GloMarGridding

Release 0.2.0

NOC Surface Processes

Mar 10, 2025

CONTENTS:

CHAPTER

ONE

INTRODUCTION

CHAPTER TWO

GETTING STARTED

2.1 Installation

2.1.1 Via Pip

GloMarGridding is not available on PyPI, however it can be installed via pip with the following command:

```
pip install GloMarGridding@git+ssh://git@git.noc.ac.uk/nocsurfaceprocesses/glomar_
↪gridding.git
```

2.1.2 From Source

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

```
git clone git@git.noc.ac.uk/nocsurfaceprocesses/glomar_gridding.git
cd glomar_gridding
python -m venv venv
source venv/bin/activate
pip install -e .
```

CHAPTER THREE

CREDITS

3.1 Development Lead

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

3.2 Contributoring Developers

- Steven Chan <stchan@noc.ac.uk> @stchan
- Richard C. Cornes <rcornes@noc.ac.uk> @ricorne
- Elizabeth C. Kent <eck@noc.ac.uk> @eck

CHAPTER

FOUR

USERS GUIDE

4.1 Grid

Functions for creating grids and mapping observations to a grid

`glomar_gridding.grid.`**`assign_to_grid`**`(`*values*, *grid_idx*, *grid*, *mask_grid=False*, *mask_value=nan*`)`

Assign a vector of values to a grid, using a list of grid index values. The default value for grid values is 0.0.

Optionally, if the grid is a mask, apply the mask to the output grid.

Parameters

- **`values`** (*`pl.Series`*) – The values to map onto the output grid.
- **`grid_idx`** (*`pl.Series`*) – The 1d index of the grid (assuming "C" style ravelling) for each value.
- **`grid`** (*`xarray.DataArray`*) – The grid used to define the output grid.
- **`mask_grid`** (*`bool`*) – Optionally use values in the grid to mask the output grid.
- **`mask_value`** (*`Any`*) – The value in the grid to use for masking the output grid.

Returns

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

Return type

xarray.DataArray

`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.

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
- **`coord_names`** (*`list[str]`*) – List of coordinate names

Returns

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

Return type

xarray.DataArray:

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

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.

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

`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

4.2 Variograms

Varigram classes for construction of spatial covariance structure from distance matrices.

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

Exponential Model

Parameters

- **`psill`** (*`float | numpy.ndarray`*) – The variance of the variogram.
- **`nugget`** (*`float | numpy.ndarray`*)
- **`effective_range`** (*`float | numpy.ndarray | None`*)
- **`range`** (*`float | numpy.ndarray | None`*)

`fit``(`*distance_matrix*`)`

Fit the ExponentialVariogram model to a distance matrix

Return type

`ndarray | DataArray`

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

Gaussian Model

Parameters

- **`psill`** (*`float | np.ndarray`*) – The variance of the variogram.
- **`nugget`** (*`float | np.ndarray`*)
- **`effective_range`** (*`float | np.ndarray | None`*)
- **`range`** (*`float | np.ndarray | None`*)

`fit``(`*distance_matrix*`)`

Fit the GaussianVariogram model to a distance matrix

Return type

`ndarray | DataArray`

`class glomar_gridding.variogram.`**`LinearVariogram`**`(`*slope*, *nugget*`)`

Linear model

Parameters

- **`slope`** (*`float | np.ndarray`*)
- **`nugget`** (*`float | np.ndarray`*)

`fit``(`*distance_matrix*`)`

Fit the LinearVariogram model to a distance matrix

Return type

`ndarray | DataArray`

`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, 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.

Reference; 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

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

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*) – 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

Return type

`ndarray | DataArray`

`class glomar_gridding.variogram.`**`PowerVariogram`**(*scale*, *exponent*, *nugget*)

Power model

Parameters

- **scale** (*float | np.ndarray*)
- **exponent** (*float | np.ndarray*)
- **nugget** (*float | np.ndarray*)

`fit`(*distance_matrix*)

Fit the PowerVariogram model to a distance matrix

Return type
`ndarray | DataArray`

`class glomar_gridding.variogram.`**`Variogram`**

Generic Variogram Class - defines the abstract class

`abstractmethod` **`fit`**(*distance_matrix*)

Fit the Variogram model to a distance matrix

Return type
`ndarray | DataArray`

`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

4.3 Kriging

Functions for performing Kriging.

Interpolation using a Gaussian Process. Available methods are Simple and Ordinary Kriging.

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

Compute the spatial mean accounting for auto-correlation.

Parameters

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

Return type
`float`

Returns

- **spatial_mean** (*float*) – The spatial mean defined as (1^T x C^{-1} x 1)^{-1} * (1^T x C^{-1} x z)
- *Reference*
- ———
- **https** (*//www.css.cornell.edu/faculty/dgr2/_static/files/distance_ed_geostats/ov5.pdf*)

`glomar_gridding.kriging.`**`get_unmasked_obs_indices`**(*unmask_idx*, *unique_obs_idx*)

Get grid indices with observations from un-masked grid-box indices and unique grid-box indices with observations.

Parameters

- **`unmask_idx`** (*`np.ndarray[int]`*) – List of all unmasked grid-box indices.
- **`unique_obs_idx`** (*`np.ndarray[int]`*) – Indices of grid-boxes with observations.

Returns
obs_idx – Subset of grid-box indices containing observations that are unmasked.

Return type
np.ndarray[int]

`glomar_gridding.kriging.`**`kriging`**(*obs_idx*, *weights*, *obs*, *interp_cov*, *error_cov*, *remove_obs_mean=0*, *obs_bias=None*, *method='simple'*)

Perform Kriging using a chosen method.

Get array of krigged observations and anomalies for all grid points in the domain.

Parameters

- **`obs_idx`** (*`np.ndarray[int]`*) – Grid indices with observations. It is expected that this should be an ordering that lines up with the 1st dimension of weights. If *observations.dist_weights* or *observations.get_weights* was used to get the weights then this is the ordering of *sorted(df[“gridbox”].unique())*, which is a sorting on lat and lon
- **`weights`** (*`np.ndarray[float]`*) – Weight matrix (inverse of counts of observations).
- **`obs`** (*`np.ndarray[float]`*) – All point observations/measurements for the chosen date.
- **`interp_cov`** (*`np.ndarray[float]`*) – interpolation covariance of all output grid points (each point in time and all points against each other).
- **`error_cov`** (*`np.ndarray[float]`*) – Measurement/Error covariance matrix.
- **`remove_obs_mean`** (*`int`*) – Should the mean or median from grib_obs be removed and added back onto grib_obs? 0 = No (default action) 1 = the mean is removed 2 = the median is removed 3 = the spatial meam os removed
- **`obs_bias`** (*`np.ndarray[float] | None`*) – Bias of all measurement points for a chosen date (corresponds to x_obs).
- **`method`** (*`KrigMethod`*) – The kriging method to use to fill in the output grid. One of “simple” or “ordinary”.

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 the chosen kriging method.
- **dz** (*np.ndarray[float]*) – Uncertainty associated with the chosen kriging method.

`glomar_gridding.kriging.`**`kriging_ordinary`**`(`*S*, *Ss*, *grid_obs*, *interp_cov*`)`

Perform Ordinary Kriging with unknown but constant mean.

Parameters

- **`S`** (*`np.ndarray[float]`*) – Spatial covariance between all measured grid points plus the covariance due to measurements (i.e. measurement noise, bias noise, and sampling noise).
- **`Ss`** (*`np.ndarray[float]`*) – Covariance between the all (predicted) grid points and measured points.
- **`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.

`glomar_gridding.kriging.`**`kriging_simple`**`(`*S*, *Ss*, *grid_obs*, *interp_cov*, *mean=0.0*`)`

Perform Simple Kriging assuming a constant known mean.

Parameters

- **`S`** (*`np.ndarray[float]`*) – Spatial covariance between all measured grid points plus the covariance due to measurements (i.e. measurement noise, bias noise, and sampling noise).
- **`Ss`** (*`np.ndarray[float]`*) – Covariance between the all (predicted) grid points and measured points.
- **`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.

`glomar_gridding.kriging.`**`unmasked_kriging`**`(`*unmask_idx*, *unique_obs_idx*, *weights*, *obs*, *interp_cov*, *error_cov*, *remove_obs_mean=0*, *obs_bias=None*, *method='simple'*`)`

Perform Kriging on a masked grid using a chosen method.

Get array of krigged observations and anomalies for all grid points in the domain.

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.
- **interp_cov** (*np.ndarray[float]*) – Interpolation covariance of all output grid points (each point in time and all points against each other).
- **error_cov** (*np.ndarray[float]*) – Measurement/Error covariance matrix.
- **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
- **obs_bias** (*np.ndarray[float] | None*) – Bias of all measurement points for a chosen date (corresponds to x_obs).
- **method** (*KrigMethod*) – The kriging method to use to fill in the output grid. One of "simple" or "ordinary".

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 the chosen kriging method.
- **dz** (*np.ndarray[float]*) – Uncertainty associated with the chosen kriging method.

4.4 Climatology

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

4.5 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.

`glomar_gridding.mask.`**`mask_array`**`(`*grid*, *mask*, *varname*, *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.
- **`varname`** (*`str`*) – Name of the variable in the observational DataArray 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.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

`glomar_gridding.mask.`**`mask_from_obs_frame`**(*obs*, *coords*, *datetime_col*, *value_col*)

Compute a mask from observations.

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.

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"].
- **datetime_col** (*str*) – Name of the datetime column. Any positions that contain no records at any datetime value are masked.
- **value_col** (*str*) – Name of the column containing values from which the mask will be defined.

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*, *mask_varname='mask'*, *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.
- **mask_varname** (*str*) – Name of the mask variable in the mask DataArray.
- **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

4.6 Distances

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>*, *lat_col='lat'*, *lon_col='lon'*`)`

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.

Returns
dist – A matrix of pairwise distances.

Return type
np.ndarray[float]

`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).

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

d = SQRT((x_2-x_1)**2 + (y_2-y_1)**2 + (z_2-z_1)**2)

where

(x_n y_n z_n) = (Rcos(lat)cos(lon) Rcos(lat)sin(lon) Rsin(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

`glomar_gridding.distances.`**`haversine_distance`**`(`*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.`**`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.`**`tau_dist`**`(`*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

4.7 Interpolation Covariance

Functions for computing (components of) the interpolation covariance matrix used for the interpolation step.

`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

4.8 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.
- ****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.*

Functions for helping with perturbations/random drawing

`glomar_gridding.perturbation.`**`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

4.9 Utils

Utility functions for GlomarGridding

`exception` `glomar_gridding.utils.`**`ColumnNotFoundError`**

Error class for Column Not Being Found

`class` `glomar_gridding.utils.`**`ConfigParserMultiValues`**

Internal Helper Class

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

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[nc.Variable] | nc.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*`)`

Adjusts small negative values (with absolute value < 1e-8) in matrix to 0 in-place.

Raises a warning if any small negative values are detected.

Parameters
`mat` (*`np.ndarray[float]`*) – Squared uncertainty associated with chosen kriging method Derived from the diagonal of the matrix

Return type
`ndarray`

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

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

Return type
`None`

`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.

Return type
`ndarray`

`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.

Return type
`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`** (*`Iterable`*) – The array to search for nearest values.
- **`values`** (*`Iterable`*) – 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.

`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

`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

`glomar_gridding.utils.`**`init_logging`**`(`*file*, *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

- **`array`** (*`b (array) - 1-D`*)

- **`array`**

Returns

- *1-D array, c, of common values found in two arrays, a and b, sorted in order*
- *List of indices, where the common values are located, for array a*
- *List of indices, where the common values are located, for array b*

`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

PYTHON MODULE INDEX

INDEX