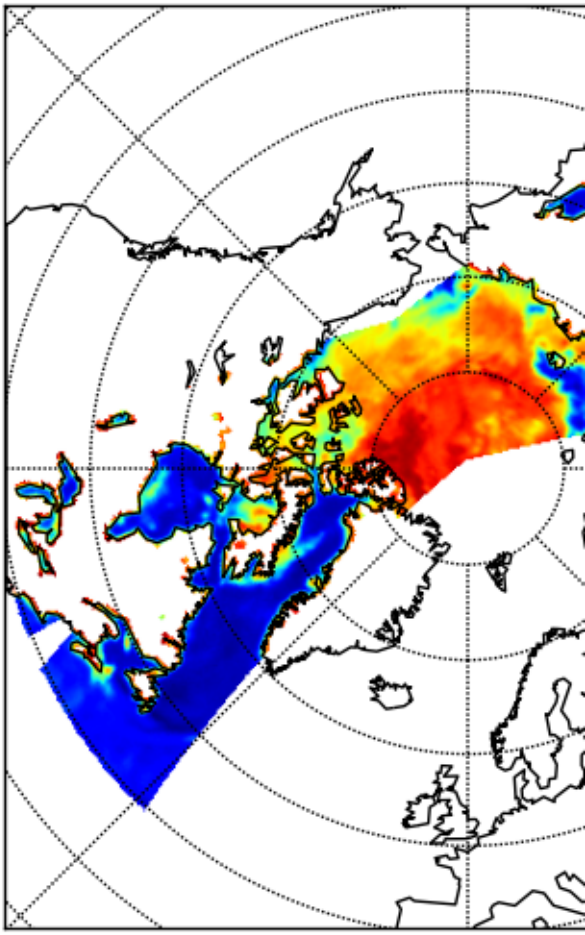
Uncertainty estimates in the form of weighted standard deviation can be obtained from the **resample_custom** and **resample_gauss** functions. By default the functions return the result of the resampling as a single numpy array. If the

functions are given the keyword argument **with_uncert=True** then the following list of numpy arrays will be returned instead: **(result, stddev, count)**. **result** is the usual result. **stddev** is the weighted standard deviation for each element in the result. **count** is the number of data values used in the weighting for each element in the result.

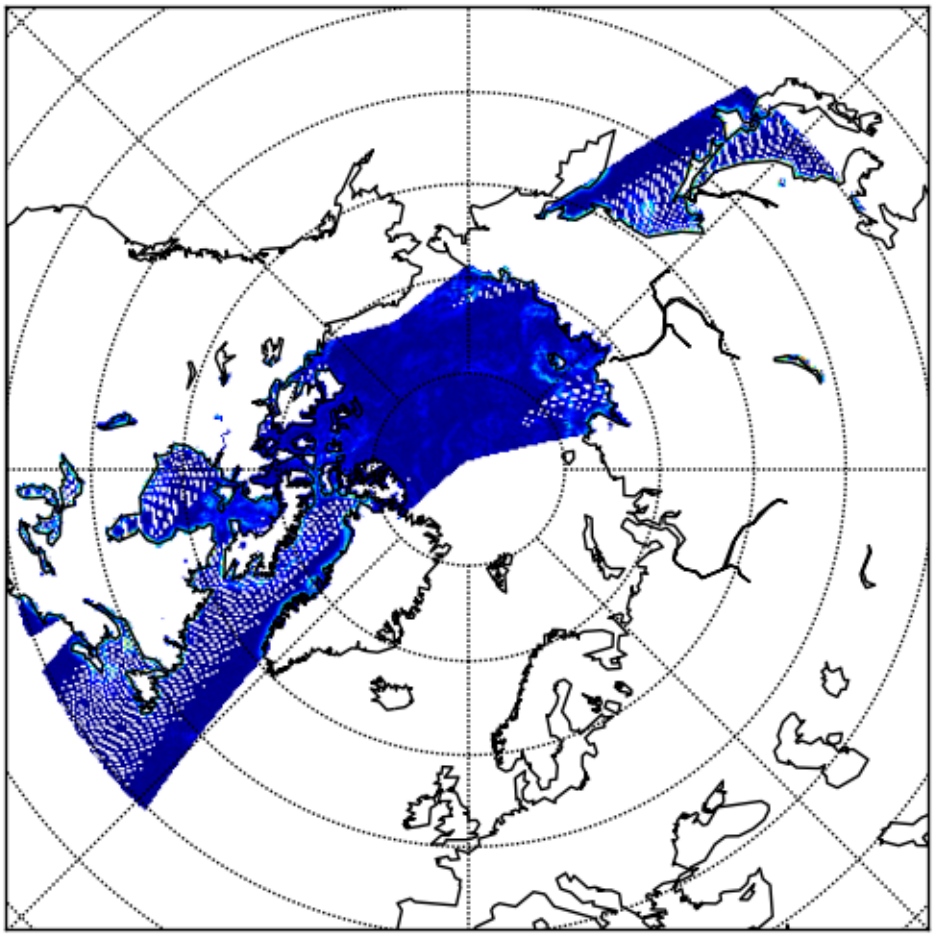
The principle is to view the calculated value for each element in the result as a weighted average of values sampled from a statistical variable. An estimate of the standard deviation of the distribution is calculated using the unbiased weighted estimator given as **stddev = sqrt((V1 / (V1 ** 2 + V2)) * sum(wi * (xi - result) ** 2))** where **result** is the result of the resampling. **xi** is the value of a contributing neighbour and **wi** is the corresponding weight. The coefficients are given as **V1 = sum(wi)** and **V2 = sum(wi ** 2)**. The standard deviation is only calculated for elements in the result where more than one neighbour has contributed to the weighting. The **count** numpy array can be used for filtering at a higher number of contributing neigbours.

Usage only differs in the number of return values from **resample_gauss** and **resample_custom**. E.g.:

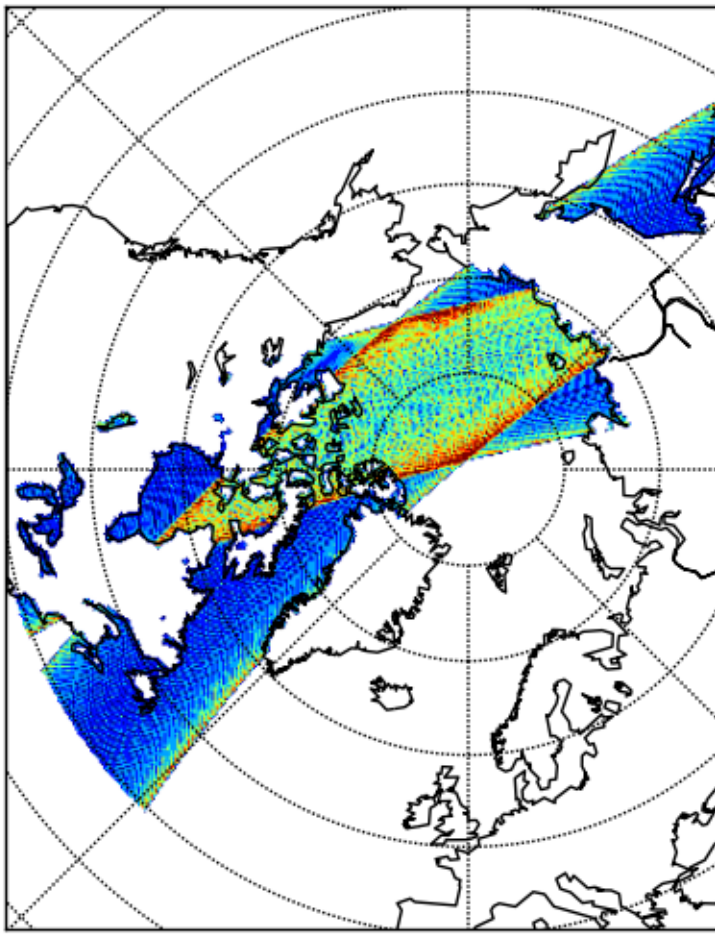
```
>>> result, stddev, count = pr.kd_tree.resample_gauss(swath_def, ice_conc, area_def,
...                                                   radius_of_influence=20000,
...                                                   sigmas=pr.utils.
↪fwhm2sigma(35000),
...                                                   fill_value=None, with_
↪uncert=True)
```



Below is shown a plot of the result of the resampling using a real data set:



The corresponding standard deviations:



And the number of contributing neighbours for each element:

Notice the standard deviation is only calculated where there are more than one contributing neighbour.

Resampling from neighbour info

The resampling can be split in two steps:

First get arrays containing information about the nearest neighbours to each grid point. Then use these arrays to retrive the resampling result.

This approch can be useful if several datasets based on the same swath are to be resampled. The computational heavy task of calculating the neighbour information can be done once and the result can be used to retrieve the resampled data from each of the datasets fast.

```
>>> import numpy as np
>>> from pyresample import kd_tree, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
```

(continues on next page)

(continued from previous page)

```
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> swath_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> valid_input_index, valid_output_index, index_array, distance_array = \
...                         kd_tree.get_neighbour_info(swath_def,
...                                                    area_def, 50000,
...                                                   neighbours=1)
>>> res = kd_tree.get_sample_from_neighbour_info('nn', area_def.shape, data,
...                                              valid_input_index, valid_output_
↪index,
...                                              index_array)
```

Note the keyword argument **neighbours=1**. This specifies only to consider one neighbour for each grid point (the nearest neighbour). Also note **distance_array** is not a required argument for **get_sample_from_neighbour_info** when using nearest neighbour resampling

Segmented resampling

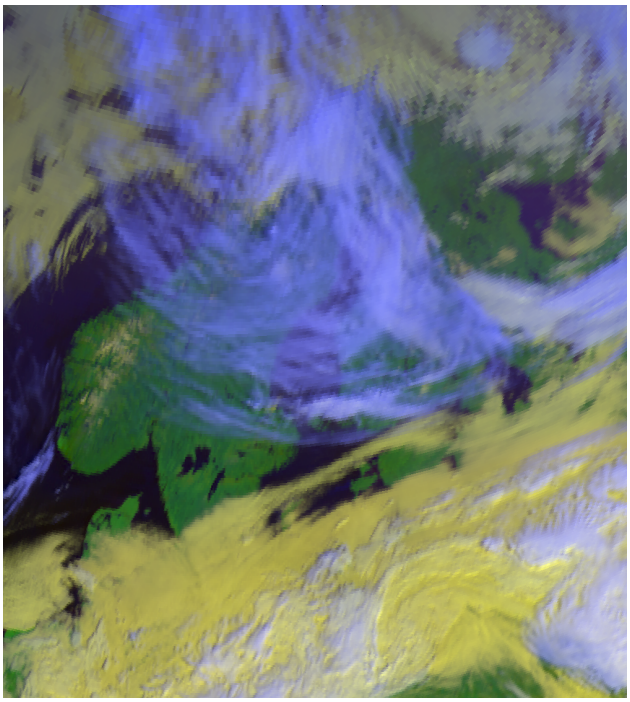
Whenever a resampling function takes the keyword argument **segments** the number of segments to split the resampling process in can be specified. This affects the memory footprint of pyresample. If the value of **segments** is left to default pyresample will estimate the number of segments to use.

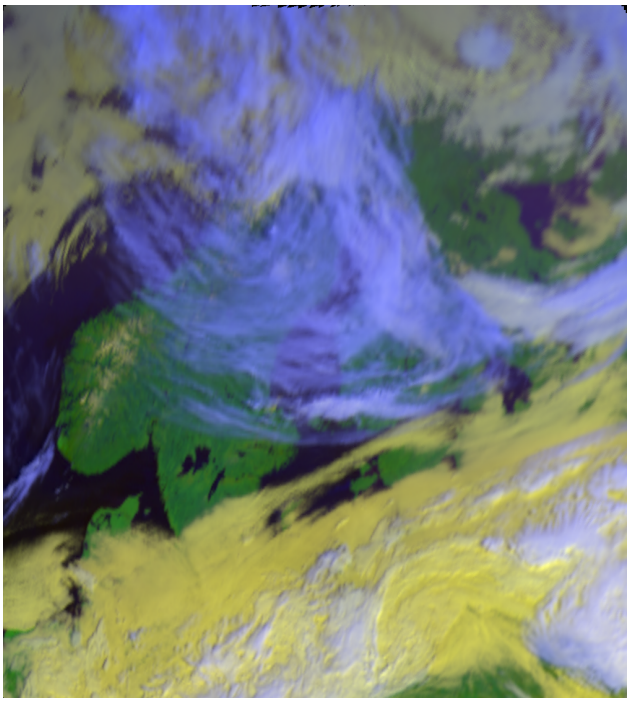
1.6.3 pyresample.bilinear

Compared to nearest neighbour resampling, bilinear interpolation produces smoother results near swath edges of polar satellite data and edges of geostationary satellites.

The algorithm is implemented from http://www.ahinson.com/algorithms_general/Sections/InterpolationRegression/InterpolationIrregularBilinear.pdf

Below is shown a comparison between image generated with nearest neighbour resampling (top) and with bilinear interpolation (bottom):





Click images to see the full resolution versions.

The *perceived* sharpness of the bottom image is lower, but there is more detail present.

resample_bilinear

Function for resampling using bilinear interpolation for irregular source grids.

```
>>> import numpy as np
>>> from pyresample import bilinear, geometry
>>> target_def = geometry.AreaDefinition('areaD',
...                                      'Europe (3km, HRV, VTC)',
...                                      'areaD',
...                                      {'a': '6378144.0', 'b': '6356759.0',
...                                       'lat_0': '50.00', 'lat_ts': '50.00',
...                                       'lon_0': '8.00', 'proj': 'stere'},
...                                      800, 800,
...                                      [-1370912.72, -909968.64,
...                                       1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (500, 100))
>>> lons = np.fromfunction(lambda y, x: 3 + x * 0.1, (500, 100))
>>> lats = np.fromfunction(lambda y, x: 75 - y * 0.1, (500, 100))
>>> source_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> result = bilinear.resample_bilinear(data, source_def, target_def,
...                                     radius=50e3, neighbours=32,
...                                     nprocs=1, fill_value=0,
...                                     reduce_data=True, segments=None,
...                                     epsilon=0)
```

The **target_area** needs to be an area definition with **proj_str** attribute.

Keyword arguments which are passed to **kd_tree**:

- **radius**: radius around each target pixel in meters to search for neighbours in the source data

- **neighbours**: number of closest locations to consider when selecting the four data points around the target location. Note that this value needs to be large enough to ensure "surrounding" the target!
- **nprocs**: number of processors to use for finding the closest pixels
- **fill_value**: fill invalid pixel with this value. If **fill_value=None** is used, masked arrays will be returned
- **reduce_data**: do/don't do preliminary data reduction before calculating the neigbour info
- **segments**: number of segments to use in neighbour search
- **epsilon**: maximum uncertainty allowed in neighbour search

The example above shows the default value for each keyword argument.

Resampling from bilinear coefficients

As for nearest neighbour resampling, also bilinear interpolation can be split in two steps.

- Calculate interpolation coefficients, input data reduction matrix and mapping matrix
- Use this information to resample several datasets between these two areas/swaths

Only the first step is computationally expensive operation, so by re-using this information the overall processing time is reduced significantly. This is also done internally by the **resample_bilinear** function, but separating these steps makes it possible to cache the coefficients if the same transformation is done over and over again. This is very typical in operational geostationary satellite image processing. Note that the output shape is now defined so that the result is reshaped to correct shape. This reshaping is done internally in **resample_bilinear**.

```
>>> import numpy as np
>>> from pyresample import bilinear, geometry
>>> target_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)',
...                                      'areaD',
...                                      {'a': '6378144.0', 'b': '6356759.0',
...                                       'lat_0': '50.00', 'lat_ts': '50.00',
...                                       'lon_0': '8.00', 'proj': 'stere'},
...                                      800, 800,
...                                      [-1370912.72, -909968.64,
...                                       1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> source_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> t_params, s_params, input_idxs, idx_ref = \
...     bilinear.get_bil_info(source_def, target_def, radius=50e3, nprocs=1)
>>> res = bilinear.get_sample_from_bil_info(data.ravel(), t_params, s_params,
...                                         input_idxs, idx_ref,
...                                         output_shape=target_def.shape)
```

1.6.4 pyresample.ewa

Pyresample makes it possible to resample swath data to a uniform grid using an Elliptical Weighted Averaging algorithm or EWA for short. This algorithm behaves differently than the KDTree based resampling algorithms that pyresample provides. The KDTree-based algorithms process each output grid pixel by searching for all "nearby" input pixels and applying a certain interpolation (nearest neighbor, gaussian, etc). The EWA algorithm processes each input pixel mapping it to one or more output pixels. Once each input pixel has been analyzed the intermediate results are averaged to produce the final gridded result.

The EWA algorithm also has limitations on how the input data is structured compared to the generic KDTree algorithms. EWA assumes that data in the array is organized geographically; adjacent data in the array is adjacent data geographically. The algorithm uses this to configure parameters based on the size and location of the swath pixels.

The EWA algorithm consists of two steps: ll2cr and fornav. The algorithm was originally part of the MODIS Swath to Grid Toolbox (ms2gt) created by the NASA National Snow & Ice Data Center (NSIDC). Its default parameters work best with MODIS L1B data, but it has been proven to produce high quality images from VIIRS and AVHRR data with the right parameters.

Note: This code was originally part of the Polar2Grid project. This documentation and the API documentation for this algorithm may still use references or concepts from Polar2Grid until everything can be updated.

Gridding

The first step is called 'll2cr' which stands for "longitude/latitude to column/row". This step maps the pixel location (lon/lat space) into area (grid) space. Areas in pyresample are defined by a PROJ.4 projection specification. An area is defined by the following parameters:

- Grid Name
- PROJ.4 String (either lat/lon or metered projection space)
- Width (number of pixels in the X direction)
- Height (number of pixels in the Y direction)
- Cell Width (pixel size in the X direction in grid units)
- Cell Height (pixel size in the Y direction in grid units)
- X Origin (upper-left X coordinate in grid units)
- Y Origin (upper-left Y coordinate in grid units)

Resampling

The second step of EWA remapping is called "fornav", short for "forward navigation". This EWA algorithm processes one input scan line at a time. The algorithm weights the effect of an input pixel on an output pixel based on its location in the scan line and other calculated coefficients. It can also handle swaths that are not scan based by specifying *rows_per_scan* as the number of rows in the entire swath. How the algorithm treats the data can be configured with various keyword arguments, see the API documentation for more information. Both steps provide additional information to inform the user how much data was used in the result. The first returned value of ll2cr tells you how many of the input swath pixels overlap the grid. The first returned value of fornav tells you how many grid points have valid data values in them.

Example

Note: EWA resampling in pyresample is still in an alpha stage. As development continues, EWA resampling may be called differently.

```
>>> import numpy as np
>>> from pyresample.ewa import ll2cr, fornav
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> swath_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> # ll2cr converts swath longitudes and latitudes to grid columns and rows
>>> swath_points_in_grid, cols, rows = ll2cr(swath_def, area_def)
>>> # if the data is scan based, specify how many data rows make up one scan
>>> rows_per_scan = 5
>>> # fornav resamples the swath data to the gridded area
>>> num_valid_points, gridded_data = fornav(cols, rows, area_def, data, rows_per_
↪scan=rows_per_scan)
```

1.6.5 pyresample.bucket

See `BucketResampler` API documentation for the details of method parameters.

1.7 Using multiple processor cores

1.7.1 Multi core processing

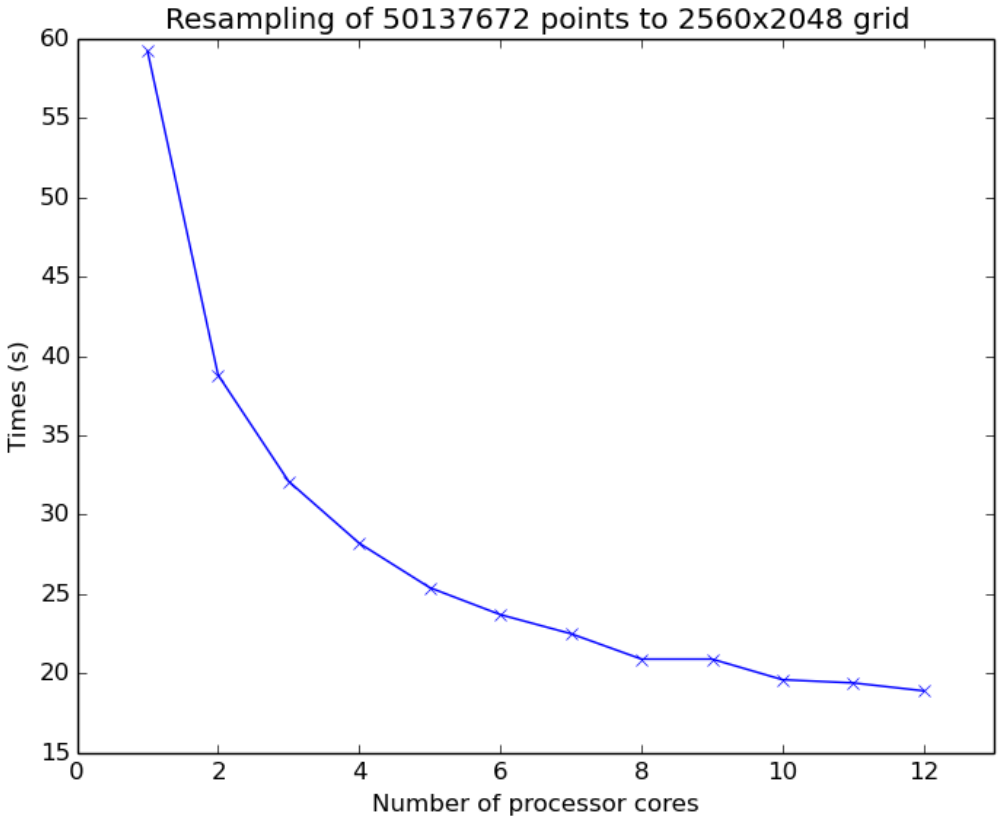
Bottlenecks of pyresample can be executed in parallel. Parallel computing can be executed if the pyresample function has the **nprocs** keyword argument. **nprocs** specifies the number of processes to be used for calculation. If a class takes the constructor argument **nprocs** this sets **nprocs** for all methods of this class

Example of resampling in parallel using 4 processes:

```
>>> import numpy
>>> from pyresample import kd_tree, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> data = numpy.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = numpy.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = numpy.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> swath_def = geometry.SwathDefinition(lons=lons, lats=lats)
>>> result = kd_tree.resample_nearest(swath_def, data.ravel(),
... area_def, radius_of_influence=50000, nprocs=4)
```

Note: Do not use more processes than available processor cores. As there is a process creation overhead there might be neglible performance improvement using say 8 compared to 4 processor cores. Test on the actual system to determine the most sensible number of processes to use.

Here is an example of the performance for a varying number of processors on a 64-bit ubuntu 14.04, 32 GB RAM, 2 x Intel Xeo



1.8 Preprocessing of grids

When resampling is performed repeatedly to the same grid significant execution time can be save by preprocessing grid information.

1.8.1 Preprocessing for grid resampling

Using the function **generate_quick_linesample_arrays** or **generate_nearest_neighbour_linesample_arrays** from **pyresample.utils** arrays containing the rows and cols indices used to calculate the result in **image.resample_area_quick** or **resample_area_nearest_neighbour** can be obtained. These can be fed to the method **get_array_from_linesample** of an **ImageContainer** object to obtain the resample result.

```
>>> import numpy
>>> from pyresample import utils, image, geometry
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
```

(continues on next page)

(continued from previous page)

```
>>> msg_area = geometry.AreaDefinition('msg_full', 'Full globe MSG image 0 degrees',
...                                    'msg_full',
...                                    {'a': '6378169.0', 'b': '6356584.0',
...                                     'h': '35785831.0', 'lon_0': '0',
...                                     'proj': 'geos'},
...                                    3712, 3712,
...                                    [-5568742.4, -5568742.4,
...                                     5568742.4, 5568742.4])
>>> data = numpy.ones((3712, 3712))
>>> msg_con = image.ImageContainer(data, msg_area)
>>> row_indices, col_indices = \
...            utils.generate_nearest_neighbour_linesample_arrays(msg_area, area_def,␣
↪50000)
>>> result = msg_con.get_array_from_linesample(row_indices, col_indices)
```

The numpy arrays returned by **generate_*_linesample_arrays** can be and used with the **ImageContainer.get_array_from_linesample** method when the same resampling is to be performed again thus eliminating the need for calculating the reprojection.

Numpy arrays can be saved and loaded using **numpy.save** and **numpy.load**.

1.9 Plotting with pyresample and Cartopy

Pyresample supports basic integration with Cartopy.

1.9.1 Displaying data quickly

Pyresample has some convenience functions for displaying data from a single channel. The *show_quicklook* function shows a Cartopy generated image of a dataset for a specified AreaDefinition. The function *save_quicklook* saves the Cartopy image directly to file.

Example usage:

In this simple example below we use GCOM-W1 AMSR-2 data loaded using Satpy. Satpy simplifies the reading of this data, but is not necessary for using pyresample or plotting data.

First we read in the data with Satpy:

```
>>> from satpy.scene import Scene
>>> from glob import glob
>>> SCENE_FILES = glob("./GW1AM2_20191122????_156*h5")
>>> scn = Scene(reader='amsr2_l1b', filenames=SCENE_FILES)
>>> scn.load(["btemp_36.5v"])
>>> lons, lats = scn["btemp_36.5v"].area.get_lonlats()
>>> tb37v = scn["btemp_36.5v"].data.compute()
```

Data for this example can be downloaded from zenodo.

If you have your own data, or just want to see that the example code here runs, you can set the three arrays `lons`, `lats` and `tb37v` accordingly, e.g.:

```
>>> from pyresample.geometry import AreaDefinition
>>> area_id = 'ease_sh'
>>> description = 'Antarctic EASE grid'
```

(continues on next page)

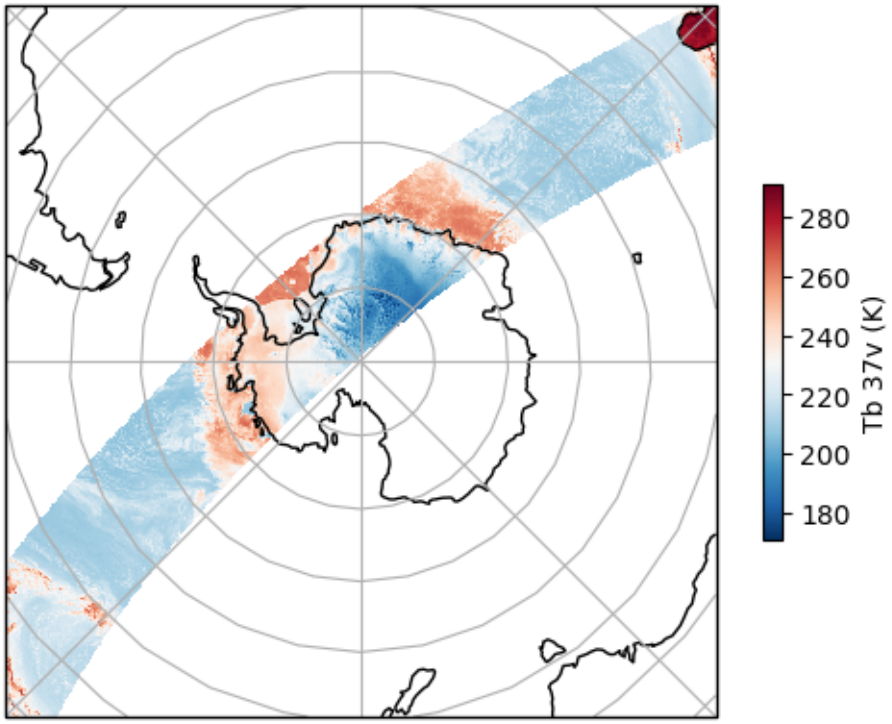
(continued from previous page)

```
>>> proj_id = 'ease_sh'
>>> projection = {'proj': 'laea', 'lat_0': -90, 'lon_0': 0, 'a': 6371228.0, 'units':
↪'m'}
>>> width = 425
>>> height = 425
>>> area_extent = (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
```

But here we go on with the loaded AMSR-2 data. Make sure you have an `areas.yaml` file that defines the `ease_sh` area, or see *the area definition section* on how to define one.

```
>>> from pyresample import load_area, save_quicklook, SwathDefinition
>>> from pyresample.kd_tree import resample_nearest
>>> swath_def = SwathDefinition(lons, lats)
>>> result = resample_nearest(swath_def, tb37v, area_def,
...                           radius_of_influence=20000, fill_value=None)
>>> save_quicklook('tb37v_quick.png', area_def, result, label='Tb 37v (K)')
```

Assuming **lons**, **lats** and **tb37v** are initialized with real data (as in the above Satpy example) the result might look something like this:



The data passed to the functions is a 2D array matching the AreaDefinition.

The Plate Carree projection

The Plate Carree projection (regular lon/lat grid) is named **eqc** in Proj.4. Pyresample uses the Proj.4 naming.

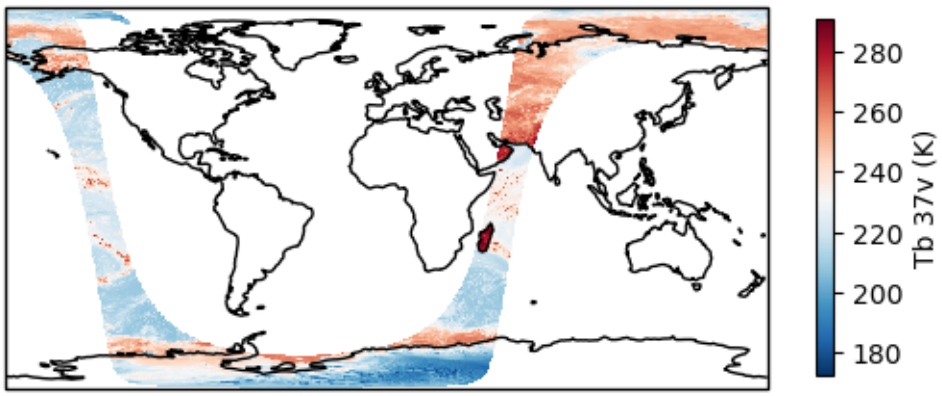
Assuming the file **areas.yaml** has the following area definition:

```
pc_world:
  description: Plate Carree world map
  projection:
    proj: eqc
    ellps: WGS84
  shape:
    height: 480
    width: 640
  area_extent:
    lower_left_xy: [-20037508.34, -10018754.17]
    upper_right_xy: [20037508.34, 10018754.17]
```

Example usage:

```
>>> import matplotlib.pyplot as plt
>>> from pyresample import load_area, save_quicklook
>>> from pyresample.kd_tree import resample_nearest
>>> area_def = load_area('areas.yaml', 'pc_world')
>>> result = resample_nearest(swath_def, tb37v, area_def, radius_of_influence=20000,␣
↪fill_value=None)
>>> save_quicklook('tb37v_pc.png', area_def, result, num_meridians=None, num_
↪parallels=None, label='Tb 37v (K)')
```

Assuming **lons**, **lats** and **tb37v** are initialized with real data (like above we use AMSR-2 data in this example) the result might look something like this:



The Globe projections

From v0.7.12 pyresample can use the geos, ortho and nsper projections with Basemap. Starting with v1.9.0 quicklooks are now generated with Cartopy which should also work with these projections. Again assuming the area-config file **areas.yaml** has the following definition for an ortho projection area:

```
ortho:
  description: Ortho globe
  projection:
    proj: ortho
    lon_0: 40.
    lat_0: -40.
    a: 6370997.0
  shape:
    height: 480
```

(continues on next page)

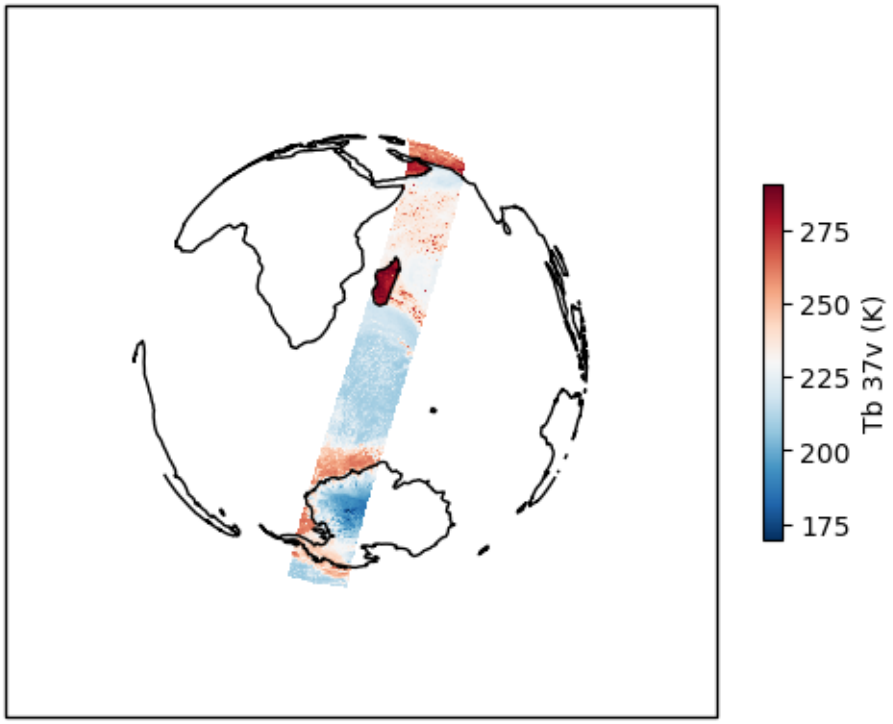
(continued from previous page)

```
    width: 640
  area_extent:
    lower_left_xy: [-10000000, -10000000]
    upper_right_xy: [10000000, 10000000]
```

Example usage:

```
>>> from pyresample import load_area, save_quicklook, SwathDefinition
>>> from pyresample.kd_tree import resample_nearest
>>> from pyresample import load_area
>>> area_def = load_area('areas.yaml', 'ortho')
>>> swath_def = SwathDefinition(lons, lats)
>>> result = resample_nearest(swath_def, tb37v, area_def, radius_of_influence=20000,␣
↪fill_value=None)
>>> save_quicklook('tb37v_ortho.png', area_def, result, num_meridians=None, num_
↪parallels=None, label='Tb 37v (K)')
```

Assuming **lons**, **lats** and **tb37v** are initialized with real data, like in the above examples, the result might look something like this:



1.9.2 Getting a Cartopy CRS

To make more advanced plots than the preconfigured quicklooks Cartopy can be used to work with mapped data alongside matplotlib. The below code is based on this Cartopy gallery example. Pyresample allows any *AreaDefinition* to be converted to a Cartopy CRS as long as Cartopy can represent the projection. Once an *AreaDefinition* is converted to a CRS object it can be used like any other Cartopy CRS object.

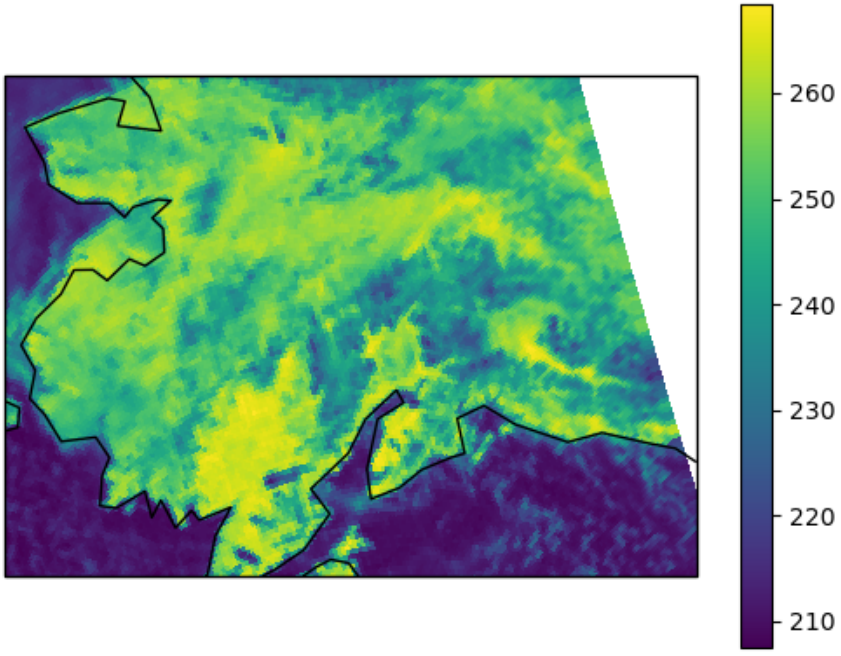
```
>>> import matplotlib.pyplot as plt
>>> from pyresample.kd_tree import resample_nearest
```

(continues on next page)

(continued from previous page)

```
>>> from pyresample.geometry import AreaDefinition
>>> area_id = 'alaska'
>>> description = 'Alaska Lambert Equal Area grid'
>>> proj_id = 'alaska'
>>> projection = {'proj': 'stere', 'lat_0': 62., 'lon_0': -152.5, 'ellps': 'WGS84',
↪'units': 'm'}
>>> width = 2019
>>> height = 1463
>>> area_extent = (-757214.993104, -485904.321517, 757214.993104, 611533.818622)
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
>>> result = resample_nearest(swath_def, tb37v, area_def, radius_of_influence=20000,␣
↪fill_value=None)
>>> crs = area_def.to_cartopy_crs()
>>> fig, ax = plt.subplots(subplot_kw=dict(projection=crs))
>>> coastlines = ax.coastlines()
>>> ax.set_global()
>>> img = plt.imshow(result, transform=crs, extent=crs.bounds, origin='upper')
>>> cbar = plt.colorbar()
>>> fig.savefig('amsr2_tb37v_cartopy.png')
```

Assuming **lons**, **lats**, and **i04_data** are initialized with real data the result might look something like this:



1.9.3 Getting a Basemap object

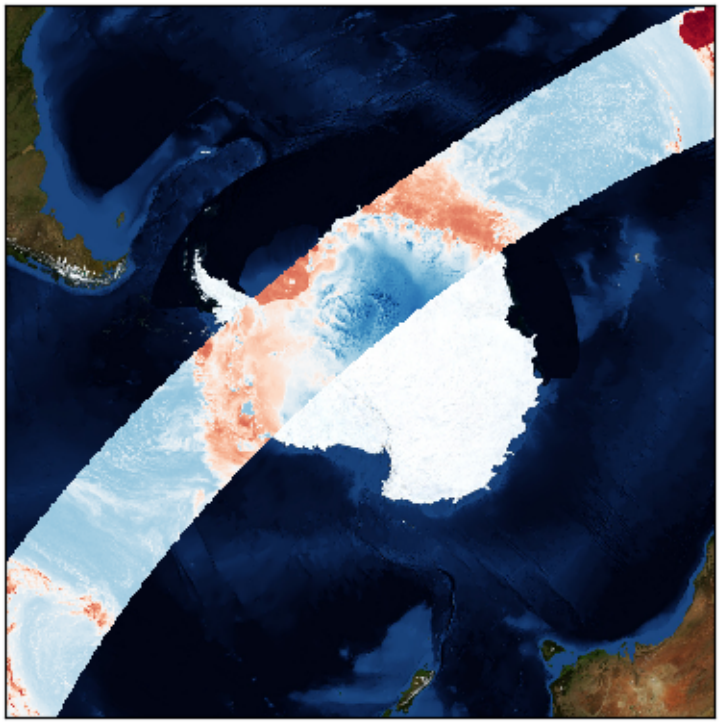
Warning: Basemap is no longer maintained. Cartopy (see above) should be used instead. Basemap does not support Matplotlib 3.0+ either.

In order to make more advanced plots than the preconfigured quicklooks a Basemap object can be generated from an AreaDefinition using the *area_def2basemap* function.

Example usage:

```
>>> import matplotlib.pyplot as plt
>>> from pyresample.kd_tree import resample_nearest
>>> from pyresample.geometry import AreaDefinition
>>> area_id = 'ease_sh'
>>> description = 'Antarctic EASE grid'
>>> proj_id = 'ease_sh'
>>> projection = {'proj': 'laea', 'lat_0': -90, 'lon_0': 0, 'a': 6371228.0, 'units':
↪'m'}
>>> width = 425
>>> height = 425
>>> area_extent = (-5326849.0625, -5326849.0625, 5326849.0625, 5326849.0625)
>>> area_def = AreaDefinition(area_id, description, proj_id, projection,
...                           width, height, area_extent)
>>> from pyresample import area_def2basemap
>>> result = resample_nearest(swath_def, tb37v, area_def,
...                           radius_of_influence=20000, fill_value=None)
>>> bmap = area_def2basemap(area_def)
>>> bmng = bmap.bluemarble()
>>> col = bmap.imshow(result, origin='upper', cmap='RdBu_r')
>>> plt.savefig('tb37v_bmng.png', bbox_inches='tight')
```

Assuming **lons**, **lats** and **tb37v** are initialized with real data as in the previous examples the result might look something like this:



Any keyword arguments (not concerning the projection) passed to **plot.area_def2basemap** will be passed directly to the Basemap initialization.

For more information on how to plot with Basemap please refer to the Basemap and matplotlib documentation.

1.9.4 Adding background maps with Cartopy

As mentioned in the above warning Cartopy should be used rather than Basemap as the latter is not maintained anymore.

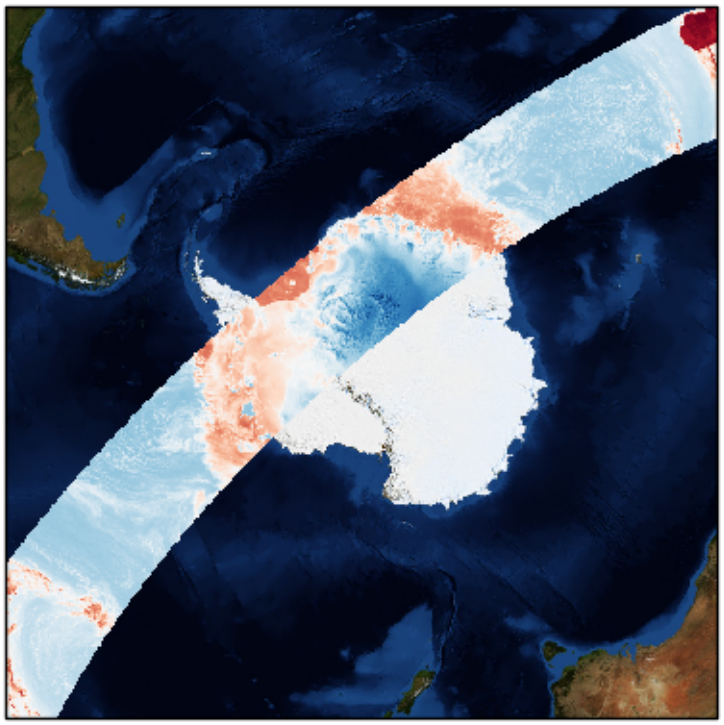
The above image can be generated using Cartopy instead by utilizing the method *to_cartopy_crs* of the *AreaDefinition* object.

Example usage:

```
>>> from pyresample.kd_tree import resample_nearest
>>> import matplotlib.pyplot as plt
>>> result = resample_nearest(swath_def, tb37v, area_def,
...                           radius_of_influence=20000, fill_value=None)
>>> crs = area_def.to_cartopy_crs()
>>> ax = plt.axes(projection=crs)
>>> ax.background_img(name='BM')
>>> plt.imshow(result, transform=crs, extent=crs.bounds, origin='upper', cmap='RdBu_r
↪')
>>> plt.savefig('tb37v_bmng.png', bbox_inches='tight')
```

The above provides you have the Bluemarble background data available in the Cartopy standard place or in a directory pointed to by the environment parameter *CARTOPY_USER_BACKGROUNDS*.

With real data (same AMSR-2 as above) this might look like this:



1.10 Reduction of swath data

Given a swath and a cartesian grid or grid lons and lats pyresample can reduce the swath data to only the relevant part covering the grid area. The reduction is coarse in order not to risk removing relevant data.

From **data_reduce** the function **swath_from_lonlat_grid** can be used to reduce the swath data set to the area covering the lon lat grid

```
>>> import numpy as np
>>> from pyresample import geometry, data_reduce
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> grid_lons, grid_lats = area_def.get_lonlats()
>>> reduced_lons, reduced_lats, reduced_data = \
...                               data_reduce.swath_from_lonlat_grid(grid_lons, grid_
↪lats,
...                               lons, lats, data,
...                               radius_of_influence=3000)
```

radius_of_influence is used to calculate a buffer zone around the grid where swath data points are not reduced.

The function **get_valid_index_from_lonlat_grid** returns a boolean array of same size as the swath indicating the

relevant swath data points compared to the grid

```
>>> import numpy as np
>>> from pyresample import geometry, data_reduce
>>> area_def = geometry.AreaDefinition('areaD', 'Europe (3km, HRV, VTC)', 'areaD',
...                                    {'a': '6378144.0', 'b': '6356759.0',
...                                     'lat_0': '50.00', 'lat_ts': '50.00',
...                                     'lon_0': '8.00', 'proj': 'stere'},
...                                    800, 800,
...                                    [-1370912.72, -909968.64,
...                                     1029087.28, 1490031.36])
>>> data = np.fromfunction(lambda y, x: y*x, (50, 10))
>>> lons = np.fromfunction(lambda y, x: 3 + x, (50, 10))
>>> lats = np.fromfunction(lambda y, x: 75 - y, (50, 10))
>>> grid_lons, grid_lats = area_def.get_lonlats()
>>> valid_index = data_reduce.get_valid_index_from_lonlat_grid(grid_lons, grid_lats,
...                                                            lons, lats,
...                                                            radius_of_influence=3000)
```

1.11 pyresample package

1.11.1 Subpackages

pyresample.bilinear package

Submodules

pyresample.bilinear.xarr module

XArray version of bilinear interpolation.

class pyresample.bilinear.xarr.**XArrayResamplerBilinear**(*source_geo_def*, *target_geo_def*, *radius_of_influence*, *neighbours=32*, *epsilon=0*, *reduce_data=True*)

Bases: object

Bilinear interpolation using XArray.

get_bil_info()

Return neighbour info.

Returns

- **t__** (*numpy array*) – Vertical fractional distances from corner to the new points
- **s__** (*numpy array*) – Horizontal fractional distances from corner to the new points
- **valid_input_index** (*numpy array*) – Valid indices in the input data
- **index_array** (*numpy array*) – Mapping array from valid source points to target points

get_sample_from_bil_info(*data*, *fill_value=None*, *output_shape=None*)

Resample using pre-computed resampling LUTs.

pyresample.bilinear.xarr.**lonlat2xyz**(*lons*, *lats*)
Convert geographic coordinates to cartesian 3D coordinates.

pyresample.bilinear.xarr.**query_no_distance**(*target_lons*, *target_lats*, *valid_output_index*, *kdtree*, *neighbours*, *epsilon*, *radius*)
Query the kdtree. No distances are returned.

Module contents

Code for resampling using bilinear algorithm for irregular grids.

The algorithm is taken from

http://www.ahinson.com/algorithms_general/Sections/InterpolationRegression/InterpolationIrregularBilinear.pdf

pyresample.bilinear.**get_bil_info**(*source_geo_def*, *target_area_def*, *radius=50000.0*, *neighbours=32*, *nprocs=1*, *masked=False*, *reduce_data=True*, *segments=None*, *epsilon=0*)
Calculate information needed for bilinear resampling.

source_geo_def [object] Geometry definition of source data

target_area_def [object] Geometry definition of target area

radius [float, optional] Cut-off distance in meters

neighbours [int, optional] Number of neighbours to consider for each grid point when searching the closest corner points

nprocs [int, optional] Number of processor cores to be used for getting neighbour info

masked [bool, optional] If true, return masked arrays, else return np.nan values for invalid points (default)

reduce_data [bool, optional] Perform initial coarse reduction of source dataset in order to reduce execution time

segments [int or None] Number of segments to use when resampling. If set to None an estimate will be calculated

epsilon [float, optional] Allowed uncertainty in meters. Increasing uncertainty reduces execution time

Returns

- **t__** (*numpy array*) – Vertical fractional distances from corner to the new points
- **s__** (*numpy array*) – Horizontal fractional distances from corner to the new points
- **input_idxs** (*numpy array*) – Valid indices in the input data
- **idx_arr** (*numpy array*) – Mapping array from valid source points to target points

pyresample.bilinear.**get_sample_from_bil_info**(*data*, *t__*, *s__*, *input_idxs*, *idx_arr*, *output_shape=None*)
Resample data using bilinear interpolation.

Parameters

- **data** (*numpy array*) – 1d array to be resampled
- **t** (*numpy array*) – Vertical fractional distances from corner to the new points
- **s** (*numpy array*) – Horizontal fractional distances from corner to the new points
- **input_idxs** (*numpy array*) – Valid indices in the input data

- **idx_arr** (*numpy array*) – Mapping array from valid source points to target points
- **output_shape** (*tuple, optional*) – Tuple of (y, x) dimension for the target projection. If None (default), do not reshape data.

Returns **result** – Source data resampled to target geometry

Return type numpy array

pyresample.bilinear.**resample_bilinear**(*data*, *source_geo_def*, *target_area_def*, *radius=50000.0*, *neighbours=32*, *nprocs=1*, *fill_value=0*, *reduce_data=True*, *segments=None*, *epsilon=0*)

Resample using bilinear interpolation.

data [numpy array] Array of single channel data points or (source_geo_def.shape, k) array of k channels of datapoints

source_geo_def [object] Geometry definition of source data

target_area_def [object] Geometry definition of target area

radius [float, optional] Cut-off distance in meters

neighbours [int, optional] Number of neighbours to consider for each grid point when searching the closest corner points

nprocs [int, optional] Number of processor cores to be used for getting neighbour info

fill_value [{int, None}, optional] Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked

reduce_data [bool, optional] Perform initial coarse reduction of source dataset in order to reduce execution time

segments [int or None] Number of segments to use when resampling. If set to None an estimate will be calculated

epsilon [float, optional] Allowed uncertainty in meters. Increasing uncertainty reduces execution time

Returns **data** – Source data resampled to target geometry

Return type numpy array

pyresample.bucket package

Module contents

pyresample.ewa package

Module contents

pyresample.gradient package

Module contents

pyresample.utils package

Submodules

pyresample.utils.cartopy module

pyresample.utils.cf module

Load an AreaDefinition object from a netCDF/CF file.

pyresample.utils.cf.**load_cf_area**(*nc_file*, *variable=None*, *y=None*, *x=None*)

Load an AreaDefinition object from a netCDF/CF file.

Parameters

- **nc_file** (*string or object*) – path to a netCDF/CF file, or opened xarray.Dataset object
- **variable** (*string, optional*) – name of the variable to load the AreaDefinition from. If the variable is not a CF grid_mapping container variable, it should be a variable having a :grid_mapping attribute. If variable is None the file will be searched for valid CF area definitions
- **y** (*string, optional*) – name of the variable to use as 'y' axis of the CF area definition If y is None an appropriate 'y' axis will be deduced from the CF file
- **x** (*string, optional*) – name of the variable to use as 'x' axis of the CF area definition If x is None an appropriate 'x' axis will be deduced from the CF file

Returns **are_def, cf_info** – cf_info holds info about how the AreaDefinition was defined in the CF file.

Return type geometry.AreaDefinition object, dict

pyresample.utils.proj4 module

pyresample.utils.proj4.**convert_proj_floats**(*proj_pairs*)

Convert PROJ.4 parameters to floats if possible.

pyresample.utils.proj4.**proj4_dict_to_str**(*proj4_dict*, *sort=False*)

Convert a dictionary of PROJ.4 parameters to a valid PROJ.4 string

pyresample.utils.proj4.**proj4_radius_parameters**(*proj4_dict*)

Calculate 'a' and 'b' radius parameters.

Parameters **proj4_dict** (*str or dict*) – PROJ.4 parameters

Returns equatorial and polar radius

Return type a (float), b (float)

pyresample.utils.proj4.**proj4_str_to_dict**(*proj4_str*)

Convert PROJ.4 compatible string definition to dict

EPSG codes should be provided as "EPSG:XXXX" where "XXXX" is the EPSG number code. It can also be provided as `"+init=EPSG:XXXX"` as long as the underlying PROJ library supports it (deprecated in PROJ 6.0+).

Note: Key only parameters will be assigned a value of *True*.

pyresample.utils.rasterio module

pyresample.utils.rasterio.**get_area_def_from_raster**(*source*, *area_id=None*, *name=None*, *proj_id=None*, *proj_dict=None*)

Construct AreaDefinition object from raster.

Parameters

- **source** (*str, Dataset, DatasetReader or DatasetWriter*) – A file name. Also it can be `osgeo.gdal.Dataset`, `rasterio.io.DatasetReader` or `rasterio.io.DatasetWriter`
- **area_id** (*str, optional*) – ID of area
- **name** (*str, optional*) – Name of area
- **proj_id** (*str, optional*) – ID of projection
- **proj_dict** (*dict, optional*) – PROJ.4 parameters

Returns **area_def** – AreaDefinition object

Return type object

Module contents

Miscellaneous utility functions for pyresample.

pyresample.utils.**check_and_wrap**(*lons*, *lats*)

Wrap longitude to [-180:+180[and check latitude for validity.

Parameters

- **lons** (*ndarray*) – Longitude degrees
- **lats** (*ndarray*) – Latitude degrees

Returns

Longitude degrees in the range [-180:180[and the original latitude array

Return type lons, lats

Raises `ValueError` – If latitude array is not between -90 and 90

pyresample.utils.**check_slice_orientation**(*sli*)

Check that the slice is slicing the right way.

pyresample.utils.**convert_def_to_yaml**(**args*, ***kwargs*)

pyresample.utils.**create_area_def**(**args*, ***kwargs*)

pyresample.utils.**fwhm2sigma**(*fwhm*)

Calculate sigma for gauss function from FWHM (3 dB level)

Parameters **fwhm** (*float*) – FWHM of gauss function (3 dB level of beam footprint)

Returns **sigma** – sigma for use in resampling gauss function

Return type float

pyresample.utils.**generate_nearest_neighbour_linesample_arrays**(*source_area_def*, *target_area_def*, *radius_of_influence*, *nprocs=1*)

Generate linesample arrays for nearest neighbour grid resampling

Parameters

- **source_area_def** (*object*) – Source area definition as geometry definition object
- **target_area_def** (*object*) – Target area definition as geometry definition object
- **radius_of_influence** (*float*) – Cut off distance in meters
- **nprocs** (*int, optional*) – Number of processor cores to be used

Returns (row_indices, col_indices)

Return type tuple of numpy arrays

pyresample.utils.**generate_quick_linesample_arrays**(*source_area_def*, *target_area_def*, *nprocs=1*)

Generate linesample arrays for quick grid resampling

Parameters

- **source_area_def** (*object*) – Source area definition as geometry definition object
- **target_area_def** (*object*) – Target area definition as geometry definition object
- **nprocs** (*int, optional*) – Number of processor cores to be used

Returns (row_indices, col_indices)

Return type tuple of numpy arrays

pyresample.utils.**get_area_def**(**args*, ***kwargs*)

pyresample.utils.**is_pyproj2**()

Determine whether the current pyproj version is >= 2.0

pyresample.utils.**load_area**(**args*, ***kwargs*)

pyresample.utils.**parse_area_file**(**args*, ***kwargs*)

pyresample.utils.**recursive_dict_update**(*d*, *u*)

Recursive dictionary update using

Copied from:

http://stackoverflow.com/questions/3232943/update-value-of-a-nested-dictionary-of-varying-depth

pyresample.utils.**wrap_longitudes**(*lons*)

Wrap longitudes to the [-180:+180[validity range (preserves dtype)

Parameters **lons** (*numpy array*) – Longitudes in degrees

Returns **lons** – Longitudes wrapped into [-180:+180[validity range

Return type numpy array

1.11.2 Submodules

1.11.3 pyresample.area_config module

exception pyresample.area_config.**AreaNotFound**
Bases: KeyError

Exception raised when specified are is no found in file.

class pyresample.area_config.**DataArray**(*data*, *attrs=None*)
Bases: object

Stand-in for DataArray for holding units information.

pyresample.area_config.**convert_def_to_yaml**(*def_area_file*, *yaml_area_file*)
Convert a legacy area def file to the yaml counter partself.

yaml_area_file will be overwritten by the operation.

pyresample.area_config.**create_area_def**(*area_id*, *projection*, *width=None*, *height=None*, *area_extent=None*, *shape=None*, *upper_left_extent=None*, *center=None*, *resolution=None*, *radius=None*, *units=None*, ***kwargs*)
Takes data the user knows and tries to make an area definition from what can be found.

Parameters

- **area_id** (*str*) – ID of area
- **projection** (*dict or str*) – Projection parameters as a proj4_dict or proj4_string
- **description** (*str, optional*) – Description/name of area. Defaults to area_id
- **proj_id** (*str, optional*) – ID of projection (deprecated)
- **units** (*str, optional*) – Units that provided arguments should be interpreted as. This can be one of 'deg', 'degrees', 'meters', 'metres', and any parameter supported by the cs2cs -lu command. Units are determined in the following priority:
  1. units expressed with each variable through a DataArray's attrs attribute.
  2. units passed to `units`
  3. units used in `projection`
  4. meters
- **width** (*str, optional*) – Number of pixels in the x direction
- **height** (*str, optional*) – Number of pixels in the y direction
- **area_extent** (*list, optional*) – Area extent as a list (lower_left_x, lower_left_y, upper_right_x, upper_right_y)
- **shape** (*list, optional*) – Number of pixels in the y and x direction (height, width)
- **upper_left_extent** (*list, optional*) – Upper left corner of upper left pixel (x, y)
- **center** (*list, optional*) – Center of projection (x, y)
- **resolution** (*list or float, optional*) – Size of pixels: (dx, dy)
- **radius** (*list or float, optional*) – Length from the center to the edges of the projection (dx, dy)

- **rotation** (*float, optional*) – rotation in degrees(negative is cw)
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **lons** (*numpy array, optional*) – Grid lons
- **lats** (*numpy array, optional*) – Grid lats
- **optimize_projection** – Whether the projection parameters have to be optimized for a DynamicAreaDefinition.

Returns **AreaDefinition or DynamicAreaDefinition** – If shape and area_extent are found, an AreaDefinition object is returned. If only shape or area_extent can be found, a DynamicAreaDefinition object is returned

Return type *AreaDefinition* or *DynamicAreaDefinition*

Raises ValueError: – If neither shape nor area_extent could be found

---

Notes

- `resolution` and `radius` can be specified with one value if dx == dy
- If `resolution` and `radius` are provided as angles, center must be given or findable. In such a case, they represent [projection x distance from center[0] to center[0]+dx, projection y distance from center[1] to center[1]+dy]

---

pyresample.area_config.**get_area_def**(*area_id*, *area_name*, *proj_id*, *proj4_args*, *width*, *height*, *area_extent*, *rotation=0*)

Construct AreaDefinition object from arguments

Parameters

- **area_id** (*str*) – ID of area
- **area_name** (*str*) – Description of area
- **proj_id** (*str*) – ID of projection
- **proj4_args** (*list, dict, or str*) – Proj4 arguments as list of arguments or string
- **width** (*int*) – Number of pixel in x dimension
- **height** (*int*) – Number of pixel in y dimension
- **rotation** (*float*) – Rotation in degrees (negative is cw)
- **area_extent** (*list*) – Area extent as a list of ints (LL_x, LL_y, UR_x, UR_y)

Returns **area_def** – AreaDefinition object

Return type object

pyresample.area_config.**load_area**(*area_file_name*, **regions*)

Load area(s) from area file.

Parameters

- **area_file_name** (*str, pathlib.Path, stream, or list thereof*) – List of paths or streams. Any str or pathlib.Path will be interpreted as a path to a file. Any stream will be interpreted as containing a yaml definition. To read directly from a string, use *load_area_from_string()*.
- **regions** (*str argument list*) – Regions to parse. If no regions are specified all regions in the file are returned

Returns **area_defs** – If one area name is specified a single AreaDefinition object is returned. If several area names are specified a list of AreaDefinition objects is returned

Return type *AreaDefinition* or list

Raises AreaNotFound: – If a specified area name is not found

pyresample.area_config.**load_area_from_string**(*area_strs*, **regions*)
Load area(s) from area strings.

Like *load_area()*, but load from string directly.

Parameters

- **area_strs** (*str or List[str]*) – Strings containing yaml definitions.
- **regions** (*str*) – Regions to parse.

Returns **area_defs** – If one area name is specified a single AreaDefinition object is returned. If several area names are specified a list of AreaDefinition objects is returned

Return type *AreaDefinition* or list

pyresample.area_config.**parse_area_file**(*area_file_name*, **regions*)
Parse area information from area file

Parameters

- **area_file_name** (*str or list*) – One or more paths to area definition files
- **regions** (*str argument list*) – Regions to parse. If no regions are specified all regions in the file are returned

Returns **area_defs** – List of AreaDefinition objects

Return type list

Raises AreaNotFound: – If a specified area is not found

1.11.4 pyresample.boundary module

The Boundary classes.

class pyresample.boundary.**AreaBoundary**(**sides*)
Bases: *pyresample.boundary.Boundary*

Area boundary objects.

contour()
Get the (lons, lats) tuple of the boundary object.

decimate(*ratio*)
Remove some points in the boundaries, but never the corners.

class pyresample.boundary.**AreaDefBoundary**(*area*, *frequency=1*)
Bases: *pyresample.boundary.AreaBoundary*

Boundaries for area definitions (pyresample).

class pyresample.boundary.**Boundary**(*lons=None*, *lats=None*, *frequency=1*)
Bases: object

Boundary objects.

contour()

contour_poly
Get the Spherical polygon corresponding to the Boundary

draw(*mapper*, *options*, ***more_options*)
Draw the current boundary on the *mapper*

class pyresample.boundary.**SimpleBoundary**(*side1*, *side2*, *side3*, *side4*)
Bases: object

Container for geometry boundary. Labelling starts in upper left corner and proceeds clockwise

1.11.5 pyresample.data_reduce module

Reduce data sets based on geographical information

pyresample.data_reduce.**get_valid_index_from_cartesian_grid**(*cart_grid*, *lons*, *lats*, *radius_of_influence*)
Calculates relevant data indices using coarse data reduction of swath data by comparison with cartesian grid

Parameters

- **chart_grid** (*numpy array*) – Grid of area cartesian coordinates
- **lons** (*numpy array*) – Swath lons
- **lats** (*numpy array*) – Swath lats
- **data** (*numpy array*) – Swath data
- **radius_of_influence** (*float*) – Cut off distance in meters

Returns **valid_index** – Boolean array of same size as lons and lats indicating relevant indices

Return type numpy array

pyresample.data_reduce.**get_valid_index_from_lonlat_boundaries**(*boundary_lons*, *boundary_lats*, *lons*, *lats*, *radius_of_influence*)
Find relevant indices from grid boundaries using the winding number theorem

pyresample.data_reduce.**get_valid_index_from_lonlat_grid**(*grid_lons*, *grid_lats*, *lons*, *lats*, *radius_of_influence*)
Calculates relevant data indices using coarse data reduction of swath data by comparison with lon lat grid

Parameters

- **chart_grid** (*numpy array*) – Grid of area cartesian coordinates
- **lons** (*numpy array*) – Swath lons
- **lats** (*numpy array*) – Swath lats
- **data** (*numpy array*) – Swath data
- **radius_of_influence** (*float*) – Cut off distance in meters

Returns **valid_index** – Boolean array of same size as lon and lat indicating relevant indices

Return type numpy array

pyresample.data_reduce.**swath_from_cartesian_grid**(*cart_grid*, *lons*, *lats*, *data*, *radius_of_influence*)
Makes coarse data reduction of swath data by comparison with cartesian grid

Parameters

- **chart_grid** (*numpy array*) – Grid of area cartesian coordinates
- **lons** (*numpy array*) – Swath lons
- **lats** (*numpy array*) – Swath lats
- **data** (*numpy array*) – Swath data
- **radius_of_influence** (*float*) – Cut off distance in meters

Returns **(lons, lats, data)** – Reduced swath data and coordinate set

Return type list of numpy arrays

pyresample.data_reduce.**swath_from_lonlat_boundaries**(*boundary_lons*, *boundary_lats*, *lons*, *lats*, *data*, *radius_of_influence*)

Makes coarse data reduction of swath data by comparison with lon lat boundary

Parameters

- **boundary_lons** (*numpy array*) – Grid of area lons
- **boundary_lats** (*numpy array*) – Grid of area lats
- **lons** (*numpy array*) – Swath lons
- **lats** (*numpy array*) – Swath lats
- **data** (*numpy array*) – Swath data
- **radius_of_influence** (*float*) – Cut off distance in meters

Returns **(lons, lats, data)** – Reduced swath data and coordinate set

Return type list of numpy arrays

pyresample.data_reduce.**swath_from_lonlat_grid**(*grid_lons*, *grid_lats*, *lons*, *lats*, *data*, *radius_of_influence*)

Makes coarse data reduction of swath data by comparison with lon lat grid

Parameters

- **grid_lons** (*numpy array*) – Grid of area lons
- **grid_lats** (*numpy array*) – Grid of area lats
- **lons** (*numpy array*) – Swath lons
- **lats** (*numpy array*) – Swath lats
- **data** (*numpy array*) – Swath data
- **radius_of_influence** (*float*) – Cut off distance in meters

Returns **(lons, lats, data)** – Reduced swath data and coordinate set

Return type list of numpy arrays

1.11.6 pyresample.geo_filter module

class pyresample.geo_filter.**GridFilter**(*area_def*, *filter*, *nprocs=1*)

Bases: object

Geographic filter from a grid.

Parameters

- **grid_ll_x** (*float*) – Projection x coordinate of lower left corner of lower left pixel
- **grid_ll_y** (*float*) – Projection y coordinate of lower left corner of lower left pixel
- **grid_ur_x** (*float*) – Projection x coordinate of upper right corner of upper right pixel
- **grid_ur_y** (*float*) – Projection y coordinate of upper right corner of upper right pixel
- **proj4_string** (*str*) – Projection definition as a PROJ.4 string.
- **mask** (*numpy array*) – Mask as boolean numpy array

filter(*geometry_def*, *data*)

get_valid_index(*geometry_def*)
Calculates valid_index array based on lons and lats

Parameters

- **lons** (*numpy array*) – Longitude degrees array
- **lats** (*numpy array*) – Latitude degrees array

Returns Boolean numpy array of same shape as lons and lats

1.11.7 pyresample.geometry module

Classes for geometry operations.

class pyresample.geometry.**AreaDefinition**(*area_id*, *description*, *proj_id*, *projection*, *width*, *height*, *area_extent*, *rotation=None*, *nprocs=1*, *lons=None*, *lats=None*, *dtype=<class 'numpy.float64'>*)

Bases: *pyresample.geometry.BaseDefinition*

Holds definition of an area.

Parameters

- **area_id** (*str*) – Identifier for the area
- **description** (*str*) – Human-readable description of the area
- **proj_id** (*str*) – ID of projection
- **projection** (*dict or str or pyproj.crs.CRS*) – Dictionary of PROJ parameters or string of PROJ or WKT parameters. Can also be a pyproj.crs.CRS object.
- **width** (*int*) – x dimension in number of pixels, aka number of grid columns
- **height** (*int*) – y dimension in number of pixels, aka number of grid rows
- **area_extent** (*list*) – Area extent as a list (lower_left_x, lower_left_y, upper_right_x, upper_right_y)
- **rotation** (*float, optional*) – rotation in degrees (negative is clockwise)
- **nprocs** (*int, optional*) – Number of processor cores to be used for certain calculations

area_id
Identifier for the area

Type str

description
Human-readable description of the area

Type str

proj_id
ID of projection

Type str

projection
Dictionary or string with Proj.4 parameters

Type dict or str

width
x dimension in number of pixels, aka number of grid columns

Type int

height
y dimension in number of pixels, aka number of grid rows

Type int

rotation
rotation in degrees (negative is cw)

Type float

size
Number of points in grid

Type int

area_extent
Area extent as a tuple (lower_left_x, lower_left_y, upper_right_x, upper_right_y)

Type tuple

area_extent_ll
Area extent in lons lats as a tuple (lower_left_lon, lower_left_lat, upper_right_lon, upper_right_lat)

Type tuple

pixel_size_x
Pixel width in projection units

Type float

pixel_size_y
Pixel height in projection units

Type float

upper_left_extent
Coordinates (x, y) of upper left corner of upper left pixel in projection units

Type tuple

pixel_upper_left
Coordinates (x, y) of center of upper left pixel in projection units

Type tuple

pixel_offset_x
x offset between projection center and upper left corner of upper left pixel in units of pixels.

Type float

pixel_offset_y
y offset between projection center and upper left corner of upper left pixel in units of pixels..

Type float

crs
Coordinate reference system object similar to the PROJ parameters in *proj_dict* and *proj_str*. This is the preferred attribute to use when working with the *pyproj* library. Note, however, that this object is not thread-safe and should not be passed between threads.

Type pyproj.crs.CRS

crs_wkt
WellKnownText version of the CRS object. This is the preferred way of describing CRS information as a string.

Type str

cartesian_coords
Grid cartesian coordinates

Type object

aggregate(***dims*)
Return an aggregated version of the area.

colrow2lonlat(*cols*, *rows*)
Return lons and lats for the given image columns and rows.

Both scalars and arrays are supported. To be used with scarse data points instead of slices (see get_lonlats).

copy(***override_kwargs*)
Make a copy of the current area.

This replaces the current values with anything in *override_kwargs*.

create_areas_def()
Generate YAML formatted representation of this area.

create_areas_def_legacy()
Create area definition in legacy format.

crop_around(*other_area*)
Crop this area around *other_area*.

classmethod from_area_of_interest(*area_id*, *projection*, *shape*, *center*, *resolution*, *units=None*, ***kwargs*)
Create an AreaDefinition from center, resolution, and shape.

Parameters

- **area_id** (*str*) – ID of area
- **projection** (*dict or str*) – Projection parameters as a proj4_dict or proj4_string
- **shape** (*list*) – Number of pixels in the y and x direction (height, width)
- **center** (*list*) – Center of projection (x, y)
- **resolution** (*list or float*) – Size of pixels: (dx, dy). Can be specified with one value if dx == dy

- **units** (*str, optional*) – Units that provided arguments should be interpreted as. This can be one of 'deg', 'degrees', 'meters', 'metres', and any parameter supported by the cs2cs -lu command. Units are determined in the following priority:
  1. units expressed with each variable through a DataArray's attrs attribute.
  2. units passed to `units`
  3. units used in `projection`
  4. meters
- **description** (*str, optional*) – Description/name of area. Defaults to area_id
- **proj_id** (*str, optional*) – ID of projection
- **rotation** (*float, optional*) – rotation in degrees (negative is cw)
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **lons** (*numpy array, optional*) – Grid lons
- **lats** (*numpy array, optional*) – Grid lats

Returns AreaDefinition

Return type *AreaDefinition*

classmethod from_cf(*cf_file*, *variable=None*, *y=None*, *x=None*)

Create an AreaDefinition object from a netCDF/CF file.

Parameters

- **nc_file** (*string or object*) – path to a netCDF/CF file, or opened xarray.Dataset object
- **variable** (*string, optional*) – name of the variable to load the AreaDefinition from If variable is None the file will be searched for valid CF area definitions
- **y** (*string, optional*) – name of the variable to use as 'y' axis of the CF area definition If y is None an appropriate 'y' axis will be deduced from the CF file
- **x** (*string, optional*) – name of the variable to use as 'x' axis of the CF area definition If x is None an appropriate 'x' axis will be deduced from the CF file

Returns AreaDefinition

Return type *AreaDefinition*

classmethod from_circle(*area_id*, *projection*, *center*, *radius*, *shape=None*, *resolution=None*, *units=None*, ***kwargs*)

Create an AreaDefinition from center, radius, and shape or from center, radius, and resolution.

Parameters

- **area_id** (*str*) – ID of area
- **projection** (*dict or str*) – Projection parameters as a proj4_dict or proj4_string
- **center** (*list*) – Center of projection (x, y)
- **radius** (*list or float*) – Length from the center to the edges of the projection (dx, dy)
- **shape** (*list, optional*) – Number of pixels in the y and x direction (height, width)
- **resolution** (*list or float, optional*) – Size of pixels: (dx, dy)

- **units** (*str, optional*) – Units that provided arguments should be interpreted as. This can be one of 'deg', 'degrees', 'meters', 'metres', and any parameter supported by the cs2cs -lu command. Units are determined in the following priority:
  1. units expressed with each variable through a DataArray's attrs attribute.
  2. units passed to `units`
  3. units used in `projection`
  4. meters
- **description** (*str, optional*) – Description/name of area. Defaults to area_id
- **proj_id** (*str, optional*) – ID of projection
- **rotation** (*float, optional*) – rotation in degrees (negative is cw)
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **lons** (*numpy array, optional*) – Grid lons
- **lats** (*numpy array, optional*) – Grid lats
- **optimize_projection** – Whether the projection parameters have to be optimized for a DynamicAreaDefinition.

Returns **AreaDefinition or DynamicAreaDefinition** – If shape or resolution are provided, an AreaDefinition object is returned. Else a DynamicAreaDefinition object is returned

Return type *AreaDefinition* or *DynamicAreaDefinition*

Notes

- `resolution` and `radius` can be specified with one value if dx == dy

---

classmethod from_epsg(*code*, *resolution*)

Create an AreaDefinition object from an epsg code (string or int) and a resolution.

classmethod from_extent(*area_id*, *projection*, *shape*, *area_extent*, *units=None*, ***kwargs*)

Create an AreaDefinition object from area_extent and shape.

Parameters

- **area_id** (*str*) – ID of area
- **projection** (*dict or str*) – Projection parameters as a proj4_dict or proj4_string
- **shape** (*list*) – Number of pixels in the y and x direction (height, width)
- **area_extent** (*list*) – Area extent as a list (lower_left_x, lower_left_y, upper_right_x, upper_right_y)
- **units** (*str, optional*) – Units that provided arguments should be interpreted as. This can be one of 'deg', 'degrees', 'meters', 'metres', and any parameter supported by the cs2cs -lu command. Units are determined in the following priority:
  1. units expressed with each variable through a DataArray's attrs attribute.
  2. units passed to `units`
  3. units used in `projection`
  4. meters
- **description** (*str, optional*) – Description/name of area. Defaults to area_id

- **proj_id** (*str, optional*) – ID of projection
- **rotation** (*float, optional*) – rotation in degrees (negative is cw)
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **lons** (*numpy array, optional*) – Grid lons
- **lats** (*numpy array, optional*) – Grid lats

Returns AreaDefinition

Return type *AreaDefinition*

classmethod from_ul_corner(*area_id*, *projection*, *shape*, *upper_left_extent*, *resolution*, *units=None*, ***kwargs*)

Create an AreaDefinition object from upper_left_extent, resolution, and shape.

Parameters

- **area_id** (*str*) – ID of area
- **projection** (*dict or str*) – Projection parameters as a proj4_dict or proj4_string
- **shape** (*list*) – Number of pixels in the y and x direction (height, width)
- **upper_left_extent** (*list*) – Upper left corner of upper left pixel (x, y)
- **resolution** (*list or float*) – Size of pixels in **meters**: (dx, dy). Can be specified with one value if dx == dy
- **units** (*str, optional*) – Units that provided arguments should be interpreted as. This can be one of 'deg', 'degrees', 'meters', 'metres', and any parameter supported by the cs2cs -lu command. Units are determined in the following priority:
  1. units expressed with each variable through a DataArray's attrs attribute.
  2. units passed to `units`
  3. units used in `projection`
  4. meters
- **description** (*str, optional*) – Description/name of area. Defaults to area_id
- **proj_id** (*str, optional*) – ID of projection
- **rotation** (*float, optional*) – rotation in degrees (negative is cw)
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **lons** (*numpy array, optional*) – Grid lons
- **lats** (*numpy array, optional*) – Grid lats

Returns AreaDefinition

Return type *AreaDefinition*

geocentric_resolution(*ellps='WGS84'*, *radius=None*)

Find best estimate for overall geocentric resolution.

This method is extremely important to the results of KDTree-based resamplers like the nearest neighbor resampling. This is used to determine how far the KDTree should be queried for valid pixels before giving up (*radius_of_influence*). This method attempts to make a best guess at what geocentric resolution (the units used by the KDTree) represents the majority of an area.

To do this this method will:

1. Create a vertical mid-line and a horizontal mid-line.
2. Convert these coordinates to geocentric coordinates.
3. Compute the distance between points along these lines.
4. Take the histogram of each set of distances and find the bin with the most points.
5. Take the average of the edges of that bin.
6. Return the maximum of the vertical and horizontal bin edge averages.

get_area_slices(*area_to_cover*, *shape_divisible_by=None*)
Compute the slice to read based on an *area_to_cover*.

get_lonlat(*row*, *col*)
Retrieve lon and lat values of single point in area grid.

Parameters

- **row** (*int*) –
- **col** (*int*) –

Returns **(lon, lat)**

Return type tuple of floats

get_lonlats(*nprocs=None*, *data_slice=None*, *cache=False*, *dtype=None*, *chunks=None*)
Return lon and lat arrays of area.

Parameters

- **nprocs** (*int, optional*) – Number of processor cores to be used. Defaults to the nprocs set when instantiating object
- **data_slice** (*slice object, optional*) – Calculate only coordinates for specified slice
- **cache** (*bool, optional*) – Store result the result. Requires data_slice to be None
- **dtype** (*numpy.dtype, optional*) – Data type of the returned arrays
- **chunks** (*int or tuple, optional*) – Create dask arrays and use this chunk size

Returns **(lons, lats)** – Grids of area lons and and lats

Return type tuple of numpy arrays

get_lonlats_dask(*chunks=None*, *dtype=None*)
Get longitudes and latitudes.

get_proj_coords(*data_slice=None*, *dtype=None*, *chunks=None*)
Get projection coordinates of grid.

Parameters

- **data_slice** (*slice object, optional*) – Calculate only coordinates for specified slice
- **dtype** (*numpy.dtype, optional*) – Data type of the returned arrays
- **chunks** (*int or tuple, optional*) – Create dask arrays and use this chunk size

Returns

- **(target_x, target_y)** (*tuple of numpy arrays*) – Grids of area x- and y-coordinates in projection units

- .. *versionchanged:: 1.11.0* – Removed 'cache' keyword argument and add 'chunks' for creating dask arrays.

get_proj_coords_dask(*chunks=None*, *dtype=None*)
Get projection coordinates.

get_proj_vectors(*dtype=None*, *chunks=None*)
Calculate 1D projection coordinates for the X and Y dimension.

Parameters

- **dtype** (*numpy.dtype*) – Numpy data type for the returned arrays
- **chunks** (*int or tuple*) – Return dask arrays with the chunk size specified. If this is a tuple then the first element is the Y array's chunk size and the second is the X array's chunk size.

Returns

- **tuple** (*(X, Y) where X and Y are 1-dimensional numpy arrays*)
- *The data type of the returned arrays can be controlled with the*
- *dtype* keyword argument. If *chunks* is provided then dask arrays
- *are returned instead.*

get_proj_vectors_dask(*chunks=None*, *dtype=None*)
Get projection vectors.

get_xy_from_lonlat(*lon*, *lat*)
Retrieve closest x and y coordinates.

Retrieve closest x and y coordinates (column, row indices) for the specified geolocation (lon,lat) if inside area. If lon,lat is a point a ValueError is raised if the return point is outside the area domain. If lon,lat is a tuple of sequences of longitudes and latitudes, a tuple of masked arrays are returned.

Input

lon : point or sequence (list or array) of longitudes lat : point or sequence (list or array) of latitudes

Returns

(x, y) : tuple of integer points/arrays

get_xy_from_proj_coords(*xm*, *ym*)
Find closest grid cell index for a specified projection coordinate.

If xm, ym is a tuple of sequences of projection coordinates, a tuple of masked arrays are returned.

Parameters

- **xm** (*list or array*) – point or sequence of x-coordinates in meters (map projection)
- **ym** (*list or array*) – point or sequence of y-coordinates in meters (map projection)

Returns column and row grid cell indexes as 2 scalars or arrays

Return type x, y

Raises ValueError – if the return point is outside the area domain

lonlat2colrow(*lons*, *lats*)
Return image columns and rows for the given lons and lats.

Both scalars and arrays are supported. Same as get_xy_from_lonlat, renamed for convenience.

name
Return area name.

outer_boundary_corners
Return the lon,lat of the outer edges of the corner points.

proj4_string
Return projection definition as Proj.4 string.

proj_dict
Return the PROJ projection dictionary.

This is no longer the preferred way of describing CRS information. Switch to the *crs* or *crs_wkt* properties for the most flexibility.

proj_str
Return PROJ projection string.

This is no longer the preferred way of describing CRS information. Switch to the *crs* or *crs_wkt* properties for the most flexibility.

projection_x_coords
Return projection X coordinates.

projection_y_coords
Return projection Y coordinates.

resolution
Return area resolution in X and Y direction.

shape
Return area shape.

to_cartopy_crs()
Convert projection to cartopy CRS object.

update_hash(*the_hash=None*)
Update a hash, or return a new one if needed.

x_size
Return area width.

y_size
Return area height.

class pyresample.geometry.**BaseDefinition**(*lons=None*, *lats=None*, *nprocs=1*)
Bases: object

Base class for geometry definitions.

Changed in version 1.8.0: *BaseDefinition* no longer checks the validity of the provided longitude and latitude coordinates to improve performance. Longitude arrays are expected to be between -180 and 180 degrees, latitude -90 to 90 degrees. Use *check_and_wrap()* to preprocess your arrays.

corners
Return the corners of the current area.

get_area()
Get the area of the convex area defined by the corners of the curren area.

get_area_extent_for_subset(*row_LR*, *col_LR*, *row_UL*, *col_UL*)
Calculate extent for a subdomain of this area.

Rows are counted from upper left to lower left and columns are counted from upper left to upper right.

Parameters

- **row_LR** (*int*) – row of the lower right pixel
- **col_LR** (*int*) – col of the lower right pixel
- **row_UL** (*int*) – row of the upper left pixel
- **col_UL** (*int*) – col of the upper left pixel

Returns Area extent (LL_x, LL_y, UR_x, UR_y) of the subset

Return type area_extent (tuple)

Author: Ulrich Hamann

get_area_slices(*area_to_cover*)
Compute the slice to read based on an *area_to_cover*.

get_bbox_lonlats()
Return the bounding box lons and lats.

get_boundary_lonlats()
Return Boundary objects.

get_cartesian_coords(*nprocs=None*, *data_slice=None*, *cache=False*)
Retrieve cartesian coordinates of geometry definition.

Parameters

- **nprocs** (*int, optional*) – Number of processor cores to be used. Defaults to the nprocs set when instantiating object
- **data_slice** (*slice object, optional*) – Calculate only cartesian coordnates for the defined slice
- **cache** (*bool, optional*) – Store result the result. Requires data_slice to be None

Returns **cartesian_coords**

Return type numpy array

get_lonlat(*row*, *col*)
Retrieve lon and lat of single pixel.

Parameters

- **row** (*int*) –
- **col** (*int*) –

Returns **(lon, lat)**

Return type tuple of floats

get_lonlats(*data_slice=None*, *chunks=None*, ***kwargs*)
Get longitude and latitude arrays representing this geometry.

Returns **(lon, lat)** – If *chunks* is provided then the arrays will be dask arrays with the provided chunk size. If *chunks* is not provided then the returned arrays are the same as the internal data types of this geometry object (numpy or dask).

Return type tuple of numpy arrays

get_lonlats_dask(*chunks=None*)
Get the lon lats as a single dask array.

intersection(*other*)
Return the corners of the intersection polygon of the current area with *other*.

Parameters **other** (*object*) – Instance of subclass of BaseDefinition

Returns **(corner1, corner2, corner3, corner4)**

Return type tuple of points

overlap_rate(*other*)
Get how much the current area overlaps an *other* area.

Parameters **other** (*object*) – Instance of subclass of BaseDefinition

Returns **overlap_rate**

Return type float

overlaps(*other*)
Test if the current area overlaps the *other* area.

This is based solely on the corners of areas, assuming the boundaries to be great circles.

Parameters **other** (*object*) – Instance of subclass of BaseDefinition

Returns **overlaps**

Return type bool

class pyresample.geometry.**CoordinateDefinition**(*lons*, *lats*, *nprocs=1*)
Bases: *pyresample.geometry.BaseDefinition*

Base class for geometry definitions defined by lons and lats only.

append(*other*)
Append another coordinate definition to existing one.

concatenate(*other*)
Concatenate coordinate definitions.

geocentric_resolution(*ellps='WGS84'*, *radius=None*, *nadir_factor=2*)
Calculate maximum geocentric pixel resolution.

If *lons* is a `xarray.DataArray` object with a *resolution* attribute, this will be used instead of loading the longitude and latitude data. In this case the resolution attribute is assumed to mean the nadir resolution of a swath and will be multiplied by the *nadir_factor* to adjust for increases in the spatial resolution towards the limb of the swath.

Parameters

- **ellps** (*str*) – PROJ Ellipsoid for the Cartographic projection used as the target geocentric coordinate reference system. Default: 'WGS84'. Ignored if *radius* is provided.
- **radius** (*float*) – Spherical radius of the Earth to use instead of the definitions in *ellps*.
- **nadir_factor** (*int*) – Number to multiply the nadir resolution attribute by to reflect pixel size on the limb of the swath.

Returns: Estimated maximum pixel size in meters on a geocentric coordinate system (X, Y, Z) representing the Earth.

Raises: RuntimeError if a simple search for valid longitude/latitude data points found no valid data points.

exception pyresample.geometry.**DimensionError**
Bases: ValueError

Wrap ValueError.

class pyresample.geometry.**DynamicAreaDefinition**(*area_id=None*, *description=None*, *projection=None*, *width=None*, *height=None*, *area_extent=None*, *resolution=None*, *optimize_projection=False*, *rotation=None*)
Bases: object

An AreaDefintion containing just a subset of the needed parameters.

The purpose of this class is to be able to adapt the area extent and shape of the area to a given set of longitudes and latitudes, such that e.g. polar satellite granules can be resampled optimally to a given projection.

Parameters

- **area_id** – The name of the area.
- **description** – The description of the area.
- **projection** – The dictionary or string of projection parameters. Doesn't have to be complete. If not complete, proj_info must be provided to freeze to "fill in" any missing parameters.
- **width** – x dimension in number of pixels, aka number of grid columns
- **height** – y dimension in number of pixels, aka number of grid rows
- **shape** – Corresponding array shape as (height, width)
- **area_extent** – The area extent of the area.
- **pixel_size_x** – Pixel width in projection units
- **pixel_size_y** – Pixel height in projection units
- **resolution** – Resolution of the resulting area as (pixel_size_x, pixel_size_y) or a scalar if pixel_size_x == pixel_size_y.
- **optimize_projection** – Whether the projection parameters have to be optimized.
- **rotation** – Rotation in degrees (negative is cw)

compute_domain(*corners*, *resolution=None*, *shape=None*)
Compute shape and area_extent from corners and [shape or resolution] info.

Corners represents the center of pixels, while area_extent represents the edge of pixels.

Note that shape is (rows, columns) and resolution is (x_size, y_size); the dimensions are flipped.

freeze(*lonslats=None*, *resolution=None*, *shape=None*, *proj_info=None*)
Create an AreaDefinition from this area with help of some extra info.

Parameters

- **lonlats** (SwathDefinition *or* *tuple*) – The geographical coordinates to contain in the resulting area. A tuple should be (lons, lats).
- **resolution** – the resolution of the resulting area.
- **shape** – the shape of the resulting area.
- **proj_info** – complementing parameters to the projection info.

- **and shape parameters are ignored if the instance is created** (*Resolution*) –
- **the optimize_projection flag set to True.** (*with*) –

pixel_size_x
Return pixel size in X direction.

pixel_size_y
Return pixel size in Y direction.

class pyresample.geometry.**GridDefinition**(*lons*, *lats*, *nprocs=1*)
Bases: *pyresample.geometry.CoordinateDefinition*

Grid defined by lons and lats.

Parameters

- **lons** (*numpy array*) –
- **lats** (*numpy array*) –
- **nprocs** (*int, optional*) – Number of processor cores to be used for calculations.

shape
Grid shape as (rows, cols)

Type tuple

size
Number of elements in grid

Type int

lons
Grid lons

Type object

lats
Grid lats

Type object

cartesian_coords
Grid cartesian coordinates

Type object

exception pyresample.geometry.**IncompatibleAreas**
Bases: ValueError

Error when the areas to combine are not compatible.

class pyresample.geometry.**StackedAreaDefinition**(**definitions*, ***kwargs*)
Bases: *pyresample.geometry.BaseDefinition*

Definition based on muliple vertically stacked AreaDefinitions.

append(*definition*)
Append another definition to the area.

get_lonlats(*nprocs=None*, *data_slice=None*, *cache=False*, *dtype=None*, *chunks=None*)
Return lon and lat arrays of the area.

get_lonlats_dask(*chunks=None*, *dtype=None*)
Return lon and lat dask arrays of the area.

height
Return height of the area definition.

proj4_string
Return projection definition as Proj.4 string.

proj_str
Return projection definition as Proj.4 string.

shape
Return shape of the area definition.

size
Return size of the area definition.

squeeze()
Generate a single AreaDefinition if possible.

update_hash(*the_hash=None*)
Update the hash.

width
Return width of the area definition.

x_size
Return width of the area definition.

y_size
Return height of the area definition.

class pyresample.geometry.**SwathDefinition**(*lons*, *lats*, *nprocs=1*)
Bases: *pyresample.geometry.CoordinateDefinition*

Swath defined by lons and lats.

Parameters

- **lons** (*numpy array*) –
- **lats** (*numpy array*) –
- **nprocs** (*int, optional*) – Number of processor cores to be used for calculations.

shape
Swath shape

Type tuple

size
Number of elements in swath

Type int

ndims
Swath dimensions

Type int

lons
Swath lons

Type object

lats
Swath lats

Type object

cartesian_coords
Swath cartesian coordinates

Type object

aggregate(***dims*)
Aggregate the current swath definition by averaging.

For example, averaging over 2x2 windows: *sd.aggregate(x=2, y=2)*

compute_bb_proj_params(*proj_dict*)
Compute BB projection parameters.

compute_optimal_bb_area(*proj_dict=None*)
Compute the "best" bounding box area for this swath with *proj_dict*.

By default, the projection is Oblique Mercator (*omerc* in proj.4), in which case the right projection angle *alpha* is computed from the swath centerline. For other projections, only the appropriate center of projection and area extents are computed.

The height and width are computed so that the resolution is approximately the same across dimensions.

copy()
Copy the current swath.

get_edge_lonlats()
Get the concatenated boundary of the current swath.

update_hash(*the_hash=None*)
Update the hash.

pyresample.geometry.**combine_area_extents_vertical**(*area1*, *area2*)
Combine the area extents of areas 1 and 2.

pyresample.geometry.**concatenate_area_defs**(*area1*, *area2*, *axis=0*)
Append *area2* to *area1* and return the results.

pyresample.geometry.**get_array_hashable**(*arr*)
Compute a hashable form of the array *arr*.

Works with numpy arrays, dask.array.Array, and xarray.DataArray.

pyresample.geometry.**get_geostationary_angle_extent**(*geos_area*)
Get the max earth (vs space) viewing angles in x and y.

pyresample.geometry.**get_geostationary_bounding_box**(*geos_area*, *nb_points=50*)
Get the bbox in lon/lats of the valid pixels inside *geos_area*.

Parameters **nb_points** – Number of points on the polygon

pyresample.geometry.**invproj**(*data_x*, *data_y*, *proj_dict*)
Perform inverse projection.

pyresample.geometry.**ordered_dump**(*data*, *stream=None*, *Dumper=<class 'yaml.dumper.Dumper'>*, ***kwds*)
Dump the data to YAML in ordered fashion.

1.11.8 pyresample.grid module

Resample image from one projection to another using nearest neighbour method in cartesian projection coordinate systems

pyresample.grid.**get_image_from_linesample**(*row_indices*, *col_indices*, *source_image*, *fill_value=0*)

Samples from image based on index arrays.

Parameters

- **row_indices** (*numpy array*) – Row indices. Dimensions must match col_indices
- **col_indices** (*numpy array*) – Col indices. Dimensions must match row_indices
- **source_image** (*numpy array*) – Source image
- **fill_value** (*int or None, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked

Returns **image_data** – Resampled image

Return type numpy array

pyresample.grid.**get_image_from_lonlats**(*lons*, *lats*, *source_area_def*, *source_image_data*, *fill_value=0*, *nprocs=1*)

Samples from image based on lon lat arrays using nearest neighbour method in cartesian projection coordinate systems.

Parameters

- **lons** (*numpy array*) – Lons. Dimensions must match lats
- **lats** (*numpy array*) – Lats. Dimensions must match lons
- **source_area_def** (*object*) – Source definition as AreaDefinition object
- **source_image_data** (*numpy array*) – Source image data
- **fill_value** (*int or None, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **nprocs** (*int, optional*) – Number of processor cores to be used

Returns **image_data** – Resampled image data

Return type numpy array

pyresample.grid.**get_linesample**(*lons*, *lats*, *source_area_def*, *nprocs=1*)

Returns index row and col arrays for resampling

Parameters

- **lons** (*numpy array*) – Lons. Dimensions must match lats
- **lats** (*numpy array*) – Lats. Dimensions must match lons
- **source_area_def** (*object*) – Source definition as AreaDefinition object
- **nprocs** (*int, optional*) – Number of processor cores to be used

Returns **(row_indices, col_indices)** – Arrays for resampling area by array indexing

Return type tuple of numpy arrays

pyresample.grid.**get_resampled_image**(*target_area_def*, *source_area_def*, *source_image_data*, *fill_value=0*, *nprocs=1*, *segments=None*)

Resamples image using nearest neighbour method in cartesian projection coordinate systems.

Parameters

- **target_area_def** (*object*) – Target definition as AreaDefinition object
- **source_area_def** (*object*) – Source definition as AreaDefinition object

- **source_image_data** (*numpy array*) – Source image data
- **fill_value** (*{int, None} optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **segments** (*{int, None} optional*) – Number of segments to use when resampling. If set to None an estimate will be calculated.

Returns **image_data** – Resampled image data

Return type numpy array

1.11.9 pyresample.image module

Handles resampling of images with assigned geometry definitions

class pyresample.image.**ImageContainer**(*image_data*, *geo_def*, *fill_value=0*, *nprocs=1*)

Bases: object

Holds image with geometry definition. Allows indexing with linesample arrays.

Parameters

- **image_data** (*numpy array*) – Image data
- **geo_def** (*object*) – Geometry definition
- **fill_value** (*int or None, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **nprocs** (*int, optional*) – Number of processor cores to be used

image_data

Image data

Type numpy array

geo_def

Geometry definition

Type object

fill_value

Resample result fill value

Type int or None

nprocs

Number of processor cores to be used for geometry operations

Type int

get_array_from_linesample(*row_indices*, *col_indices*)

Samples from image based on index arrays.

Parameters

- **row_indices** (*numpy array*) – Row indices. Dimensions must match col_indices
- **col_indices** (*numpy array*) – Col indices. Dimensions must match row_indices

Returns **image_data** – Resampled image data

Return type numpy_array

get_array_from_neighbour_info(**args*, ***kwargs*)
Base method for resampling from preprocessed data.

resample(*target_geo_def*)
Base method for resampling

class pyresample.image.**ImageContainerBilinear**(*image_data*, *geo_def*, *radius_of_influence*, *epsilon=0*, *fill_value=0*, *reduce_data=False*, *nprocs=1*, *segments=None*, *neighbours=32*)

Bases: *pyresample.image.ImageContainer*

Holds image with geometry definition. Allows bilinear to new geometry definition.

Parameters

- **image_data** (*numpy array*) – Image data
- **geo_def** (*object*) – Geometry definition
- **radius_of_influence** (*float*) – Cut off distance in meters
- **epsilon** (*float, optional*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time
- **fill_value** (*int or None, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **reduce_data** (*bool, optional*) – Perform coarse data reduction before resampling in order to reduce execution time
- **nprocs** (*int, optional*) – Number of processor cores to be used for geometry operations
- **segments** (*int or None*) – Number of segments to use when resampling. If set to None an estimate will be calculated

image_data
Image data

Type numpy array

geo_def
Geometry definition

Type object

radius_of_influence
Cut off distance in meters

Type float

epsilon
Allowed uncertainty in meters

Type float

fill_value
Resample result fill value

Type int or None

reduce_data
Perform coarse data reduction before resampling

Type bool

nprocs
Number of processor cores to be used

Type int

segments
Number of segments to use when resampling

Type int or None

resample(*target_geo_def*)
Resamples image to area definition using bilinear approach

Parameters **target_geo_def** (*object*) – Target geometry definition

Returns **image_container** – ImageContainerBilinear object of resampled geometry

Return type object

class pyresample.image.**ImageContainerNearest**(*image_data*, *geo_def*, *radius_of_influence*, *epsilon=0*, *fill_value=0*, *reduce_data=True*, *nprocs=1*, *segments=None*)
Bases: *pyresample.image.ImageContainer*

Holds image with geometry definition. Allows nearest neighbour to new geometry definition.

Parameters

- **image_data** (*numpy array*) – Image data
- **geo_def** (*object*) – Geometry definition
- **radius_of_influence** (*float*) – Cut off distance in meters
- **epsilon** (*float, optional*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time
- **fill_value** (*int or None, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **reduce_data** (*bool, optional*) – Perform coarse data reduction before resampling in order to reduce execution time
- **nprocs** (*int, optional*) – Number of processor cores to be used for geometry operations
- **segments** (*int or None*) – Number of segments to use when resampling. If set to None an estimate will be calculated

image_data
Image data

Type numpy array

geo_def
Geometry definition

Type object

radius_of_influence
Cut off distance in meters

Type float

epsilon
Allowed uncertainty in meters

Type float

fill_value
Resample result fill value

Type int or None

reduce_data
Perform coarse data reduction before resampling

Type bool

nprocs
Number of processor cores to be used

Type int

segments
Number of segments to use when resampling

Type int or None

resample(*target_geo_def*)
Resamples image to area definition using nearest neighbour approach

Parameters **target_geo_def** (*object*) – Target geometry definition

Returns **image_container** – ImageContainerNearest object of resampled geometry

Return type object

class pyresample.image.**ImageContainerQuick**(*image_data*, *geo_def*, *fill_value=0*, *nprocs=1*, *segments=None*)
Bases: *pyresample.image.ImageContainer*

Holds image with area definition. ‘ Allows quick resampling within area.

Parameters

- **image_data** (*numpy array*) – Image data
- **geo_def** (*object*) – Area definition as AreaDefinition object
- **fill_value** (*int or None, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **nprocs** (*int, optional*) – Number of processor cores to be used for geometry operations
- **segments** (*int or None*) – Number of segments to use when resampling. If set to None an estimate will be calculated

image_data
Image data

Type numpy array

geo_def
Area definition as AreaDefinition object

Type object

fill_value
Resample result fill value If fill_value is None a masked array is returned with undetermined pixels masked

Type int or None

nprocs
Number of processor cores to be used

Type int

segments
Number of segments to use when resampling

Type int or None

resample(*target_area_def*)
Resamples image to area definition using nearest neighbour approach in projection coordinates.

Parameters **target_area_def** (*object*) – Target area definition as AreaDefinition object

Returns **image_container** – ImageContainerQuick object of resampled area

Return type object

1.11.10 pyresample.kd_tree module

Handles reprojection of geolocated data. Several types of resampling are supported

exception pyresample.kd_tree.**EmptyResult**
Bases: ValueError

class pyresample.kd_tree.**XArrayResamplerNN**(*source_geo_def*, *target_geo_def*, *radius_of_influence=None*, *neighbours=1*, *epsilon=0*)
Bases: object

get_neighbour_info(*mask=None*)
Return neighbour info.

Returns

- *(valid_input_index, valid_output_index,*
- **index_array, distance_array)** (*tuple of numpy arrays*) – Neighbour resampling info

get_sample_from_neighbour_info(*data*, *fill_value=nan*)
Get the pixels matching the target area.

This method should work for any dimensionality of the provided data array as long as the geolocation dimensions match in size and name in `data.dims`. Where source area definition are *AreaDefinition* objects the corresponding dimensions in the data should be `('y', 'x')`.

This method also attempts to preserve chunk sizes of dask arrays, but does require loading/sharing the fully computed source data before it can actually compute the values to write to the destination array. This can result in large memory usage for large source data arrays, but is a necessary evil until fancier indexing is supported by dask and/or pykdtree.

Parameters

- **data** (*xarray.DataArray*) – Source data pixels to sample
- **fill_value** (*float*) – Output fill value when no source data is near the target pixel. When omitted, if the input data is an integer array then the maximum value for that integer type is used, but otherwise, NaN is used and can be detected in the result with `res.isnull()`.

Returns

The resampled array. The dtype of the array will be the same as the input data. Pixels with no matching data from the input array will be filled (see the *fill_value* parameter description above).

Return type dask.array.Array

query_resample_kdtree(*resample_kdtree*, *tlons*, *tlats*, *valid_oi*, *mask*)
Query kd-tree on slice of target coordinates.

pyresample.kd_tree.**get_neighbour_info**(*source_geo_def*, *target_geo_def*, *radius_of_influence*, *neighbours=8*, *epsilon=0*, *reduce_data=True*, *nprocs=1*, *segments=None*)
Returns neighbour info

Parameters

- **source_geo_def** (*object*) – Geometry definition of source
- **target_geo_def** (*object*) – Geometry definition of target
- **radius_of_influence** (*float*) – Cut off distance in meters
- **neighbours** (*int, optional*) – The number of neigbours to consider for each grid point
- **epsilon** (*float, optional*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time
- **reduce_data** (*bool, optional*) – Perform initial coarse reduction of source dataset in order to reduce execution time
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **segments** (*int or None*) – Number of segments to use when resampling. If set to None an estimate will be calculated

Returns

- *(valid_input_index, valid_output_index,*
- **index_array, distance_array)** (*tuple of numpy arrays*) – Neighbour resampling info

pyresample.kd_tree.**get_sample_from_neighbour_info**(*resample_type*, *output_shape*, *data*, *valid_input_index*, *valid_output_index*, *index_array*, *distance_array=None*, *weight_funcs=None*, *fill_value=0*, *with_uncert=False*)
Resamples swath based on neighbour info

Parameters

- **resample_type** (*{'nn', 'custom'}*) – ‘nn’: Use nearest neighbour resampling ‘custom’: Resample based on weight_funcs
- **output_shape** (*(int, int)*) – Shape of output as (rows, cols)
- **data** (*numpy array*) – Source data
- **valid_input_index** (*numpy array*) – valid_input_index from get_neighbour_info
- **valid_output_index** (*numpy array*) – valid_output_index from get_neighbour_info
- **index_array** (*numpy array*) – index_array from get_neighbour_info

- **distance_array** (*numpy array, optional*) – distance_array from get_neighbour_info Not needed for 'nn' resample type
- **weight_funcs** (*list of function objects or function object, optional*) – List of weight functions f(dist) to use for the weighting of each channel 1 to k. If only one channel is resampled weight_funcs is a single function object. Must be supplied when using 'custom' resample type
- **fill_value** (*int, float, numpy floating, numpy integer or None, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked

Returns **result** – Source data resampled to target geometry

Return type numpy array

pyresample.kd_tree.**lonlat2xyz**(*lons*, *lats*)

pyresample.kd_tree.**query_no_distance**(*target_lons*, *target_lats*, *valid_output_index*, *mask=None*, *valid_input_index=None*, *neighbours=None*, *epsilon=None*, *radius=None*, *kdtree=None*)

Query the kdtree. No distances are returned.

NOTE: Dask array arguments must always come before other keyword arguments for *da.blockwise* arguments to work.

pyresample.kd_tree.**resample_custom**(*source_geo_def*, *data*, *target_geo_def*, *radius_of_influence*, *weight_funcs*, *neighbours=8*, *epsilon=0*, *fill_value=0*, *reduce_data=True*, *nprocs=1*, *segments=None*, *with_uncert=False*)

Resamples data using kd-tree custom radial weighting neighbour approach

Parameters

- **source_geo_def** (*object*) – Geometry definition of source
- **data** (*numpy array*) – Array of single channel data points or (source_geo_def.shape, k) array of k channels of datapoints
- **target_geo_def** (*object*) – Geometry definition of target
- **radius_of_influence** (*float*) – Cut off distance in meters
- **weight_funcs** (*list of function objects or function object*) – List of weight functions f(dist) to use for the weighting of each channel 1 to k. If only one channel is resampled weight_funcs is a single function object.
- **neighbours** (*int, optional*) – The number of neigbours to consider for each grid point
- **epsilon** (*float, optional*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time
- **fill_value** (*{int, None}, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **reduce_data** (*bool, optional*) – Perform initial coarse reduction of source dataset in order to reduce execution time
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **segments** (*{int, None}*) – Number of segments to use when resampling. If set to None an estimate will be calculated

Returns

- **data** (*numpy array (default)*) – Source data resampled to target geometry
- **data, stddev, counts** (*numpy array, numpy array, numpy array (if with_uncert == True)*) – Source data resampled to target geometry. Weighted standard devaition for all pixels having more than one source value Counts of number of source values used in weighting per pixel

pyresample.kd_tree.**resample_gauss**(*source_geo_def*, *data*, *target_geo_def*, *radius_of_influence*, *sigmas*, *neighbours=8*, *epsilon=0*, *fill_value=0*, *reduce_data=True*, *nprocs=1*, *segments=None*, *with_uncert=False*)

Resamples data using kd-tree gaussian weighting neighbour approach.

Parameters

- **source_geo_def** (*object*) – Geometry definition of source
- **data** (*numpy array*) – Array of single channel data points or (source_geo_def.shape, k) array of k channels of datapoints
- **target_geo_def** (*object*) – Geometry definition of target
- **radius_of_influence** (*float*) – Cut off distance in meters
- **sigmas** (*list of floats or float*) – List of sigmas to use for the gauss weighting of each channel 1 to k, w_k = exp(-dist^2/sigma_k^2). If only one channel is resampled sigmas is a single float value.
- **neighbours** (*int, optional*) – The number of neigbours to consider for each grid point
- **epsilon** (*float, optional*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time
- **fill_value** (*{int, None}, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **reduce_data** (*bool, optional*) – Perform initial coarse reduction of source dataset in order to reduce execution time
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **segments** (*int or None*) – Number of segments to use when resampling. If set to None an estimate will be calculated
- **with_uncert** (*bool, optional*) – Calculate uncertainty estimates

Returns

- **data** (*numpy array (default)*) – Source data resampled to target geometry
- **data, stddev, counts** (*numpy array, numpy array, numpy array (if with_uncert == True)*) – Source data resampled to target geometry. Weighted standard devaition for all pixels having more than one source value Counts of number of source values used in weighting per pixel

pyresample.kd_tree.**resample_nearest**(*source_geo_def*, *data*, *target_geo_def*, *radius_of_influence*, *epsilon=0*, *fill_value=0*, *reduce_data=True*, *nprocs=1*, *segments=None*)

Resamples data using kd-tree nearest neighbour approach

Parameters

- **source_geo_def** (*object*) – Geometry definition of source

- **data** (*numpy array*) – 1d array of single channel data points or (source_size, k) array of k channels of datapoints
- **target_geo_def** (*object*) – Geometry definition of target
- **radius_of_influence** (*float*) – Cut off distance in meters
- **epsilon** (*float, optional*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time
- **fill_value** (*int, float, numpy floating, numpy integer or None, optional*) – Set undetermined pixels to this value. If fill_value is None a masked array is returned with undetermined pixels masked
- **reduce_data** (*bool, optional*) – Perform initial coarse reduction of source dataset in order to reduce execution time
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **segments** (*int or None*) – Number of segments to use when resampling. If set to None an estimate will be calculated

Returns **data** – Source data resampled to target geometry

Return type numpy array

1.11.11 pyresample.plot module

Utility functions for quick and easy display.

pyresample.plot.**area_def2basemap**(*area_def*, ***kwargs*)

Get Basemap object from an AreaDefinition object.

Parameters

- **area_def** (*object*) – geometry.AreaDefinition object
- ****kwargs** (*Keyword arguments*) – Additional initialization arguments for Basemap

Returns **bmap**

Return type Basemap object

pyresample.plot.**ellps2axis**(*ellps_name*)

Get semi-major and semi-minor axis from ellipsis definition.

Parameters **ellps_name** (*str*) – Standard name of ellipsis

Returns **(a, b)**

Return type semi-major and semi-minor axis

pyresample.plot.**save_quicklook**(*filename*, *area_def*, *data*, *vmin=None*, *vmax=None*, *label='Variable (units)'*, *num_meridians=45*, *num_parallels=10*, *coast_res='110m'*, *cmap='RdBu_r'*)

Display and save default quicklook plot.

Parameters

- **filename** (*str*) – path to output file
- **area_def** (*object*) – geometry.AreaDefinition object
- **data** (*numpy array | numpy masked array*) – 2D array matching area_def. Use masked array for transparent values

- **vmin** (*float, optional*) – Min value for luminescence scaling
- **vmax** (*float, optional*) – Max value for luminescence scaling
- **label** (*str, optional*) – Label for data
- **num_meridians** (*int, optional*) – Number of meridians to plot on the globe
- **num_parallels** (*int, optional*) – Number of parallels to plot on the globe
- **coast_res** (*{'c', 'l', 'i', 'h', 'f'}, optional*) – Resolution of coastlines

pyresample.plot.**show_quicklook**(*area_def*, *data*, *vmin=None*, *vmax=None*, *label='Variable (units)'*, *num_meridians=45*, *num_parallels=10*, *coast_res='110m'*, *cmap='RdBu_r'*)

Display default quicklook plot.

Parameters

- **area_def** (*object*) – geometry.AreaDefinition object
- **data** (*numpy array | numpy masked array*) – 2D array matching area_def. Use masked array for transparent values
- **vmin** (*float, optional*) – Min value for luminescence scaling
- **vmax** (*float, optional*) – Max value for luminescence scaling
- **label** (*str, optional*) – Label for data
- **num_meridians** (*int, optional*) – Number of meridians to plot on the globe
- **num_parallels** (*int, optional*) – Number of parallels to plot on the globe
- **coast_res** (*{'c', 'l', 'i', 'h', 'f'}, optional*) – Resolution of coastlines

Returns **bmap**

Return type Basemap object

1.11.12 pyresample.resampler module

Base resampler class made for subclassing.

class pyresample.resampler.**BaseResampler**(*source_geo_def*, *target_geo_def*)

Bases: object

Base abstract resampler class.

compute(*data*, ***kwargs*)

Do the actual resampling.

This must be implemented by subclasses.

get_hash(*source_geo_def=None*, *target_geo_def=None*, ***kwargs*)

Get hash for the current resample with the given *kwargs*.

precompute(***kwargs*)

Do the precomputation.

This is an optional step if the subclass wants to implement more complex features like caching or can share some calculations between multiple datasets to be processed.

resample(*data*, *cache_dir=None*, *mask_area=None*, ***kwargs*)
Resample *data* by calling *precompute* and *compute* methods.

Only certain resampling classes may use *cache_dir* and the *mask* provided when *mask_area* is True. The return value of calling the *precompute* method is passed as the *cache_id* keyword argument of the *compute* method, but may not be used directly for caching. It is up to the individual resampler subclasses to determine how this is used.

Parameters

- **data** (*xarray.DataArray*) – Data to be resampled
- **cache_dir** (*str*) – directory to cache precomputed results (default False, optional)
- **mask_area** (*bool*) – Mask geolocation data where data values are invalid. This should be used when data values may affect what neighbors are considered valid.

Returns (xarray.DataArray): Data resampled to the target area

pyresample.resampler.**hash_dict**(*the_dict*, *the_hash=None*)
Calculate a hash for a dictionary.

1.11.13 pyresample.spherical module

Some generalized spherical functions.

base type is a numpy array of size (n, 2) (2 for lon and lats)

class pyresample.spherical.**Arc**(*start*, *end*)
Bases: object

An arc of the great circle between two points.

angle(*other_arc*)
Oriented angle between two arcs.

get_next_intersection(*arcs*, *known_inter=None*)
Get the next intersection between the current arc and *arcs*

intersection(*other_arc*)
Return where, if the current arc and the *other_arc* intersect.

None is returned if there is not intersection. An arc is defined as the shortest tracks between two points.

intersections(*other_arc*)
Gives the two intersections of the greats circles defined by the current arc and *other_arc*. From http://williams.best.vwh.net/intersect.htm

intersects(*other_arc*)
Check if the current arc and the *other_arc* intersect.

An arc is defined as the shortest tracks between two points.

class pyresample.spherical.**CCoordinate**(*cart*)
Bases: object

Cartesian coordinates

cross(*point*)
cross product with another vector.

dot(*point*)
dot product with another vector.

norm()
Euclidean norm of the vector.

normalize()
normalize the vector.

to_spherical()

class pyresample.spherical.**SCoordinate**(*lon*, *lat*)
Bases: object

Spherical coordinates.

cross2cart(*point*)
Compute the cross product, and convert to cartesian coordinates.

distance(*point*)
Vincenty formula.

hdistance(*point*)
Haversine formula.

to_cart()
Convert to cartesian.

class pyresample.spherical.**SphPolygon**(*vertices*, *radius=1*)
Bases: object

Spherical polygon.

Vertices as a 2-column array of (col 1) lons and (col 2) lats is given in radians. The inside of the polygon is defined by the vertices being defined clockwise around it.

aedges()
Iterator over the edges, in arcs of Coordinates.

area()
Find the area of a polygon.

The inside of the polygon is defined by having the vertices enumerated clockwise around it.

Uses the algorithm described in [bev1987].

Note: The article mixes up longitudes and latitudes in equation 3! Look at the fortran code appendix for the correct version.

edges()
Iterator over the edges, in geographical coordinates.

intersection(*other*)
Return the intersection of this and *other* polygon.

inverse()
Return an inverse of the polygon.

invert()
Invert the polygon.

union(*other*)
Return the union of this and *other* polygon.

pyresample.spherical.**modpi**(*val*, *mod=3.141592653589793*)
Puts *val* between -*mod* and *mod*.

1.11.14 pyresample.spherical_geometry module

Classes for spherical geometry operations

class pyresample.spherical_geometry.**Arc**(*start*, *end*)
Bases: object

An arc of the great circle between two points.

angle(*other_arc*, *snap=True*)
Oriented angle between two arcs.

Parameters

- **other_arc** (pyresample.spherical_geometry.Arc) –
- **snap** (*boolean*) – Snap small angles to 0. Allows for detecting colinearity. Disable snapping when calculating polygon areas as it might lead to negative area values.

center_angle()
Angle of an arc at the center of the sphere.

end = None

intersection(*other_arc*)
Says where, if two arcs defined by the current arc and the *other_arc* intersect. An arc is defined as the shortest tracks between two points.

intersections(*other_arc*)
Gives the two intersections of the greats circles defined by the current arc and *other_arc*.

intersects(*other_arc*)
Says if two arcs defined by the current arc and the *other_arc* intersect. An arc is defined as the shortest tracks between two points.

start = None

class pyresample.spherical_geometry.**Coordinate**(*lon=None*, *lat=None*, *x__=None*, *y__=None*, *z__=None*, *R__=1*)
Bases: object

Point on earth in terms of lat and lon.

cross(*point*)
cross product with another vector.

cross2cart(*point*)
Compute the cross product, and convert to cartesian coordinates (assuming radius 1).

distance(*point*)
Vincenty formula.

dot(*point*)
dot product with another vector.

lat = None

lon = None

norm()
Return the norm of the vector.

normalize()
normalize the vector.

x__ = None

y__ = None

z__ = None

pyresample.spherical_geometry.**get_first_intersection**(*b__*, *boundaries*)
Get the first intersection on *b__* with *boundaries*.

pyresample.spherical_geometry.**get_intersections**(*b__*, *boundaries*)
Get the intersections of *b__* with *boundaries*. Returns both the intersection coordinates and the concerned boundaries.

pyresample.spherical_geometry.**get_next_intersection**(*p__*, *b__*, *boundaries*)
Get the next intersection from the intersection of arcs *p__* and *b__* along segment *b__* with *boundaries*.

pyresample.spherical_geometry.**get_polygon_area**(*corners*)
Get the area of the convex area defined by *corners*.

pyresample.spherical_geometry.**intersection_polygon**(*area_corners*, *segment_corners*)
Get the intersection polygon between two areas.

pyresample.spherical_geometry.**modpi**(*val*)
Puts *val* between -pi and pi.

pyresample.spherical_geometry.**point_inside**(*point*, *corners*)
Is a point inside the 4 corners ? This uses great circle arcs as area boundaries.

1.11.15 pyresample.version module

Git implementation of _version.py.

exception pyresample.version.**NotThisMethod**
Bases: Exception

Exception raised if a method is not valid for the current scenario.

class pyresample.version.**VersioneerConfig**
Bases: object

Container for Versioneer configuration parameters.

pyresample.version.**get_config**()
Create, populate and return the VersioneerConfig() object.

pyresample.version.**get_keywords**()
Get the keywords needed to look up the version information.

pyresample.version.**get_versions**()
Get version information or return default if unable to do so.

pyresample.version.**git_get_keywords**(*versionfile_abs*)
Extract version information from the given file.

pyresample.version.**git_pieces_from_vcs**(*tag_prefix*, *root*, *verbose*, *run_command=<function run_command>*)
Get version from 'git describe' in the root of the source tree.

This only gets called if the git-archive 'subst' keywords were *not* expanded, and _version.py hasn't already been rewritten with a short version string, meaning we're inside a checked out source tree.

pyresample.version.**git_versions_from_keywords**(*keywords*, *tag_prefix*, *verbose*)
Get version information from git keywords.

pyresample.version.**plus_or_dot**(*pieces*)
Return a + if we don't already have one, else return a .

pyresample.version.**register_vcs_handler**(*vcs*, *method*)
Decorator to mark a method as the handler for a particular VCS.

pyresample.version.**render**(*pieces*, *style*)
Render the given version pieces into the requested style.

pyresample.version.**render_git_describe**(*pieces*)
TAG[-DISTANCE-gHEX][-dirty].

Like 'git describe –tags –dirty –always'.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

pyresample.version.**render_git_describe_long**(*pieces*)
TAG-DISTANCE-gHEX[-dirty].

Like 'git describe –tags –dirty –always -long'. The distance/hash is unconditional.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

pyresample.version.**render_pep440**(*pieces*)
Build up version string, with post-release "local version identifier".

Our goal: TAG[+DISTANCE.gHEX[.dirty]] . Note that if you get a tagged build and then dirty it, you'll get TAG+0.gHEX.dirty

Exceptions: 1: no tags. git_describe was just HEX. 0+untagged.DISTANCE.gHEX[.dirty]

pyresample.version.**render_pep440_old**(*pieces*)
TAG[.postDISTANCE[.dev0]] .

The ".dev0" means dirty.

Eexceptions: 1: no tags. 0.postDISTANCE[.dev0]

pyresample.version.**render_pep440_post**(*pieces*)
TAG[.postDISTANCE[.dev0]+gHEX] .

The ".dev0" means dirty. Note that .dev0 sorts backwards (a dirty tree will appear "older" than the corresponding clean one), but you shouldn't be releasing software with -dirty anyways.

Exceptions: 1: no tags. 0.postDISTANCE[.dev0]

pyresample.version.**render_pep440_pre**(*pieces*)
TAG[.post.devDISTANCE] – No -dirty.

Exceptions: 1: no tags. 0.post.devDISTANCE

pyresample.version.**run_command**(*commands*, *args*, *cwd=None*, *verbose=False*, *hide_stderr=False*, *env=None*)
Call the given command(s).

pyresample.version.**versions_from_parentdir**(*parentdir_prefix*, *root*, *verbose*)
Try to determine the version from the parent directory name.

Source tarballs conventionally unpack into a directory that includes both the project name and a version string. We will also support searching up two directory levels for an appropriately named parent directory

1.11.16 Module contents

pyresample.**load_area**(*area_file_name*, **regions*)

Load area(s) from area file.

Parameters

- **area_file_name** (*str, pathlib.Path, stream, or list thereof*) – List of paths or streams. Any str or pathlib.Path will be interpreted as a path to a file. Any stream will be interpreted as containing a yaml definition. To read directly from a string, use `load_area_from_string()`.
- **regions** (*str argument list*) – Regions to parse. If no regions are specified all regions in the file are returned

Returns **area_defs** – If one area name is specified a single AreaDefinition object is returned. If several area names are specified a list of AreaDefinition objects is returned

Return type *AreaDefinition* or list

Raises AreaNotFound: – If a specified area name is not found

pyresample.**create_area_def**(*area_id*, *projection*, *width=None*, *height=None*, *area_extent=None*, *shape=None*, *upper_left_extent=None*, *center=None*, *resolution=None*, *radius=None*, *units=None*, ***kwargs*)

Takes data the user knows and tries to make an area definition from what can be found.

Parameters

- **area_id** (*str*) – ID of area
- **projection** (*dict or str*) – Projection parameters as a proj4_dict or proj4_string
- **description** (*str, optional*) – Description/name of area. Defaults to area_id
- **proj_id** (*str, optional*) – ID of projection (deprecated)
- **units** (*str, optional*) – Units that provided arguments should be interpreted as. This can be one of 'deg', 'degrees', 'meters', 'metres', and any parameter supported by the cs2cs -lu command. Units are determined in the following priority:
  1. units expressed with each variable through a DataArray's attrs attribute.
  2. units passed to `units`
  3. units used in `projection`
  4. meters
- **width** (*str, optional*) – Number of pixels in the x direction
- **height** (*str, optional*) – Number of pixels in the y direction
- **area_extent** (*list, optional*) – Area extent as a list (lower_left_x, lower_left_y, upper_right_x, upper_right_y)
- **shape** (*list, optional*) – Number of pixels in the y and x direction (height, width)
- **upper_left_extent** (*list, optional*) – Upper left corner of upper left pixel (x, y)
- **center** (*list, optional*) – Center of projection (x, y)
- **resolution** (*list or float, optional*) – Size of pixels: (dx, dy)
- **radius** (*list or float, optional*) – Length from the center to the edges of the projection (dx, dy)

- **rotation** (*float, optional*) – rotation in degrees(negative is cw)
- **nprocs** (*int, optional*) – Number of processor cores to be used
- **lons** (*numpy array, optional*) – Grid lons
- **lats** (*numpy array, optional*) – Grid lats
- **optimize_projection** – Whether the projection parameters have to be optimized for a DynamicAreaDefinition.

Returns **AreaDefinition or DynamicAreaDefinition** – If shape and area_extent are found, an AreaDefinition object is returned. If only shape or area_extent can be found, a DynamicAreaDefinition object is returned

Return type *AreaDefinition* or *DynamicAreaDefinition*

Raises ValueError: – If neither shape nor area_extent could be found

---

Notes

- `resolution` and `radius` can be specified with one value if dx == dy
- If `resolution` and `radius` are provided as angles, center must be given or findable. In such a case, they represent [projection x distance from center[0] to center[0]+dx, projection y distance from center[1] to center[1]+dy]

---

pyresample.**get_area_def**(*area_id*, *area_name*, *proj_id*, *proj4_args*, *width*, *height*, *area_extent*, *rotation=0*)

Construct AreaDefinition object from arguments

Parameters

- **area_id** (*str*) – ID of area
- **area_name** (*str*) – Description of area
- **proj_id** (*str*) – ID of projection
- **proj4_args** (*list, dict, or str*) – Proj4 arguments as list of arguments or string
- **width** (*int*) – Number of pixel in x dimension
- **height** (*int*) – Number of pixel in y dimension
- **rotation** (*float*) – Rotation in degrees (negative is cw)
- **area_extent** (*list*) – Area extent as a list of ints (LL_x, LL_y, UR_x, UR_y)

Returns **area_def** – AreaDefinition object

Return type object

pyresample.**parse_area_file**(*area_file_name*, **regions*)

Parse area information from area file

Parameters

- **area_file_name** (*str or list*) – One or more paths to area definition files
- **regions** (*str argument list*) – Regions to parse. If no regions are specified all regions in the file are returned

Returns **area_defs** – List of AreaDefinition objects

Return type list

Raises AreaNotFound: – If a specified area is not found

pyresample.**convert_def_to_yaml**(*def_area_file*, *yaml_area_file*)

Convert a legacy area def file to the yaml counter partself.

yaml_area_file will be overwritten by the operation.

Bibliography

[bev1987] , Michael Bevis and Greg Cambareri, “Computing the area of a spherical polygon of arbitrary shape”, in *Mathematical Geology*, May 1987, Volume 19, Issue 4, pp 335-346.

Python Module Index

p

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z