GloMarGridding

Release 1.0.1

NOC Surface Processes

Nov 10, 2025

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Global surface temperature datasets - such as those used in the Intergovernmental Panel on Climate Change (IPCC) Assessment Report [IPCC] rely on a range of techniques to generate smoothed, infilled gridded fields from available observations [Lenssen], [Kadow], [RohdeBerkeley], [Morice_2021], [Huang] . The construction of these datasets involves numerous processing decisions, and variations in how each dataset represents the temperature field contribute to what is known as structural uncertainty [Thorne]. This structural uncertainty has two primary sources: (1) differences in the processing of the input temperature measurements, and (2) differences in the spatial interpolation methods applied. Because these steps are often tightly integrated, it is difficult to determine their individual contributions.

glomar_gridding is a software package developed to support the evaluation of structural uncertainty by offering flexible tools for spatial interpolation. The package enables users to spatially interpolate grid-box average observations and their associated uncertainty estimates using Gaussian Process Regression Modelling (GPRM, often called *Kriging*) [Rasmussen], [Cressie] . This technique builds on established methods for generating surface temperature fields [Karspeck], [Morice_2012]. By decoupling interpolation from earlier processing stages - such as homogenization, quality control, and aggregation to grid-cell averages - *glomar_gridding* allows users to create spatially complete fields while independently assessing the effects of upstream data processing choices.

CHAPTER

TWO

CITATION

Richard C. Cornes, Steven. C. Chan, Archie Cable et al. GloMarGridding: A Python Package for Spatial Interpolation to Support Structural Uncertainty Assessment of Climate Datasets, 22 August 2025, PREPRINT (Version 1) available at Research Square **`https://doi.org/10.21203/rs.3.rs-7427869/v1`_**

CHAPTER

THREE

GETTING STARTED

3.1 Installation

3.1.1 Via Pip

GloMarGridding is available on PyPI, and can be installed with *pip* or uv:

```
pip install glomar_gridding
```

```
uv add glomar_gridding
```

3.1.2 From Source

Alternatively, you can clone the repository and install using pip (or uv if preferred).

```
git clone https://github.com/NOCSurfaceProcesses/GloMarGridding.git
cd GloMarGridding
python -m venv venv
source venv/bin/activate
pip install -e .
```

```
git clone https://github.com/NOCSurfaceProcesses/GloMarGridding.git
cd GloMarGridding
uv sync --all-extras --python 3.11
```

CHAPTER

FOUR

CREDITS

4.1 Development Lead

- Agnieszka Faulkner <agfaul@noc.ac.uk> @agfaul
- Joseph T. Siddons <josidd@noc.ac.uk> @josidd

4.2 Contributoring Developers

- Archie Cable <acable@noc.ac.uk> @acable
- Steven Chan <stchan@noc.ac.uk> @stchan
- Richard C. Cornes <rcornes@noc.ac.uk> @ricorne
- Elizabeth C. Kent <eck@noc.ac.uk> @eck
- Duo Chan <Duo.Chan@soton.ac.uk>

4.3 Acknowledgements

- Thanks to Simon Williams for providing the original Matlab code that GloMarGridding is based upon.
- Supported by the Natural Environmental Research Council through National Capability funding (AtlantiS: NE/Y005589/1)

4.4 License

CHAPTER

FIVE

EXAMPLE WORKFLOW

Here we present a simple example, where we use Ordinary Kriging to interpolate observational data.

A full example can be found in the notebooks directory of the repository.

5.1 Load Observations

In this hypothetical example, we are working with data from a *csv* file. This is assumed to be point-observation data.

```
obs = pl.read_csv("/path/to/obs.csv", ...)
```

5.2 Output Grid

The first step is to create the output grid, using *glomar_gridding.grid.grid_from_resolution()* to specify a global grid with a fixed resolution. Here, the grid has a 5-degree resolution.

The grid is essentially an empty *xarray.DataArray* with a coordinate system defined by the input parameters. This can be used to map to the observations, and to create a covariance matrix with consistent coordinates.

```
grid = grid_from_resolution(
    resolution=5,
    bounds=[(-87.5, 90), (-177.5, 180)],
    coord_names=["latitude", "longitude"],
)
```

5.3 Align Observations

The input observations may not be located at grid-box locations, i.e. they may be located somewhere between grid-box centres. *glomar_gridding.grid.map_to_grid()* can be used to map each point observation to a grid-box by identifying the nearest grid-box position from an input grid.

This adds an index column to the observational data-frame, which is the 1-dimensional index value of the coordinate. This allows for easy mapping to the indices of covariance matrices, etc.

```
obs = map_to_grid(
    obs=obs,
    grid=grid,
)
```

In this example, it is assumed that the observations are such that at most one observation is associated with a grid-box. If this is not the case, the observations can be combined into a grid-box *super*-observation with a weighting using *glomar_gridding.kriging.prep_obs_for_kriging()*.

Extract the observation values, and the grid-box index for each observation. In this example, the observation value is stored in the *"val"* column.

```
grid_obs = obs["val"]
grid_idx = obs["grid_idx"]
```

5.4 Create or Load Spatial Covariance

The grid can be converted in to a distance matrix, and finally to a covariance matrix using a `glomar_gridding.variogram.Variogram` object, for example *glomar_gridding.variogram.GaussianVariogram*.

```
dist = grid_to_distance_matrix(
    grid=grid,
    lat_coord="latitude",
    lon_coord="longitude",
)

variogram = GaussianVariogram(
    range=1200,
    psill=1.2,
    nugget=0.0,
).fit(dist)

covariance = variogram_to_covariance(variogram, variance=1.2)
```

Alternatively, the covariance matrix can be loaded from disk. A non-stationary (varying parameter) covariance matrix can be estimated using ellipse-based models. See *glomar_gridding.ellipse.EllipseModel*.

5.5 Optionally Load Error Covariance

In this example an error covariance matrix is loaded from a netCDF file on disk, using *glomar_gridding.io.load_array()*

```
error_cov = load_array("/path/to/error_cov.nc", var="error_covariance")
```

Alternatively, an error covariance matrix can be computed component wise.

5.6 Ordinary Kriging

In this example, we will infill the observations using Ordinary Kriging. For this, we use *glomar_gridding.kriging.OrdinaryKriging*, which requires a spatial covariance matrix, observation grid indices, observation values, and (optionally) error covariance as inputs.

```
ok = OrdinaryKriging(
    covariance.values,
    idx=grid_idx,
    obs=grid_obs,
    error_cov=error_cov,
)
```

We can now use this class-instance to solve the system, using the *solve* method.

```
result = ok.solve()
```

Finally, the output can be mapped back on to the grid using `glomar_gridding.grid.assign_to_grid()`

```
gridded_result = assign_to_grid(
     values=result,
     grid_idx=np.arange(grid.size),
     grid=grid,
)
```

CHAPTER

SIX

STATIONARY INTERPOLATION COVARIANCE

Outside of the observation values and positions, the spatial covariance structure is the most important component for Kriging. The Kriging classes in this library all require this matrix as the input to the class. This covariance matrix can be provided as a pre-computed matrix, loaded into the environment with *glomar_gridding.interpolation_covariance.load_covariance()*. Alternatively the covariance structure can be estimated using functions and classes contained within *glomar_gridding*.

A commonly used approach is to compute a stationary covariance structure, using a *Variogram* with fixed scales. The use of *stationary* here suggests that the range of covariance is constant across all positions - each position has the same influence over a fixed distance.

```python
from glomar_gridding.grid import grid_from_resolution, grid_to_distance_matrix
from glomar_gridding.variogram import GaussianVariogram, variogram_to_covariance


# Initialise a grid
grid = grid_from_resolution(
    resolution=5,
    bounds=[(-87.5, 90), (-177.5, 180)],
    coord_names=["latitude", "longitude"],
)

# Compute a distance matrix
dist = grid_to_distance_matrix(
    grid=grid,
    lat_coord="latitude",
    lon_coord="longitude",
)

# Define and compute a Variogram
variogram = GaussianVariogram(
    nugget=0.0,
    psill=1.2,
    range=1300,
).fit(dist)

# Convert to covariance
covariance = variogram_to_covariance(variogram, sill=1.2)
```

6.1 Grid

glomar_gridding.grid.**grid_from_resolution**(*resolution*, *bounds*, *coord_names*)

Generate a grid from a resolution value, or a list of resolutions for given boundaries and coordinate names.

Note that all list inputs must have the same length, the ordering of values in the lists is assumed align.

The constructed grid will be regular, in the sense that the grid spacing is constant. However, the resolution in each direction can be different, allowing for finer resolution in some direction.

Parameters

- **resolution** (*float | list[float]*) – Resolution of the grid. Can be a single resolution value that will be applied to all coordinates, or a list of values mapping a resolution value to each of the coordinates.
- **bounds** (*list[tuple[float, float]]*) – A list of bounds of the form *(lower_bound, upper_bound)* indicating the bounding box of the returned grid. Typically, one would set the lower bound to be the centre of the first grid box. The upper bound is an open bound (similar to usage in *range*). For example a 5 degree resolution longitude range between -180, 180 could be defined with bounds *(-177.5, 180)*.
- **coord_names** (*list[str]*) – List of coordinate names in the same order as the bounds and resolution(s).

Returns

grid – The grid defined by the resolution and bounding box.

Return type

xarray.DataArray:

Examples

```
>>> grid_from_resolution(
        resolution=5,
        bounds=[(-87.5, 90), (-177.5, 180)],  # Lower bound is centre
        coord_names=["lat", "lon"]
    )
<xarray.DataArray (lat: 36, lon: 72)> Size: 21kB
array([[nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       ...,
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan]], shape=(36, 72))
Coordinates:
  * lat      (lat) float64 288B -87.5 -82.5 -77.5 ... 77.5 82.5 87.5
  * lon      (lon) float64 576B -177.5 -172.5 ... 172.5 177.5
```

glomar_gridding.grid.**grid_to_distance_matrix**(*grid*, *dist_func=<function haversine_distance_from_frame>*, *lat_coord='lat'*, *lon_coord='lon'*, ***dist_kwargs*)

Calculate a distance matrix between all positions in a grid. Orientation of latitude and longitude will be maintained in the returned distance matrix.

Parameters

- **grid** (*xarray.DataArray*) – A 2-d grid containing latitude and longitude indexes specified in decimal degrees.
- **dist_func** (*Callable*) – Distance function to use to compute pairwise distances. See glomar_gridding.distances.calculate_distance_matrix for more information.
- **lat_coord** (*str*) – Name of the latitude coordinate in the input grid.
- **lon_coord** (*str*) – Name of the longitude coordinate in the input grid.
- ****dist_kwargs** – Keyword arguments to pass to the distance function.

Returns

dist – A DataArray containing the distance matrix with coordinate system defined with grid cell index ("index_1" and "index_2"). The coordinates of the original grid are also kept as coordinates related to each index (the coordinate names are suffixed with "_1" or "_2" respectively).

Return type

xarray.DataArray

Examples

```
>>> grid = grid_from_resolution(
        resolution=5,
        bounds=[(-87.5, 90), (-177.5, 180)],  # Lower bound is centre
        coord_names=["lat", "lon"]
    )
>>> grid_to_distance_matrix(grid, lat_coord="lat", lon_coord="lon")
<xarray.DataArray 'dist' (index_1: 2592, index_2: 2592)> Size: 54MB
array([[    0.        ,    24.24359308,    48.44112457, ...,
        19463.87158499, 19461.22915012, 19459.64166305],
       [   24.24359308,     0.        ,    24.24359308, ...,
        19467.56390938, 19463.87158499, 19461.22915012],
       [   48.44112457,    24.24359308,     0.        , ...,
        19472.29905588, 19467.56390938, 19463.87158499],
       ...,
       [19463.87158499, 19467.56390938, 19472.29905588, ...,
            0.        ,    24.24359308,    48.44112457],
       [19461.22915012, 19463.87158499, 19467.56390938, ...,
           24.24359308,     0.        ,    24.24359308],
       [19459.64166305, 19461.22915012, 19463.87158499, ...,
           48.44112457,    24.24359308,     0.        ]],
      shape=(2592, 2592))
Coordinates:
  * index_1  (index_1) int64 21kB 0 1 2 3 4 ... 2587 2588 2589 2590 2591
  * index_2  (index_2) int64 21kB 0 1 2 3 4 ... 2587 2588 2589 2590 2591
    lat_1    (index_1) float64 21kB -87.5 -87.5 -87.5 ... 87.5 87.5
    lon_1    (index_1) float64 21kB -177.5 -172.5 ... 172.5 177.5
    lat_2    (index_2) float64 21kB -87.5 -87.5 -87.5 ... 87.5 87.5 87.5
    lon_2    (index_2) float64 21kB -177.5 -172.5 ... 172.5 177.5
```

6.2 Variograms

`class glomar_gridding.variogram.`**`ExponentialVariogram`**(*psill*, *nugget*, *range=None*, *effective_range=None*)

Exponential Model

Parameters

- **psill** (*float | np.ndarray*) – Sill of the variogram where it will flatten out. Values in the variogram will not exceed psill + nugget. This value is the variance.
- **nugget** (*float | np.ndarray*) – The value of the independent variable at distance 0
- **effective_range** (*float | np.ndarray | None*) – Effective Range, this is the distance where 95% of the sill are exceeded. This is not the range parameter, which is defined as r/3.
- **range** (*float | ndarray | None*) – The range parameter. One of range and effective_range must be set. If range is not set, it will be computed from effective_range.

fit(*distance_matrix*)

Fit the ExponentialVariogram model to a distance matrix

Parameters

distance_matrix (*numpy.ndarray | xarray.DataArray*) – The distance matrix indicating the distance between each pair of points in the grid.

Returns

A matrix containing the variogram values at each distance.

Return type

numpy.ndarray | xarray.DataArray

Examples

```
>>> ExponentialVariogram(range=1200, psill=1.2, nugget=0.0).fit(dist)
```

class glomar_gridding.variogram.**SphericalVariogram**(*psill*, *nugget*, *effective_range=None*, *range=None*)

Spherical Model

Parameters

- **psill** (*float | np.ndarray*) – Sill of the variogram where it will flatten out. Values in the variogram will not exceed psill + nugget. This value is the variance.
- **nugget** (*float | np.ndarray*) – The value of the independent variable at distance 0
- **effective_range** (*float | np.ndarray | None*) – Effective Range, this is the distance where 95% of the sill are exceeded. This is not the range parameter, which is equal to the effective range in the SphericalVariogram case.
- **range** (*float | ndarray | None*) – The range parameter. One of range and effective_range must be set. If range is not set, it will be computed from effective_range.

fit(*distance_matrix*)

Fit the SphericalVariogram model to a distance matrix

Parameters

distance_matrix (*numpy.ndarray | xarray.DataArray*) – The distance matrix indicating the distance between each pair of points in the grid.

Returns

A matrix containing the variogram values at each distance.

Return type

numpy.ndarray | xarray.DataArray

Examples

```
>>> SphericalVariogram(range=1200, psill=1.2, nugget=0.0).fit(dist)
```

class glomar_gridding.variogram.**GaussianVariogram**(*psill*, *nugget*, *effective_range=None*, *range=None*)

Gaussian Model

Parameters

- **psill** (*float | np.ndarray*) – Sill of the variogram where it will flatten out. Values in the variogram will not exceed psill + nugget. This value is the variance.
- **nugget** (*float | np.ndarray*) – The value of the independent variable at distance 0
- **effective_range** (*float | np.ndarray | None*) – Effective Range, this is the distance where 95% of the sill are exceeded. This is not the range parameter, which is defined as r/2.
- **range** (*float | ndarray | None*) – The range parameter. One of range and effective_range must be set. If range is not set, it will be computed from effective_range.

fit(*distance_matrix*)

Fit the GaussianVariogram model to a distance matrix

Parameters

distance_matrix (*numpy.ndarray | xarray.DataArray*) – The distance matrix indicating the distance between each pair of points in the grid.

Returns

A matrix containing the variogram values at each distance.

Return type

numpy.ndarray | xarray.DataArray

Examples

```
>>> GaussianVariogram(range=1200, psill=1.2, nugget=0.0).fit(dist)
```

class glomar_gridding.variogram.**MaternVariogram**(*psill*, *nugget*, *effective_range=None*, *range=None*, *nu=0.5*, *method='sklearn'*)

Matern Models

Same args as the Variogram classes with additional nu (ν), method parameters.

Sklearn:

1) This is called “sklearn” because if d/range = 1.0 and $\nu = 0.5$, it gives 1/e correlation…
2) This is NOT the same formulation as in GSTAT nor in papers about non-stationary anistropic covariance models (aka Karspeck paper).
3) It is perhaps the most intitutive (because of (1)) and is used in sklearn GP and HadCRUT5 and other UKMO dataset.
4) nu defaults to 0.5 (exponential; used in HADSST4 and our kriging). HadCRUT5 uses 1.5.
5) The “2” is inside the square root for middle and right.

GeoStatic:

Similar to Sklearn MaternVariogram model but uses the range scaling in gstat. Note: there are no square root 2 or nu in middle and right

Yields the same answer to sklearn MaternVariogram if $\nu = 0.5$ but are otherwise different.

Karspeck:

Similar to Sklearn MaternVariogram model but uses the form in Karspeck paper Note: Note the 2 is outside the square root for middle and right e-folding distance is now at d/SQRT(2) for $\nu = 0.5$

References

see chapter 4.2 of Rasmussen, C. E., & Williams, C. K. I. (2005). Gaussian Processes for Machine Learning. The MIT Press. https://doi.org/10.7551/mitpress/3206.001.0001

Parameters

- **psill** (*float | np.ndarray*) – Sill of the variogram where it will flatten out. Values in the variogram will not exceed psill + nugget. This value is the variance.
- **nugget** (*float | np.ndarray*) – The value of the independent variable at distance 0
- **effective_range** (*float | np.ndarray | None*) – Effective Range, this is the lag where 95% of the sill are exceeded. This is not the range parameter, which is defined as r/3 if nu < 0.5 or nu > 10, otherwise r/2 (where r is the effective range). One of effective_range and range must be set.
- **range** (*float | ndarray | None*) – The range parameter. One of range and effective_range must be set. If range is not set, it will be computed from effective_range.
- **nu** (*float | np.ndarray*) – ν, smoothing parameter, shapes to a smooth or rough variogram function
- **method** (*MaternModel*) – One of "sklearn", "gstat", or "karspeck":
  - sklearn: https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.Matern.html#sklearn.gaussian_process.kernels.Matern
  - gstat: https://scikit-gstat.readthedocs.io/en/latest/reference/models.html#matern-model
  - karspeck: https://rmets.onlinelibrary.wiley.com/doi/10.1002/qj.900

fit(*distance_matrix*)

Fit the MaternVariogram model to a distance matrix.

Parameters

distance_matrix (*numpy.ndarray | xarray.DataArray*) – The distance matrix indicating the distance between each pair of points in the grid.

Returns

A matrix containing the variogram values at each distance.

Return type

numpy.ndarray | xarray.DataArray

Examples

```
>>> MaternVariogram(
        range=1200, psill=1.2, nugget=0.0, nu=1.5, method="karspeck"
    ).fit(dist)
```

glomar_gridding.variogram.**variogram_to_covariance**(*variogram*, *variance*)

Convert a variogram matrix to a covariance matrix.

This is given by:
covariance = variance - variogram

Parameters

- **variogram** (*numpy.ndarray | xarray.DataArray*) – The variogram matrix, output of Variogram.fit.
- **variance** (*numpy.ndarray | float*) – The variance

Returns
cov – The covariance matrix

Return type
numpy.ndarray | xarray.DataArray

6.3 Covariance

I/O functionality for loading a covariance matrix from disk.

glomar_gridding.interpolation_covariance.**load_covariance**(*path*, *cov_var_name='covariance'*, ***kwargs*)

Load a covariance matrix from a netCDF file. Can input a filename or a string to format with keyword arguments.

Parameters

- **path** (*str*) – Full filename (including path), or filename with replacements using str.format with named replacements. For example: /path/to/global_covariance_{month:02d}.nc
- **cov_var_name** (*str*) – Name of the variable for the covariance matrix
- ****kwargs** – Keywords arguments matching the replacements in the input path.

Returns
covariance – A numpy matrix containing the covariance matrix loaded from the netCDF file determined by the input arguments.

Return type
numpy.ndarray

CHAPTER SEVEN

ELLIPSES: NON-STATIONARY INTERPOLATION COVARIANCE

7.1 Ellipse Models

class glomar_gridding.ellipse.**EllipseModel**(*anisotropic*, *rotated*, *physical_distance*, *v*, *unit_sigma=False*)

The class that contains variogram/ellipse fitting methods and parameters

This class assumes your input to be a standardised correlation matrix They are easier to handle because stdevs in the covariance function become 1

Parameters

- **anisotropic** (*bool*) – Should the output be an ellipse? Set to False for circle.
- **rotated** (*bool*) – Can the ellipse be rotated. If anisotropic is False this value cannot be True.
- **physical_distance** (*bool*) – Use physical distances rather than lat/lon distance.
- **v** (*float*) – Matern Shape Parameter. Must be > 0.0.
- **unit_sigma=True** (*bool*) – When MLE fitting the Matern parameters, assuming the Matern parameters themselves are normally distributed, there is standard deviation within the log likelihood function.

  See Wikipedia entry for Maximum Likelihood under: - Continuous distribution, continuous parameter space

  Its actual value is not important to the best (MLE) estimate of the Matern parameters. If one assumes the parameters are normally distributed, the mean (best estimate) is independent of its variance. In fact in Karspeck et al 2012 [Karspeck], it is simply set to 1 Eq B1). This value can however be computed. It serves a similar purpose as the original standard deviation: in this case, how the actual observed semivariance disperses around the fitted variogram.

  The choice to default to 1 follows Karspeck et al. 2012 [Karspeck]

fit(*X*, *y*, *guesses=None*, *bounds=None*, *opt_method='Nelder-Mead'*, *tol=None*, *estimate_SE='bootstrap_parallel'*, *n_sim=500*, *n_jobs=4*, *backend='loky'*, *random_seed=1234*)

Default solver in Nelder-Mead as used in Karspeck et al. 2012 [Karspeck] i.e. https://docs.scipy.org/doc/scipy/reference/optimize.minimize-neldermead.html default max-iter is 200 x (number_of_variables) for 3 variables (Lx, Ly, theta) –> 200x3 = 600 note: unlike variogram fitting, no nugget, no sill, and no residue variance (normalised data but Fisher transform needed?) can be adjusted using “maxiter” within “options” kwargs

Much of the variable names are defined the same way as earlier

Parameters

- **X** (*numpy.ndarray*) – Array of displacements. Expected to be 1-dimensional if the ellipse model is not anisotropic, 2-dimensional otherwise. In units of km if the ellipse uses physical distances, otherwise in degrees. The displacements are from each position within the test region to the centre of the ellipse.
- **y** (*numpy.ndarray*) – Vector of observed correlations between the centre of the ellipse and each test point.
- **guesses=None** (*list[float] | None*) – List of initial values to scipy.optimize.minimize, default guesses for the ellipse model are used if not set.
- **bounds=None** (*list[tuple[float, float]] | None*) – Tuples/lists of bounds for fitted parameters. Default bounds for the ellipse model are used if not set.
- **opt_method** (*str*) – scipy.optimize.minimize optimisation method. Defaults to "Nelder-Mead". See https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html for valid values.
- **tol=None** (*float | None*) – scipy.optimize.minimize convergence tolerance
- **estimate_SE='bootstrap_parallel'** (*str | None*) – How to estimate standard error if needed. If not set no standard error is computed.
- **n_sim=500** (*int*) – Number of bootstrap to estimate standard error
- **n_jobs=DEFAULT_N_JOBS** (*int*) – Number of threads for bootstrapping if *estimate_SE* is set to "bootstrap_parallel".
- **backend=DEFAULT_BACKEND** (*str*) – joblib backend for bootstrapping.
- **random_seed=1234** (*int*) – Random seed for bootstrap

Return type

`tuple[OptimizeResult, float | None, list[tuple[float, float]]]`

Returns

- **results** (*OptimizeResult*) – Output of scipy.optimize.minimize
- **SE** (*float | None*) – Standard error of the fitted parameters
- **bounds** (*list[tuple[float, …]]*) – Bounds of fitted parameters

negative_log_likelihood(*X*, *y*, *params*, *arctanh_transform=True*)

Compute the negative log-likelihood given observed X independent observations (displacements) and y dependent variable (the observed correlation), and Matern parameters params. Namely does the Matern covariance function using params, how close it explains the observed displacements and correlations.

log(LL) = SUM (f (y,x|params)) params = Maximise (log(LL)) params = Minimise (-log(LL)) which is how usually the computer solves it assuming errors of params are normally distributed

There is a hidden scale/standard deviation in stats.norm.logpdf(scale, which defaults to 1) but since we have scaled our values to covariance to correlation (and even used Fisher transform) as part of the function, it can be dropped

Otherwise, you need to have stdev as the last value of params, and should be set to the scale parameter

Parameters

- **X** (*np.ndarray*) – Observed displacements to the centre of the ellipse.
- **y** (*np.ndarray*) – Observed correlation against the centre of the ellipse.
- **params** (*list[float]*) – Ellipse parameters (in the current optimize iteration) or if you want to compute the actual negative log-likelihood.

- `arctanh_transform` (*bool*) – Should the Fisher (arctanh) transform be used This is usually option, but it does make the computation more stable if they are close to 1 (or -1; doesn't apply here)

Returns
nLL – The negative log likelihood

Return type
float

`negative_log_likelihood_function`(*X*, *y*)

Creates a function that can be fed into scipy.optimizer.minimize

Return type
`Callable`[[`list`[`float`]], `float`]

7.2 Ellipse Parameter Estimation

class `glomar_gridding.ellipse.EllipseBuilder`(*data_array*, *coords*)

Class to build spatial covariance and correlation matrices used to estimate ellipse parameterss using an instance of EllipseModel which sets up the defaults for a given configuration.

To fit ellipse parameters to the correlation of the input data_array, call self.fit_ellipse_model.

Parameters

- `data_array` (*numpy.ndarray | numpy.ma.MaskedArray*) – Training data stored within a numpy array. In general, this input should be extracted from an xarray.DataArray, and masked appropriately.
- `coords` (*xarray.Coordinates*) – The coordinates associated with the data_array value. It is expected that these are [“time”, “latitude”, “longitude”]

`calc_cov`(*rounding=None*)

Calculate covariance and correlation matrices.

Parameters
`rounding` (*int | None*) – Round the values of the output.

Return type
`None`

`compute_params`(*default_value*, *matern_ellipse*, *max_distance=6000*, *min_distance=0.3*, *delta_x_method='Modified_Met_Office'*, *guesses=None*, *bounds=None*, *opt_method='Nelder-Mead'*, *tol=0.0001*, *estimate_SE=None*, *n_jobs=4*, *n_sim=500*, *physical_distance_selection=True*, *return_r2=False*)

Fit ellipses/covariance models using adhoc local covariances to all unmasked grid points

The form of the covariance model depends on the “fform” attribute of the Ellipse model:

- isotropic (radial distance only)
- anistropic (x and y are different, but not rotated)
- anistropic_rotated (rotated)

If the “fform” attribute ends with _pd then physical distances are used instead of degrees

range is defined max_distance (either in km and degrees) default is in degrees, but needs to be km if fform is from _pd series <— likely to be wrong: max_distance should only be in degrees there is also a min_distance in which values, matern function is not defined at the origin, so the 0.0 needs to removed

v = matern covariance function shape parameter Karspeck et al. [Karspeck] and Paciorek & Schervish [PaciorekSchervish] but 0.5 and 1.5 are popular 0.5 gives an exponential decay lim v–>inf, Gaussian shape

delta_x_method: only meaningful for _pd fits:

- "Met_Office": Cylindrical Earth delta_x = 6400km x delta_lon (in radians)
- "Modified_Met_Office": uses the average zonal dist at different lat

Parameters

- **default_value** (*Any*) – Default value(s) to fill arrays where parameter estimation is not possible (typically due to masking). Typically, one should set a value that is appropriate to the type of the field. If a single value is provided, this is used for all fields. If not, the length of the list of default values must equal the number of parameters of the *EllipseModel*
- **matern_ellipse** (EllipseModel) – EllipseModel to use for parameter estimation
- **max_distance** (*float*) – Maximum separation in distance unit that data will be fed into parameter fitting Units depend on fform (it is usually either degrees or km)
- **min_distance** (*float*) – Minimum separation in distance unit that data will be fed into parameter fitting Units depend on fform (it is usually either degrees or km) Note: Due to the way we compute the Matern function, it is undefined at dist == 0 even if the limit -> zero is obvious.
- **delta_x_method="Modified_Met_Office"** (*str*) – How to compute distances between grid points For istropic variogram/covariances, this is a trivial problem; you can just take the haversine or Euclidean ("tunnel") distance as they are non-directional.

  But it is non trivial for anistropic cases, you have to define a set of orthogonal space. In HadSST4, Earth is assumed to be cylindrical "tin can" Earth, so you can just define the orthogonal space by lines of constant lat and lon (delta_x_method="Met_Office").

  The modified "Modified_Met_Office" is a variation to that, but allow the tin can get squished at the poles. (Sinusoidal projection). This does results in a problem: the zonal displacement now depends in which latitude you compute on (at the beginning latitude or at the end latitude). Here we take the average of the two.
- **guesses=None** (*tuple of floats; None uses default guess values*) – Initial guess values that get feeds in the optimizer for MLE. In scipy, you are required to do so (but R often doesn't). You should anyway; sometimes they do funny things if you don't (per recommendation of David Stephenson)
- **bounds=None** (*tuple of floats; None uses default bounds values*) – This is essentially a Bayesian "uniformative prior" that forces convergence if the optimizer hits the bound. For lower resolution fitting, this is rarely a problem. For higher resolution fits, this often interacts with the limit of the data you can put into the fit the optimizer may fail to converge if the input data is very smooth (aka ENSO region, where anomalies are smooth over very large (~10000km) scales).
- **opt_method='Nelder-Mead'** (*str*) – scipy.optimize method. Nelder-Mead is the one used by [Karspeck]. See https://docs.scipy.org/doc/scipy/tutorial/optimize.html for valid options
- **tol=0.001** (*float*) – Set convergence tolerance for scipy optimize. See https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize

Note on new tol kwarg: For N-M, this sets the value to both xatol and fatol Default is 1E-4 Since it affects accuracy of all values including rotation rotation angle 0.001 rad ~ 0.05 deg

- **estimate_SE=None** (*str | None*) – The code can estimate the standard error if the Matern parameters. This is not usually used or discussed for the purpose of kriging. Certain opt_method (gradient descent) can do this automatically using Fisher Info for certain covariance function, but is not possible for some nasty functions (aka Bessel func) gets involved nor it is possible for some optimisers (such as Nelder-Mead). The code does it using bootstrapping.
- **n_jobs=DEFAULT_N_JOBS** (*int*) – If parallel processing, number of threads to use.
- **n_sim** (*int*) – Number of simulations to bootstrap for SE estimation.
- **physical_distance_selection** (*bool*) – Select training data using physical distance (haversine distance) or Euclidean distance of degree difference between each position and all possible training data. Data that falls within the min and max distance values using the selected distance method are selected as training data.
- **return_r2** (*bool*) – Add R2 / Correlation Matrix to output fields.

Returns

params – Containing arrays for each parameter in the ellipse model class. Note that one array is likely to be “qc_code”, which takes values:

- 0: success
- 2: success but with one parameter reaching upper boundaries
- 3: success with multiple parameters reaching the boundaries (aka both Lx and Ly), can be both at lower or upper boundaries
- 9: fail, probably due to running out of maxiter (see scipy.optimize.minimize kwargs “options”)

Return type

xarray.Dataset

find_nearest_xy_index_in_cov_matrix(*lonlat*, *use_full=False*)

Find the nearest column/row index of the covariance that corresponds to a specific lat lon

Return type

tuple[int, ndarray]

fit_ellipse_model(*xy_point*, *matern_ellipse*, *max_distance=6000*, *min_distance=0.3*, *delta_x_method='Modified_Met_Office'*, *guesses=None*, *bounds=None*, *opt_method='Nelder-Mead'*, *tol=0.001*, *estimate_SE=None*, *n_jobs=4*, *n_sim=500*, *physical_distance_selection=True*, *return_r2=False*)

Fit ellipses/covariance models using adhoc local covariances

the form of the covariance model depends on the “fform” attribute of the Ellipse model:

> isotropic (radial distance only) anistropic (x and y are different, but not rotated) anistropic_rotated (rotated)

If the “fform” attribute ends with _pd then physical distances are used instead of degrees

range is defined max_distance (either in km and degrees) default is in degrees, but needs to be km if fform is from _pd series <— likely to be wrong: max_distance should only be in degrees

there is also a min_distance in which values, matern function is not defined at the origin, so the 0.0 needs to removed v = matern covariance function shape parameter Karspeck et al. [Karspeck] and Paciorek & Schervish [PaciorekSchervish] use 3 and 4 but 0.5 and 1.5 are popular. 0.5 gives an exponential decay: lim v–>inf, Gaussian shape

delta_x_method: only meaningful for _pd fits:

- "Met_Office": Cylindrical Earth delta_x = 6400km x delta_lon (in radians)
- "Modified_Met_Office": uses the average zonal dist at different lat

Parameters

- `xy_point` (*int*) – The index point where ellipses will be fitted to
- `max_distance` (*float*) – Maximum separation in distance unit that data will be fed into parameter fitting Units depend on fform (it is usually either degrees or km)
- `min_distance` (*float*) – Minimum separation in distance unit that data will be fed into parameter fitting Units depend on fform (it is usually either degrees or km) Note: Due to the way we compute the Matern function, it is undefined at dist == 0 even if the limit -> zero is obvious.
- `delta_x_method="Modified_Met_Office"` (*str*) – How to compute distances between grid points For istropic variogram/covariances, this is a trivial problem; you can just take the haversine or Euclidean ("tunnel") distance as they are non-directional.

  But it is non trivial for anistropic cases, you have to define a set of orthogonal space. In HadSST4, Earth is assumed to be cylindrical "tin can" Earth, so you can just define the orthogonal space by lines of constant lat and lon (delta_x_method="Met_Office").

  The modified "Modified_Met_Office" is a variation to that, but allow the tin can get squished at the poles. (Sinusoidal projection). This does results in a problem: the zonal displacement now depends in which latitude you compute on (at the beginning latitude or at the end latitude). Here we take the average of the two.
- `guesses=None` (*tuple of floats; None uses default guess values*) – Initial guess values that get feeds in the optimizer for MLE. In scipy, you are required to do so (but R often doesn't). You should anyway; sometimes they do funny things if you don't (per recommendation of David Stephenson)
- `bounds=None` (*tuple of floats; None uses default bounds values*) – This is essentially a Bayesian "uniformative prior" that forces convergence if the optimizer hits the bound. For lower resolution fitting, this is rarely a problem. For higher resolution fits, this often interacts with the limit of the data you can put into the fit the optimizer may fail to converge if the input data is very smooth (aka ENSO region, where anomalies are smooth over very large (~10000km) scales).
- `opt_method='Nelder-Mead'` (*str*) – scipy.optimize method. Nelder-Mead is the one used by [Karspeck]. See https://docs.scipy.org/doc/scipy/tutorial/optimize.html for valid options
- `tol=0.001` (*float*) – Set convergence tolerance for scipy optimize. See https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize

  Note on new tol kwarg: For N-M, this sets the value to both xatol and fatol Default is 1E-4 (?) Since it affects accuracy of all values including rotation rotation angle 0.001 rad ~ 0.05 deg
- `estimate_SE=None` (*str | None*) – The code can estimate the standard error if the Matern parameters. This is not usually used or discussed for the purpose of kriging. Cer-

tain opt_method (gradient descent) can do this automatically using Fisher Info for certain covariance function, but is not possible for some nasty functions (aka Bessel func) gets involved nor it is possible for some optimisers (such as Nelder-Mead). The code does it using bootstrapping.

- **n_jobs=DEFAULT_N_JOBS** (*int*) – If parallel processing, number of threads to use.
- **n_sim** (*int*) – Number of simulations to bootstrap for SE estimation.
- **physical_distance_selection** (*bool*) – Select training data using physical distance (haversine distance) or Euclidean distance of degree difference between each position and all possible training data. Data that falls within the min and max distance values using the selected distance method are selected as training data.
- **return_r2** (*bool*) – Add R2 / Correlation Matrix to output fields.

Returns

Dictionary with results of the fit and the observed correlation matrix.

Return type

dict

7.3 Ellipse-based Covariance Estimation

class glomar_gridding.ellipse.**EllipseCovarianceBuilder**(*Lx*, *Ly*, *theta*, *stdev*, *lats*, *lons*, *v*, *delta_x_method='Modified_Met_Office'*, *max_dist=None*, *precision=<class 'numpy.float32'>*, *covariance_method='array'*, *batch_size=None*)

Compute covariance from Ellipse parameters and positions.

v = Matern covariance shape parameter

Lx - an numpy array of horizontal length scales (Ly - an numpy array of meridonal length scales theta - an numpy array of rotation angles (RADIANS ONLY)

sdev - standard deviation – right now it just takes a numeric array if you have multiple contribution to sdev (uncertainties derived from different sources), you need to put them into one array

Rules: Valid (ocean) point: 1) cov_ns and cor_ns are computed out to max_dist; out of range = 0.0 2) Masked points are ignored

Invalid (masked) points: 1) Skipped over

Parameters

- **Lx** (*numpy.ndarray*) – Arrays with non-stationary parameters
- **Ly** (*numpy.ndarray*) – Arrays with non-stationary parameters
- **theta** (*numpy.ndarray*) – Arrays with non-stationary parameters
- **stdev** (*numpy.ndarray*) – Arrays with non-stationary parameters
- **lats** (*numpy.ndarray*) – Arrays containing the latitude and longitude values
- **lons** (*numpy.ndarray*) – Arrays containing the latitude and longitude values
- **v** (*float*) – Matern shape parameter
- **delta_x_method** (*str*) – How are displacements computed between points

- **max_dist** (*float | None*) – If the Haversine distance between 2 points exceed max_dist, covariance is set to 0. If set to None then an infinite max dist is assumed and covariance between all pairs of positions will be computed.
- **precision** (*type*) – Floating point precision of the output covariance numpy defaults to np.float32.
- **covariance_method** (*CovarianceMethod*) – Set the covariance method used:
  - array (default): faster but uses significantly more memory as more pre-computation is performed. Values are computed in a vectorised method.
  - loop: slower iterative process, computes each value individually
  - batched: combines the above approaches.

  If the number of grid-points exceeds 10_000 and "array" method is used, the method will be overwritten to "loop".
- **batch_size** (*int | None*) – Size of the batch to use for the "batched" method. Must be set if the covariance_method is set to "batched".

c_ij_anisotropic_array(*i_s*, *j_s*)

Compute the covariances between pairs of ellipses, at displacements.

Each ellipse is defined by values from Lxs, Lys, and thetas, with standard deviation in stdevs.

The displacements between each pair of ellipses are x_is and x_js.

For N ellipses, the number of displacements should be 1/2 * N * (N - 1), i.e. the displacement between each pair combination of ellipses. This function will return the upper triangular values of the covariance matrix (excluding the diagonal).

itertools.combinations is used to handle ordering, so the displacements must be ordered in the same way.

Parameters

- **i_s** (*numpy.ndarray*) – The row indices for the covariance matrix.
- **j_s** (*numpy.ndarray*) – The column indices for the covariance matrix.

Returns

c_ij – A vector containing the covariance values between each pair of ellipses. This will return the components of the upper triangle of the covariance matrix as a vector (excluding the diagonal).

Return type

numpy.ndarray

References

1. Paciorek and Schevrish 2006 [PaciorekSchervish] Equation 8
2. Karspeck et al. 2012 [Karspeck] Equation 17

calculate_cor()

Calculate correlation matrix from the covariance matrix

Return type

None

calculate_covariance_array()

Calculate the covariance matrix from the ellipse parameters

Return type
`None`

calculate_covariance_batched()

Compute the covariance matrix from ellipse parameters, using a batched approach. This approach is more memory safe and appropriate for low-memory operations, but is slower than self.calculate_covariance which pre-computes values at all upper triangle points. This approach performs pre-computation at all points within the current batch.

Each ellipse is defined by values from Lxs, Lys, and thetas, with standard deviation in stdevs.

Requires a batch_size parameter.

Return type
`None`

References

1. Paciorek and Schevrish 2006 [PaciorekSchervish] Equation 8
2. Karspeck et al. 2012 [Karspeck] Equation 17

calculate_covariance_loop()

Compute the covariance matrix from ellipse parameters, using a loop. This approach is more memory safe and appropriate for low-memory operations, but is significantly slower than self.calculate_covariance which uses a lot of pre-computation and a vectorised approach.

Each ellipse is defined by values from Lxs, Lys, and thetas, with standard deviation in stdevs.

Return type
`None`

References

1. Paciorek and Schevrish 2006 [PaciorekSchervish] Equation 8
2. Karspeck et al. 2012 [Karspeck] Equation 17

uncompress_cov(*diag_fill_value=nan*, *fill_value=nan*)

Convert the covariance matrix to full grid size.

Optionally, fill the array with along the diagonal with a *diag_fill_value* and off the diagonal with a *fill_value*, which both default to *np.nan*.

Overwrites the *cov_ns* attribute.

Parameters

- **diag_fill_value** (*Any*) – Value to assign to diagonal masked values. Defaults to *np.nan*
- **fill_value** (*Any*) – Value to assign to off-diagonal masked values. Defaults to *np.nan*

Return type
`None`

CHAPTER

EIGHT

ERROR COVARIANCE

Functions for computing correlated and uncorrelated components of the error covariance. These values are determined from standard deviation (sigma) values assigned to groupings within the observational data.

The correlated components will form a matrix that is permutationally equivalent to a block diagonal matrix (i.e. the matrix will be block diagonal if the observational data is sorted by the group).

The uncorrelated components will form a diagonal matrix.

Further a distance-based component can be constructed, where distances between records within the same grid box are evaluated.

The functions in this module are valid for observational data where there could be more than 1 observation in a gridbox.

glomar_gridding.error_covariance.**correlated_components**(*df*, *group_col*, *bias_sig_col=None*, *bias_sig_map=None*)

Returns measurements covariance matrix updated by adding bias uncertainty to the measurements based on a grouping within the observational data.

The result is equivalent to a block diagonal matrix via permutation. If the input observational data is sorted by the group column then the resulting matrix is block diagonal, where the blocks are the size of each grouping. The values in each block are the square of the sigma value associated with the grouping.

Note that in most cases the output is not a block-diagonal, as the input is not usually sorted by the group column. In most processing cases, the input dataframe will be sorted by the gridbox index.

The values can either be pre-defined in the observational dataframe, and can be indicated by the “bias_val_col” argument. Alternatively, a mapping can be passed, the values will be then assigned by this mapping of group to sigma.

Parameters

- **df** (*polars.DataFrame*) – Observational DataFrame including group information and bias uncertainty values for each grouping. It is assumed that a single bias uncertainty value applies to the whole group, and is applied as cross terms in the covariance matrix (plus to the diagonal).
- **group_col** (*str*) – Name of the column that can be used to partition the observational DataFrame.
- **bias_sig_col** (*str | None*) – Name of the column containing bias uncertainty values for each of the groups identified by ‘group_col’. It is assumed that a single bias uncertainty value applies to the whole group, and is applied as cross terms in the covariance matrix (plus to the diagonal).
- **bias_sig_map** (*dict[str, float] | None*) – Mapping between values in the group_col and bias uncertainty values, if bias_val_col is not in the DataFrame.

Return type
The correlated components of the error covariance.

`glomar_gridding.error_covariance.`**`dist_weight`**`(`*df*, *dist_fn*, *grid_idx='grid_idx'*, ***dist_kwargs*`)`

Compute the distance and weight matrices over gridboxes for an input Frame.

This function acts as a wrapper for a distance function, allowing for computation of the distances between positions in the same gridbox using any distance metric.

The weightings from this function are for the gridbox mean of the observations within a gridbox.

Parameters

- **df** (*polars.DataFrame*) – The observation DataFrame, containing the columns required for computation of the distance matrix. Contains the "grid_idx" column which indicates the gridbox for a given observation. The index of the DataFrame should match the index ordering for the output distance matrix/weights.
- **dist_fn** (*Callable*) – The function used to compute a distance matrix for all points in a given grid-cell. Takes as input a polars.DataFrame as first argument. Any other arguments should be constant over all gridboxes, or can be a look-up table that can use values in the DataFrame to specify values specific to a gridbox. The function should return a numpy matrix, which is the distance matrix for the gridbox only. This wrapper function will correctly apply this matrix to the larger distance matrix using the index from the DataFrame.

  If dist_fn is None, then no distances are computed and None is returned for the dist value.
- **grid_idx** (*str*) – Name of the column containing the grid index values
- ****dist_kwargs** – Arguments to be passed to dist_fn. In general these should be constant across all gridboxes. It is possible to pass a look-up table that contains pre-computed values that are gridbox specific, if the keys can be matched to a column in df.

Return type
`tuple`[`ndarray`, `ndarray`]

Returns

- **dist** (*numpy.matrix*) – The distance matrix, which contains the same number of rows and columns as rows in the input DataFrame df. The values in the matrix are 0 if the indices of the row/column are for observations from different gridboxes, and non-zero if the row/column indices fall within the same gridbox. Consequently, with appropriate re-arrangement of rows and columns this matrix can be transformed into a block-diagonal matrix. If the DataFrame input is pre-sorted by the gridbox column, then the result is a block-diagonal matrix.

  If dist_fn is None, then this value will be None.
- **weights** (*numpy.matrix*) – A matrix of weights. This has dimensions n x p where n is the number of unique gridboxes and p is the number of observations (the number of rows in df). The values are 0 if the row and column do not correspond to the same gridbox and equal to the inverse of the number of observations in a gridbox if the row and column indices fall within the same gridbox. The rows of weights are in a sorted order of the gridbox. Should this be incorrect, one should re-arrange the rows after calling this function.

`glomar_gridding.error_covariance.`**`get_weights`**`(`*df*, *grid_idx='grid_idx'*`)`

Get just the weight matrices over gridboxes for an input Frame.

The weightings from this function are for the gridbox mean of the observations within a gridbox.

Parameters

- **df** (*polars.DataFrame*) – The observation DataFrame, containing the columns required for computation of the distance matrix. Contains the "grid_idx" column which indicates the gridbox for a given observation. The index of the DataFrame should match the index ordering for the output weights.
- **grid_idx** (*str*) – Name of the column containing the gridbox index from the output grid.

Returns

weights – A matrix of weights. This has dimensions n x p where n is the number of unique gridboxes and p is the number of observations (the number of rows in df). The values are 0 if the row and column do not correspond to the same gridbox and equal to the inverse of the number of observations in a gridbox if the row and column indices fall within the same gridbox. The rows of weights are in a sorted order of the gridbox. Should this be incorrect, one should re-arrange the rows after calling this function.

Return type

numpy.matrix

glomar_gridding.error_covariance.**uncorrelated_components**(*df*, *group_col='data_type'*, *obs_sig_col=None*, *obs_sig_map=None*)

Calculates the covariance matrix of the measurements (observations). This is the uncorrelated component of the covariance.

The result is a diagonal matrix. The diagonal is formed by the square of the sigma values associated with the values in the grouping.

The values can either be pre-defined in the observational dataframe, and can be indicated by the "bias_val_col" argument. Alternatively, a mapping can be passed, the values will be then assigned by this mapping of group to sigma.

Parameters

- **df** (*polars.DataFrame*) – The observational DataFrame containing values to group by.
- **group_col** (*str*) – Name of the group column to use to set observational sigma values.
- **obs_sig_col** (*str | None*) – Name of the column containing observational sigma values. If set and present in the DataFrame, then this column is used as the diagonal of the returned covariance matrix.
- **obs_sig_map** (*dict[str, float] | None*) – Mapping between group and observational sigma values used to define the diagonal of the returned covariance matrix.

Return type

ndarray

Returns

- *A diagonal matrix representing the uncorrelated components of the error*
- *covariance matrix.*

CHAPTER

NINE

KRIGING

The *glomar_gridding.kriging* module contains classes and functions for interpolation via Kriging. Two methods of Kriging are supported by *glomar_gridding*:

- Simple Kriging
- Ordinary Kriging

For each Kriging method there is a class and a function. The recommended approach is to use the classes, the functions will be deprecated in a future version of *glomar_gridding*. The classes require the full grid spatial covariance structure, the observation values, and the grid index of each observation, and an optional error covariance matrix as inputs. Each grid index should be a single index value, and represents the flattened index, and connects directly to the corresponding index of the covariance matrices. If an error covariance matrix is provided, the covariance matrix will be automatically subset to the grid index values, if the resulting matrix contains *nan* or *0* values on the diagonal, then the observation values and indices are filtered to exclude these points, and the error covariance matrix is subset again. Just initialising the class does not solve the system, this requires the *solve* method to be called.

9.1 Preparation

glomar_gridding provides functionality for preparing your data for the interpolation. The *grid* module has functionality for defining the output grid (`glomar_gridding.grid.grid_from_resolution()`) which allows the user to create a coordinate system for the output, that can easily be mapped to a covariance matrix. The grid object is an *xarray.DataArray* object, with a coordinate system. Once the grid is defined, the observations can be mapped to the grid. This creates a 1-dimensional index value that should match to the covariance matrices used in the interpolation.

`glomar_gridding.grid.`**`map_to_grid`**`(`*obs*, *grid*, *obs_coords=['lat', 'lon']*, *grid_coords=['latitude', 'longitude']*, *sort=True*, *bounds=None*, *add_grid_pts=True*, *grid_prefix='grid_'*`)`

Align an observation dataframe to a grid defined by an xarray DataArray.

Maps observations to the nearest grid-point, and sorts the data by the 1d index of the DataArray in a row-major format.

The grid defined by the latitude and longitude coordinates of the input DataArray is then used as the output grid of the Gridding process.

Parameters

- **obs** (*polars.DataFrame*) – The observational DataFrame containing positional data with latitude, longitude values within the *obs_latname* and *obs_lonname* columns respectively. Observations are mapped to the nearest grid-point in the grid.
- **grid** (*xarray.DataArray*) – Contains the grid coordinates to map observations to.
- **obs_coords** (*list[str]*) – Names of the column containing positional values in the input observational DataFrame.

- **grid_coords** (*list[str]*) – Names of the coordinates in the input grid DataArray used to define the grid.
- **sort** (*bool*) – Sort the observational DataFrame by the grid index
- **bounds** (*list[tuple[float, float]] | None*) – Optionally filter the grid and DataFrame to fall within spatial bounds. This list must have the same size and ordering as *obs_coords* and *grid_coords* arguments.
- **add_grid_pts** (*bool*) – Add the grid positional information to the observational DataFrame.
- **grid_prefix** (*str*) – Prefix to use for the new grid columns in the observational DataFrame.

Returns

obs – Containing additional *grid_**, and *grid_idx* values indicating the positions and grid index of the observation respectively. The DataFrame is also sorted (ascendingly) by the *grid_idx* columns for consistency with the gridding functions.

Return type

pandas.DataFrame

Examples

```
>>> obs = pl.read_csv("/path/to/obs.csv")
>>> grid = grid_from_resolution(
        resolution=5,
        bounds=[(-87.5, 90), (-177.5, 180)],  # Lower bound is centre
        coord_names=["lat", "lon"]
    )
>>> obs = map_to_grid(obs, grid, grid_coords=["lat", "lon"])
```

For Kriging, the interpolation requires at most a single observation value in each grid box. If the data contains multiple values in a single grid cell then these need to be combined.

glomar_gridding.kriging.**prep_obs_for_kriging**(*unmask_idx*, *unique_obs_idx*, *weights*, *obs*, *remove_obs_mean=0*, *obs_bias=None*, *error_cov=None*)

Prep masked observations for Kriging. Combines observations in the same grid box to a single averaged observation using a weighted average.

Parameters

- **unmask_idx** (*np.ndarray[int]*) – Indices of all un-masked points for chosen date.
- **unique_obs_idx** (*np.ndarray[int]*) – Unique indices of all measurement points for a chosen date, representative of the indices of gridboxes, which have => 1 measurement.
- **weights** (*np.ndarray[float]*) – Weight matrix (inverse of counts of observations).
- **obs** (*np.ndarray[float]*) – All point observations/measurements for the chosen date.
- **remove_obs_mean** (*int*) – Should the mean or median from obs be removed and added back onto obs?
  - 0 = No (default action)
  - 1 = the mean is removed
  - 2 = the median is removed
  - 3 = the spatial meam os removed

  Note that the mean will need to be reapplied to the Kriging result.

- **obs_bias** (*np.ndarray[float] | None*) – Bias of all measurement points for a chosen date (corresponds to x_obs).

Return type

`tuple[ndarray, ndarray]`

Returns

- **obs_idx** (*numpy.ndarray[int]*) – Subset of grid-box indices containing observations that are unmasked.
- **grid_obs** (*numpy.ndarray[float]*) – Unmasked and combined observations

9.2 Simple Kriging

`class glomar_gridding.kriging.SimpleKriging`(*covariance*, *idx*, *obs*, *error_cov=None*)

Class for SimpleKriging.

The equation for simple Kriging is:

$$(C_{obs} + E)^{-1} \times C_{cross} \times y + \mu$$

Where μ is a constant known mean, typically this is 0.

Parameters

- **covariance** (*numpy.ndarray*) – The spatial covariance matrix. This can be a pre-computed matrix loaded into the environment, or computed from a Variogram class or using Ellipse methods.
- **idx** (*numpy.ndarray[int] | list[int]*) – The 1d indices of observation grid points. These values should be between 0 and (N * M) - 1 where N, M are the number of longitudes and latitudes respectively. Note that these values should also be computed using "C" ordering in numpy reshaping. They can be computed from a grid using glomar_gridding.grid.map_to_grid. Each value should only appear once. Points that contain more than 1 observation should be averaged
- **obs** (*numpy.ndarray[float]*) – The observation values. If there are multiple observations in any grid box then these values need to be averaged into one value per grid box.
- **error_cov** (*numpy.ndarray | None*) – Optionally add error covariance values to the covariance between observation grid points.

`constraint_mask()`

Compute the observational constraint mask (A14 in [Morice_2021]) to determine if a grid point should be masked/weights modified by how far it is to its near observed point

Note: typo in Section A4 in [Morice_2021] (confirmed by authors).

Equation to use is A14 is incorrect. Easily noticeable because dimensionally incorrect is wrong, but the correct answer is easy to figure out.

Correct Equation (extra matrix inverse for $C_{obs} + E$):

$$\frac{1 - diag(C - C_{cross}^T \times (C_{obs} + E)^{-1} \times C_{cross})}{diag(C)} < \alpha$$

This can be re-written as:

$$\frac{diag(C_{cross}^T \times (C_{obs} + E)^{-1} \times C_{cross})}{diag(C)} < \alpha$$

α is chosen to be 0.25 in the UKMO paper

Written by S. Chan, modified by J. Siddons.

This requires that the *kriging_weights* attribute is set.

Returns
constraint_mask – Constraint mask values, the left-hand-side of equation A14 from Morice et al. (2021). This is a vector of length *k_obs.size[0]*.

Return type
numpy.ndarray

References

[Morice_2021]: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2019JD032361

get_kriging_weights()

Compute the Kriging weights from the flattened grid indices where there is an observation. Optionally add an error covariance to the covariance between observation grid points.

The Kriging weights are calculated as:

$$(C_{obs} + E)^{-1} \times C_{cross}$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, and C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points).

Sets the *kriging_weights* attribute.

Return type
`None`

get_uncertainty()

Compute the kriging uncertainty. This requires the attribute *kriging_weights* to be computed.

Returns
uncert – The Kriging uncertainty.

Return type
numpy.ndarray

kriging_weights_from_inverse(*inv*)

Compute the Kriging weights from the flattened grid indices where there is an observation, using a pre-computed inverse of the covariance between grid-points with observations.

The Kriging weights are calculated as:

$$(C_{obs} + E)^{-1} \times C_{cross}$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, and C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points).

Sets the *kriging_weights* attribute.

Parameters
inv (*numpy.ndarray*) – The pre-computed inverse of the covariance between grid-points with observations. $(C_{obs} + E)^{-1}$

Return type
`None`

`solve`(*mean=0.0*)

Solves the simple Kriging problem. Computes the Kriging weights if the *kriging_weights* attribute is not already set. The solution to Kriging is:

$$(C_{obs} + E)^{-1} \times C_{cross} \times y$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points), and y are the observation values.

Parameters

`mean` (*`numpy.ndarray | float`*) – Constant, known, mean value of the system. Defaults to 0.0.

Returns

The solution to the simple Kriging problem (as a Vector, this may need to be re-shaped appropriately as a post-processing step).

Return type

numpy.ndarray

Examples

```
>>> SK = SimpleKriging(interp_covariance)
>>> SK.solve(obs, idx, error_covariance)
```

`glomar_gridding.kriging.`**`kriging_simple`**(*obs_obs_cov*, *obs_grid_cov*, *grid_obs*, *interp_cov*, *mean=0.0*)

Perform Simple Kriging assuming a constant known mean.

This function is deprecated in favour of SimpleKriging class. It will be removed in version 1.0.0.

Parameters

- **`obs_obs_cov`** (*`np.ndarray[float]`*) – Covariance between all measured grid points plus the covariance due to measurements (i.e. measurement noise, bias noise, and sampling noise). Can include error covariance terms.
- **`obs_grid_cov`** (*`np.ndarray[float]`*) – Covariance between the all (predicted) grid points and measured points. Does not contain error covarance.
- **`grid_obs`** (*`np.ndarray[float]`*) – Gridded measurements (all measurement points averaged onto the output gridboxes).
- **`interp_cov`** (*`np.ndarray[float]`*) – interpolation covariance of all output grid points (each point in time and all points against each other).
- **`mean`** (*`float`*) – The constant mean of the output field.

Return type

`tuple`[`ndarray`, `ndarray`]

Returns

- **z_obs** (*np.ndarray[float]*) – Full set of values for the whole domain derived from the observation points using simple kriging.
- **dz** (*np.ndarray[float]*) – Uncertainty associated with the simple kriging.

9.3 Ordinary Kriging

class glomar_gridding.kriging.**OrdinaryKriging**(*covariance*, *idx*, *obs*, *error_cov=None*)

Class for OrdinaryKriging.

The equation for ordinary Kriging is:

$$(C_{obs} + E)^{-1} \times C_{cross} \times y$$

with a constant but unknown mean.

In this case, the C_{obs}, C_{cross} and y values are extended with a Lagrange multiplier term, ensuring that the Kriging weights are constrained to sum to 1.

The matrix C_{obs} is extended by one row and one column, each containing the value 1, except at the diagonal point, which is 0. The C_{cross} matrix is extended by an extra row containing values of 1. Finally, the grid observations y is extended by a single value of 0 at the end of the vector.

Parameters

- **covariance** (*numpy.ndarray*) – The spatial covariance matrix. This can be a pre-computed matrix loaded into the environment, or computed from a Variogram class or using Ellipse methods.
- **idx** (*numpy.ndarray*) – The 1d indices of observation grid points. These values should be between 0 and (N * M) - 1 where N, M are the number of longitudes and latitudes respectively. Note that these values should also be computed using "C" ordering in numpy reshaping. They can be computed from a grid using glomar_gridding.grid.map_to_grid. Each value should only appear once. Points that contain more than 1 observation should be averaged. Used to compute the Kriging weights.
- **obs** (*numpy.ndarray*) – The observation values. If there are multiple observations in any grid box then these values need to be averaged into one value per grid box.
- **error_cov** (*numpy.ndarray | None*) – Optionally add error covariance values to the covariance between observation grid points.

constraint_mask(*simple_kriging_weights=None*)

Compute the observational constraint mask (A14 in [Morice_2021]) to determine if a grid point should be masked/weights modified by how far it is to its near observed point

Note: typo in Section A4 in [Morice_2021] (confirmed by authors).

Equation to use is A14 is incorrect. Easily noticeable because dimensionally incorrect is wrong, but the correct answer is easy to figure out.

Correct Equation (extra matrix inverse for $C_{obs} + E$):

$$\frac{1 - diag(C - C_{cross}^T \times (C_{obs} + E)^{-1} \times C_{cross})}{diag(C)} < \alpha$$

This can be re-written as:

$$\frac{diag(C_{cross}^T \times (C_{obs} + E)^{-1} \times C_{cross})}{diag(C)} < \alpha$$

α is chosen to be 0.25 in the UKMO paper

Written by S. Chan, modified by J. Siddons.

This requires the Kriging weights from simple Kriging. If these are not provided as an input, then they are calculated.

Parameters
`simple_kriging_weights` (*numpy.ndarray | None,*) – The Kriging weights for the equivalent simple Kriging system. error covariance.

Returns
constraint_mask – Constraint mask values, the left-hand-side of equation A14 from Morice et al. (2021). This is a vector of length *k_obs.size[0]*.

Return type
numpy.ndarray

References

[Morice_2021]: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2019JD032361

extended_inverse(*simple_inv*)

Compute the inverse of a covariance matrix $S = C_{obs} + E$, and use that to compute the inverse of the extended version of the covariance matrix with Lagrange multipliers, used by Ordinary Kriging.

This is useful when one needs to perform BOTH simple and ordinary Kriging, or when one wishes to compute the constraint mask for ordinary Kriging which requires the Kriging weights for the equivalent simple Kriging problem.

The extended form of S is given by:

$$\begin{pmatrix} & & & 1 \\ & S & & \vdots \\ & & & 1 \\ 1 & \dots & 1 & 0 \end{pmatrix}$$

This approach follows Guttman 1946 10.1214/aoms/1177730946

Parameters
`simple_inv` (*numpy.matrix*) – Inverse of the covariance between observation grid-points

Returns
Inverse of the extended covariance matrix between observation grid-points including the Lagrange multiplier factors.

Return type
numpy.matrix

get_kriging_weights()

Compute the Kriging weights from the flattened grid indices where there is an observation. Optionally add an error covariance to the covariance between observation grid points.

The Kriging weights are calculated as:

$$(C_{obs} + E)^{-1} \times C_{cross}$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, and C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points).

In this case, the C_{obs}, C_{cross} and are extended with a Lagrange multiplier term, ensuring that the Kriging weights are constrained to sum to 1.

The matrix C_{obs} is extended by one row and one column, each containing the value 1, except at the diagonal point, which is 0. The C_{cross} matrix is extended by an extra row containing values of 1.

Sets the *kriging_weights* attribute.

Return type
`None`

`get_uncertainty()`

Compute the kriging uncertainty. This requires the attribute *kriging_weights* to be computed.

Returns
uncert – The Kriging uncertainty.

Return type
numpy.ndarray

`kriging_weights_from_inverse(inv)`

Compute the Kriging weights from the flattened grid indices where there is an observation, using a pre-computed inverse of the covariance between grid-points with observations.

The Kriging weights are calculated as:

$$(C_{obs} + E)^{-1} \times C_{cross}$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, and C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points).

In this case, the inverse matrix must be computed from the covariance between observation grid-points with the Lagrange multiplier applied.

This method is appropriate if one wants to compute the constraint mask which requires simple Kriging weights, which can be computed from the unextended covariance inverse. The extended inverse can then be calculated from that inverse.

Sets the *kriging_weights* attribute.

Parameters
inv (*numpy.ndarray*) – The pre-computed inverse of the covariance between grid-points with observations. $(C_{obs} + E)^{-1}$

Return type
`None`

`solve()`

Solves the ordinary Kriging problem. Computes the Kriging weights if the *kriging_weights* attribute is not already set. The solution to Kriging is:

$$(C_{obs} + E)^{-1} \times C_{cross} \times y$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points), and y are the observation values.

In this case, the C_{obs}, C_{cross} and are extended with a Lagrange multiplier term, ensuring that the Kriging weights are constrained to sum to 1.

The matrix C_{obs} is extended by one row and one column, each containing the value 1, except at the diagonal point, which is 0. The C_{cross} matrix is extended by an extra row containing values of 1.

Returns
The solution to the ordinary Kriging problem (as a Vector, this may need to be re-shaped appropriately as a post-processing step).

Return type
numpy.ndarray

Examples

```
>>> OK = OrdinaryKriging(interp_covariance)
>>> OK.solve(obs, idx, error_covariance)
```

`glomar_gridding.kriging.`**`kriging_ordinary`**`(`*obs_obs_cov*, *obs_grid_cov*, *grid_obs*, *interp_cov*`)`

Perform Ordinary Kriging with unknown but constant mean.

This function is deprecated in favour of OrdinaryKriging class. It will be removed in version 1.0.0.

Parameters

- **obs_obs_cov** (*np.ndarray[float]*) – Covariance between all measured grid points plus the covariance due to measurements (i.e. measurement noise, bias noise, and sampling noise). Can include error covariance terms, if these are being used.
- **obs_grid_cov** (*np.ndarray[float]*) – Covariance between the all (predicted) grid points and measured points. Does not contain error covarance.
- **grid_obs** (*np.ndarray[float]*) – Gridded measurements (all measurement points averaged onto the output gridboxes).
- **interp_cov** (*np.ndarray[float]*) – Interpolation covariance of all output grid points (each point in time and all points against each other).

Return type

`tuple`[`ndarray`, `ndarray`]

Returns

- **z_obs** (*np.ndarray[float]*) – Full set of values for the whole domain derived from the observation points using ordinary kriging.
- **dz** (*np.ndarray[float]*) – Uncertainty associated with the ordinary kriging.

9.4 Perturbed Gridded Fields

An additional two-stage combined Kriging class is provided in the *stochastic* module. In this case, the *solve* method has an additional optional *simulated_state* argument, if this is set, then the value is used as the simulated system state used as the base of the perturbed field (from which observations are simulated), otherwise the value is computed. This allows for pre-computation of a sequence of simulated states as part of ensemble processing, for example.

`class` `glomar_gridding.stochastic.`**`StochasticKriging`**`(`*covariance*, *idx*, *obs*, *error_cov*`)`

Class for the combined two-stage Kriging approach following [Morice_2021] The first stage is to produce a gridded field from the observations using Ordinary Kriging. The second stage is to apply a perturbation.

The perturbation is constructed by first generating a simulated state from the covariance matrix. A set of simulated observations is drawn from the error covariance matrix. A simulated gridded field is then computed using Simple Kriging with the simulated observations as input. Finally, the perturbation is then the difference between the simulated gridded field and the simulated state. This perturbation is added to the gridded field from the first stage.

The equation for ordinary Kriging is:

$$(C_{obs} + E)^{-1} \times C_{cross} \times y$$

with a constant but unknown mean.

In this case, the C_{obs}, C_{cross} and y values are extended with a Lagrange multiplier term for the first stage, ensuring that the Kriging weights are constrained to sum to 1.

Additionally, the matrix C_{obs} is extended by one row and one column, each containing the value 1, except at the diagonal point, which is 0 for the first Ordinary Kriging stage. The C_{cross} matrix is extended by an extra row containing values of 1. Finally, the grid observations y is extended by a single value of 0 at the end of the vector.

Parameters

- **covariance** (*numpy.ndarray*) – The spatial covariance matrix. This can be a pre-computed matrix loaded into the environment, or computed from a Variogram class or using Ellipse methods.
- **idx** (*numpy.ndarray*) – The 1d indices of observation grid points. These values should be between 0 and (N * M) - 1 where N, M are the number of longitudes and latitudes respectively. Note that these values should also be computed using "C" ordering in numpy reshaping. They can be computed from a grid using glomar_gridding.grid.map_to_grid. Each value should only appear once. Points that contain more than 1 observation should be averaged. Used to compute the Kriging weights.
- **obs** (*numpy.ndarray*) – The observation values. If there are multiple observations in any grid box then these values need to be averaged into one value per grid box.
- **error_cov** (*numpy.ndarray*) – Error covariance values to the covariance between observation grid points.

`constraint_mask()`

Compute the observational constraint mask (A14 in [Morice_2021]) to determine if a grid point should be masked/weights modified by how far it is to its near observed point

Note: typo in Section A4 in [Morice_2021] (confired by authors).

Equation to use is A14 is incorrect. Easily noticeable because dimensionally incorrect is wrong, but the correct answer is easy to figure out.

Correct Equation (extra matrix inverse for $C_{obs} + E$):

$$\frac{1 - diag(C - C_{cross}^T \times (C_{obs} + E)^{-1} \times C_{cross})}{diag(C)} < \alpha$$

This can be re-written as:

$$\frac{diag(C_{cross}^T \times (C_{obs} + E)^{-1} \times C_{cross})}{diag(C)} < \alpha$$

α is chosen to be 0.25 in the UKMO paper

Written by S. Chan, modified by J. Siddons.

This requires the Kriging weights from simple Kriging, set as the *simple_kriging_weights* attribute.

Returns

constraint_mask – Constraint mask values, the left-hand-side of equation A14 from [Morice_2021]. This is a vector of length *k_obs.size[0]*.

Return type

numpy.ndarray

References

[Morice_2021]: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2019JD032361

`get_kriging_weights()`

Compute the Kriging weights from the flattened grid indices where there is an observation. Optionally add an error covariance to the covariance between observation grid points.

The Kriging weights are calculated as:

$$(C_{obs} + E)^{-1} \times C_{cross}$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, and C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points).

In this case, the C_{obs}, C_{cross} and are extended with a Lagrange multiplier term, ensuring that the Kriging weights are constrained to sum to 1, for the first stage of the StochasticKriging process.

The matrix C_{obs} is extended by one row and one column, each containing the value 1, except at the diagonal point, which is 0. The C_{cross} matrix is extended by an extra row containing values of 1.

Sets the *kriging_weights* and *simple_kriging_weights* attributes.

Return type
`None`

`get_uncertainty()`

Compute the kriging uncertainty. This requires the attribute *kriging_weights* to be computed.

Returns
uncert – The Kriging uncertainty.

Return type
numpy.ndarray

`kriging_weights_from_inverse(`*inv*`)`

Compute the Kriging weights from the flattened grid indices where there is an observation, using a pre-computed inverse of the covariance between grid-points with observations.

The Kriging weights are calculated as:

$$(C_{obs} + E)^{-1} \times C_{cross}$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, and C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points).

This method is appropriate if one wants to compute the constraint mask which requires simple Kriging weights, which can be computed from the unextended covariance inverse. The extended inverse can then be calculated from that inverse.

The inverse matrix is used to compute the inverse of the extended covariance matrix used for the first Ordinary Kriging stage.

Sets the *kriging_weights* and *simple_kriging_weights* attributes.

Parameters
inv (*numpy.ndarray*) – The pre-computed inverse of the covariance between grid-points with observations. $(C_{obs} + E)^{-1}$

Return type
`None`

`set_simple_kriging_weights`(*simple_kriging_weights*)

Set Simple Kriging Weights. For use in the second Simple Kriging stage for computing the simulated gridded field.

Sets the *simple_kriging_weights* attribute.

Parameters

`simple_kriging_weights` (*numpy.ndarray*) – The pre-computed simple_kriging_weights to use.

Return type

`None`

`solve`(*simulated_state=None*)

Solves the combined Stochastic Kriging problem. Computes the Kriging weights if the *kriging_weights* attribute is not already set.

Stochastic Kriging is a combined Ordinary and Simple Kriging approach. First a gridded field is generated for the observations using Ordinary Kriging. Secondly, a simulated gridded field is generated using Simple Kriging from a simulated state and simulated observations. The simulated state can be pre-computed or computed by this method. The simulated observations are drawn from the error covariance and located at the simulated state. The difference between the simulated gridded field and the simulated state is added to the gridded field.

The solution to Kriging is:

$$(C_{obs} + E)^{-1} \times C_{cross} \times y$$

Where C_{obs} is the spatial covariance between grid-points with observations, E is the error covariance between grid-points with observations, C_{cross} is the covariance between grid-points with observations and all grid-points (including observation grid-points), and y are the observation values.

In this case, the C_{obs}, C_{cross} and are extended with a Lagrange multiplier term, ensuring that the Kriging weights are constrained to sum to 1. For the Ordinary Kriging component.

The matrix C_{obs} is extended by one row and one column, each containing the value 1, except at the diagonal point, which is 0. The C_{cross} matrix is extended by an extra row containing values of 1. For the Ordinary Kriging component.

This additionally sets the following attributes:

- *gridded_field* - The unperturbed gridded field
- *simulated_grid* - The simulated gridded field
- *epsilon* - The perturbation

Parameters

`simulated_state` (*numpy.ndarray | None*) – Flattened simulated state, used as the location basis for the simulated observation draw. If this is not provided it will be calculated. Often it is better to pre-compute a series of states in advance, since this can be a time consuming step (drawing a single state takes approximately the same time as drawing 200).

Returns

The solution to the stochastic Kriging problem (as a Vector, this may need to be re-shaped appropriately as a post-processing step).

Return type

numpy.ndarray

Examples

```
>>> SK = StochasticKriging(interp_covariance)
>>> SK.solve(obs, idx, error_covariance)
```

glomar_gridding.stochastic.**scipy_mv_normal_draw**(*loc*, *cov*, *ndraws=1*, *eigen_rtol=1e-06*, *eigen_fudge=1e-08*)

Do a random multivariate normal draw using scipy.stats.multivariate_normal.rvs

numpy.random.multivariate_normal can also, but fixing seeds are more difficult using numpy

This function has similar API as GP_draw with less kwargs.

Warning/possible future scipy version may change this: It seems if one uses stats.Covariance, you have to have add [0] from rvs function. The above behavior applies to scipy v1.14.0

Parameters

- **loc** (*float*) – the location for the normal dry
- **cov** (*numpy.ndarray*) – not a xarray/iris cube! Some of our covariances are saved in numpy format and not netCDF files
- **n_draws** (*int*) – number of simulations, this is usually set to 1 except during
- **testing** (*unit*)
- **eigen_rtol** (*float*) – relative tolerance to negative eigenvalues
- **eigen_fudge** (*float*) – forced minimum value of eigenvalues if negative values are detected

Returns

draw – The draw(s) from the multivariate random normal distribution defined by the loc and cov parameters. If the cov parameter is not positive-definite then a new covariance will be determined by adjusting the eigen decomposition such that the modified covariance should be positive-definite.

Return type

np.ndarray

Examples

```
>>> A = np.random.rand(5, 5)
>>> cov = np.dot(A, A.T)
>>> scipy_mv_normal_draw(np.zeros(5), cov, ndraws=5)
array([[-0.35972806, -0.51289612,  0.85307028, -0.11580307,  0.6677707 ],
       [-1.38214628, -1.29331638, -0.4879436 , -1.42310831, -0.19369562],
       [-1.04502143, -1.97686163, -2.058605  , -1.97202206, -2.90116796],
       [-1.97981119, -2.72330373,  0.0088662 , -2.53521893, -0.03670664],
       [ 0.49948228,  0.54695988,  0.33864294,  0.53730282,  0.14743019]])
```

9.5 Outputs

The outputs to the solvers, `glomar_gridding.SimpleKriging.solve()` for example will be vectors, they should be re-shaped to the grid.

Outputs can also be re-mapped to the grid glomar_gridding.grid.**assign_to_grid**(*values*, *grid_idx*, *grid*, *fill_value=nan*)

Assign a vector of values to a grid, using a list of grid index values.

Parameters

- **values** (*numpy.ndarray*) – The values to map onto the output grid.
- **grid_idx** (*numpy.ndarray*) – The 1d index of the grid (assuming “C” style ravelling) for each value.
- **grid** (*xarray.DataArray*) – The grid used to define the output grid.
- **fill_value** (*Any*) – The value to fill unassigned grid boxes. Must be a valid value of the input *values* data type.

Returns

out_grid – A new grid containing the values mapped onto the grid.

Return type

xarray.DataArray

CHAPTER TEN

MISCALLANEOUS MODULES

10.1 Climatologies

Functions for mapping climatologies and computing anomalies

`glomar_gridding.climatology.`**`join_climatology_by_doy`**(*obs_df*, *climatology_365*, *lat_col='lat'*, *lon_col='lon'*, *date_col='date'*, *var_col='sst'*, *clim_lat='latitude'*, *clim_lon='longitude'*, *clim_doy='doy'*, *clim_var='climatology'*, *temp_from_kelvin=True*)

Merge a climatology from an xarray.DataArray into a polars.DataFrame using the **day of year** value and position.

This function accounts for leap years by taking the average of the climatology values for 28th Feb and 1st March for observations that were made on the 29th of Feb.

The climatology is merged into the DataFrame and anomaly values are computed.

Parameters

- **obs_df** (*polars.DataFrame*) – Observational DataFrame.
- **climatology_365** (*xarray.DataArray*) – DataArray containing daily climatology values (for 365 days).
- **lat_col** (*str*) – Name of the latitude column in the observational DataFrame.
- **lon_col** (*str*) – Name of the longitude column in the observational DataFrame.
- **date_col** (*str*) – Name of the datetime column in the observational DataFrame. Day of year values are computed from this value.
- **var_col** (*str*) – Name of the variable column in the observational DataFrame. The merged climatology names will have this name prefixed to "_climatology", the anomaly values will have this name prefixed to "_anomaly".
- **clim_lat** (*str*) – Name of the latitude coordinate in the climatology DataArray.
- **clim_lon** (*str*) – Name of the longitude coordinate in the climatology DataArray.
- **clim_doy** (*str*) – Name of the day of year coordinate in the climatology DataArray.
- **clim_var** (*str*) – Name of the climatology variable in the climatology DataArray.
- **temp_from_kelvin** (*bool*) – Optionally adjust the climatology from Kelvin to Celsius if the variable is a temperature.

Returns

obs_df – With the climatology merged and anomaly computed. The new columns are "_climatology" and "_anomaly" prefixed by the *var_col* value respectively.

Return type
polars.DataFrame

`glomar_gridding.climatology.`**`read_climatology`**(*clim_path*, *min_lat=-90*, *max_lat=90*, *min_lon=-180*, *max_lon=180*, *lat_var='lat'*, *lon_var='lon'*, ***kwargs*)

Load a climatology dataset from a netCDF file.

Parameters

- **`clim_path`** (*`str`*) – Path to the climatology file. Can contain format blocks to be replaced by the values passed to kwargs.
- **`min_lat`** (*`float`*) – Minimum latitude to load.
- **`max_lat`** (*`float`*) – Maximum latitude to load.
- **`min_lon`** (*`float`*) – Minimum longitude to load.
- **`max_lon`** (*`float`*) – Maximum longitude to load.
- **`lat_var`** (*`str`*) – Name of the latitude variable.
- **`lon_var`** (*`str`*) – Name of the longitude variable.
- **`**kwargs`** – Replacement values for the climatology path.

Returns
clim_ds – Containing the climatology bounded by the min/max arguments provided.

Return type
xarray.Dataset

10.2 Masking

Functions for applying masks to grids and DataFrames

`glomar_gridding.mask.`**`get_mask_idx`**(*mask*, *mask_val=nan*, *masked=True*)

Get the 1d indices of masked values from a mask array.

Parameters

- **`mask`** (*`xarray.DataArray`*) – The mask array, containing values indicated a masked value.
- **`mask_val`** (*`Any`*) – The value that indicates the position should be masked.
- **`masked`** (*`bool`*) – Return indices where values in the mask array equal this value. If set to False it will return indices where values are not equal to the mask value. Can be used to get unmasked indices if this value is set to False.

Return type
An array of integers indicating the indices which are masked.

Examples

```
>>> data = np.random.rand(4, 4)
>>> data[data > 0.65] = np.nan
>>> mask = xr.DataArray(data)
>>> get_mask_idx(mask)
array([[1],
       [3],
       [4],
```

(continues on next page)

(continued from previous page)

```
       [5],
       [8]])
```

glomar_gridding.mask.**mask_array**(*grid*, *mask*, *masked_value=nan*, *mask_value=True*)

Apply a mask to a DataArray.

The grid and mask must already align for this function to work. An error will be raised if the coordinate systems cannot be aligned.

Parameters

- **grid** (*xarray.DataArray*) – Observational DataArray to be masked by positions in the mask DataArray.
- **mask** (*xarray.DataArray*) – Array containing values used to mask the observational DataFrame.
- **masked_value** (*Any*) – Value indicating masked values in the DataArray.
- **mask_value** (*Any*) – Value to set masked values to in the observational DataFrame.

Returns

grid – Input xarray.DataArray with the variable masked by the mask DataArray.

Return type

xarray.DataArray

glomar_gridding.mask.**mask_dataset**(*dataset*, *mask*, *varnames*, *masked_value=nan*, *mask_value=True*)

Apply a mask to a DataSet.

The grid and mask must already align for this function to work. An error will be raised if the coordinate systems cannot be aligned.

Parameters

- **dataset** (*xarray.Dataset*) – Observational Dataset to be masked by positions in the mask DataArray.
- **mask** (*xarray.DataArray*) – Array containing values used to mask the observational DataFrame.
- **varnames** (*str | list[str]*) – A list containing the names of variables in the observational Dataser to apply the mask to.
- **masked_value** (*Any*) – Value indicating masked values in the DataArray.
- **mask_value** (*Any*) – Value to set masked values to in the observational DataFrame.

Returns

grid – Input xarray.Dataset with the variables masked by the mask DataArray.

Return type

xarray.Dataset

glomar_gridding.mask.**mask_from_obs_array**(*obs*, *datetime_idx*)

Infer a mask from an input array. Mask values are those where all values are NaN along the time dimension.

An example use-case would be to infer land-points from a SST data array.

Parameters

- **obs** (*numpy.ndarray*) – Array containing the observation values. Records that are numpy.nan will count towards the mask, if all values in the datetime dimension are numpy.nan.
- **datetime_idx** (*int*) – The index of the datetime, or grouping, dimension. If all records at a point along this dimension are NaN then this point will be masked.

Returns

mask – A boolean array with dimension reduced along the datetime dimension. A True value indicates that all values along the datetime dimension for this index are numpy.nan and are masked.

Return type

numpy.ndarray | xarray.DataArray

`glomar_gridding.mask.`**`mask_from_obs_frame`**`(`*obs*, *coords*, *value_col*, *datetime_col=None*, *grid=None*, *grid_coords=None*`)`

Compute a mask from observations and an optional output grid..

Positions defined by the "coords" values that do not have any observations, at any datetime value in the "datetime_col", for the "value_col" field are masked.

An example use-case would be to identify land positions from sst records.

If a grid is supplied, the observations are mapped to the grid and any positions from the grid that do not contain observations.

If no grid is supplied, then it is assumed that the observation frame represents the full grid, and any positions without observations are included with null values in the value_col.

Parameters

- **obs** (*polars.DataFrame*) – DataFrame containing observations over space and time. The values in the "value_col" field will be used to define the mask.
- **coords** (*str | list[str]*) – A list of columns containing the coordinates used to define the mask. For example ["lat", "lon"].
- **value_col** (*str*) – Name of the column containing values from which the mask will be defined.
- **datetime_col** (*str | None*) – Name of the datetime column. Any positions that contain no records at any datetime value are masked.
- **grid** (*xarray.DataArray | None*) – Optional grid, used to map observations so that empty positions can be identified. If not supplied, it is assumed that the observations frame contains the full grid, and includes nulls in empty positions.
- **grid_coords** (*str | list[str] | None*) – Optional grid coordinate names. Must be set if grid is set.

Return type

`DataFrame`

Returns

- *polars.DataFrame containing coordinate columns and a Boolean "mask" column*
- *indicating positions that contain no observations and would be a mask value.*

`glomar_gridding.mask.`**`mask_observations`**`(`*obs*, *mask*, *varnames*, *masked_value=nan*, *mask_value=True*, *obs_coords=['lat', 'lon']*, *mask_coords=['latitude', 'longitude']*, *align_to_mask=False*, *drop=False*, *mask_grid_prefix='_mask_grid_'*`)`

Mask observations in a DataFrame subject to a mask DataArray.

Parameters

- **obs** (*polars.DataFrame*) – Observational DataFrame to be masked by positions in the mask DataArray.
- **mask** (*xarray.DataArray*) – Array containing values used to mask the observational DataFrame.
- **varnames** (*str | list[str]*) – Columns in the observational DataFrame to apply the mask to.
- **masked_value** (*Any*) – Value indicating masked values in the DataArray.
- **mask_value** (*Any*) – Value to set masked values to in the observational DataFrame.
- **obs_coords** (*list[str]*) – A list of coordinate names in the observational DataFrame. Used to map the mask DataArray to the observational DataFrame. The order must align with the coordinates of the mask DataArray.
- **mask_coords** (*list[str]*) – A list of coordinate names in the mask DataArray. These coordinates are mapped onto the observational DataFrame in order to apply the mask. The ordering of the coordinate names in this list must match those in the obs_coords list.
- **align_to_mask** (*bool*) – Optionally align the observational DataFrame to the mask DataArray. This essentially sets the mask's grid as the output grid for interpolation.
- **drop** (*bool*) – Drop masked values in the observational DataFrame.
- **mask_grid_prefix** (*str*) – Prefix to use for the mask gridbox index column in the observational DataFrame.

Returns

obs – Input polars.DataFrame containing additional column named by the mask_varname argument, indicating records that are masked. Masked values are dropped if the drop argument is set to True.

Return type

polars.DataFrame

10.3 Distances and Distance Matrices

Functions for calculating distances or distance-based covariance components.

Some functions can be used for computing pairwise-distances, for example via squareform. Some functions can be used as a distance function for glomar_gridding.error_covariance.dist_weights, accounting for the distance component to an error covariance matrix.

Functions for computing covariance using Matern Tau by Steven Chan (@stchan).

glomar_gridding.distances.**calculate_distance_matrix**(*df*, *dist_func=<function haversine_distance_from_frame>*, *lat_col='lat'*, *lon_col='lon'*, **dist_kwargs)

Create a distance matrix from a DataFrame containing positional information, typically latitude and longitude, using a distance function.

Available functions are *haversine_distance*, *euclidean_distance*. A custom function can be used, requiring that the function takes the form: (tuple[float, float], tuple[float, float]) -> float

Parameters

- **df** (*polars.DataFrame*) – DataFrame containing latitude and longitude columns indicating the positions between which distances are computed to form the distance matrix
- **dist_func** (*Callable*) – The function used to calculate the pairwise distances. Functions available for this function are *haversine_distance* and *euclidean_distance*. A custom function can be based, that takes as input two tuples of positions (computing a single distance value between the pair of positions). (tuple[float, float], tuple[float, float]) -> float
- **lat_col** (*str*) – Name of the column in the input DataFrame containing latitude values.
- **lon_col** (*str*) – Name of the column in the input DataFrame containing longitude values.
- ****dist_kwargs** – Keyword arguments to pass to the distance function.

Returns

dist – A matrix of pairwise distances.

Return type

np.ndarray[float]

`glomar_gridding.distances.`**displacements**(*lats*, *lons*, *lats2=None*, *lons2=None*, *delta_x_method=None*)

Calculate east-west and north-south displacement matrices for all pairs of input positions.

The results are not scaled by any radius, this should be performed outside of this function.

Parameters

- **lats** (*numpy.ndarray*) – The latitudes of the positions, should be provided in degrees.
- **lons** (*numpy.ndarray*) – The longitudes of the positions, should be provided in degrees.
- **lats2** (*numpy.ndarray*) – The latitudes of the optional second positions, should be provided in degrees.
- **lons2** (*numpy.ndarray*) – The longitudes of the optional second positions, should be provided in degrees.
- **delta_x_method** (*str | None*) – One of "Met_Office" or "Modified_Met_Office". If set to None, the displacements will be returned in degrees, rather than actual distance values. Set to "Met_Office" to use a cylindrical approximation, set to "Modified_Met_Office" to use an approximation that uses the average of the latitudes to set the horizontal displacement scale.

Return type

`tuple`[ndarray, ndarray]

Returns

- **disp_y** (*numpy.ndarray*) – The north-south displacements.
- **disp_x** (*numpy.ndarray*) – The east-west displacements.

`glomar_gridding.distances.`**euclidean_distance**(*df*, *radius=6371.0*)

Calculate the Euclidean distance in kilometers between pairs of lat, lon points on the earth (specified in decimal degrees).

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

where

$$(x_n, y_n, z_n) = (R\cos(lat)\cos(lon), R\cos(lat)\sin(lon), R\sin(lat))$$

Parameters

- **df** (*polars.DataFrame*) – DataFrame containing latitude and longitude columns indicating the positions between which distances are computed to form the distance matrix
- **radius** (*float*) – The radius of the sphere used for the calculation. Defaults to the radius of the earth in km (6371.0 km).

Returns
dist – The direct pairwise distance between the positions in the input DataFrame through the sphere defined by the radius parameter.

Return type
float

References

https://math.stackexchange.com/questions/29157/how-do-i-convert-the-distance-between-two-lat-long-points-into-feet-meters
https://cesar.esa.int/upload/201709/Earth_Coordinates_Booklet.pdf

`glomar_gridding.distances.`**`haversine_distance_from_frame`**`(`*df*, *radius=6371*`)`

Calculate the great circle distance in kilometers between pairs of lat, lon points on the earth (specified in decimal degrees).

Parameters

- **df** (*polars.DataFrame*) – DataFrame containing latitude and longitude columns indicating the positions between which distances are computed to form the distance matrix
- **radius** (*float*) – The radius of the sphere used for the calculation. Defaults to the radius of the earth in km (6371.0 km).

Returns
dist – The pairwise haversine distances between the inputs in the DataFrame, on the sphere defined by the radius parameter.

Return type
numpy.ndarray

`glomar_gridding.distances.`**`haversine_gaussian`**`(`*df*, *R=6371.0*, *r=40*, *s=0.6*`)`

Gaussian Haversine Model

Parameters

- **df** (*polars.DataFrame*) – Observations, required columns are “lat” and “lon” representing latitude and longitude respectively.
- **R** (*float*) – Radius of the sphere on which Haversine distance is computed. Defaults to radius of earth in km.
- **r** (*float*) – Gaussian model range parameter
- **s** (*float*) – Gaussian model scale parameter

Returns
C – Distance matrix for the input positions. Result has been modified using the Gaussian model.

Return type
np.ndarray

`glomar_gridding.distances.`**`inv_2d`**`(`*mat*`)`

Compute the inverse of a 2 x 2 matrix

Return type
`ndarray`

glomar_gridding.distances.**mahal_dist_func**(*delta_x*, *delta_y*, *Lx*, *Ly*, *theta=None*)

Calculate tau from displacements, Lx, Ly, and theta (if it is known). For an array of displacements, for a set of scalar ellipse parameters, Lx, Ly, and theta.

Parameters

- **delta_x** (*numpy.ndarray*) – displacement to remote point as in: (delta_x) i + (delta_y) j in old school vector notation
- **delta_y** (*numpy.ndarray*) – displacement to remote point as in: (delta_x) i + (delta_y) j in old school vector notation
- **Lx** (*float*) – Lx, Ly scale (km or degrees)
- **Ly** (*float*) – Lx, Ly scale (km or degrees)
- **theta** (*float | None*) – rotation angle in radians

Returns
tau – Mahalanobis distance

Return type
float

glomar_gridding.distances.**radial_dist**(*lat1*, *lon1*, *lat2*, *lon2*)

Computes a distance matrix of the coordinates using a spherical metric.

Parameters

- **lat1** (*float*) – latitude of point A
- **lon1** (*float*) – longitude of point A
- **lat2** (*float*) – latitude of point B
- **lon2** (*float*) – longitude of point B

Return type
Radial distance between point A and point B

glomar_gridding.distances.**rot_mat**(*angle*)

Compute a 2d rotation matrix from an angle.

The input angle must be in radians

Return type
`ndarray`

glomar_gridding.distances.**sigma_rot_func**(*Lx*, *Ly*, *theta*)

Equation 15 in Karspeck el al 2011 and Equation 6 in Paciorek and Schervish 2006, assuming Sigma(Lx, Ly, theta) locally/moving-window invariant or we have already taken the mean (Sigma overbar, PP06 3.1.1)

Lx, Ly - anistropic variogram length scales theta - angle relative to lines of constant latitude theta should be radians, and the fitting code outputs radians by default

Returns
sigma – 2 x 2 matrix

Return type
np.ndarray

glomar_gridding.distances.**tau_dist**(*dE*, *dN*, *sigma*)

Eq.15 in Karspeck paper but it is standard formulation to the Mahalanobis distance https://en.wikipedia.org/wiki/Mahalanobis_distance 10.1002/qj.900

Return type
`ndarray`

`glomar_gridding.distances.`**`tau_dist_from_frame`**`(`*df*`)`

Compute the tau/Mahalanobis matrix for all records within a gridbox

Can be used as an input function for observations.dist_weight.

Eq.15 in Karspeck paper but it is standard formulation to the Mahalanobis distance https://en.wikipedia.org/wiki/Mahalanobis_distance 10.1002/qj.900

By Steven Chan - @stchan

Parameters
df (*polars.DataFrame*) – The observational DataFrame, containing positional information for each observation ("lat", "lon"), gridbox specific positional information ("grid_lat", "grid_lon"), and ellipse length-scale parameters used for computation of *sigma* ("grid_lx", "grid_ly", "grid_theta").

Returns
tau – A matrix of dimension n x n where n is the number of rows in *df* and is the tau/Mahalanobis distance.

Return type
numpy.matrix

10.4 Covariance Tools and Eigenvalue Clipping

When estimating covariance matrices, using ellipse-based methods for example, the results may not be positive-definite. This can be problematic, for instance the Kriging equations may not be solvable as the inverse matrix cannot be computed, or simulated states cannot be constructed using *`glomar_gridding.stochastic.scipy_mv_normal_draw()`*. To account for this, *glomar_gridding* includes a few tools for *fixing* these covariance matrices. In general, these methods attempt to coerce the input matrix to be positive-definite by updating the eigenvalues and re-computing the matrix from these updated eigenvalues and the original eigenvectors. The indicators of an invalid covariance matrix include

1. Un-invertible covariance matrices with 0 eigenvalues
2. Covariance matrices with eigenvalues less than zero

This can typically be a consequence of

1. Multicollinearity: but nearly all very large cov matrices will have rounding errors to have this occur
2. Number of spatial points >> length of time series (for ESA monthly pentads: this ratio is about 150)
3. Covariance is estimated using partial data

In most cases, the most likely causes are 2 and 3.

There are a number of methods included in this module. In general, the approach is to adjust the eigenvalues to ensure small or negative eigenvalues are increased to some minimum threshold. The covariance matrix is then re-calculated using these modified eigenvalues and the original eigenvectors.

In general, the recommended approach is Original Clipping, see *`glomar_gridding.covariance_tools.eigenvalue_clip()`*.

Fixes:

1. Simple clipping - *`glomar_gridding.covariance_tools.simple_clipping()`*:

Cut off the negative, zero, and small positive eigenvalues; this is method used in statsmodels.stats.correlation_tools but the version here has better thresholds based on the accuracy of the eigenvalues, plus a iterative version which is slower but more stable with big matrices. The iterative version is recommended for SST/MAT covariances.

This is used for SST covariance matrices which have less dominant modes than MAT; it also preserves more noise.

Trace (aka total variance) of the covariance matrix is not conserved, but it is less disruptive than EOF chop off (method 3).

It is more difficult to use for covariance matrices with one large dominant mode because that raises the bar of accuracy of the eigenvalues, which requires clipping off a lot more eigenvectors.

Note, this will adjust all negative eigenvalues to be positive.

2. Original clipping - *`glomar_gridding.covariance_tools.eigenvalue_clip()`*:

   Determine a noise eigenvalue threshold and replace all eigenvalues below using the average of them, preserving the original trace (aka total variance) of the covariance matrix, but this will require a full computation of all eigenvectors, which may be slow and cause memory problems

   Note, this will adjust all negative eigenvalues to be positive.

Other methods not implemented here -

a. **shrinkage methods**
   https://scikit-learn.org/stable/modules/covariance.html

b. **reprojection (aka Higham's method) [Higham]**
   https://github.com/mikecroucher/nearest_correlation

   https://nhigham.com/2013/02/13/the-nearest-correlation-matrix/

Author S. Chan.

Modified by J. Siddons.

`glomar_gridding.covariance_tools.`**`check_symmetric`**(*a*, *rtol=1e-05*, *atol=1e-08*)

Helper function for perturb_sym_matrix_2_positive_definite

Return type
`bool`

`glomar_gridding.covariance_tools.`**`clean_small`**(*matrix*, *atol=1e-05*)

Set small values (abs(x) < atol) in an matrix to 0

Return type
`ndarray`

`glomar_gridding.covariance_tools.`**`csum_up_to_val`**(*vals*, *target*, *reverse=True*, *niter=0*, *csum=0.0*)

Find csum and sample index that target is surpassed. Displays a warning if the target is not exceeded or the input *vals* is empty.

Can provide an initial *niter* and/or *csum* value(s), if working with multiple arrays in an iterative process.

If *reverse* is set, the returned index will be negative and will correspond to the index required for the non-reversed array. Reverse is the default.

Parameters

- **`vals`** (*`numpy.ndarray`*) – Vector of values to sum cumulatively.
- **`target`** (*`float`*) – Value for which the cumulative sum must exceed.

- **reverse** (*bool*) – Reverse the array. The index will be negative.
- **niter** (*int*) – Initial number of iterations.
- **csum** (*float*) – Initial cumulative sum value.

Return type

tuple[float, int]

Returns

- **csum** (*float*) – The cumulative sum at the index when the target has been exceeded.
- **niter** (*int*) – The index of the value that results in the cumulative sum exceeding the target.

Note

It is actually faster to compute a full cumulative sum with *np.cumsum* and then look for the value that exceeds the target. This is not performed in this function.

Examples

```
>>> vals = np.random.rand(1000)
>>> target = 301.1
>>> csum_up_to_val(vals, target)
```

glomar_gridding.covariance_tools.**eigenvalue_clip**(*cov*, *method='explained_variance'*, ***kwargs*)

Denoise symmetric damaged covariance/correlation matrix cov by clipping eigenvalues

Explained variance or aspect ratio based threshold Aspect ratios is based on dimensionless parameters (number of independent variable and observation size)

$$q = N/T = \frac{\text{num of independent variable}}{\text{num of observation per independent variable}}$$

Does not give the same results as in eig_clip

explained_variance here does not have the same meaning. The trace of a correlation, by definition, equals the number of diagonal elements, which isn't intituatively linked to actual explained variance in climate science sense

This is done by KEEPING the largest explained variance in which (number of basis vectors to be kept) >> (number of rows) In ESA data, keeping 95% variance means keeping top ~15% of the eigenvalues

Parameters

- **cov** (*numpy.ndarray*) – Input covariance matrix to be adjusted to positive definite.
- **method** (*"explained_variance" | "Laloux_2000"*) – Method used to identify the index of the eigenvalues to clip. If set to "explained_variance" then the sorted eigenvalues below the target variance are *clipped*. If "Laloux_2000" is set, then the method of [Laloux] is used.
- ****kwargs**

Returns

Adjusted covariance matrix.

Return type

numpy.ndarray

See also

glomar_gridding.covariance_tools.explained_variance_clip()

See also

glomar_gridding.covariance_tools.laloux_clip()

`glomar_gridding.covariance_tools.`**`explained_variance_clip`**(*cov*, *target_variance_fraction=0.95*)

Clip all EOFs beyond a certain level of explained variance Starting from the EOF with the largest eigenvalue (explained variance) descending down the eigenvalues until the cumulative sum of the eigenvalues just goes beyond (Trace(cov) x target_variance_fraction)

All the larger eigenvalues up to the above thresholds are left unchanged.

Eigenvalues beyond (smaller positive and negative ones) are clipped instead. If n is the first eigenvalue to be clipped, and there are N eigenvalues, sorted in descending order:

$$(\lambda)_{clipped} = (\lambda_n, \lambda_{n+1}, ..., \lambda_N)$$

They now get a revised eigenvalue of E(lambda_clipped). Checks are in place within _eigenvalue_clip to ensure the threshold is suitable.

The threshold itself is subjective, but they are based on on making intelligent guesses what may be good enough to perturb the matrix to make it positive (semi) definite. This approach would always give an acceptable and closest stable matrix ([Higham_blog_nearest]) as long as the thresholds are set sensibly.

The automatic method (laloux_clip) (sort-of) does it for you, but it only works after the covariance is standardised (or the diagonal of the covariance matrix is constant). It is also an aggressive method which is based on noisiness and probability distribution of eigenvalues. Noting that closeness to the original matrix is our primary goal, not the noisiness of eigenvalues.

- [Jolliffe] recommends 70-90% for truncation
- Graphical guidances (“spectrum of eigenvalues”) are helpful ([Wilks], [Laloux])

Noting that [Wilks] and [Jolliffe] mostly concern with TRUNCATION not clipping. However, clipping is essentially a modified version of truncation but with the explained variance fully conserved, nor we cannot use covariances based on truncated EOFs; it will have 0 eigenvalues, many of them will become negative after floating point errors.

95% is somewhat higher than [Jolliffe] 70-90% guidance. Operational experience with 5x5 monthly data indicates even at 95%, you are only retaining 200-400 eigenvalues out of 36 x 72 = 2592 possible eigenvalues (if data is global). There are only ~10-20 negative eigenvalues plus the 2000-ish eigenvalues that sits out of 95% threshold. The magitude of the negative values are 2-plus order of mangitude smaller than the largest positive eigenvalues.

Parameters

- **`cov`** (*`numpy.ndarray`*) – Input covariance matrix to be adjusted to positive definite.
- **`target_variance_fraction`** (*`float`*) – Starting from the largest eigenvalue and descending, all eigenvalues larger than (the trace of cov x target_variance_fraction) are left unmodified. Eigenvalues beyond are modified.

Returns

Adjusted covariance matrix.

Return type

numpy.ndarray

References

- [Laloux]
- [Jolliffe]
- [Wilks]

See also

glomar_gridding.covariance_tools.laloux_clip()

`glomar_gridding.covariance_tools.laloux_clip`(*cov*, *num_grid_pts=None*, *num_time_pts=40*)

Estimate the largest eigenvalue that one will get from covariance or correlation matrices that are generated by random uncorrelated matrices aka the `noise` level of the eigenvalues.

Eq 0.3 in [Laloux] says eigenvalues of covariances generated by uncorrelated random vectors with constant variance have a max of:

$$\lambda_{max} = \sigma^2(1 + Q + 2\sqrt{Q})$$

in which:

$$Q = \frac{\text{num of features}}{\text{length of each feature}}$$

One can make a simple Monte-Carlo simulation to check that (a few lines of Python code will do). With variable (but realistic observed) variance, equation above is not exactly correct. However, Monte-Carlo simulations of λ_{max} show differences are not large; it becomes a somewhat larger, determined by the larger variances of the variable variances, but the estimate is correct up to the first significant digit.

Hence λ_{max} can be thought of a floor for non-noise EOFs, as this is the largest eigenvalue that can be generated by random uncorrelated data, and is only a function of Q (aka aspect ratio of the number of features with the length of each feature vector).

Any eigenvalues smaller than λ_{max} (including negative ones) are essentially noise and can possibly be trimmed.

This approach is aggressive. For climate sciences, Q can be large; for example if number of features are on the order of 1000-2000 (a global 5x5 grid land-only, sea-only, or of any terrain (up to 2592)) with 30-40 data points per feature (one value per year), the threshold can be reached within first 3-5 EOFs. Q would only worsen if number of features increases while number of values per feature decreases (aka higher resolution remote sensing data).

The explained variance approach (default), even if set to a generous value (say 70%; see [Wilks] chapter 11) would still keep a lot more EOFs. Noting that just because some (many) EOFs are noisy or degenerate, they are still part of the original covariance; they are actual bits of our best estimate to the data. It is desirable to minimise the perturbation applied to the covariance, keeping it as close as it is original; noise and degeneracy may be interesting by their own right (say in the physical interpret the EOFs – a contentious issue), but that is not our primary concern here. See [North] Section 5 and [Higham_blog_nearest] for further discussion.

Above requires q and sigma be the same for all features/independent variables. A way to get to around that is standardise the covariance into correlation and to apply the trimming to the correlation. The trimmed correlation can then converted back to the covariance by putting the correct standard deviations back into the diagonal. This effectively forces/averages the variance to constant.

Parameters

- **cov** (*numpy.ndarray*) – Input covariance matrix to be adjusted to positive definite.
- **num_grid_pts** (*int | None*) – Number of spatial grid points for covariance, this should usually be the shape of the covariance matrix. If unset, this will default to the size of the input covariance matrix (*cov.shape[0]*).
- **num_time_pts** (*int*) – Length of the time series that is behind the covariance generation Default 40 (aka 40 Jans from 1981-2020), since this is what used originally when covariances are generated for modern satellite era data (early 1980-ish to 2020-ish). It is important this value to be set correctly.

Returns

Adjusted covariance matrix.

Return type

numpy.ndarray

References

- [Laloux]
- [Bun]

See also

glomar_gridding.covariance_tools.explained_variance_clip()

`glomar_gridding.covariance_tools.`**`perturb_cov_to_positive_definite`**(*cov*, *threshold=1e-15*)

Force an estimated covariance matrix to be positive definite using the eigenvalue clipping with statsmodels.stats.correlation_tools.cov_nearest function.

Deprecated in favour of *glomar_gridding.covariance_tools.simple_clipping()*.

Parameters

- **cov** (*numpy.ndarray*) – The estimated covariance matrix that is not positive definite.
- **threshold** (*float | 'auto'*) – Eigenvalues below this value are set to 0. If the input is 'auto' then the value is determined using the floating-point precision and magnitude of the largest eigenvalues.

Returns

cov_adj – Adjusted covariance matrix

Return type

numpy.ndarray

See also

`glomar_gridding.covariance_tool.simple_clipping()`

Notes

Other methods:

- https://nhigham.com/2021/02/16/diagonally-perturbing-a-symmetric-matrix-to-make-it-positive-definite/
- https://nhigham.com/2013/02/13/the-nearest-correlation-matrix/
- https://academic.oup.com/imajna/article/22/3/329/708688

`glomar_gridding.covariance_tools.`**`simple_clipping`**`(`*cov*, *threshold='auto'*, *method='iterative'*`)`

A modified version of: https://www.statsmodels.org/dev/generated/statsmodels.stats.correlation_tools.corr_nearest.html

Force an estimated covariance matrix to be positive definite using the eigenvalue clipping with statsmodels.stats.correlation_tools.cov_nearest function.

This is appropriate for covariance matrices which have less dominant modes; it also preserves more noise.

Trace (aka total variance) of the covariance matrix is not conserved, but it is less disruptive than EOF chop off.

Parameters

- **`cov`** (*`numpy.ndarray`*) – The estimated covariance matrix that is not positive definite.
- **`threshold`** (*`float | 'auto'`*) – Eigenvalues below this value are set to 0. If the input is ‘auto’ then the value is determined using the floating-point precision and magnitude of the largest eigenvalues.

Return type

`tuple[ndarray, dict[str, Any]]`

Returns

- **cov_adj** (*numpy.ndarray*) – Adjusted covariance matrix
- **summary_dict** (*dict[str, Any]*) – A dictionary containing a summary of the input and results with the following keys:
  - ”threshold”
  - ”smallest_eigv”
  - ”determinant”
  - ”total_variance”

See also

`statsmodels.stats.correlation_tools.cov_nearest()`

Notes

Other methods:

- https://nhigham.com/2021/02/16/diagonally-perturbing-a-symmetric-matrix-to-make-it-positive-definite/
- https://nhigham.com/2013/02/13/the-nearest-correlation-matrix/
- https://academic.oup.com/imajna/article/22/3/329/708688

10.5 IO

glomar_gridding includes functionality for loading datasets or arrays from *netCDF* files using python format strings. This can be useful for loading pre-computed inputs for the Kriging process, for example covariance matrices or observations. The allowance for passing a string containing format components (e.g. python t-string) allows for dynamic configuration if processing a series of monthly inputs for example.

Also included is a function for recursively getting sub-keys from a python *dict* style object. This can be useful for working with *yaml* formatting configuration files for instance.

`glomar_gridding.io.`**`get_recurse`**(*config*, **keys*, *default=None*)

Recursively get sub keys from a python dict object.

If a dictionary object contains keys whose values are themselves dictionaries, get a value from a sub dictionary by specifying the key-path to get to the desired value.

Equivalent to:

```
config[keys[0]][keys[1]]...[keys[n]]
```

Or:

```
config.get(keys[0]).get(keys[1])...get(keys[n])
```

Parameters

- **`config`** (*`dict`*) – The layered dictionary containing sub dictionaries.
- **`*keys`** – The sequence of keys to recurse through to get the final value. If any key in the sequence is not found, or is not a dictionary (and is not the final key), then the default value is returned.
- **`default`** (*`Any`*) – The default value, returned if the sequence of keys cannot be completed or the final key is not present.

Return type

`Any`

Returns

- *The value associated with the final key, or the default value if the final*
- *key cannot be reached.*

`glomar_gridding.io.`**`load_array`**(*path*, *var='covariance'*, ***kwargs*)

Load an xarray.DataArray from a netCDF file. Can input a filename or a string to format with keyword arguments.

Parameters

- **`path`** (*`str`*) – Full filename (including path), or filename with replacements using str.format with named replacements. For example:

  /path/to/global_covariance_{month:02d}.nc

- **`var`** (*`str`*) – Name of the variable to select from the input file
- **`**kwargs`** – Keywords arguments matching the replacements in the input path.

Returns

arr – An array containing the values of the variable specified by var

Return type
xarray.DataArray

`glomar_gridding.io.load_dataset`(*path*, ***kwargs*)

Load an xarray.Dataset from a netCDF file. Can input a filename or a string to format with keyword arguments.

Parameters

- **path** (*str*) – Full filename (including path), or filename with replacements using str.format with named replacements. For example:

  /path/to/global_covariance_{month:02d}.nc

- ****kwargs** – Keywords arguments matching the replacements in the input path.

Returns
arr – The netcdf dataset as an xarray.Dataset.

Return type
xarray.Dataset

10.6 Utilities

Utility functions for *glomar_gridding*

exception `glomar_gridding.utils.ColumnNotFoundError`

Error class for Column Not Being Found

class `glomar_gridding.utils.MonthName`(**values*)

Name of month from int

`glomar_gridding.utils.add_empty_layers`(*nc_variables*, *timestamps*, *shape*)

Add empty layers to a netcdf file. This adds a layer of zeros to the netCDF file.

Parameters

- **nc_variables** (*Iterable[netcdf.Variable] | netcdf.Variable*) – Name(s) of the variables to add empty layers to
- **timestamps** (*Iterable[int] | int*) – Indices to add empty layers
- **shape** (*tuple[int, int]*) – Shape of the layer to add

Return type
None

`glomar_gridding.utils.adjust_small_negative`(*mat*, *atol=1e-08*)

Adjusts small negative values below an absolute tolerance value in a matrix to 0.

Raises a warning if any small negative values are detected.

Parameters

- **mat** (*numpy.ndarray[float]*) – Squared uncertainty associated with chosen kriging method Derived from the diagonal of the matrix
- **atol** (*float, default = 1e-8*) – Absolute tolerance value.

Returns
With negatice values below an absolute tolerance replaced with 0.

Return type
numpy.ndarray

Examples

```
>>> arr = np.array([[1, -1e-10], [-1e-10, 1]])
>>> adjust_small_negative(arr, atol=1e-8)
array([[1., 0.],
       [0., 1.]])
```

`glomar_gridding.utils.`**`batched`**(*iterable*, *n*, ** (Keyword-only parameters separator (PEP 3102))*, *strict=False*)

Implementation of itertools.batched for use if python version is < 3.12.

Examples

```
>>> list(batched("ABCDEFG", 3))
[("A", "B", "C"), ("D", "E", "F"), ("G", )]
```

`glomar_gridding.utils.`**`check_cols`**(*df*, *cols*)

Check that all columns in a list of columns are in a DataFrame.

Parameters

- **df** (*polars.DataFrame*)
- **cols** (*list[str]*) – List of columns to check for in *df*

Raises

ColumnNotFoundError – If any columns in *cols* are not present in *df*. The missing columns are displayed in the error message.

Return type

None

`glomar_gridding.utils.`**`cor_2_cov`**(*cor*, *variances*, *rounding=None*)

Compute covariance matrix from correlation matrix and variances

Parameters

- **cor** (*numpy.ndarray*) – Correlation Matrix
- **variances** (*numpy.ndarray*) – Variances to scale the correlation matrix.
- **rounding** (*int*) – round the values of the output

Return type

ndarray

`glomar_gridding.utils.`**`cov_2_cor`**(*cov*, *rounding=None*)

Normalises the covariance matrices within the class instance and return correlation matrices https://gist.github.com/wiso/ce2a9919ded228838703c1c7c7dad13b

Parameters

- **cov** (*numpy.ndarray*) – Covariance matrix
- **rounding** (*int*) – round the values of the output

Return type

ndarray

`glomar_gridding.utils.`**`days_since_by_month`**(*year*, *day*)

Get the number of days since *year*-01-*day* for each month. This is used to set the time values in a netCDF file where temporal resolution is monthly and the units are days since some date.

Parameters

- **year** (*int*) – Get a value for each month in this year.
- **day** (*int*) – Day of month for each returned datetime value in the sequence.

Returns

Containing 12 values, one for each month in the year containing the number of days since *year*-01-*day*.

Return type

numpy.ndarray

Examples

```
>>> days_since_by_month(1988, 14)
array([  0,  31,  60,  91, 121, 152, 182, 213, 244, 274, 305, 335])
```

glomar_gridding.utils.**deg_to_km**(*deg*)

Convert degree latitude change to km

Parameters

deg (*float*) – The difference in latitude in degrees

Returns

The latitude difference in kilometers

Return type

float

glomar_gridding.utils.**deg_to_nm**(*deg*)

Convert degree latitude change to nautical miles

Parameters

deg (*float*) – The difference in latitude in degrees

Returns

The latitude difference in nautical miles

Return type

float

glomar_gridding.utils.**filter_bounds**(*df*, *bounds*, *bound_cols*, *closed='left'*)

Filter a polars DataFrame based on a set of lower and upper bounds.

Parameters

- **df** (*polars.DataFrame*) – The data to be filtered by the bounds
- **bounds** (*list[tuple[float, float]]*) – A list of tuples containing lower and upper bounds for a column
- **bound_cols** (*list[str]*) – A list of column names to be filtered by the bounds, the length of the bounds list must equal the length of the bound_cols list.
- **closed** (*str | list[str]*) – One of "both", "left", "right", "none" indicating the closedness of the bounds. If the input is a single instance then all bounds will have that closedness. If it is a list of closed values then its length must match the length of the bounds list.

Returns

DataFrame filtered by the positional bounds

Return type
polars.DataFrame

`glomar_gridding.utils.`**`find_nearest`**`(`*array*, *values*`)`

Get the indices and values from an array that are closest to the input values.

A single index, value pair is returned for each look-up value in the values list.

Parameters

- **array** (*numpy.ndarray*) – The array to search for nearest values.
- **values** (*numpy.ndarray*) – The values to look-up in the array.

Return type
tuple[list[int], ndarray]

Returns

- **idx_list** (*list[int]*) – The indices of nearest values
- **array_values_list** (*list*) – The list of values in array that are closest to the input values.

Examples

```
>>> array = [1.0, 2.5, 2.7, 2.1, 4.5]
>>> tests = [1.1, 4.4, 2.2]
>>> find_nearest(array, tests)
([np.int64(0), np.int64(4), np.int64(3)], array([1. , 4.5, 2.1]))
```

`glomar_gridding.utils.`**`get_date_index`**`(`*year*, *month*, *start_year*`)`

Get the index of a given year-month in a monthly sequence of dates starting from month 1 in a specific start year

Parameters

- **year** (*int*) – The year for the date to find the index of.
- **month** (*int*) – The month for the date to find the index of.
- **start_year** (*int*) – The start year of the date series, the result assumes that the date time series starts in the first month of this year.

Returns
index – The index of the input date in the monthly datetime series starting from the first month of year *start_year*.

Return type
int

Examples

```
>>> get_date_index(2009, 14, start_year=1988)
265
```

`glomar_gridding.utils.`**`get_month_midpoint`**`(`*dates*`)`

Get the month midpoint for a series of datetimes.

The midpoint of a month is the exact half-way point between the start and end of the month.

For example, the midpoint of January 1990 is 1990-01-16 12:00.

Return type
`Series`

`glomar_gridding.utils.`**`get_pentad_range`**`(`*centre_date*`)`

Get the start and date of a pentad centred at a centre date. If the pentad includes the leap date of 29th Feb then the pentad will include 6 days. This follows the * pentad convention.

The start and end date are first calculated from a non-leap year.

If the centre date value is 29th Feb then the pentad will be a pentad starting on 27th Feb and ending on 2nd March.

Parameters
`centre_date` (*`datetime.date`*) – The centre date of the pentad. The start date will be 2 days before this date, and the end date will be 2 days after.

Return type
`tuple`[`date`, `date`]

Returns

- **start_date** (*datetime.date*) – Two days before centre_date
- **end_date** (*datetime.date*) – Two days after centre_date

Examples

```
>>> get_pentad_range(date(2008, 2, 29))
(datetime.date(2008, 2, 27), datetime.date(2008, 3, 2))
```

`glomar_gridding.utils.`**`get_spatial_mean`**`(`*grid_obs*, *covx*`)`

Compute the spatial mean accounting for auto-correlation. See [Cornell]

Parameters

- **`grid_obs`** (*`numpy.ndarray`*) – Vector containing observations
- **`covx`** (*`numpy.ndarray`*) – Observation covariance matrix

Returns
spatial_mean – The spatial mean defined as (1^T x C^{-1} x 1)^{-1} * (1^T x C^{-1} x z)

Return type
float

References

[Cornell] https://www.css.cornell.edu/faculty/dgr2/_static/files/distance_ed_geostats/ov5.pdf

`glomar_gridding.utils.`**`init_logging`**`(`*file=None*, *level='DEBUG'*`)`

Initialise the logger

Parameters

- **`file`** (*`str`*) – File to send log messages to. If set to None (default) then print log messages to STDout
- **`level`** (*`str`*) – Level of logging, one of: “debug”, “info”, “warn”, “error”, “critical”.

Return type
`None`

`glomar_gridding.utils.`**`intersect_mtlb`**`(`*a*, *b*`)`

Returns data common between two arrays, a and b, in a sorted order and index vectors for a and b arrays Reproduces behaviour of Matlab's intersect function.

Parameters

- **a** (*numpy.ndarray*)
- **b** (*numpy.ndarray*)

Returns

- Intersection
- List of indices, where the common values are located, for array a
- List of indices, where the common values are located, for array b

Return type

tuple[numpy.ndarray]

Examples

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([1, 1, 2, 5, 6])
>>> intersect_mtlb(a, b)
(array([1, 2]), array([0, 1]), array([0, 2]))
```

`glomar_gridding.utils.`**`is_iter`**`(`*val*`)`

Determine if a value is an iterable

Return type

`bool`

`glomar_gridding.utils.`**`km_to_deg`**`(`*km*`)`

Convert meridonal km change to degree latitude

Parameters

km (*float*) – The meridonal difference in kilometers

Returns

The meridonal difference in degrees

Return type

float

`glomar_gridding.utils.`**`mask_array`**`(`*arr*`)`

Forces numpy array to be an instance of np.ma.MaskedArray

Parameters

arr (*np.ndarray*) – Can be masked or not masked

Returns

arr – array is now an instance of np.ma.MaskedArray

Return type

np.ndarray

`glomar_gridding.utils.`**`select_bounds`**`(`*x*, *bounds=[(-90, 90), (-180, 180)]*, *variables=['lat', 'lon']*`)`

Filter an xarray.DataArray or xarray.Dataset by a set of bounds.

Parameters

- **x** (*xarray.DataArray | xarray.Dataset*) – The data to filter
- **bounds** (*list[tuple[float, float]]*) – A list of tuples containing the lower and upper bounds for each dimension.
- **variables** (*list[str]*) – Names of the dimensions (the order must match the bounds).

Returns
x – The input data filtered by the bounds.

Return type
xarray.DataArray | xarray.Dataset

glomar_gridding.utils.**sizeof_fmt**(*num*, *suffix='B'*)

Convert numbers to kilo/mega… bytes, for interactive printing of code progress.

Parameters

- **num** (*float*) – The number (typically of bytes) to format
- **suffix** (*str*)

Returns
The formatted number using power of 1024 base.

Return type
str

Examples

```
>>> sizeof_fmt(123456789)
'117.7MiB'
```

glomar_gridding.utils.**uncompress_masked**(*compressed_array*, *mask*, *fill_value=0.0*, *apply_mask=False*, *dtype=None*)

Un-compress a compressed array using a mask.

Parameters

- **compressed_array** (*numpy.ndarray*) – The compressed array, originally compressed by the mask
- **mask** (*numpy.ndarray*) – The mask - a boolean numpy array
- **fill_value** (*Any*) – The value to fill masked points. If *apply_mask* is set, then this will be removed in the output.
- **apply_mask** (*bool*) – Apply the mask to the result, returning a MaskedArray rather than a ndarray.
- **dtype** (*type | None*) – Optionally set a dtype for the returned array, if not set then the dtype of the compressed_array is used.

Returns
uncompressed – The uncompressed array, masked points are filled with the fill_value if apply_mask is False. If apply_mask is True, then the result is an instance of numpy.ma.MaskedArray with the mask applied to the uncompressed result.

Return type
numpy.ndarray | numpy.ma.MaskedArray

Examples

```
>>> arr = np.random.rand(16)
>>> mask = arr > 0.65
>>> arr = np.ma.masked_where(mask, arr).compressed()
>>> uncompress_masked(arr, mask, fill_value=-999.0)
array([ 2.79245414e-01, -9.99000000e+02,  3.93541024e-01, -9.99000000e+02,
        8.07814120e-03,  3.34164220e-01, -9.99000000e+02,  2.08200564e-01,
        3.32044850e-01,  1.83166093e-01, -9.99000000e+02,  2.57339943e-02,
        1.76017461e-01,  3.56673893e-01,  1.59393168e-01,  2.17047382e-01])
```

BIBLIOGRAPHY

[Bun] Bun, J., Bouchaud, J.-P., and Potters, M: Cleaning large correlation matrices: Tools from Random Matrix Theory, Physics Reports, Volume 666, 2017, Pages 1-109, ISSN 0370-1573, https://doi.org/10.1016/j.physrep.2016.10.005.

[Cornell] https://www.css.cornell.edu/faculty/dgr2/_static/files/distance_ed_geostats/ov5.pdf

[Cressie] Cressie, N. A. C.: Statistics for Spatial Data, Wiley Series in Probability and Statistics, Wiley, 1 edn., ISBN 978-0-471-00255-0 978-1-119-11515-1, https://doi.org/10.1002/9781119115151, 1993.

[Higham] Higham, N. J., Strabi´c, N., and Šego, V.: Restoring Definiteness via Shrinking, with an Application to Correlation Matrices with a Fixed255 Block, SIAM Review, 58, 245–263, https://www.jstor.org/stable/24778894, 2016.

[Higham_blog_nearest] https://nhigham.com/2021/01/26/what-is-the-nearest-positive-semidefinite-matrix/

[Huang] Huang, B., Yin, X., Menne, M. J., Vose, R., and Zhang, H.-M.: Improvements to the Land Surface Air Temperature Reconstruction in NOAAGlobalTemp: An Artificial Neural Network Approach, Artificial Intelligence for the Earth Systems, 1, e220 032, https://doi.org/10.1175/AIES-D-22-0032.1, 2022.

[IPCC] Intergovernmental Panel On Climate Change: Climate Change 2021 – The Physical Science Basis: Working Group I Contribution to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change, Cambridge University Press, 1 edn., ISBN 978-1-00-915789-6, https://doi.org/10.1017/9781009157896, 2021.

[Jolliffe] Jolliffe, I. T., Principal Component Analysis, pp 487, Springer, 2002

[Kadow] Kadow, C., Hall, D. M., and Ulbrich, U.: Artificial intelligence reconstructs missing climate information, Nature Geoscience, 13, 408–413, https://doi.org/10.1038/s41561-020-0582-5, 2020

[Karspeck] Karspeck, A. R., Kaplan, A., and Sain, S. R.: Bayesian modelling and ensemble reconstruction of mid-scale spatial variability in North Atlantic sea-surface temperatures for 1850-2008, Quarterly Journal of the Royal Meteorological Society, 138, 234–248, https://doi.org/10.1002/qj.900, 2012.

[Laloux] Laloux, L., Cizeau, P., Potters, M. and Bouchaud, J.P.: Random matrix theory and financial correlations. International Journal of Theoretical and Applied Finance, 3(03), 391-397. https://www.worldscientific.com/doi/abs/10.1142/S0219024900000255, 2000

[Lenssen] Lenssen, N. J. L., Schmidt, G. A., Hansen, J. E., Menne, M. J., Persin, A., Ruedy, R., and Zyss, D.: Improvements in the GISTEMP Uncertainty Model, Journal of Geophysical Research: Atmospheres, 124, 6307–6326, https://doi.org/10.1029/2018JD029522, 2019.

[Morice_2012] Morice, C. P., Kennedy, J. J., Rayner, N. A., and Jones, P. D.: Quantifying uncertainties in global and regional temperature change using an ensemble of observational estimates: The HadCRUT4 data set: THE HADCRUT4 DATASET, Journal of Geophysical Research: Atmospheres, 117, n/a–n/a, https://doi.org/10.1029/2011JD017187, 2012

[Morice_2021] Morice, C. P., Kennedy, J. J., Rayner, N. A., Winn, J. P., Hogan, E., Killick, R. E., Dunn, R. J. H., Osborn, T. J., Jones, P. D., and Simpson, 285 I. R.: An Updated Assessment of Near-Surface Temperature Change From 1850: The HadCRUT5 Data Set, Journal of Geophysical Research: Atmospheres, 126, e2019JD032 361, https://doi.org/10.1029/2019JD032361, 2021.

[North] North, G. R., Bell, T. L., Cahalan, R. F., and Moeng, F. J.: Sampling Errors in the Estimation of Empirical Orthogonal Functions. Monthly Weather Review, 110(7), 699–706. https://doi.org/10.1175/1520-0493(1982)110<0699:SEITEO>2.0.CO;2, 1982.

[PaciorekSchervish] Paciorek, C. J. and Schervish, M. J.: Spatial modelling using a new class of nonstationary covariance functions, Environmetrics, 17, 483–506, https://doi.org/10.1002/env.785, 2006

[Rasmussen] Rasmussen, C. E. and Williams, C. K. I.: Gaussian Processes for Machine Learning, The MIT Press, ISBN 978-0-262-25683-4, https://doi.org/10.7551/mitpress/3206.001.0001, 2005.

[RohdeBerkeley] Rohde, R. A. and Hausfather, Z.: The Berkeley Earth Land/Ocean Temperature Record, Earth System Science Data, 12, 3469–3479, https://doi.org/10.5194/essd-12-3469-2020, 2020

[Thorne] Thorne, P. W., D. E. Parker, J. R. Christy, and C. A. Mears: Uncertainties in climate trends: Lessons from upper-air temperature records, Bull. Am. Meteorol. Soc., 86, 1437–1442, https://doi.org/10.1175/BAMS-86-10-1437, 2005

[Wilks] Wilks, D.: Statistical Methods in Atmospheric Sciences, 2nd Edition, Elsevier, 2006

PYTHON MODULE INDEX

INDEX