repurpose Documentation

Release 0.8

Wolfgang Preimesberger

Sep 06, 2023

Contents

This package provides routines for the conversion of image formats to time series and vice versa. It is part of the poets project and works best with the readers and writers supported there. The main use case is for data that is sampled irregularly in space or time. If you have data that is sampled in regular intervals then there are alternatives to this package which might be better for your use case. See *Alternatives* for more detail.

The readers and writers have to conform to the API specifications of the base classes defined in pygeobase to work without adpation.

CHAPTER 1

Citation

If you use the software in a publication then please cite it using the Zenodo DOI. Be aware that this badge links to the latest package version.

Please select your specific version at https://doi.org/10.5281/zenodo.593577 to get the DOI of that version. You should normally always use the DOI for the specific version of your record in citations. This is to ensure that other researchers can access the exact research artefact you used for reproducibility.

You can find additional information regarding DOI versioning at http://help.zenodo.org/#versioning

CHAPTER 2

Installation

This package should be installable through pip:

```
pip install repurpose
```

CHAPTER 3

Modules

It includes two main modules:

- `img2ts` for image/swath to time series conversion, including support for spatial resampling.
- `ts2img` for time series to image conversion, including support for temporal resampling. This module is very experimental at the moment.
- `resample` for spatial resampling of (regular or irregular) gridded data to different resolutions.

CHAPTER 4

Alternatives

If you have data that can be represented as a 3D datacube then these projects might be better suited to your needs.

- PyReshaper is a package that works with NetCDF input and output and converts time slices into a time series representation.
- Climate Data Operators (CDO) can work with several input formats, stack them and change the chunking to allow time series optimized access. It assumes regular sampling in space and time as far as we know.
- netCDF Operators (NCO) are similar to CDO with a stronger focus on netCDF.

CHAPTER 5

Contribute

We are happy if you want to contribute. Please raise an issue explaining what is missing or if you find a bug. We will also gladly accept pull requests against our master branch for new features or bug fixes.

5.1 Development setup

For Development we recommend a `conda` environment

5.2 Guidelines

If you want to contribute please follow these steps:

- Fork the repurpose repository to your account
- make a new feature branch from the repurpose master branch
- Add your feature
- Please include tests for your contributions in one of the test directories. We use py.test so a simple function called test_my_feature is enough
- submit a pull request to our master branch

CHAPTER 6

Note

This project has been set up using PyScaffold 2.4.4. For details and usage information on PyScaffold see http://pyscaffold.readthedocs.org/.

CHAPTER 7

ts2img

7.1 Introduction

Conversion of time series data to images (ts2img) is a general problem that we have to deal with often for all kinds of datasets. Although most of the external datasets we use come in image format most of them are converted into a time series format for analysis. This is fairly straightforward for image stacks but gets more complicated for orbit data.

Orbit data mostly comes as several data values accompanied by latitude, longitude information and must often be resampled to fit on a grid that lends itself for time series analysis. A more or less general solution for this already exists in the img2ts module.

7.2 Possible steps involved in the conversion

The steps that a ts2img program might have to perform are:

1. Read time series in a geographic region - constrained by memory
2. Aggregate time series in time
    - methods for doing this aggregation can vary for each dataset so the method is best specified by the user
    - resolution in time has to be chosen and is probably also best specified by the user
    - after aggregation every point in time and in space must have a value which can of course be NaN
    - time series might have to be split into separate images during conversion, e.g. ASCAT time series are routinely split into images for ascending and descending satellite overpasses. **This means that we can not assume that the output dataset has the same number or names of variables as the input dataset.**
3. Put the now uniform time series of equal length into a 2D array per variable
4. A resampling step could be performed here but since only a part of the dataset is available edge cases would not be resolved correctly. A better solution would be to develop a good resampling tool which might already exist in pyresample and pytesmo functions that use it.
5. write this data into a file

- this can be a netCDF file with dimensions of the grid into which the data is written
- this could be any other file format, the interface to this format just has to make sure that in the end a consistent image dataset is built out of the parts that are written.

7.3 Solution

The chosen first solution uses netCDF as an output format. The output will be a stack of images in netCDF format. This format can easily be converted into substacks or single images if that is needed for a certain user or project.

The chosen solution will **not** do resampling since this is better and easier done using the whole converted dataset. This also means that if the input dataset is e.g. a dataset defined over land only then the resulting "image" will also not contain land points. I think it is best to let this be decided by the input dataset.

The output of the resulting netCDF can have one of two possible "shapes":

- 2D variables with time on one axis and gpi on the other. This is kind of how SWI time series are stored already.
- 3D variables with latitude, longitude and time as the three dimensions.

The decision of which it will be is dependent on the grid on which the input data is stored. If the grid has a 2D shape then the 3D solution will be chosen. If the input grid has only a 1D shape then only the 2D solution is possible.

7.3.1 Time Series aggregation

The chosen solution will use a custom function for each dataset to perform the aggregation if necessary. A simple example of a function that gets a time time series and aggregates it to a monthly time series could look like *agg_tsmonthly*

Simple example of a aggregation function

```
def agg_tsmonthly(ts, **kwargs):
    """
    Parameters
    ----------
    ts : pandas.DataFrame
        time series of a point
    kwargs : dict
        any additional keyword arguments that are given to the ts2img object
        during initialization

    Returns
    -------
    ts_agg : pandas.DataFrame
        aggregated time series, they all must have the same length
        otherwise it can not work
        each column of this DataFrame will be a layer in the image
    """
    # very simple example
    # aggregate to monthly timestamp
    # should also make sure that the output has a certain length
    return ts.asfreq("M")
```

7.3.2 Time series iteration

The function `agg_tsmonthly` will be called for every time series in the input dataset. The input dataset must have a `iter_ts` iterator that iterates over the grid points in a sensible order.

7.3.3 Interface to the netCDF writer

The netCDF writer will be initialized outside the *ts2img* class with a filename and other attributes it needs. So the *ts2img* class only gets a writer object. This writer object already knows about the start and end date of the time series as well as the target grid and has initialized the correct dimensions in the netCDF file. This object must have a method `write_ts` which takes a array of gpi's and a 2D array containing the time series for these gpis. This should be enough to write the gpi's into the correct position of the netCDF file.

This approach should also work if another output format is supposed to be used.

7.3.4 Implementation of the main ts2img class

The ts2img class will automatically use a the function given in `agg_ts2img` if no custom `agg_ts2img` function is provided. If the tsreader implements a method called `agg_ts2img` this function will be used instead.

```
class Ts2Img(object):

    """
    Takes a time series dataset and converts it
    into an image dataset.
    A custom aggregate function should be given otherwise
    a daily mean will be used

    Parameters
    ----------
    tsreader: object
        object that implements a iter_ts method which iterates over
        pandas time series and has a grid attribute that is a pytesmo
        BasicGrid or CellGrid
    imgwriter: object
        writer object that implements a write_ts method that takes
        a list of grid point indices and a 2D array containing the time series data
    agg_func: function
        function that takes a pandas DataFrame and returns
        an aggregated pandas DataFrame
    ts_buffer: int
        how many time series to read before writing to disk,
        constrained by the working memory the process should use.

    """

    def __init__(self, tsreader, imgwriter,
                 agg_func=None,
                 ts_buffer=1000):

        self.agg_func = agg_func
        if self.agg_func is None:
            try:
                self.agg_func = tsreader.agg_ts2img
            except AttributeError:
                self.agg_func = agg_tsmonthly
        self.tsreader = tsreader
        self.imgwriter = imgwriter
        self.ts_buffer = ts_buffer

    def calc(self, **tsaggkw):
```

(continues on next page)

(continued from previous page)

```
        """
        does the conversion from time series to images
        """
        for gpis, ts in self.tsbulk(**tsaggkw):
            self.imgwriter.write_ts(gpis, ts)

    def tsbulk(self, gpis=None, **tsaggkw):
        """
        iterator over gpi and time series arrays of size self.ts_buffer

        Parameters
        ----------
        gpis: iterable, optional
            if given these gpis will be used, can be practical
            if the gpis are managed by an external class e.g. for parallel
            processing
        tsaggkw: dict
            Keywords to give to the time series aggregation function


        Returns
        -------
        gpi_array: numpy.array
            numpy array of gpis in this batch
        ts_bulk: dict of numpy arrays
            for each variable one numpy array of shape
            (len(gpi_array), len(ts_aggregated))
        """
        # have to use the grid iteration as long as iter_ts only returns
        # data frame and no time series object including relevant metadata
        # of the time series
        i = 0
        gpi_bulk = []
        ts_bulk = {}
        ts_index = None
        if gpis is None:
            gpis, _, _, _ = self.tsreader.grid.grid_points()
        for gpi in gpis:
            gpi_bulk.append(gpi)
            ts = self.tsreader.read_ts(gpi)
            ts_agg = self.agg_func(ts, **tsaggkw)
            for column in ts_agg.columns:
                try:
                    ts_bulk[column].append(ts_agg[column].values)
                except KeyError:
                    ts_bulk[column] = []
                    ts_bulk[column].append(ts_agg[column].values)

            if ts_index is None:
                ts_index = ts_agg.index

            i += 1
            if i >= self.ts_buffer:
                for key in ts_bulk:
                    ts_bulk[key] = np.vstack(ts_bulk[key])
                gpi_array = np.hstack(gpi_bulk)
                yield gpi_array, ts_bulk
```

(continues on next page)

(continued from previous page)

```
                ts_bulk = {}
                gpi_bulk = []
                i = 0
        if i > 0:
            for key in ts_bulk:
                ts_bulk[key] = np.vstack(ts_bulk[key])
            gpi_array = np.hstack(gpi_bulk)
            yield gpi_array, ts_bulk
```

CHAPTER 8

resample

8.1 Spatial Resampling

The `resample` module contains functions to convert gridded data to different spatial resolutions.

8.2 Contents

8.2.1 ts2img

Introduction

Conversion of time series data to images (ts2img) is a general problem that we have to deal with often for all kinds of datasets. Although most of the external datasets we use come in image format most of them are converted into a time series format for analysis. This is fairly straightforward for image stacks but gets more complicated for orbit data.

Orbit data mostly comes as several data values accompanied by latitude, longitude information and must often be resampled to fit on a grid that lends itself for time series analysis. A more or less general solution for this already exists in the img2ts module.

Possible steps involved in the conversion

The steps that a ts2img program might have to perform are:

1. Read time series in a geographic region - constrained by memory
2. Aggregate time series in time
   - methods for doing this aggregation can vary for each dataset so the method is best specified by the user
   - resolution in time has to be chosen and is probably also best specified by the user
   - after aggregation every point in time and in space must have a value which can of course be NaN

   - time series might have to be split into separate images during conversion, e.g. ASCAT time series are routinely split into images for ascending and descending satellite overpasses. **This means that we can not assume that the output dataset has the same number or names of variables as the input dataset.**

3. Put the now uniform time series of equal length into a 2D array per variable
4. A resampling step could be performed here but since only a part of the dataset is available edge cases would not be resolved correctly. A better solution would be to develop a good resampling tool which might already exist in pyresample and pytesmo functions that use it.
5. write this data into a file
   - this can be a netCDF file with dimensions of the grid into which the data is written
   - this could be any other file format, the interface to this format just has to make sure that in the end a consistent image dataset is built out of the parts that are written.

Solution

The chosen first solution uses netCDF as an output format. The output will be a stack of images in netCDF format. This format can easily be converted into substacks or single images if that is needed for a certain user or project.

The chosen solution will **not** do resampling since this is better and easier done using the whole converted dataset. This also means that if the input dataset is e.g. a dataset defined over land only then the resulting "image" will also not contain land points. I think it is best to let this be decided by the input dataset.

The output of the resulting netCDF can have one of two possible "shapes":

- 2D variables with time on one axis and gpi on the other. This is kind of how SWI time series are stored already.
- 3D variables with latitude, longitude and time as the three dimensions.

The decision of which it will be is dependent on the grid on which the input data is stored. If the grid has a 2D shape then the 3D solution will be chosen. If the input grid has only a 1D shape then only the 2D solution is possible.

Time Series aggregation

The chosen solution will use a custom function for each dataset to perform the aggregation if necessary. A simple example of a function that gets a time time series and aggregates it to a monthly time series could look like *agg_tsmonthly*

Simple example of a aggregation function

```
def agg_tsmonthly(ts, **kwargs):
    """
    Parameters
    ----------
    ts : pandas.DataFrame
        time series of a point
    kwargs : dict
        any additional keyword arguments that are given to the ts2img object
        during initialization

    Returns
    -------
    ts_agg : pandas.DataFrame
        aggregated time series, they all must have the same length
        otherwise it can not work
        each column of this DataFrame will be a layer in the image
```

(continues on next page)

(continued from previous page)

```
    """
    # very simple example
    # aggregate to monthly timestamp
    # should also make sure that the output has a certain length
    return ts.asfreq("M")
```

Time series iteration

The function `agg_tsmonthly` will be called for every time series in the input dataset. The input dataset must have a `iter_ts` iterator that iterates over the grid points in a sensible order.

Interface to the netCDF writer

The netCDF writer will be initialized outside the *ts2img* class with a filename and other attributes it needs. So the *ts2img* class only gets a writer object. This writer object already knows about the start and end date of the time series as well as the target grid and has initialized the correct dimensions in the netCDF file. This object must have a method `write_ts` which takes a array of gpi's and a 2D array containing the time series for these gpis. This should be enough to write the gpi's into the correct position of the netCDF file.

This approach should also work if another output format is supposed to be used.

Implementation of the main ts2img class

The ts2img class will automatically use a the function given in `agg_ts2img` if no custom `agg_ts2img` function is provided. If the tsreader implements a method called `agg_ts2img` this function will be used instead.

```
class Ts2Img(object):

    """
    Takes a time series dataset and converts it
    into an image dataset.
    A custom aggregate function should be given otherwise
    a daily mean will be used

    Parameters
    ----------
    tsreader: object
        object that implements a iter_ts method which iterates over
        pandas time series and has a grid attribute that is a pytesmo
        BasicGrid or CellGrid
    imgwriter: object
        writer object that implements a write_ts method that takes
        a list of grid point indices and a 2D array containing the time series data
    agg_func: function
        function that takes a pandas DataFrame and returns
        an aggregated pandas DataFrame
    ts_buffer: int
        how many time series to read before writing to disk,
        constrained by the working memory the process should use.

    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, tsreader, imgwriter,
                 agg_func=None,
                 ts_buffer=1000):

        self.agg_func = agg_func
        if self.agg_func is None:
            try:
                self.agg_func = tsreader.agg_ts2img
            except AttributeError:
                self.agg_func = agg_tsmonthly
        self.tsreader = tsreader
        self.imgwriter = imgwriter
        self.ts_buffer = ts_buffer

    def calc(self, **tsaggkw):
        """
        does the conversion from time series to images
        """
        for gpis, ts in self.tsbulk(**tsaggkw):
            self.imgwriter.write_ts(gpis, ts)

    def tsbulk(self, gpis=None, **tsaggkw):
        """
        iterator over gpi and time series arrays of size self.ts_buffer

        Parameters
        ----------
        gpis: iterable, optional
            if given these gpis will be used, can be practical
            if the gpis are managed by an external class e.g. for parallel
            processing
        tsaggkw: dict
            Keywords to give to the time series aggregation function


        Returns
        -------
        gpi_array: numpy.array
            numpy array of gpis in this batch
        ts_bulk: dict of numpy arrays
            for each variable one numpy array of shape
            (len(gpi_array), len(ts_aggregated))
        """
        # have to use the grid iteration as long as iter_ts only returns
        # data frame and no time series object including relevant metadata
        # of the time series
        i = 0
        gpi_bulk = []
        ts_bulk = {}
        ts_index = None
        if gpis is None:
            gpis, _, _, _ = self.tsreader.grid.grid_points()
        for gpi in gpis:
            gpi_bulk.append(gpi)
            ts = self.tsreader.read_ts(gpi)
            ts_agg = self.agg_func(ts, **tsaggkw)
            for column in ts_agg.columns:
```

(continues on next page)

(continued from previous page)

```
                try:
                    ts_bulk[column].append(ts_agg[column].values)
                except KeyError:
                    ts_bulk[column] = []
                    ts_bulk[column].append(ts_agg[column].values)

            if ts_index is None:
                ts_index = ts_agg.index

            i += 1
            if i >= self.ts_buffer:
                for key in ts_bulk:
                    ts_bulk[key] = np.vstack(ts_bulk[key])
                gpi_array = np.hstack(gpi_bulk)
                yield gpi_array, ts_bulk
                ts_bulk = {}
                gpi_bulk = []
                i = 0
        if i > 0:
            for key in ts_bulk:
                ts_bulk[key] = np.vstack(ts_bulk[key])
            gpi_array = np.hstack(gpi_bulk)
            yield gpi_array, ts_bulk
```

8.2.2 resample

Spatial Resampling

The `resample` module contains functions to convert gridded data to different spatial resolutions.

8.2.3 License

ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

8.2.4 Contributors

- Christoph Paulik <cpaulik@vandersat.com>
- Wolfgang Preimesberger <wolfgang.preimesberger@geo.tuwien.ac.at>

8.2.5 repurpose

repurpose package

Submodules

repurpose.img2ts module

class repurpose.img2ts.**Img2Ts**(*input_dataset*, *outputpath*, *startdate*, *enddate*, *input_kwargs={}*, *input_grid=None*, *target_grid=None*, *imgbuffer=100*, *variable_rename=None*, *unlim_chunksize=100*, *cellsize_lat=180.0*, *cellsize_lon=360.0*, *r_methods='nn'*, *r_weightf=None*, *r_min_n=1*, *r_radius=18000*, *r_neigh=8*, *r_fill_values=None*, *filename_templ='%04d.nc'*, *gridname='grid.nc'*, *global_attr=None*, *ts_attributes=None*, *ts_dtypes=None*, *time_units='days since 1858-11-17 00:00:00'*, *zlib=True*)

Bases: `object`

class that uses the read_img iterator of the input_data dataset to read all images between startdate and enddate and saves them in netCDF time series files according to the given netCDF class and the cell structure of the outputgrid

Parameters

- **input_dataset** (*DatasetImgBase like class instance*) – must implement a daily_images iterator that yields data : dict

  dictionary of numpy arrays that hold the image data for each variable of the dataset

  timestamp : exact timestamp of the image lon : numpy.array or None

  array of longitudes, if None self.grid will be assumed

  lat [numpy.array or None] array of latitudes, if None self.grid will be assumed

  jd [numpy.array or None] array of observation times in julian days, if None all observations have the same timestamp

- **outputpath** (*string*) – path where to save the time series to
- **startdate** (*date*) – date from which the time series should start. Of course images have to be available from this date onwards.
- **enddate** (*date*) – date when the time series should end. Images should be availabe up until this date

- **input_kwargs** (*dict, optional*) – keyword arguments which should be used in the read_img method of the input_dataset
- **input_grid** (grid instance as defined in **:module:'pytesmo.grids.grid'**, optional) – the grid on which input data is stored. If not given then the grid of the input dataset will be used. If the input dataset has no grid object then resampling to the target_grid is performed.
- **target_grid** (grid instance as defined in **:module:'pytesmo.grids.grid'**, optional) – the grid on which the time series will be stored. If not given then the grid of the input dataset will be used
- **imgbuffer** (*int, optional*) – number of days worth of images that should be read into memory before a time series is written. This parameter should be chosen so that the memory of your machine is utilized. It depends on the daily data volume of the input dataset
- **variable_rename** (*dict, optional*) – if the variables should have other names than the names that are returned as keys in the dict by the daily_images iterator. A dictionary can be provided that changes these names for the time series.
- **unlim_chunksize** (*int, optional*) – netCDF chunksize for unlimited variables.
- **cellsize_lat** (*float, optional*) – if outgrid or input_data.grid are not cell grids then the cellsize in latitude direction can be specified here. Default is 1 global cell.
- **cellsize_lon** (*float, optional*) – if outgrid or input_data.grid are not cell grids then the cellsize in longitude direction can be specified here. Default is 1 global cell.
- **r_methods** (*string or dict, optional*) – resample methods to use if resampling is necessary, either 'nn' for nearest neighbour or 'custom' for custom weight function. Can also be a dictionary in which the method is specified for each variable
- **r_weightf** (*function or dict, optional*) – if r_methods is custom this function will be used to calculate the weights depending on distance. This can also be a dict with a separate weight function for each variable.
- **r_min_n** (*int, optional*) – Minimum number of neighbours on the target_grid that are required for a point to be resampled.
- **r_radius** (*float, optional*) – resample radius in which neighbours should be searched given in meters
- **r_neigh** (*int, optional*) – maximum number of neighbours found inside r_radius to use during resampling. If more are found the r_neigh closest neighbours will be used.
- **r_fill_values** (*number or dict, optional*) – if given the resampled output array will be filled with this value if no valid resampled value could be computed, if not a masked array will be returned can also be a dict with a fill value for each variable
- **filename_templ** (*string, optional*) – filename template must be a string with a string formatter for the cell number. e.g. '%04d.nc' will translate to the filename '0001.nc' for cell number 1.
- **gridname** (*string, optional*) – filename of the grid which will be saved as netCDF
- **global_attr** (*dict, optional*) – global attributes for each file
- **ts_attributes** (*dict, optional*) – dictionary of attributes that should be set for the netCDF time series. Can be either a dictionary of attributes that will be set for all variables in input_data or a dictionary of dictionaries. In the second case the first dictionary has to have a key for each variable returned by input_data and the second level dictionary will be the dictionary of attributes for this time series.

- **ts_dtype** (*numpy.dtype or dict of numpy.dtypes*) – data type to use for the time series, if it is a dict then a key must exist for each variable returned by input_data. Default : None, no change from input data
- **time_units** (*string, optional*) – units the time axis is given in. Default: "days since 1858-11-17 00:00:00" which is modified julian date for regular images this can be set freely since the conversion is done automatically, for images with irregular timestamp this will be ignored for now
- **zlib** (*boolean, optional*) – if True the saved netCDF files will be compressed Default: True

calc()

go through all images and retrieve a stack of them then go through all grid points in cell order and write to netCDF file

img_bulk()

Yields numpy array of self.const.imgbuffer images, start and enddate until all dates have been read

Returns

- **img_stack_dict** (*dict of numpy.array*) – stack of daily images for each variable
- **startdate** (*date*) – date of first image in stack
- **enddate** (*date*) – date of last image in stack
- **datetimestack** (*np.array*) – array of the timestamps of each image
- **jd_stack** (*np.array or None*) – if None all observations in an image have the same observation timestamp. Otherwise it gives the julian date of each observation in img_stack_dict

exception repurpose.img2ts.**Img2TsError**

Bases: Exception

repurpose.resample module

repurpose.resample.**hamming_window**(*radius*, *distances*)

Hamming window filter.

Parameters

- **radius** (*float32*) – Radius of the window.
- **distances** (*numpy.ndarray*) – Array with distances.

Returns **weights** – Distance weights.

Return type numpy.ndarray

repurpose.resample.**resample_to_grid**(*input_data*, *src_lon*, *src_lat*, *target_lon*, *target_lat*, *methods='nn'*, *weight_funcs=None*, *min_neighbours=1*, *search_rad=18000*, *neighbours=8*, *fill_values=None*)

resamples data from dictionary of numpy arrays using pyresample to given grid. Searches for the neighbours and then resamples the data to the grid given in togrid if at least min_neighbours neighbours are found

Parameters

- **input_data** (*dict of numpy.arrays*) –
- **src_lon** (*numpy.array*) – longitudes of the input data
- **src_lat** (*numpy.array*) – src_latitudes of the input data

- **target_lon** (*numpy.array*) – longitudes of the output data
- **target_src_lat** (*numpy.array*) – src_latitudes of the output data
- **methods** (*string or dict, optional*) – method of spatial averaging. this is given to pyresample and can be 'nn' : nearest neighbour 'custom' : custom weight function has to be supplied in weight_funcs see pyresample documentation for more details can also be a dictionary with a method for each array in input data dict
- **weight_funcs** (*function or dict of functions, optional*) – if method is 'custom' a function like func(distance) has to be given can also be a dictionary with a function for each array in input data dict
- **min_neighbours** (*int, optional*) – if given then only points with at least this number of neighbours will be resampled Default : 1
- **search_rad** (*float, optional*) – search radius in meters of neighbour search Default : 18000
- **neighbours** (*int, optional*) – maximum number of neighbours to look for for each input grid point Default : 8
- **fill_values** (*number or dict, optional*) – if given the output array will be filled with this value if no valid resampled value could be computed, if not a masked array will be returned can also be a dict with a fill value for each variable

Returns **data** – resampled data on given grid

Return type dict of numpy.arrays

Raises *ValueError :* – if empty dataset is resampled

repurpose.resample.**resample_to_grid_only_valid_return**(*input_data, src_lon, src_lat, target_lon, target_lat, methods='nn', weight_funcs=None, min_neighbours=1, search_rad=18000, neighbours=8, fill_values=None*)

resamples data from dictionary of numpy arrays using pyresample to given grid. Searches for the neighbours and then resamples the data to the grid given in togrid if at least min_neighbours neighbours are found

Parameters

- **input_data** (*dict of numpy.arrays*) –
- **src_lon** (*numpy.array*) – longitudes of the input data
- **src_lat** (*numpy.array*) – src_latitudes of the input data
- **target_lon** (*numpy.array*) – longitudes of the output data
- **target_src_lat** (*numpy.array*) – src_latitudes of the output data
- **methods** (*string or dict, optional*) – method of spatial averaging. this is given to pyresample and can be 'nn' : nearest neighbour 'custom' : custom weight function has to be supplied in weight_funcs see pyresample documentation for more details can also be a dictionary with a method for each array in input data dict
- **weight_funcs** (*function or dict of functions, optional*) – if method is 'custom' a function like func(distance) has to be given can also be a dictionary with a function for each array in input data dict

- **min_neighbours** (*int, optional*) – if given then only points with at least this number of neighbours will be resampled Default : 1
- **search_rad** (*float, optional*) – search radius in meters of neighbour search Default : 18000
- **neighbours** (*int, optional*) – maximum number of neighbours to look for for each input grid point Default : 8
- **fill_values** (*number or dict, optional*) – if given the output array will be filled with this value if no valid resampled value could be computed, if not a masked array will be returned can also be a dict with a fill value for each variable

Returns

- **data** (*dict of numpy.arrays*) – resampled data on part of the target grid over which data was found
- **mask** (*numpy.ndarray*) – boolean mask into target grid that specifies where data was resampled

Raises *ValueError :* – if empty dataset is resampled

repurpose.ts2img module

class repurpose.ts2img.**Ts2Img**(*tsreader*, *imgwriter*, *agg_func=None*, *ts_buffer=1000*)

Bases: object

Takes a time series dataset and converts it into an image dataset. A custom aggregate function should be given otherwise a daily mean will be used

Parameters

- **tsreader** (*object*) – object that implements a iter_ts method which iterates over pandas time series and has a grid attribute that is a pytesmo BasicGrid or CellGrid
- **imgwriter** (*object*) – writer object that implements a write_ts method that takes a list of grid point indices and a 2D array containing the time series data
- **agg_func** (*function*) – function that takes a pandas DataFrame and returns an aggregated pandas DataFrame
- **ts_buffer** (*int*) – how many time series to read before writing to disk, constrained by the working memory the process should use.

calc(***tsaggkw*)

does the conversion from time series to images

tsbulk(*gpis=None*, ***tsaggkw*)

iterator over gpi and time series arrays of size self.ts_buffer

Parameters

- **gpis** (*iterable, optional*) – if given these gpis will be used, can be practical if the gpis are managed by an external class e.g. for parallel processing
- **tsaggkw** (*dict*) – Keywords to give to the time series aggregation function

Returns

- **gpi_array** (*numpy.array*) – numpy array of gpis in this batch

- **ts_bulk** (*dict of numpy arrays*) – for each variable one numpy array of shape (len(gpi_array), len(ts_aggregated))

`repurpose.ts2img.`**`agg_tsmonthly`**(*ts*, ***kwargs*)

Parameters

- **`ts`** (*`pandas.DataFrame`*) – time series of a point
- **`kwargs`** (*`dict`*) – any additional keyword arguments that are given to the ts2img object during initialization

Returns **ts_agg** – aggregated time series, they all must have the same length otherwise it can not work each column of this DataFrame will be a layer in the image

Return type pandas.DataFrame

Module contents

8.3 Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index