# Contents

1 PyKrige .......................... 1
   1.1 Purpose .......................... 2
   1.2 Installation .......................... 2
   1.3 Features .......................... 2
      1.3.1 Kriging algorithms ..................... 2
      1.3.2 Wrappers .......................... 2
      1.3.3 Tools .......................... 2
      1.3.4 Kriging Parameters Tuning ............... 3
      1.3.5 Regression Kriging ..................... 3
   1.4 License .......................... 3

2 Variogram Models ............... 5
   2.1 References .......................... 6

3 API Reference .................. 7
   3.1 Krigging algorithms ..................... 7
      3.1.1 pykrige.ok.OrdinaryKriging .............. 7
      3.1.2 pykrige.uk.UniversalKriging .............. 12
      3.1.3 pykrige.ok3d.OrdinaryKriging3D ............. 17
      3.1.4 pykrige.uk3d.UniversalKriging3D ............. 23
      3.1.5 pykrige.rk.RegressionKriging .............. 28
   3.2 Wrappers .......................... 30
      3.2.1 pykrige.rk.Krige ..................... 30
   3.3 Tools .......................... 33
      3.3.1 pykrige.kriging_tools.write_asc_grid ....... 33
      3.3.2 pykrige.kriging_tools.read_asc_grid ....... 33

4 Examples .................... 35
   4.1 Universal Kriging Example ................. 35
   4.2 Ordinary Kriging Example ................. 36
   4.3 GSTools Interface .................... 38
   4.4 Exact Values .......................... 40
   4.5 Regression kriging .......................... 41
   4.6 Geometric example .................... 42
   4.7 Three-Dimensional Kriging Example ............ 45
   4.8 Krige CV .......................... 47
   4.9 1D Kriging .......................... 51
Kriging Toolkit for Python.
1.1 Purpose

The code supports 2D and 3D ordinary and universal kriging. Standard variogram models (linear, power, spherical, gaussian, exponential) are built in, but custom variogram models can also be used. The 2D universal kriging code currently supports regional-linear, point-logarithmic, and external drift terms, while the 3D universal kriging code supports a regional-linear drift term in all three spatial dimensions. Both universal kriging classes also support generic ‘specified’ and ‘functional’ drift capabilities. With the ‘specified’ drift capability, the user may manually specify the values of the drift(s) at each data point and all grid points. With the ‘functional’ drift capability, the user may provide callable function(s) of the spatial coordinates that define the drift(s). The package includes a module that contains functions that should be useful in working with ASCII grid files (*.asc).

See the documentation at http://pykrige.readthedocs.io/ for more details and examples.

1.2 Installation

PyKrige requires Python 3.5+ as well as numpy, scipy. It can be installed from PyPi with,

```bash
pip install pykrige
```

scikit-learn is an optional dependency needed for parameter tuning and regression kriging. matplotlib is an optional dependency needed for plotting.

If you use conda, PyKrige can be installed from the conda-forge channel with,

```bash
conda install -c conda-forge pykrige
```

1.3 Features

1.3.1 Kriging algorithms

- OrdinaryKriging: 2D ordinary kriging with estimated mean
- UniversalKriging: 2D universal kriging providing drift terms
- OrdinaryKriging3D: 3D ordinary kriging
- UniversalKriging3D: 3D universal kriging
- RegressionKriging: An implementation of Regression-Kriging

1.3.2 Wrappers

- rk.Krige: A scikit-learn wrapper class for Ordinary and Universal Kriging

1.3.3 Tools

- kriging_tools.write_asc_grid: Writes gridded data to ASCII grid file (*.asc)
- kriging_tools.read_asc_grid: Reads ASCII grid file (*.asc)
1.3.4 Kriging Parameters Tuning

A scikit-learn compatible API for parameter tuning by cross-validation is exposed in sklearn.model_selection.GridSearchCV. See the Krige CV example for a more practical illustration.

1.3.5 Regression Kriging

Regression kriging can be performed with pykrige.rk.RegressionKriging. This class takes as parameters a scikit-learn regression model, and details of either either the OrdinaryKriging or the UniversalKriging class, and performs a correction steps on the ML regression prediction.

A demonstration of the regression kriging is provided in the corresponding example.

1.4 License

PyKrige uses the BSD 3-Clause License.
PyKrige internally supports the six variogram models listed below. Additionally, the code supports user-defined variogram models via the 'custom' variogram model keyword argument.

- **Gaussian Model**
  \[ p \cdot (1 - e^{-\frac{d^2}{(4r)^2}}) + n \]

- **Exponential Model**
  \[ p \cdot (1 - e^{-\frac{d}{r/3}}) + n \]

- **Spherical Model**
  \[
  \begin{cases} 
  p \cdot \left( \frac{3d^2}{2r} - \frac{d^3}{2r^3} \right) + n & d \leq r \\
  p + n & d > r 
  \end{cases}
  \]

- **Linear Model**
  \[ s \cdot d + n \]

Where \( s \) is the slope and \( n \) is the nugget.

- **Power Model**
  \[ s \cdot d^e + n \]

Where \( s \) is the scaling factor, \( e \) is the exponent (between 0 and 2), and \( n \) is the nugget term.

- **Hole-Effect Model**
  \[ p \cdot \left(1 - \left(1 - \frac{d}{r/3}\right) e^{-\frac{d}{r/3}}\right) + n \]

Variables are defined as:
\( d \) = distance values at which to calculate the variogram
\( p \) = partial sill (psill = sill - nugget)
\( r \) = range
\[ n = \text{nugget} \]
\[ s = \text{scaling factor or slope} \]
\[ e = \text{exponent for power model} \]

For stationary variogram models (gaussian, exponential, spherical, and hole-effect models), the partial sill is defined as the difference between the full sill and the nugget term. The sill represents the asymptotic maximum spatial variance at longest lags (distances). The range represents the distance at which the spatial variance has reached ~95% of the sill variance. The nugget effectively takes up ‘noise’ in measurements. It represents the random deviations from an overall smooth spatial data trend. (The name nugget is an allusion to kriging’s mathematical origin in gold exploration; the nugget effect is intended to take into account the possibility that when sampling you randomly hit a pocket gold that is anomalously richer than the surrounding area.)

For nonstationary models (linear and power models, with unbounded spatial variances), the nugget has the same meaning. The exponent for the power-law model should be between 0 and 2\(^1\).

A few important notes:

The PyKrige user interface by default takes the full sill. This default behavior can be changed with a keyword flag, so that the user can supply the partial sill instead. The code internally uses the partial sill (psill = sill - nugget) rather than the full sill, as it’s safer to perform automatic variogram estimation using the partial sill.

The exact definitions of the variogram models here may differ from those used elsewhere. Keep that in mind when switching from another kriging code over to PyKrige.

According to\(^1\), the hole-effect variogram model is only correct for the 1D case. It’s implemented here for completeness and should be used cautiously.

### 2.1 References

### 3.1 Krigging algorithms

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pykrige.ok.OrdinaryKriging(x, y, z[, ...])</code></td>
<td>Convenience class for easy access to 2D Ordinary Kriging.</td>
</tr>
<tr>
<td><code>pykrige.uk.UniversalKriging(x, y, z[, ...])</code></td>
<td>Provides greater control over 2D kriging by utilizing drift terms.</td>
</tr>
<tr>
<td><code>pykrige.ok3d.OrdinaryKriging3D(x, y, z, val)</code></td>
<td>Three-dimensional ordinary kriging.</td>
</tr>
<tr>
<td><code>pykrige.uk3d.UniversalKriging3D(x, y, z, val)</code></td>
<td>Three-dimensional universal kriging.</td>
</tr>
<tr>
<td><code>pykrige.rk.RegressionKriging([...])</code></td>
<td>An implementation of Regression-Kriging.</td>
</tr>
</tbody>
</table>

#### 3.1.1 pykrige.ok.OrdinaryKriging

```python
class pykrige.ok.OrdinaryKriging(x, y, z, variogram_model='linear', variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_scaling=1.0, anisotropy_angle=0.0, enable_plotting=False, enable_statistics=False, coordinates_type='euclidean', exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv')
```

Convenience class for easy access to 2D Ordinary Kriging.

**Parameters**

- `x (array_like)` – X-coordinates of data points.
- `y (array_like)` – Y-coordinates of data points.
- `z (array-like)` – Values at data points.
- `variogram_model (str or GSTools CovModel, optional)` – Specifies which variogram model to use; may be one of the following: linear, power, gaussian,
spherical, exponential, hole-effect. Default is linear variogram model. To utilize a
custom variogram model, specify ‘custom’; you must also provide variogram_parameters
and variogram_function. Note that the hole-effect model is only technically correct for
one-dimensional problems. You can also use a GSTools CovModel.

• **variogram_parameters** *(list or dict, optional)* – Parameters that define
the specified variogram model. If not provided, parameters will be automatically calculated
using a “soft” L1 norm minimization scheme. For variogram model parameters provided in
a dict, the required dict keys vary according to the specified variogram model:

```python
# linear
{'slope': slope, 'nugget': nugget}
# power
{'scale': scale, 'exponent': exponent, 'nugget': nugget}
# gaussian, spherical, exponential and hole-effect:
{'sill': s, 'range': r, 'nugget': n}
# OR
{'psill': p, 'range': r, 'nugget': n}
```

Note that either the full sill or the partial sill (psill = sill - nugget) can be specified in the
dict. For variogram model parameters provided in a list, the entries must be as follows:

```python
# linear
[slope, nugget]
# power
[scope, exponent, nugget]
# gaussian, spherical, exponential and hole-effect:
[sill, range, nugget]
```

Note that the full sill (NOT the partial sill) must be specified in the list format. For a
custom variogram model, the parameters are required, as custom variogram models will not
automatically be fit to the data. Furthermore, the parameters must be specified in list format,
in the order in which they are used in the callable function (see variogram_function for more
information). The code does not check that the provided list contains the appropriate number
of parameters for the custom variogram model, so an incorrect parameter list in such a case
will probably trigger an esoteric exception someplace deep in the code. NOTE that, while
the list format expects the full sill, the code itself works internally with the partial sill.

• **variogram_function** *(callable, optional)* – A callable function that must be
provided if variogram_model is specified as ‘custom’. The function must take only two
arguments: first, a list of parameters for the variogram model; second, the distances at
which to calculate the variogram model. The list provided in variogram_parameters will be
passed to the function as the first argument.

• **nlags** *(int, optional)* – Number of averaging bins for the semivariogram. Default is
6.

• **weight** *(bool, optional)* – Flag that specifies if semivariance at smaller lags should
be weighted more heavily when automatically calculating variogram model. The routine
is currently hard-coded such that the weights are calculated from a logistic function, so
weights at small lags are ~1 and weights at the longest lags are ~0; the center of the logistic
weighting is hard-coded to be at 70% of the distance from the shortest lag to the largest
lag. Setting this parameter to True indicates that weights will be applied. Default is False.
(Kitanidis suggests that the values at smaller lags are more important in fitting a variogram
model, so the option is provided to enable such weighting.)

• **anisotropy_scaling** *(float, optional)* – Scalar stretching value to take into
account anisotropy. Default is 1 (effectively no stretching). Scaling is applied in the
y-direction in the rotated data frame (i.e., after adjusting for the anisotropy_angle, if anisotropy_angle is not 0). This parameter has no effect if coordinate_types is set to 'geographic'.

- **anisotropy_angle** *(float, optional) –* CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy. Default is 0 (no rotation). Note that the coordinate system is rotated. This parameter has no effect if coordinate_types is set to 'geographic'.

- **verbose** *(bool, optional) –* Enables program text output to monitor kriging process. Default is False (off).

- **enable_plotting** *(bool, optional) –* Enables plotting to display variogram. Default is False (off).

- **enable_statistics** *(bool, optional) –* Default is False

- **coordinates_type** *(str, optional) –* One of 'euclidean' or 'geographic'. Determines if the x and y coordinates are interpreted as on a plane ('euclidean') or as coordinates on a sphere ('geographic'). In case of geographic coordinates, x is interpreted as longitude and y as latitude coordinates, both given in degree. Longitudes are expected in [0, 360] and latitudes in [-90, 90]. Default is 'euclidean'.

- **exact_values** *(bool, optional) –* If True, interpolation provides input values at input locations. If False, interpolation accounts for variance/nugget within input values at input locations and does not behave as an exact-interpolator [2]. Note that this only has an effect if there is variance/nugget present within the input data since it is interpreted as measurement error. If the nugget is zero, the kriged field will behave as an exact interpolator.

- **pseudo_inv** *(bool, optional) –* Whether the kriging system is solved with the pseudo inverted kriging matrix. If True, this leads to more numerical stability and redundant points are averaged. But it can take more time. Default: False

- **pseudo_inv_type** *(str, optional) –* Here you can select the algorithm to compute the pseudo-inverse matrix:
  - "pinv": use pinv from scipy which uses lstsq
  - "pinv2": use pinv2 from scipy which uses SVD
  - "pinvh": use pinvh from scipy which uses eigen-values
  Default: “pinv”

References

```python
__init__(x, y, z, variogram_model='linear', variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_scaling=1.0, anisotropy_angle=0.0, verbose=False, enable_plotting=False, enable_statistics=False, coordinates_type='euclidean', exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv')
```
Initialize self. See help(type(self)) for accurate signature.

Methods

```python
__init__(x, y, z[, variogram_model, ...]) Initialize self.
```
Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>display_variogram_model()</code></td>
<td>Displays variogram model with the actual binned data.</td>
</tr>
<tr>
<td><code>execute(style, xpoints, ypoints[, mask, ...])</code></td>
<td>Calculates a kriged grid and the associated variance.</td>
</tr>
<tr>
<td><code>get_epsilon_residuals()</code></td>
<td>Returns the epsilon residuals for the variogram fit.</td>
</tr>
<tr>
<td><code>get_statistics()</code></td>
<td>Returns the Q1, Q2, and cR statistics for the variogram fit (in that order).</td>
</tr>
<tr>
<td><code>get_variogram_points()</code></td>
<td>Returns both the lags and the variogram function evaluated at each of them.</td>
</tr>
<tr>
<td><code>plot_epsilon_residuals()</code></td>
<td>Plots the epsilon residuals for the variogram fit.</td>
</tr>
<tr>
<td><code>print_statistics()</code></td>
<td>Prints out the Q1, Q2, and cR statistics for the variogram fit.</td>
</tr>
<tr>
<td><code>switch_plotting()</code></td>
<td>Allows user to switch plot display on/off.</td>
</tr>
<tr>
<td><code>switchVerbose()</code></td>
<td>Allows user to switch code talk-back on/off.</td>
</tr>
<tr>
<td><code>update_variogram_model(...)</code></td>
<td>Allows user to update variogram type and/or variogram model parameters.</td>
</tr>
</tbody>
</table>

**Attributes**

- `eps`
- `variogram_dict`

`display_variogram_model()`
Displays variogram model with the actual binned data.

`execute(style, xpoints, ypoints[, mask=None, backend='vectorized', n_closest_points=None])`
Calculates a kriged grid and the associated variance.

**Parameters**

- `style (str)` – Specifies how to treat input kriging points. Specifying ‘grid’ treats xpoints and ypoints as two arrays of x and y coordinates that define a rectangular grid. Specifying ‘points’ treats xpoints and ypoints as two arrays that provide coordinate pairs at which to solve the kriging system. Specifying ‘masked’ treats xpoints and ypoints as two arrays of x and y coordinates that define a rectangular grid and uses mask to only evaluate specific points in the grid.

- `xpoints (array_like, shape (N,) or (N, 1))` – If style is specific as ‘grid’ or ‘masked’, x-coordinates of MxN grid. If style is specified as ‘points’, x-coordinates of specific points at which to solve kriging system.

- `ypoints (array_like, shape (M,) or (M, 1))` – If style is specified as ‘grid’ or ‘masked’, y-coordinates of MxN grid. If style is specified as ‘points’, y-coordinates of specific points at which to solve kriging system. Note that in this case, xpoints and ypoints must have the same dimensions (i.e., M = N).

- `mask (bool, array_like, shape (M, N), optional) – Specifies the points in the rectangular grid defined by xpoints and ypoints that are to be excluded in the kriging calculations. Must be provided if style is specified as ‘masked’. False indicates that the point should not be masked, so the kriging system will be solved at the point. True indicates that the point should be masked, so the kriging system should not be solved at the point.

- `backend (str, optional)` – Specifies which approach to use in kriging. Specifying ‘vectorized’ will solve the entire kriging problem at once in a vectorized operation. This approach is faster but also can consume a significant amount of memory for large grids.
and/or large datasets. Specifying ‘loop’ will loop through each point at which the kriging system is to be solved. This approach is slower but also less memory-intensive. Specifying ‘C’ will utilize a loop in Cython. Default is ‘vectorized’.

- **n_closest_points** *(int, optional)* – For kriging with a moving window, specifies the number of nearby points to use in the calculation. This can speed up the calculation for large datasets, but should be used with caution. As Kitanidis notes, kriging with a moving window can produce unexpected oddities if the variogram model is not carefully chosen.

**Returns**

- **zvalues** *(ndarray, shape (M, N) or (N, 1))* – Z-values of specified grid or at the specified set of points. If style was specified as ‘masked’, zvalues will be a numpy masked array.
- **sigmasq** *(ndarray, shape (M, N) or (N, 1))* – Variance at specified grid points or at the specified set of points. If style was specified as ‘masked’, sigmasq will be a numpy masked array.

**get_epsilon_residuals()**

Returns the epsilon residuals for the variogram fit.

**get_statistics()**

Returns the Q1, Q2, and cR statistics for the variogram fit (in that order). No arguments.

**get_variogram_points()**

Returns both the lags and the variogram function evaluated at each of them.

The evaluation of the variogram function and the lags are produced internally. This method is convenient when the user wants to access to the lags and the resulting variogram (according to the model provided) for further analysis.

**Returns**

- **lags** (array) - the lags at which the variogram was evaluated
- **variogram** (array) - the variogram function evaluated at the lags

**Return type** *(tuple)* tuple containing

**plot_epsilon_residuals()**

Plots the epsilon residuals for the variogram fit.

**print_statistics()**

Prints out the Q1, Q2, and cR statistics for the variogram fit. NOTE that ideally Q1 is close to zero, Q2 is close to 1, and cR is as small as possible.

**switch_plotting()**

Allows user to switch plot display on/off. Takes no arguments.

**switch_verbose()**

Allows user to switch code talk-back on/off. Takes no arguments.

**update_variogram_model**(variogram_model, variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_scaling=1.0, anisotropy_angle=0.0)

Allows user to update variogram type and/or variogram model parameters.

**Parameters**

- **variogram_model** *(str or GSTools CovModel)* – May be any of the variogram models listed above. May also be ‘custom’, in which case variogram_parameters and variogram_function must be specified. You can also use a GSTools CovModel.
• **variogram_parameters** *(list or dict, optional)* – List or dict of variogram model parameters, as explained above. If not provided, a best fit model will be calculated as described above.

• **variogram_function** *(callable, optional)* – A callable function that must be provided if variogram_model is specified as ‘custom’. See above for more information.

• **nlags** *(int, optional)* – Number of averaging bins for the semivariogram. Default is 6.

• **weight** *(boolean, optional)* – Flag that specifies if semivariance at smaller lags should be weighted more heavily when automatically calculating the variogram model. See above for more information. True indicates that weights will be applied. Default is False.

• **anisotropy_scaling** *(float, optional)* – Scalar stretching value to take into account anisotropy. Default is 1 (effectively no stretching). Scaling is applied in the y-direction.

• **anisotropy_angle** *(float, optional)* – CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy. Default is 0 (no rotation).

### 3.1.2 pykrige.uk.UniversalKriging

**class** `pykrige.uk.UniversalKriging` *(x, y, z, variogram_model='linear', variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_angle=0.0, anisotropy_scaling=1.0, drift_terms=None, point_drift=None, external_drift=None, external_drift_x=None, external_drift_y=None, specified_drift=None, functional_drift=None, verbose=False, enable_plotting=False, exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv')

Provides greater control over 2D kriging by utilizing drift terms.

**Parameters**

• **x** *(array_like)* – X-coordinates of data points.

• **y** *(array_like)* – Y-coordinates of data points.

• **z** *(array_like)* – Values at data points.

• **variogram_model** *(str or GSTools CovModel, optional)* – Specified which variogram model to use; may be one of the following: linear, power, gaussian, spherical, exponential, hole-effect. Default is linear variogram model. To utilize a custom variogram model, specify ‘custom’; you must also provide variogram_parameters and variogram_function. Note that the hole-effect model is only technically correct for one-dimensional problems. You can also use a GSTools CovModel.

• **variogram_parameters** *(list or dict, optional)* – Parameters that define the specified variogram model. If not provided, parameters will be automatically calculated using a “soft” L1 norm minimization scheme. For variogram model parameters provided in a dict, the required dict keys vary according to the specified variogram model:

```python
# linear
    {'slope': slope, 'nugget': nugget)
# power
```
Note that either the full sill or the partial sill (psill = sill - nugget) can be specified in the dict. For variogram model parameters provided in a list, the entries must be as follows:

```
# linear
[slope, nugget]
# power
[scale, exponent, nugget]
# gaussian, spherical, exponential and hole-effect:
[sill, range, nugget]
```

Note that the full sill (NOT the partial sill) must be specified in the list format. For a custom variogram model, the parameters are required, as custom variogram models will not automatically be fit to the data. Furthermore, the parameters must be specified in list format, in the order in which they are used in the callable function (see variogram_function for more information). The code does not check that the provided list contains the appropriate number of parameters for the custom variogram model, so an incorrect parameter list in such a case will probably trigger an esoteric exception someplace deep in the code. NOTE that, while the list format expects the full sill, the code itself works internally with the partial sill.

- **variogram_function**(callable, optional) – A callable function that must be provided if variogram_model is specified as ‘custom’. The function must take only two arguments: first, a list of parameters for the variogram model; second, the distances at which to calculate the variogram model. The list provided in variogram_parameters will be passed to the function as the first argument.

- **nlags**(int, optional) – Number of averaging bins for the semivariogram. Default is 6.

- **weight**(bool, optional) – Flag that specifies if semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model. The routine is currently hard-coded such that the weights are calculated from a logistic function, so weights at small lags are ~1 and weights at the longest lags are ~0; the center of the logistic weighting is hard-coded to be at 70% of the distance from the shortest lag to the largest lag. Setting this parameter to True indicates that weights will be applied. Default is False. (Kitanidis suggests that the values at smaller lags are more important in fitting a variogram model, so the option is provided to enable such weighting.)

- **anisotropy_scaling**(float, optional) – Scalar stretching value to take into account anisotropy. Default is 1 (effectively no stretching). Scaling is applied in the y-direction in the rotated data frame (i.e., after adjusting for the anisotropy_angle, if anisotropy_angle is not 0).

- **anisotropy_angle**(float, optional) – CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy. Default is 0 (no rotation). Note that the coordinate system is rotated.

- **drift_terms**(list of strings, optional) – List of drift terms to include in universal kriging. Supported drift terms are currently ‘regional_linear’, ‘point_log’, ‘external_Z’, ‘specified’, and ‘functional’.

- **point_drift**(array_like, optional) – Array-like object that contains the coor-
dinates and strengths of the point-logarithmic drift terms. Array shape must be (N, 3), where
N is the number of point drift terms. First column (index 0) must contain x-coordinates, sec-
ond column (index 1) must contain y-coordinates, and third column (index 2) must contain
the strengths of each point term. Strengths are relative, so only the relation of the values to
each other matters. Note that the code will appropriately deal with point-logarithmic terms
that are at the same coordinates as an evaluation point or data point, but Python will still
kick out a warning message that an ln(0) has been encountered. If the problem involves
anisotropy, the well coordinates will be adjusted and the drift values will be calculated in
the adjusted data frame.

• **external_drift** (*array_like*, *optional*) – Gridded data used for the external
Z scalar drift term. Must be shape (M, N), where M is in the y-direction and N is in the
x-direction. Grid spacing does not need to be constant. If grid spacing is not constant, must
specify the grid cell sizes. If the problem involves anisotropy, the external drift values are
extracted based on the pre-adjusted coordinates (i.e., the original coordinate system).

• **external_drift_x** (*array_like*, *optional*) – X-coordinates for gridded exter-
nal Z-scalar data. Must be shape (M,) or (M, 1), where M is the number of grid cells in the
x-direction. The coordinate is treated as the center of the cell.

• **external_drift_y** (*array_like*, *optional*) – Y-coordinates for gridded exter-
nal Z-scalar data. Must be shape (N,) or (N, 1), where N is the number of grid cells in the
y-direction. The coordinate is treated as the center of the cell.

• **specified_drift** (*list of array-like objects*, *optional*) – List of ar-
rays that contain the drift values at data points. The arrays must be shape (N,) or (N, 1),
where N is the number of data points. Any number of specified-drift terms may be used.

• **functional_drift** (*list of callable objects*, *optional*) – List of
callable functions that will be used to evaluate drift terms. The function must be a func-
tion of only the two spatial coordinates and must return a single value for each coordinate
pair. It must be set up to be called with only two arguments, first an array of x values and sec-
ond an array of y values. If the problem involves anisotropy, the drift values are calculated
in the adjusted data frame.

• **verbose** (*bool*, *optional*) – Enables program text output to monitor kriging process.
Default is False (off).

• **enable_plotting** (*boolean*, *optional*) – Enables plotting to display variogram.
Default is False (off).

• **exact_values** (*bool*, *optional*) – If True, interpolation provides input values at
input locations. If False, interpolation accounts for variance/nugget within input values at
input locations and does not behave as an exact-interpolator [2]. Note that this only has
an effect if there is variance/nugget present within the input data since it is interpreted as
measurement error. If the nugget is zero, the kriged field will behave as an exact interpolator.

• **pseudo_inv** (*bool*, *optional*) – Whether the kriging system is solved with the pseudo
inverted kriging matrix. If True, this leads to more numerical stability and redundant points
are averaged. But it can take more time. Default: False

• **pseudo_inv_type** (*str*, *optional*) – Here you can select the algorithm to compute the
pseudo-inverse matrix:
  - "pinv": use pinv from scipy which uses lstsq
  - "pinv2": use pinv2 from scipy which uses SVD
  - "pinvh": use pinvh from scipy which uses eigen-values

Default: “pinv”
References

```python
__init__(x, y, z, variogram_model='linear', variogram_parameters=None, variogram_function=None, nlag=6, weight=False, anisotropy_scaling=1.0, anisotropy_angle=0.0, drift_terms=None, point_drift=None, external_drift=None, external_drift_x=None, external_drift_y=None, specified_drift=None, functional_drift=None, verbose=False, enable_plotting=False, exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv')
```

Initialize self. See help(type(self)) for accurate signature.

Methods

```python
__init__(x, y, z[, variogram_model, ...]) Initialize self.
display_variogram_model() Displays variogram model with the actual binned data.
execute(style, xpoints, ypoints[, mask, ...]) Calculates a kriged grid and the associated variance.
get_epsilon_residuals() Returns the epsilon residuals for the variogram fit.
get_statistics() Returns the Q1, Q2, and cR statistics for the variogram fit (in that order).
get_variogram_points() Returns both the lags and the variogram function evaluated at each of them.
plot_epsilon_residuals() Plots the epsilon residuals for the variogram fit.
print_statistics() Prints out the Q1, Q2, and cR statistics for the variogram fit.
switch_plotting() Allows user to switch plot display on/off.
switch_verbose() Allows user to switch code talk-back on/off.
update_variogram_model(variogram_model[, ...]) Allows user to update variogram type and/or variogram model parameters.
```

Attributes

```python
UNBIAS
eps
variogram_dict
```

```python
display_variogram_model()
    Displays variogram model with the actual binned data.
execute(style, xpoints, ypoints[, mask, ...])
    Calculates a kriged grid and the associated variance. Includes drift terms.

Parameters

- **style** (str) – Specifies how to treat input kriging points. Specifying ‘grid’ treats xpoints and ypoints as two arrays of x and y coordinates that define a rectangular grid. Specifying ‘points’ treats xpoints and ypoints as two arrays that provide coordinate pairs at which to solve the kriging system. Specifying ‘masked’ treats xpoints and ypoints as two arrays of x and y coordinates that define a rectangular grid and uses mask to only evaluate specific points in the grid.
- **xpoints** (array_like, shape (N,) or (N, 1)) – If style is specific as ‘grid’ or ‘masked’, x-coordinates of MxN grid. If style is specified as ‘points’, x-coordinates of specific points at which to solve kriging system.
• **ypoints** *(array-like, shape (M,) or (M, 1)) –* If style is specified as ‘grid’ or ‘masked’, y-coordinates of MxN grid. If style is specified as ‘points’, y-coordinates of specific points at which to solve kriging system. Note that in this case, xpoints and ypoints must have the same dimensions (i.e., M = N).

• **mask** *(boolean array, shape (M, N), optional) –* Specifies the points in the rectangular grid defined by xpoints and ypoints that are to be excluded in the kriging calculations. Must be provided if style is specified as ‘masked’. False indicates that the point should not be masked, so the kriging system will be solved at the point. True indicates that the point should be masked, so the kriging system should not be solved at the point.

• **backend** *(str, optional) –* Specifies which approach to use in kriging. Specifying ‘vectorized’ will solve the entire kriging problem at once in a vectorized operation. This approach is faster but also can consume a significant amount of memory for large grids and/or large datasets. Specifying ‘loop’ will loop through each point at which the kriging system is to be solved. This approach is slower but also less memory-intensive. Default is ‘vectorized’. Note that Cython backend is not supported for UK.

• **specified_drift_arrays** *(list of array-like objects, optional) –* Specifies the drift values at the points at which the kriging system is to be evaluated. Required if ‘specified’ drift provided in the list of drift terms when instantiating the UniversalKriging class. Must be a list of arrays in the same order as the list provided when instantiating the kriging object. Array(s) must be the same dimension as the specified grid or have the same number of points as the specified points; i.e., the arrays either must be shape (M, N), where M is the number of y grid-points and N is the number of x grid-points, or shape (M, ) or (N, 1), where M is the number of points at which to evaluate the kriging system.

Returns

• **zvalues** *(ndarray, shape (M, N) or (N, 1)) –* Z-values of specified grid or at the specified set of points. If style was specified as ‘masked’, zvalues will be a numpy masked array.

• **sigmasq** *(ndarray, shape (M, N) or (N, 1)) –* Variance at specified grid points or at the specified set of points. If style was specified as ‘masked’, sigmasq will be a numpy masked array.

get_epsilon_residuals()  
Returns the epsilon residuals for the variogram fit.

get_statistics()  
Returns the Q1, Q2, and cR statistics for the variogram fit (in that order). No arguments.

get_variogram_points()  
Returns both the lags and the variogram function evaluated at each of them.

The evaluation of the variogram function and the lags are produced internally. This method is convenient when the user wants to access to the lags and the resulting variogram (according to the model provided) for further analysis.

Returns  
lags (array) - the lags at which the variogram was evaluated  
variogram (array) - the variogram function evaluated at the lags

Return type  
tuple tuple containing

plot_epsilon_residuals()  
Plots the epsilon residuals for the variogram fit.
print_statistics()
Prints out the Q1, Q2, and cR statistics for the variogram fit. NOTE that ideally Q1 is close to zero, Q2 is close to 1, and cR is as small as possible.

switch_plotting()
Allows user to switch plot display on/off. Takes no arguments.

switch_verbose()
Allows user to switch code talk-back on/off. Takes no arguments.

update_variogram_model(variogram_model, variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_scaling_y=1.0, anisotropy_angle_y=0.0)
Allows user to update variogram type and/or variogram model parameters.

Parameters

• variogram_model (str or GSTools CovModel) – May be any of the variogram models listed above. May also be ‘custom’, in which case variogram_parameters and variogram_function must be specified. You can also use a GSTools CovModel.

• variogram_parameters (list or dict, optional) – List or dict of variogram model parameters, as explained above. If not provided, a best fit model will be calculated as described above.

• variogram_function (callable, optional) – A callable function that must be provided if variogram_model is specified as ‘custom’. See above for more information.

• nlags (int, optional) – Number of averaging bins for the semivariogram. Default is 6.

• weight (boolean, optional) – Flag that specifies if semivariance at smaller lags should be weighted more heavily when automatically calculating the variogram model. See above for more information. True indicates that weights will be applied. Default is False.

• anisotropy_scaling (float, optional) – Scalar stretching value to take into account anisotropy. Default is 1 (effectively no stretching). Scaling is applied in the y-direction.

• anisotropy_angle (float, optional) – CCW angle (in degrees) by which to rotate coordinate system in order to take into account anisotropy. Default is 0 (no rotation).

3.1.3 pykrige.ok3d.OldinaryKriging3D

class pykrige.ok3d.OldinaryKriging3D(x, y, z, val, variogram_model='linear', variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_scaling_y=1.0, anisotropy_angle_y=0.0, anisotropy_scaling=1.0, anisotropy_angle=0.0, enable_plotting=False, exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv')

Three-dimensional ordinary kriging.

Parameters

• x (array_like) – X-coordinates of data points.

• y (array_like) – Y-coordinates of data points.
• **z** *(array_like)* – Z-coordinates of data points.

• **val** *(array_like)* – Values at data points.

• **variogram_model** *(str or GSTools CovModel, optional)* – Specified which variogram model to use; may be one of the following: linear, power, gaussian, spherical, exponential, hole-effect. Default is linear variogram model. To utilize a custom variogram model, specify ‘custom’; you must also provide variogram_parameters and variogram_function. Note that the hole-effect model is only technically correct for one-dimensional problems. You can also use a GSTools CovModel.

• **variogram_parameters** *(list or dict, optional)* – Parameters that define the specified variogram model. If not provided, parameters will be automatically calculated using a “soft” L1 norm minimization scheme. For variogram model parameters provided in a dict, the required dict keys vary according to the specified variogram model:

  ```
  # linear
  {'slope': slope, 'nugget': nugget}
  # power
  {'scale': scale, 'exponent': exponent, 'nugget': nugget}
  # gaussian, spherical, exponential and hole-effect:
  {'sill': s, 'range': r, 'nugget': n}
  # OR
  {'psill': p, 'range': r, 'nugget': n}
  ```

Note that either the full sill or the partial sill (psill = sill - nugget) can be specified in the dict. For variogram model parameters provided in a list, the entries must be as follows:

```
# linear
[slope, nugget]
# power
[ scale, exponent, nugget]
# gaussian, spherical, exponential and hole-effect:
[ sill, range, nugget]
```

Note that the full sill (NOT the partial sill) must be specified in the list format. For a custom variogram model, the parameters are required, as custom variogram models will not automatically be fit to the data. Furthermore, the parameters must be specified in list format, in the order in which they are used in the callable function (see variogram_function for more information). The code does not check that the provided list contains the appropriate number of parameters for the custom variogram model, so an incorrect parameter list in such a case will probably trigger an esoteric exception someplace deep in the code. NOTE that, while the list format expects the full sill, the code itself works internally with the partial sill.

• **variogram_function** *(callable, optional)* – A callable function that must be provided if variogram_model is specified as ‘custom’. The function must take only two arguments: first, a list of parameters for the variogram model; second, the distances at which to calculate the variogram model. The list provided in variogram_parameters will be passed to the function as the first argument.

• **nlags** *(int, optional)* – Number of averaging bins for the semivariogram. Default is 6.

• **weight** *(boolean, optional)* – Flag that specifies if semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model. The routine is currently hard-coded such that the weights are calculated from a logistic function, so weights at small lags are ~1 and weights at the longest lags are ~0; the center of the logistic weighting is hard-coded to be at 70% of the distance from the shortest lag to the largest lag. Setting this parameter to True indicates that weights will be applied. Default
is False. (Kitanidis suggests that the values at smaller lags are more important in fitting a variogram model, so the option is provided to enable such weighting.)

- **anisotropy_scaling_y** *(float, optional)* – Scalar stretching value to take into account anisotropy in the y direction. Default is 1 (effectively no stretching). Scaling is applied in the y direction in the rotated data frame (i.e., after adjusting for the anisotropy_angle_x/y/z if anisotropy_angle_x/y/z is/are not 0).

- **anisotropy_scaling_z** *(float, optional)* – Scalar stretching value to take into account anisotropy in the z direction. Default is 1 (effectively no stretching). Scaling is applied in the z direction in the rotated data frame (i.e., after adjusting for the anisotropy_angle_x/y/z if anisotropy_angle_x/y/z is/are not 0).

- **anisotropy_angle_x** *(float, optional)* – CCW angle (in degrees) by which to rotate coordinate system about the x axis in order to take into account anisotropy. Default is 0 (no rotation). Note that the coordinate system is rotated. X rotation is applied first, then y rotation, then z rotation. Scaling is applied after rotation.

- **anisotropy_angle_y** *(float, optional)* – CCW angle (in degrees) by which to rotate coordinate system about the y axis in order to take into account anisotropy. Default is 0 (no rotation). Note that the coordinate system is rotated. X rotation is applied first, then y rotation, then z rotation. Scaling is applied after rotation.

- **anisotropy_angle_z** *(float, optional)* – CCW angle (in degrees) by which to rotate coordinate system about the z axis in order to take into account anisotropy. Default is 0 (no rotation). Note that the coordinate system is rotated. X rotation is applied first, then y rotation, then z rotation. Scaling is applied after rotation.

- **verbose** *(bool, optional)* – Enables program text output to monitor kriging process. Default is False (off).

- **enable_plotting** *(bool, optional)* – Enables plotting to display variogram. Default is False (off).

- **exact_values** *(bool, optional)* – If True, interpolation provides input values at input locations. If False, interpolation accounts for variance/nugget within input values at input locations and does not behave as an exact-interpolator [2]. Note that this only has an effect if there is variance/nugget present within the input data since it is interpreted as measurement error. If the nugget is zero, the kriged field will behave as an exact interpolator.

- **pseudo_inv** *(bool, optional)* – Whether the kriging system is solved with the pseudo inverted kriging matrix. If True, this leads to more numerical stability and redundant points are averaged. But it can take more time. Default: False

- **pseudo_inv_type** *(str, optional)* – Here you can select the algorithm to compute the pseudo-inverse matrix:
  - "pinv": use pinv from scipy which uses lstsq
  - "pinv2": use pinv2 from scipy which uses SVD
  - "pinvh": use pinvh from scipy which uses eigen-values

Default: "pinv"
__init__(x, y, z, val[, variogram_model, ...]) Initialize self. See help(type(self)) for accurate signature.

display_variogram_model() Displays variogram model with the actual binned data.

evaluate(style, xpoints, ypoints, zpoints[, ...]) Calculates a kriged grid and the associated variance.

generate_epsilon_residuals() Returns the epsilon residuals for the variogram fit.

generate_statistics() Returns the Q1, Q2, and cR statistics for the variogram fit (in that order).

plot_epsilon_residuals() Plots the epsilon residuals for the variogram fit.

print_statistics() Prints out the Q1, Q2, and cR statistics for the variogram fit.

switch_plotting() Allows user to switch plot display on/off.

switch_verbose() Allows user to switch code talk-back on/off.

update_variogram_model(variogram_model[, ...]) Changes the variogram model and variogram parameters for the kriging system.

eps

variogram_dict

display_variogram_model() Displays variogram model with the actual binned data.

evaluate(style, xpoints, ypoints, zpoints, mask=None, backend='vectorized', n_closest_points=None) Calculates a kriged grid and the associated variance.

This is now the method that performs the main kriging calculation. Note that currently measurements (i.e., z values) are considered ‘exact’. This means that, when a specified coordinate for interpolation is exactly the same as one of the data points, the variogram evaluated at the point is forced to be zero. Also, the diagonal of the kriging matrix is also always forced to be zero. In forcing the variogram evaluated at data points to be zero, we are effectively saying that there is no variance at that point (no uncertainty, so the value is ‘exact’).

In the future, the code may include an extra ‘exact_values’ boolean flag that can be adjusted to specify whether to treat the measurements as ‘exact’. Setting the flag to false would indicate that the variogram should not be forced to be zero at zero distance (i.e., when evaluated at data points). Instead, the uncertainty in the point will be equal to the nugget. This would mean that the diagonal of the kriging matrix would be set to the nugget instead of to zero.

Parameters

- **style (str)** – Specifies how to treat input kriging points. Specifying ‘grid’ treats
xpoints, ypoints, and zpoints as arrays of x, y, and z coordinates that define a rectangular grid. Specifying ‘points’ treats xpoints, ypoints, and zpoints as arrays that provide coordinates at which to solve the kriging system. Specifying ‘masked’ treats xpoints, ypoints, and zpoints as arrays of x, y, and z coordinates that define a rectangular grid and uses mask to only evaluate specific points in the grid.

• **xpoints** (*array_like*, *shape* (*N*,) or (*N*, 1)) – If style is specific as ‘grid’ or ‘masked’, x-coordinates of LxMxN grid. If style is specified as ‘points’, x-coordinates of specific points at which to solve kriging system.

• **ypoints** (*array-like*, *shape* (*M*,) or (*M*, 1)) – If style is specified as ‘grid’ or ‘masked’, y-coordinates of LxMxN grid. If style is specified as ‘points’, y-coordinates of specific points at which to solve kriging system. Note that in this case, xpoints, ypoints, and zpoints must have the same dimensions (i.e., L = M = N).

• **zpoints** (*array-like*, *shape* (*L*,) or (*L*, 1)) – If style is specified as ‘grid’ or ‘masked’, z-coordinates of LxMxN grid. If style is specified as ‘points’, z-coordinates of specific points at which to solve kriging system. Note that in this case, xpoints, ypoints, and zpoints must have the same dimensions (i.e., L = M = N).

• **mask** (*boolean array*, *shape* (*L*, *M*, *N*), *optional*) – Specifies the points in the rectangular grid defined by xpoints, ypoints, zpoints that are to be excluded in the kriging calculations. Must be provided if style is specified as ‘masked’. False indicates that the point should not be masked, so the kriging system will be solved at the point. True indicates that the point should be masked, so the kriging system should will not be solved at the point.

• **backend** (*str*, *optional*) – Specifies which approach to use in kriging. Specifying ‘vectorized’ will solve the entire kriging problem at once in a vectorized operation. This approach is faster but also can consume a significant amount of memory for large grids and/or large datasets. Specifying ‘loop’ will loop through each point at which the kriging system is to be solved. This approach is slower but also less memory-intensive. Default is ‘vectorized’.

• **n_closest_points** (*int*, *optional*) – For kriging with a moving window, specifies the number of nearby points to use in the calculation. This can speed up the calculation for large datasets, but should be used with caution. As Kitanidis notes, kriging with a moving window can produce unexpected oddities if the variogram model is not carefully chosen.

**Returns**

• **kvalues** (*ndarray*, *shape* (*L*, *M*, *N*) or (*N*, 1)) – Interpolated values of specified grid or at the specified set of points. If style was specified as ‘masked’, kvalues will be a numpy masked array.

• **sigmasq** (*ndarray*, *shape* (*L*, *M*, *N*) or (*N*, 1)) – Variance at specified grid points or at the specified set of points. If style was specified as ‘masked’, sigmasq will be a numpy masked array.

**get_epsilon_residuals**()
Returns the epsilon residuals for the variogram fit.

**get_statistics**()
Returns the Q1, Q2, and cR statistics for the variogram fit (in that order). No arguments.

**plot_epsilon_residuals**()
Plots the epsilon residuals for the variogram fit.

3.1. Krigging algorithms
print_statistics()
Prints out the Q1, Q2, and cR statistics for the variogram fit. NOTE that ideally Q1 is close to zero, Q2 is close to 1, and cR is as small as possible.

switch_plotting()
Allows user to switch plot display on/off. Takes no arguments.

switch_verbose()
Allows user to switch code talk-back on/off. Takes no arguments.

update_variogram_model(variogram_model, variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_scaling_y=1.0, anisotropy_scaling_z=1.0, anisotropy_angle_x=0.0, anisotropy_angle_y=0.0, anisotropy_angle_z=0.0)
Changes the variogram model and variogram parameters for the kriging system.

Parameters

- **variogram_model** *(str or GSTools CovModel)* – May be any of the variogram models listed above. May also be ‘custom’, in which case variogram_parameters and variogram_function must be specified. You can also use a GSTools CovModel.

- **variogram_parameters** *(list or dict, optional)* – List or dict of variogram model parameters, as explained above. If not provided, a best fit model will be calculated as described above.

- **variogram_function** *(callable, optional)* – A callable function that must be provided if variogram_model is specified as ‘custom’. See above for more information.

- **nlags** *(int, optional)* – Number of averaging bins for the semivariogram. Default is 6.

- **weight** *(bool, optional)* – Flag that specifies if semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model. See above for more information. True indicates that weights will be applied. Default is False.

- **anisotropy_scaling_y** *(float, optional)* – Scalar stretching value to take into account anisotropy in y-direction. Default is 1 (effectively no stretching). See above for more information.

- **anisotropy_scaling_z** *(float, optional)* – Scalar stretching value to take into account anisotropy in z-direction. Default is 1 (effectively no stretching). See above for more information.

- **anisotropy_angle_x** *(float, optional)* – Angle (in degrees) by which to rotate coordinate system about the x axis in order to take into account anisotropy. Default is 0 (no rotation). See above for more information.

- **anisotropy_angle_y** *(float, optional)* – Angle (in degrees) by which to rotate coordinate system about the y axis in order to take into account anisotropy. Default is 0 (no rotation). See above for more information.

- **anisotropy_angle_z** *(float, optional)* – Angle (in degrees) by which to rotate coordinate system about the z axis in order to take into account anisotropy. Default is 0 (no rotation). See above for more information.
class pykrige.uk3d.UniversalKriging3D(x, y, z, val, variogram_model='linear',
variogram_parameters=None, variogram_function=None, nlags=6, weight=False,
anisotropy_scaling_y=1.0, anisotropy_scaling_z=1.0,
anisotropy_angle_x=0.0, anisotropy_angle_y=0.0,
anisotropy_angle_z=0.0, drift_terms=None, specified_drift=None, functional_drift=None,
verbose=False, enable_plotting=False, exact_values=True, pseudo_inv=False,
pseudo_inv_type='pinv')

Three-dimensional universal kriging.

Parameters

• `x (array_like)` – X-coordinates of data points.
• `y (array_like)` – Y-coordinates of data points.
• `z (array_like)` – Z-coordinates of data points.
• `val (array_like)` – Values at data points.
• `variogram_model (str or GSTools CovModel, optional)` – Specified which variogram model to use; may be one of the following: linear, power, gaussian, spherical, exponential, hole-effect. Default is linear variogram model. To utilize a custom variogram model, specify ‘custom’; you must also provide variogram_parameters and variogram_function. Note that the hole-effect model is only technically correct for one-dimensional problems. You can also use a GSTools CovModel.

• `variogram_parameters (list or dict, optional)` – Parameters that define the specified variogram model. If not provided, parameters will be automatically calculated using a “soft” L1 norm minimization scheme. For variogram model parameters provided in a dict, the required dict keys vary according to the specified variogram model:

```python
# linear
{'slope': slope, 'nugget': nugget}

# power
{'scale': scale, 'exponent': exponent, 'nugget': nugget}

# gaussian, spherical, exponential and hole-effect:
{'sill': s, 'range': r, 'nugget': n}

# OR
{'psill': p, 'range': r, 'nugget': n}
```

Note that either the full sill or the partial sill (psill = sill - nugget) can be specified in the dict. For variogram model parameters provided in a list, the entries must be as follows:

```python
# linear
[slope, nugget]

# power
[ scale, exponent, nugget ]

# gaussian, spherical, exponential and hole-effect:
[sill, range, nugget]
```

Note that the full sill (NOT the partial sill) must be specified in the list format. For a custom variogram model, the parameters are required, as custom variogram models will not automatically be fit to the data. Furthermore, the parameters must be specified in list format, in the order in which they are used in the callable function (see variogram_function for more
The code does not check that the provided list contains the appropriate number of parameters for the custom variogram model, so an incorrect parameter list in such a case will probably trigger an esoteric exception somewhere deep in the code. NOTE that, while the list format expects the full sill, the code itself works internally with the partial sill.

- **variogram_function** *(callable, optional)* – A callable function that must be provided if `variogram_model` is specified as ‘custom’. The function must take only two arguments: first, a list of parameters for the variogram model; second, the distances at which to calculate the variogram model. The list provided in `variogram_parameters` will be passed to the function as the first argument.

- **nlags** *(int, optional)* – Number of averaging bins for the semivariogram. Default is 6.

- **weight** *(bool, optional)* – Flag that specifies if semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model. The routine is currently hard-coded such that the weights are calculated from a logistic function, so weights at small lags are ~1 and weights at the longest lags are ~0; the center of the logistic weighting is hard-coded to be at 70% of the distance from the shortest lag to the largest lag. Setting this parameter to True indicates that weights will be applied. Default is False. (Kitanidis suggests that the values at smaller lags are more important in fitting a variogram model, so the option is provided to enable such weighting.)

- **anisotropy_scaling_y** *(float, optional)* – Scalar stretching value to take into account anisotropy in the y direction. Default is 1 (effectively no stretching). Scaling is applied in the y direction in the rotated data frame (i.e., after adjusting for the `anisotropy_angle_x/y/z`, if `anisotropy_angle_x/y/z` is/are not 0).

- **anisotropy_scaling_z** *(float, optional)* – Scalar stretching value to take into account anisotropy in the z direction. Default is 1 (effectively no stretching). Scaling is applied in the z direction in the rotated data frame (i.e., after adjusting for the `anisotropy_angle_x/y/z`, if `anisotropy_angle_x/y/z` is/are not 0).

- **anisotropy_angle_x** *(float, optional)* – CCW angle (in degrees) by which to rotate coordinate system about the x axis in order to take into account anisotropy. Default is 0 (no rotation). Note that the coordinate system is rotated. X rotation is applied first, then y rotation, then z rotation. Scaling is applied after rotation.

- **anisotropy_angle_y** *(float, optional)* – CCW angle (in degrees) by which to rotate coordinate system about the y axis in order to take into account anisotropy. Default is 0 (no rotation). Note that the coordinate system is rotated. X rotation is applied first, then y rotation, then z rotation. Scaling is applied after rotation.

- **anisotropy_angle_z** *(float, optional)* – CCW angle (in degrees) by which to rotate coordinate system about the z axis in order to take into account anisotropy. Default is 0 (no rotation). Note that the coordinate system is rotated. X rotation is applied first, then y rotation, then z rotation. Scaling is applied after rotation.

- **drift_terms** *(list of strings, optional)* – List of drift terms to include in three-dimensional universal kriging. Supported drift terms are currently ‘regional_linear’, ‘specified’, and ‘functional’.

- **specified_drift** *(list of array-like objects, optional)* – List of arrays that contain the drift values at data points. The arrays must be shape (N,) or (N, 1), where N is the number of data points. Any number of specified-drift terms may be used.

- **functional_drift** *(list of callable objects, optional)* – List of callable functions that will be used to evaluate drift terms. The function must be a function of only the three spatial coordinates and must return a single value for each coordinate.
triplet. It must be set up to be called with only three arguments, first an array of x values, the second an array of y values, and the third an array of z values. If the problem involves
anisotropy, the drift values are calculated in the adjusted data frame.

- **verbose** *(boolean, optional)* – Enables program text output to monitor kriging
  process. Default is False (off).

- **enable_plotting** *(boolean, optional)* – Enables plotting to display variogram.
  Default is False (off).

- **exact_values** *(bool, optional)* – If True, interpolation provides input values at
  input locations. If False, interpolation accounts for variance/nugget within input values at
  input locations and does not behave as an exact-interpolator [2]. Note that this only has
  an effect if there is variance/nugget present within the input data since it is interpreted as
  measurement error. If the nugget is zero, the kriged field will behave as an exact interpolator.

- **pseudo_inv** *(bool, optional)* – Whether the kriging system is solved with the pseudo
  inverted kriging matrix. If True, this leads to more numerical stability and redundant points
  are averaged. But it can take more time. Default: False

- **pseudo_inv_type** *(str, optional)* – Here you can select the algorithm to compute the
  pseudo-inverse matrix:
  - “pinv”: use pinv from scipy which uses lstsq
  - “pinv2”: use pinv2 from scipy which uses SVD
  - “pinvh”: use pinvh from scipy which uses eigen-values
  Default: “pinv”

**References**

```python
__init__(x, y, z, val[, variogram_model='linear', variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_scaling_y=1.0, anisotropy_scaling_z=1.0, anisotropy_angle_x=0.0, anisotropy_angle_y=0.0, anisotropy_angle_z=0.0, drift_terms=None, specified_drift=None, functional_drift=None, verbose=False, enable_plotting=False, exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv')
```
Initialize self. See help(type(self)) for accurate signature.

**Methods**

```python
__init__(x, y, z, val[, variogram_model, ...]) Initialize self.
display_variogram_model() Displays semivariogram and variogram model.
execute(style, xpoints, ypoints, zpoints[, ...]) Calculates a kriged grid and the associated variance.
get_epsilon_residuals() Returns the epsilon residuals for the variogram fit.
get_statistics() Returns the Q1, Q2, and cR statistics for the variogram fit (in that order).
plot_epsilon_residuals() Plots the epsilon residuals for the variogram fit.
print_statistics() Prints out the Q1, Q2, and cR statistics for the variogram fit.
switch_plotting() Enables/disable variogram plot display.
switch_verbose() Enables/disables program text output.
```
Continued on next page
Table 8 – continued from previous page

`update_variogram_model(variogram_model, ...)`: Changes the variogram model and variogram parameters for the kriging system.

Attributes

- **UNBIAS**
- **eps**
- **variogram_dict**

**display_variogram_model()**

Displays semivariogram and variogram model.

**execute(style, xpoints, ypoints, zpoints, mask=None, backend='vectorized', specified_drift_arrays=None)**

Calculates a kriged grid and the associated variance.

This is now the method that performs the main kriging calculation. Note that currently measurements (i.e., z values) are considered ‘exact’. This means that, when a specified coordinate for interpolation is exactly the same as one of the data points, the variogram evaluated at the point is forced to be zero. Also, the diagonal of the kriging matrix is also always forced to be zero. In forcing the variogram evaluated at data points to be zero, we are effectively saying that there is no variance at that point (no uncertainty, so the value is ‘exact’).

In the future, the code may include an extra ‘exact_values’ boolean flag that can be adjusted to specify whether to treat the measurements as ‘exact’. Setting the flag to false would indicate that the variogram should not be forced to be zero at zero distance (i.e., when evaluated at data points). Instead, the uncertainty in the point will be equal to the nugget. This would mean that the diagonal of the kriging matrix would be set to the nugget instead of to zero.

**Parameters**

- **style (str)** — Specifies how to treat input kriging points. Specifying ‘grid’ treats xpoints, ypoints, and zpoints as arrays of x, y, and z coordinates that define a rectangular grid. Specifying ‘points’ treats xpoints, ypoints, and zpoints as arrays that provide coordinates at which to solve the kriging system. Specifying ‘masked’ treats xpoints, ypoints, and zpoints as arrays of x, y, and z coordinates that define a rectangular grid and uses mask to only evaluate specific points in the grid.

- **xpoints (array_like, shape (N,) or (N, 1))** — If style is specific as ‘grid’ or ‘masked’, x-coordinates of LxMxN grid. If style is specified as ‘points’, x-coordinates of specific points at which to solve kriging system.

- **ypoints (array_like, shape (M,) or (M, 1))** — If style is specified as ‘grid’ or ‘masked’, y-coordinates of LxMxN grid. If style is specified as ‘points’, y-coordinates of specific points at which to solve kriging system. Note that in this case, xpoints, ypoints, and zpoints must have the same dimensions (i.e., L = M = N).

- **zpoints (array_like, shape (L,) or (L, 1))** — If style is specified as ‘grid’ or ‘masked’, z-coordinates of LxMxN grid. If style is specified as ‘points’, z-coordinates of specific points at which to solve kriging system. Note that in this case, xpoints, ypoints, and zpoints must have the same dimensions (i.e., L = M = N).

- **mask (boolean array, shape (L, M, N), optional)** — Specifies the points in the rectangular grid defined by xpoints, ypoints, zpoints that are to be excluded in the kriging calculations. Must be provided if style is specified as ‘masked’. False indicates that the point should not be masked, so the kriging system will be solved at the point. True
indicates that the point should be masked, so the kriging system will not be solved at the point.

- **backend (string, optional)** – Specifies which approach to use in kriging. Specifying ‘vectorized’ will solve the entire kriging problem at once in a vectorized operation. This approach is faster but also can consume a significant amount of memory for large grids and/or large datasets. Specifying ‘loop’ will loop through each point at which the kriging system is to be solved. This approach is slower but also less memory-intensive. Default is ‘vectorized’.

- **specified_drift_arrays (list of array-like objects, optional)** – Specifies the drift values at the points at which the kriging system is to be evaluated. Required if ‘specified’ drift provided in the list of drift terms when instantiating the UniversalKriging3D class. Must be a list of arrays in the same order as the list provided when instantiating the kriging object. Array(s) must be the same dimension as the specified grid or have the same number of points as the specified points; i.e., the arrays either must be shape (L, M, N), where L is the number of z grid-points, M is the number of y grid-points, and N is the number of x grid-points, or shape (N,) or (N, 1), where N is the number of points at which to evaluate the kriging system.

Returns

- **kvalues (ndarray, shape (L, M, N) or (N,) or (N, 1))** – Interpolated values of specified grid or at the specified set of points. If style was specified as ‘masked’, kvalues will be a numpy masked array.

- **sigmasq (ndarray, shape (L, M, N) or (N,) or (N, 1))** – Variance at specified grid points or at the specified set of points. If style was specified as ‘masked’, sigmasq will be a numpy masked array.

**get_epsilon_residuals()**

Returns the epsilon residuals for the variogram fit. No arguments.

**get_statistics()**

Returns the Q1, Q2, and cR statistics for the variogram fit (in that order). No arguments.

**plot_epsilon_residuals()**

Plots the epsilon residuals for the variogram fit. No arguments.

**print_statistics()**

Prints out the Q1, Q2, and cR statistics for the variogram fit. NOTE that ideally Q1 is close to zero, Q2 is close to 1, and cR is as small as possible.

**switch_plotting()**

Enables/disables variogram plot display. No arguments.

**switch_verbose()**

Enables/disables program text output. No arguments.

**update_variogram_model (variogram_model, variogram_parameters=None, variogram_function=None, nlags=6, weight=False, anisotropy_scaling_y=1.0, anisotropy_scaling_z=1.0, anisotropy_angle_x=0.0, anisotropy_angle_y=0.0)**

Changes the variogram model and variogram parameters for the kriging system.

Parameters

- **variogram_model (str or GSTools CovModel)** – May be any of the variogram models listed above. May also be ‘custom’, in which case variogram_parameters and variogram_function must be specified. You can also use a GSTools CovModel.
• **variogram_parameters** *(list or dict, optional)* – List or dict of variogram model parameters, as explained above. If not provided, a best fit model will be calculated as described above.

• **variogram_function** *(callable, optional)* – A callable function that must be provided if variogram_model is specified as ‘custom’. See above for more information.

• **nlags** *(int, optional)* – Number of averaging bins for the semivariogram. Default is 6.

• **weight** *(boolean, optional)* – Flag that specifies if semivariance at smaller lags should be weighted more heavily when automatically calculating variogram model. See above for more information. True indicates that weights will be applied. Default is False.

• **anisotropy_scaling_y** *(float, optional)* – Scalar stretching value to take into account anisotropy in y-direction. Default is 1 (effectively no stretching). See above for more information.

• **anisotropy_scaling_z** *(float, optional)* – Scalar stretching value to take into account anisotropy in z-direction. Default is 1 (effectively no stretching). See above for more information.

• **anisotropy_angle_x** *(float, optional)* – Angle (in degrees) by which to rotate coordinate system about the x axis in order to take into account anisotropy. Default is 0 (no rotation). See above for more information.

• **anisotropy_angle_y** *(float, optional)* – Angle (in degrees) by which to rotate coordinate system about the y axis in order to take into account anisotropy. Default is 0 (no rotation). See above for more information.

• **anisotropy_angle_z** *(float, optional)* – Angle (in degrees) by which to rotate coordinate system about the z axis in order to take into account anisotropy. Default is 0 (no rotation). See above for more information.

### 3.1.5 pykrige.rk.RegressionKriging

```python
class pykrige.rk.RegressionKriging(regression_model=SVR(), method='ordinary', variogram_model='linear', n_closest_points=10, nlags=6, weight=False, verbose=False, exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv', variogram_parameters=None, variogram_function=None, anisotropy_scaling=(1.0, 1.0), anisotropy_angle=(0.0, 0.0, 0.0), enable_statistics=False, coordinates_type='euclidean', drift_terms=None, point_drift=None, ext_drift_grid=(None, None, None), functional_drift=None)
```

An implementation of Regression-Kriging.

As described here: https://en.wikipedia.org/wiki/Regression-Kriging

Parameters

• **regression_model** *(machine learning model instance from sklearn)* –

• **method** *(str, optional)* – type of kriging to be performed

• **variogram_model** *(str, optional)* – variogram model to be used during Kriging
• **n_closest_points** (*int*) – number of closest points to be used during Ordinary Kriging

• **nlags** (*int*) – see OK/UK class description

• **weight** (*bool*) – see OK/UK class description

• **verbose** (*bool*) – see OK/UK class description

• **exact_values** (*bool*) – see OK/UK class description

• **variogram_parameters** (*list or dict*) – see OK/UK class description

• **variogram_function** (*callable*) – see OK/UK class description

• **anisotropy_scaling** (*tuple*) – single value for 2D (UK/OK) and two values in 3D (UK3D/OK3D)

• **anisotropy_angle** (*tuple*) – single value for 2D (UK/OK) and three values in 3D (UK3D/OK3D)

• **enable_statistics** (*bool*) – see OK class description

• **coordinates_type** (*str*) – see OK/UK class description

• **drift_terms** (*list of strings*) – see UK/UK3D class description

• **point_drift** (*array_like*) – see UK class description

• **ext_drift_grid** (*tuple*) – Holding the three values external_drift, external_drift_x and external_drift_z for the UK class

• **functional_drift** (*list of callable*) – see UK/UK3D class description

---

**Initialize** (

```python
__init__([regression_model, method, ...])
```

Initialize self. See help(type(self)) for accurate signature.

---

**Methods**

```python
__init__([regression_model, method, ...])
```

Initialize self.

```python
fit(p, x, y)
```

Fit the regression method and also Krige the residual.

```python
krige_residual(x, **kwargs)
```

Calculate the residuals.

```python
predict(p, x, **kwargs)
```

Predict.

```python
score(p, x, y[, sample_weight])
```

Overloading default regression score method.

---

**fit (p, x, y)**

Fit the regression method and also Krige the residual.

**Parameters**

- **p** (*ndarray*) – (Ns, d) array of predictor variables (Ns samples, d dimensions) for regression

- **x** (*ndarray*) – ndarray of (x, y) points. Needs to be a (Ns, 2) array corresponding to the lon/lat, for example 2d regression kriging. array of Points, (x, y, z) pairs of shape (N, 3) for 3d kriging
• \textbf{y} (ndarray) – array of targets (Ns, )

\textbf{krige_residual} (x, **kwargs)

Calculate the residuals.

\textbf{Parameters} \textbf{x} (ndarray) – ndarray of (x, y) points. Needs to be a (Ns, 2) array corresponding to the lon/lat, for example.

\textbf{Returns} residual – kriged residual values

\textbf{Return type} ndarray

\textbf{predict} (p, x, **kwargs)

Predict.

\textbf{Parameters}

• \textbf{p} (ndarray) – (Ns, d) array of predictor variables (Ns samples, d dimensions) for regression

• \textbf{x} (ndarray) – ndarray of (x, y) points. Needs to be a (Ns, 2) array corresponding to the lon/lat, for example. array of Points, (x, y, z) pairs of shape (N, 3) for 3d kriging

\textbf{Returns} pred – The expected value of ys for the query inputs, of shape (Ns,).

\textbf{Return type} ndarray

\textbf{score} (p, x, y, sample_weight=None, **kwargs)

Overloading default regression score method.

\textbf{Parameters}

• \textbf{p} (ndarray) – (Ns, d) array of predictor variables (Ns samples, d dimensions) for regression

• \textbf{x} (ndarray) – ndarray of (x, y) points. Needs to be a (Ns, 2) array corresponding to the lon/lat, for example. array of Points, (x, y, z) pairs of shape (N, 3) for 3d kriging

• \textbf{y} (ndarray) – array of targets (Ns, )

### 3.2 Wrappers

\textbf{pykrige.rk.Krige}([method, variogram_model, ...])

A scikit-learn wrapper class for Ordinary and Universal Kriging.

#### 3.2.1 pykrige.rk.Krige

\textbf{class} pykrige.rk.Krige (method='ordinary', variogram_model='linear', nlags=6, weight=False, n_closest_points=10, verbose=False, exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv', variogram_parameters=None, variogram_function=None, anisotropy_scaling=(1.0, 1.0), anisotropy_angle=(0.0, 0.0, 0.0), enable_statistics=False, coordinates_type='euclidean', drift_terms=None, point_drift=None, ext_drift_grid=(None, None, None), functional_drift=None)

A scikit-learn wrapper class for Ordinary and Universal Kriging.

This works with both Grid/RandomSearchCv for finding the best Krige parameters combination for a problem.

\textbf{Parameters}
• **method**(str, optional) – type of kriging to be performed
• **variogram_model**(str, optional) – variogram model to be used during Krigeing
• **nlags**(int) – see OK/UK class description
• **weight**(bool) – see OK/UK class description
• **n_closest_points**(int) – number of closest points to be used during Ordinary Kriging
• **verbose**(bool) – see OK/UK class description
• **exact_values**(bool) – see OK/UK class description
• **variogram_parameters**(list or dict) – see OK/UK class description
• **variogram_function**(callable) – see OK/UK class description
• **anisotropy_scaling**(tuple) – single value for 2D (UK/OK) and two values in 3D (UK3D/OK3D)
• **anisotropy_angle**(tuple) – single value for 2D (UK/OK) and three values in 3D (UK3D/OK3D)
• **enable_statistics**(bool) – see OK class description
• **coordinates_type**(str) – see OK/UK class description
• **drift_terms**(list of strings) – see UK/UK3D class description
• **point_drift**(array_like) – see UK class description
• **ext_drift_grid**(tuple) – Holding the three values external_drift, external_drift_x and external_drift_z for the UK class
• **functional_drift**(list of callable) – see UK/UK3D class description

__init__**(method='ordinary', variogram_model='linear', nlags=6, weight=False, n_closest_points=10, verbose=False, exact_values=True, pseudo_inv=False, pseudo_inv_type='pinv', variogram_parameters=None, variogram_function=None, anisotropy_scaling=(1.0, 1.0), anisotropy_angle=(0.0, 0.0, 0.0), enable_statistics=False, coordinates_type='euclidean', drift_terms=None, point_drift=None, ext_drift_grid=None, functional_drift=None)**

Initialize self. See help(type(self)) for accurate signature.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>execute</strong>(points, *args, **kwargs)</td>
<td>Execute.</td>
<td></td>
</tr>
<tr>
<td><strong>fit</strong>(x, y, *args, **kwargs)</td>
<td>Fit the current model.</td>
<td></td>
</tr>
<tr>
<td><strong>get_params</strong>(deep)</td>
<td>Get parameters for this estimator.</td>
<td></td>
</tr>
<tr>
<td><strong>predict</strong>(x, *args, **kwargs)</td>
<td>Predict.</td>
<td></td>
</tr>
<tr>
<td><strong>score</strong>(X, y, sample_weight)</td>
<td>Return the coefficient of determination R(^2) of the prediction.</td>
<td></td>
</tr>
<tr>
<td><strong>set_params</strong>(**params)</td>
<td>Set the parameters of this estimator.</td>
<td></td>
</tr>
</tbody>
</table>

**execute**(points, *args, **kwargs)

Execute.

Parameters **points**(dict) –
Returns

- Prediction array
- Variance array

fit(x, y, *args, **kwargs)
Fit the current model.

Parameters

- x (ndarray) — array of Points, (x, y) pairs of shape (N, 2) for 2d kriging array of Points, (x, y, z) pairs of shape (N, 3) for 3d kriging
- y (ndarray) — array of targets (N,)

get_params(deep=True)
Get parameters for this estimator.

Parameters
depth (bool, default=True) — If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params — Parameter names mapped to their values.

Return type
mapping of string to any

predict(x, *args, **kwargs)
Predict.

Parameters

- x (ndarray) — array of Points, (x, y) pairs of shape (N, 2) for 2d kriging array of Points, (x, y, z) pairs of shape (N, 3) for 3d kriging

Returns

Return type
Prediction array

score(X, y, sample_weight=None)
Return the coefficient of determination $R^2$ of the prediction.

The coefficient $R^2$ is defined as $(1 - \frac{u}{v})$, where $u$ is the residual sum of squares $(y_{\text{true}} - y_{\text{pred}})^2$. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of $y$, disregarding the input features, would get a $R^2$ score of 0.0.

Parameters

- X (array-like of shape (n_samples, n_features)) — Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.
- y (array-like of shape (n_samples,) or (n_samples, n_outputs)) — True values for X.
- sample_weight (array-like of shape (n_samples,), default=None) — Sample weights.

Returns

score — $R^2$ of self.predict(X) wrt. y.

Return type
float
Notes

The R2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the score method of all the multioutput regressors (except for `MultiOutputRegressor`).

```python
set_params(**params)
```

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it’s possible to update each component of a nested object.

**Parameters**

- `**params` (dict) – Estimator parameters.

**Returns**

- `self` – Estimator instance.

**Return type**

- object

3.3 Tools

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pykrige.kriging_tools.write_asc_grid(x, y, z)</code></td>
<td>Writes gridded data to ASCII grid file (*.asc).</td>
</tr>
<tr>
<td><code>pykrige.kriging_tools.read_asc_grid(filename)</code></td>
<td>Reads ASCII grid file (*.asc).</td>
</tr>
</tbody>
</table>

3.3.1 `pykrige.kriging_tools.write_asc_grid`

```python
pykrige.kriging_tools.write_asc_grid(x, y, z, filename='output.asc', style=1)
```

Writes gridded data to ASCII grid file (*.asc).

This is useful for exporting data to a GIS program.

**Parameters**

- `x` (array_like, shape (N,) or (N, 1)) – X-coordinates of grid points at center of cells.
- `y` (array_like, shape (M,) or (M, 1)) – Y-coordinates of grid points at center of cells.
- `z` (array_like, shape (M, N)) – Gridded data values. May be a masked array.
- `filename` (string, optional) – Name of output *.asc file. Default name is ‘output.asc’.
- `style` (int, optional) – Determines how to write the *.asc file header. Specifying 1 writes out DX, DY, XLLCENTER, YLLCENTER. Specifying 2 writes out CELLSIZE (note DX must be the same as DY), XLLCORNER, YLLCORNER. Default is 1.

3.3.2 `pykrige.kriging_tools.read_asc_grid`

```python
pykrige.kriging_tools.read_asc_grid(filename, footer=0)
```

Reads ASCII grid file (*.asc).

**Parameters**
• **filename** *(str)* – Name of *.asc file.
• **footer** *(int, optional)* – Number of lines at bottom of *.asc file to skip.

**Returns**

• **grid_array** *(numpy array, shape (M, N))* – (M, N) array of grid values, where M is number of Y-coordinates and N is number of X-coordinates. The array entry corresponding to the lower-left coordinates is at index [M, 0], so that the array is oriented as it would be in X-Y space.
• **x** *(numpy array, shape (N,))* – 1D array of N X-coordinates.
• **y** *(numpy array, shape (M,))* – 1D array of M Y-coordinates.
• **CELLSIZE** *(tuple or float)* – Either a two-tuple of (x-cell size, y-cell size), or a float that specifies the uniform cell size.
• **NODATA** *(float)* – Value that specifies which entries are not actual data.
4.1 Universal Kriging Example

In this example we apply a regional linear trend to the kriging system.

```python
from pykrige.uk import UniversalKriging
import numpy as np
import matplotlib.pyplot as plt

data = np.array(
    [
        [0.3, 1.2, 0.47],
        [1.9, 0.6, 0.56],
        [1.1, 3.2, 0.74],
        [3.3, 4.4, 1.47],
        [4.7, 3.8, 1.74],
    ]
)
gridx = np.arange(0.0, 5.5, 0.5)
gridy = np.arange(0.0, 5.5, 0.5)

Create the universal kriging object. Required inputs are the X-coordinates of the data points, the Y-coordinates of the data points, and the Z-values of the data points. Variogram is handled as in the ordinary kriging case. drift_terms is a list of the drift terms to include; currently supported terms are ‘regional_linear’, ‘point_log’, and ‘external_Z’. Refer to UniversalKriging.__doc__ for more information.

```python
UK = UniversalKriging(
    data[:, 0],
    data[:, 1],
    data[:, 2],
    variogram_model="linear",
```
drift_terms=['regional_linear'],
)

Creates the kriged grid and the variance grid. Allows for kriging on a rectangular grid of points, on a masked rectangular grid of points, or with arbitrary points. (See UniversalKriging.__doc__ for more information.)

```python
z, ss = UK.execute("grid", gridx, gridy)
plt.imshow(z)
plt.show()
```

Total running time of the script: 0 minutes 2.684 seconds

### 4.2 Ordinary Kriging Example

First we will create a 2D dataset together with the associated x, y grids.

```python
import numpy as np
import pykrige.kriging_tools as kt
from pykrige.ok import OrdinaryKriging
import matplotlib.pyplot as plt
```
data = np.array(
    
    [0.3, 1.2, 0.47],
    [1.9, 0.6, 0.56],
    [1.1, 3.2, 0.74],
    [3.3, 4.4, 1.47],
    [4.7, 3.8, 1.74],
    
)

gridx = np.arange(0.0, 5.5, 0.5)
gridy = np.arange(0.0, 5.5, 0.5)

Create the ordinary kriging object. Required inputs are the X-coordinates of the data points, the Y-coordinates of the data points, and the Z-values of the data points. If no variogram model is specified, defaults to a linear variogram model. If no variogram model parameters are specified, then the code automatically calculates the parameters by fitting the variogram model to the binned experimental semivariogram. The verbose kwarg controls code talk-back, and the enable_plotting kwarg controls the display of the semivariogram.

OK = OrdinaryKriging(
    data[:, 0],
    data[:, 1],
    data[:, 2],
    variogram_model="linear",
    verbose=False,
    enable_plotting=False,
)

Creates the kriged grid and the variance grid. Allows for kriging on a rectangular grid of points, on a masked rectangular grid of points, or with arbitrary points. (See OrdinaryKriging.__doc__ for more information.)

z, ss = OK.execute("grid", gridx, gridy)

Writes the kriged grid to an ASCII grid file and plot it.

kt.write_asc_grid(gridx, gridy, z, filename="output.asc")
plt.imshow(z)
plt.show()
Total running time of the script: ( 0 minutes 0.109 seconds)

4.3 GSTools Interface

Example how to use the PyKrige routines with a GSTools CovModel.
import numpy as np
from pykrige.ok import OrdinaryKriging
from matplotlib import pyplot as plt
import gstools as gs

# conditioning data
data = np.array(
    [
        [0.3, 1.2, 0.47],
        [1.9, 0.6, 0.56],
        [1.1, 3.2, 0.74],
        [3.3, 4.4, 1.47],
        [4.7, 3.8, 1.74],
    ]
)

# grid definition for output field
gridx = np.arange(0.0, 5.5, 0.1)
gridy = np.arange(0.0, 6.5, 0.1)
# a GSTools based covariance model
cov_model = gs.Gaussian(dim=2, len_scale=4, anis=0.2, angles=-0.5, var=0.5, nugget=0.1)
# ordinary kriging with pykrige
OK1 = OrdinaryKriging(data[:, 0], data[:, 1], data[:, 2], cov_model)
z1, ss1 = OK1.execute("grid", gridx, gridy)
plt.imshow(z1, origin="lower")
plt.show()
4.4 Exact Values

PyKrige demonstration and usage as a non-exact interpolator in 1D.

```python
from pykrige.ok import OrdinaryKriging
import matplotlib.pyplot as plt
import numpy as np
plt.style.use("ggplot")
np.random.seed(42)
x = np.linspace(0, 12.5, 50)
xpred = np.linspace(0, 12.5, 393)
y = np.sin(x) * np.exp(-0.25 * x) + np.random.normal(-0.25, 0.25, 50)

# compare OrdinaryKriging as an exact and non exact interpolator
uk = OrdinaryKriging(x, np.zeros(x.shape), y, variogram_model="linear", exact_values=False)
uk_exact = OrdinaryKriging(x, np.zeros(x.shape), y, variogram_model="linear")

y_pred, y_std = uk.execute("grid", xpred, np.array([0.0]), backend="loop")
y_pred_exact, y_std_exact = uk_exact.execute("grid", xpred, np.array([0.0]), backend="loop")

y_pred = np.squeeze(y_pred)
y_std = np.squeeze(y_std)

y_pred_exact = np.squeeze(y_pred_exact)
y_std_exact = np.squeeze(y_std_exact)
```

![Graph showing exact and non-exact prediction comparison](image-url)

(continues on next page)
ax.scatter(x, y, label="Input Data")
ax.plot(xpred, y_pred_exact, label="Exact Prediction")
ax.plot(xpred, y_pred, label="Non Exact Prediction")

ax.fill_between(
    xpred,
    y_pred - 3 * y_std,
    y_pred + 3 * y_std,
    alpha=0.3,
    label="Confidence interval",
)
ax.legend(loc=9)
ax.set_ylim(-1.8, 1.3)
ax.legend(loc=9)
plt.xlabel("X")
plt.ylabel("Field")
plt.show()

Total running time of the script: ( 0 minutes 0.253 seconds)

4.5 Regression kriging

An example of regression kriging

Out:

import sys

from sklearn.svm import SVR

(imported on next page)
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.datasets import fetch_california_housing

from pykrige.rk import RegressionKriging
from pykrige.compat import train_test_split

svr_model = SVR(C=0.1, gamma="auto")
rf_model = RandomForestRegressor(n_estimators=100)
lr_model = LinearRegression(normalize=True, copy_X=True, fit_intercept=False)

models = [svr_model, rf_model, lr_model]

try:
    housing = fetch_california_housing()
except PermissionError:
    # this dataset can occasionally fail to download on Windows
    sys.exit(0)

# take the first 5000 as Kriging is memory intensive
p = housing["data"][::5000, :-2]
x = housing["data"][::5000, -2:]
target = housing["target"][::5000]

p_train, p_test, x_train, x_test, target_train, target_test = train_test_split(
    p, x, target, test_size=0.3, random_state=42)

for m in models:
    print("=" * 40)
    print("regression model:", m.__class__.__name__)
    m_rk = RegressionKriging(regression_model=m, n_closest_points=10)
    m_rk.fit(p_train, x_train, target_train)
    print("Regression Score: ", m_rk.regression_model.score(p_test, target_test))
    print("RK score: ", m_rk.score(p_test, x_test, target_test))

Total running time of the script: ( 0 minutes 8.383 seconds)

4.6 Geometric example

A small example script showing the usage of the ‘geographic’ coordinates type for ordinary kriging on a sphere.

from pykrige.ok import OrdinaryKriging
import numpy as np
from matplotlib import pyplot as plt

# Make this example reproducible:
np.random.seed(89239413)

# Generate random data following a uniform spatial distribution
# of nodes and a uniform distribution of values in the interval
# [2.0, 5.5]:
N = 7
lon = 360.0 * np.random.random(N)
latt = 180.0 / np.pi * np.arcsin(2 * np.random.random(N) - 1)
z = 3.5 * np.random.rand(N) + 2.0

# Generate a regular grid with 60° longitude and 30° latitude steps:
grid_lon = np.linspace(0.0, 360.0, 7)
grid_lat = np.linspace(-90.0, 90.0, 7)

# Create ordinary kriging object:
OK = OrdinaryKriging(
    lon,
latt,
z,
    variogram_model="linear",
    verbose=False,
    enable_plotting=False,
    coordinates_type="geographic",
)

# Execute on grid:
z1, ss1 = OK.execute("grid", grid_lon, grid_lat)

# Create ordinary kriging object ignoring curvature:
OK = OrdinaryKriging(
    lon, lat, z, variogram_model="linear", verbose=False, enable_plotting=False
)

# Execute on grid:
z2, ss2 = OK.execute("grid", grid_lon, grid_lat)

Print data at equator (last longitude index will show periodicity):

print("Original data:")
print("Longitude:", lon.astype(int))
print("Latitude: ", lat.astype(int))
print("z: ", np.array_str(z, precision=2))
print("\nKrig at 60° latitude:\n")
print("Longitude:", grid_lon)
print("Value: ", np.array_str(z1[5, :], precision=2))
print("Sigma^2: ", np.array_str(ss1[5, :], precision=2))
print("\nIgnoring curvature:\n")
print("Value: ", np.array_str(z2[5, :], precision=2))
print("Sigma^2: ", np.array_str(ss2[5, :], precision=2))

Out:

Original data:
Longitude: [122 166 92 138 86 122 136]
Latitude: [-46 -36 -25 -73 -25 50 -29]
z: [2.75 3.36 2.24 3.07 3.37 5.25 2.82]

Krig at 60° latitude:
======================
Longitude: [0. 60. 120. 180. 240. 300. 360.]
Value: [5.29 5.11 5.27 5.17 5.35 5.63 5.29]
Sigma^2: [2.22 1.32 0.42 1.21 2.07 2.48 2.22]
Ignoring curvature:

Value: [4.55 4.72 5.25 4.82 4.61 4.53 4.48]
Sigma^2: [3.79 2. 0.39 1.85 3.54 5.46 7.53]

We can see that the data point at longitude 122, latitude 50 correctly dominates the kriged results, since it is the closest node in spherical distance metric, as longitude differences scale with \( \cos(\text{latitude}) \). When kriging using longitude / latitude linearly, the value for grid points with longitude values further away as longitude is now incorrectly weighted equally as latitude.

```python
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(z1, extent=[0, 360, -90, 90], origin="lower")
ax1.set_title("geo-coordinates")
ax2.imshow(z2, extent=[0, 360, -90, 90], origin="lower")
ax2.set_title("non geo-coordinates")
plt.show()
```

Total running time of the script: (0 minutes 0.178 seconds)
4.7 Three-Dimensional Kriging Example

```python
from pykrige.ok3d import OrdinaryKriging3D
from pykrige.uk3d import UniversalKriging3D
import numpy as np
from matplotlib import pyplot as plt

data = np.array(
    [
        [0.1, 0.1, 0.3, 0.9],
        [0.2, 0.1, 0.4, 0.8],
        [0.1, 0.3, 0.1, 0.9],
        [0.5, 0.4, 0.4, 0.5],
        [0.3, 0.3, 0.2, 0.7],
    ]
)
gridx = np.arange(0.0, 0.6, 0.05)
gridy = np.arange(0.0, 0.6, 0.01)
gridz = np.arange(0.0, 0.6, 0.1)

Create the 3D ordinary kriging object and solves for the three-dimension kriged volume and variance. Refer to OrdinaryKriging3D.__doc__ for more information.

ok3d = OrdinaryKriging3D(
    data[:, 0], data[:, 1], data[:, 2], data[:, 3], variogram_model="linear"
)
k3d1, ss3d = ok3d.execute("grid", gridx, gridy, gridz)

Create the 3D universal kriging object and solves for the three-dimension kriged volume and variance. Refer to UniversalKriging3D.__doc__ for more information.

uk3d = UniversalKriging3D(
    data[:, 0],
    data[:, 1],
    data[:, 2],
    data[:, 3],
    variogram_model="linear",
    drift_terms=["regional_linear"],
)
k3d2, ss3d = uk3d.execute("grid", gridx, gridy, gridz)

To use the generic ‘specified’ drift term, the user must provide the drift values at each data point and at every grid point. The following example is equivalent to using a linear drift in all three spatial dimensions. Refer to UniversalKriging3D.__doc__ for more information.

zg, yg, xg = np.meshgrid(gridz, gridy, gridx, indexing="ij")
uk3d = UniversalKriging3D(
    data[:, 0],
    data[:, 1],
    data[:, 2],
    data[:, 3],
    variogram_model="linear",
    drift_terms=["specified"],
    specified_drift=[data[:, 0], data[:, 1], data[:, 2]],
)
```

(continues on next page)
To use the generic ‘functional’ drift term, the user must provide a callable function that takes only the spatial dimensions as arguments. The following example is equivalent to using a linear drift only in the x-direction. Refer to UniversalKriging3D.__doc__ for more information.

```python
func = lambda x, y, z: x
uk3d = UniversalKriging3D(
    data[:, 0],
    data[:, 1],
    data[:, 2],
    data[:, 3],
    variogram_model="linear",
    drift_terms=["functional"],
    functional_drift=[func],
)
```

Note that the use of the ‘specified’ and ‘functional’ generic drift capabilities is essentially identical in the two-dimensional universal kriging class (except for a difference in the number of spatial coordinates for the passed drift functions). See UniversalKriging.__doc__ for more information.

```python
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4)
ax1.imshow(k3d1[:, :, 0], origin="lower")
ax1.set_title("ordinary kriging")
ax2.imshow(k3d2[:, :, 0], origin="lower")
ax2.set_title("regional lin. drift")
ax3.imshow(k3d3[:, :, 0], origin="lower")
ax3.set_title("specified drift")
ax4.imshow(k3d3[:, :, 0], origin="lower")
ax4.set_title("functional drift")
plt.tight_layout()
plt.show()
```
4.8 Krige CV

Searching for optimal kriging parameters with cross validation

Out:

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
```

(continues on next page)
best_score $R^2 = -0.045$

best_params = ('method': 'universal', 'variogram_model': 'spherical')

CV results:
- mean_test_score : [-0.17196766 -0.17196766 -0.17784984 -0.16199851 -0.05679591 -0.05354486 -0.04486488]
- mean_train_score : [1. 1. 1. 1. 1. 1. 1.]
- param_method : ['ordinary' 'ordinary' 'ordinary' 'ordinary' 'universal' 'universal' 'universal' 'universal']
- param_variogram_model : ['linear' 'power' 'gaussian' 'spherical' 'linear' 'power' 'gaussian' 'spherical']

Fitting 5 folds for each of 8 candidates, totalling 40 fits
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging
n_closest_points will be ignored for UniversalKriging

[Parallel(n_jobs=1)]: Done 40 out of 40 | elapsed: 2.2s finished

R^2 = -0.097

cv_results = {'mean_test_score': [-0.25558625, -0.25299609, -0.27181187, -0.26476051, -0.11692599, -0.09740768, -0.14908596, -0.12875737],
              'mean_train_score': [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
              'param_variogram_model': ['linear', 'power', 'gaussian', 'spherical', 'linear', 'power', 'gaussian', 'spherical']}

import numpy as np
from pykrige.rk import Krige
from sklearn.model_selection import GridSearchCV

# 2D Kring param opt
param_dict = {
    "method": ["ordinary", "universal"],
    "variogram_model": ["linear", "power", "gaussian", "spherical"],
    "nlags": [4, 6, 8],
    "maxnlags": [4, 6, 8]
}
# "weight": [True, False]
}
estimator = GridSearchCV(Krige(), param_dict, verbose=True, return_train_score=True)

# dummy data
X = np.random.randint(0, 400, size=(100, 2)).astype(float)
y = 5 * np.random.rand(100)

# run the gridsearch
estimator.fit(X=X, y=y)

if hasattr(estimator, "best_score_"):
    print("best_score R² = {:.3f}".format(estimator.best_score_))
    print("best_params = ", estimator.best_params_)

print("\nCV results::")
if hasattr(estimator, "cv_results_"):
    for key in ["mean_test_score", "mean_train_score", "param_method", "param_variogram_model", ]:
        print(" - {} : {}".format(key, estimator.cv_results_[key]))

# 3D Kring param opt
param_dict3d = {
    "method": ["ordinary3d", "universal3d"],
    "variogram_model": ["linear", "power", "gaussian", "spherical"],
    # "nlags": [4, 6, 8],
    # "weight": [True, False]
}
estimator = GridSearchCV(Krige(), param_dict3d, verbose=True, return_train_score=True)

# dummy data
X3 = np.random.randint(0, 400, size=(100, 3)).astype(float)
y = 5 * np.random.rand(100)

# run the gridsearch
estimator.fit(X=X3, y=y)

if hasattr(estimator, "best_score_"):
    print("best_score R² = {:.3f}".format(estimator.best_score_))
    print("best_params = ", estimator.best_params_)

print("\nCV results::")
if hasattr(estimator, "cv_results_"):
    for key in ["mean_test_score", "mean_train_score", "param_method", "param_variogram_model", ]:
4.9 1D Kriging

An example of 1D kriging with PyKrige

import numpy as np
import matplotlib.pyplot as plt
from pykrige import OrdinaryKriging

plt.style.use("ggplot")

# fmt: off
# Data taken from
# https://blog.dominodatalab.com/fitting-gaussian-process-models-python/
X, y = np.array([[-5.01, 1.06], [-4.90, 0.92], [-4.82, 0.35], [-4.69, 0.49], [-4.56, 0.52],
                 [-4.52, 0.12], [-4.39, 0.47], [-4.32, -0.19], [-4.19, 0.08], [-4.11, -0.19],
                 [-4.00, -0.03], [-3.89, -0.03], [-3.78, -0.05], [-3.67, 0.10], [-3.59, 0.44],
                 [-3.50, 0.66], [-3.39, -0.12], [-3.28, 0.45], [-3.20, 0.14], [-3.07, -0.28],
                 [-3.01, -0.46], [-2.90, -0.32], [-2.77, -1.58], [-2.69, -1.44], [-2.60, -1.51],
                 [-2.49, -1.50], [-2.41, -2.04], [-2.28, -1.57], [-2.19, -1.25], [-2.10, -1.50],
                 [-2.00, -1.42], [-1.91, -1.10], [-1.80, -0.58], [-1.67, -1.08], [-1.61, -0.79],
                 [-1.50, -1.00], [-1.37, -0.04], [-1.30, -0.54], [-1.19, -0.15], [-1.06, -0.18],
                 [-0.98, -0.25], [-0.87, -1.20], [-0.78, -0.49], [-0.68, -0.83], [-0.57, -0.15],
                 [-0.50, 0.00], [-0.30, -1.10], [-0.29, -0.32], [-0.18, -0.60], [-0.09, -0.49],
                 [0.03, 0.50], [0.09, -0.02], [0.20, -0.47], [0.31, -0.11], [0.41, -0.28],
                 [0.53, 0.40], [0.61, 0.11], [0.70, 0.32], [0.94, 0.42], [1.02, 0.57],
                 [1.13, 0.82], [1.24, 1.18], [1.30, 0.86], [1.43, 1.11], [1.50, 0.74],
                 [1.63, 0.75], [1.74, 1.15], [1.80, 0.76], [1.93, 0.68], [2.03, 0.03],
                 [2.12, 0.31], [2.23, -0.14], [2.31, -0.88], [2.40, -1.25], [2.50, -1.62],
                 [2.63, -1.37], [2.72, -0.99], [2.80, -1.92], [2.83, -1.94], [2.91, -1.32],
                 [3.00, -1.69], [3.13, -1.84], [3.21, -2.05], [3.30, -1.69], [3.41, -0.53],
                 [3.50, -0.02], [3.59, -0.44], [3.67, 0.10], [3.78, -0.05], [3.89, -0.03],
                 [4.00, -0.03], [4.05, 0.08], [4.11, -0.19], [4.19, 0.08], [4.32, -0.19],
                 [4.41, -0.49], [4.52, 0.12], [4.69, 0.49], [4.82, 0.35], [4.90, 0.92],
                 [5.01, 1.06]])

Total running time of the script: 0 minutes 3.433 seconds


```python
[3.52, -0.55], [3.63, -0.92], [3.72, -0.76], [3.80, -0.41], [3.91, 0.12],
[4.04, 0.25], [4.13, 0.16], [4.24, 0.26], [4.32, 0.62], [4.44, 1.69],
[4.52, 1.11], [4.65, 0.36], [4.74, 0.79], [4.84, 0.87], [4.93, 1.01],
[5.02, 0.55])
```

```python
X_pred = np.linspace(-6, 6, 200)

# fmt: on
X_pred = np.linspace(-6, 6, 200)

# pykrige doesn't support 1D data for now, only 2D or 3D
# adapting the 1D input to 2D
uk = OrdinaryKriging(X, np.zeros(X.shape), y, variogram_model="gaussian")

y_pred, y_std = uk.execute("grid", X_pred, np.array([0.0]))

y_pred = np.squeeze(y_pred)

y_std = np.squeeze(y_std)

fig, ax = plt.subplots(1, 1, figsize=(10, 4))

ax.scatter(X, y, s=40, label="Input data")

ax.plot(X_pred, y_pred, label="Predicted values")

ax.fill_between(X_pred,
                 y_pred - 3 * y_std,
                 y_pred + 3 * y_std,
                 alpha=0.3,
                 label="Confidence interval",
                 )

ax.legend(loc=9)

ax.set_xlabel("x")

ax.set_ylabel("y")

ax.set_xlim(-6, 6)

ax.set_ylim(-2.8, 3.5)

plt.show()
```

**Total running time of the script:** ( 0 minutes 0.156 seconds)
5.1 Version 1.5.1

August 20, 2020

New features

• update Regression Kriging class to be compatible with all kriging features (#158)
• added option to enable/disable “exact values” to all kriging routines (#153)
• added option to use the pseudo-inverse in all kriging routines (#151)

Changes

• removed compat-layer for sklearn (#157)
• updated examples in documentation

5.2 Version 1.5.0

April 04, 2020

New features

• support for GSTools covariance models (#125)
• pre-build wheels for py35-py38 on Linux, Windows and MacOS (#142)
• GridSearchCV from the compat module sets iid=False by default (if present in sklearn) to be future prove (iid will be deprecated) (#144)

Changes

• dropped py2* and py<3.5 support (#142)
• installation now requires cython (#142)
• codebase was formatted with black (#144)
• internally use of scipys lapack/blas bindings (#142)
• PyKrige is now part of the GeoStat-Framework

5.3 Version 1.4.1

January 13, 2019

New features

• Added method to obtain variogram model points. PR[#94](https://github.com/GeoStat-Framework/PyKrige/pull/94) by [Daniel Mejía Raigosa](https://github.com/Daniel-M)

Bug fixes

• Fixed OrdinaryKriging readme example. PR[#107](https://github.com/GeoStat-Framework/PyKrige/pull/107) by [Harry Matchette-Downes](https://github.com/harrymd)
• Fixed kriging matrix not being calculated correctly for geographic coordinates. PR[99](https://github.com/GeoStat-Framework/PyKrige/pull/99) by [Mike Rilee](https://github.com/michelleerilee)

5.4 Version 1.4.0

April 24, 2018

New features

• Regression kriging algorithm. PR [#27](https://github.com/GeoStat-Framework/PyKrige/pull/27) by [Sudipta Basaks](https://github.com/basaks).
• Support for spherical coordinates. PR [#23](https://github.com/GeoStat-Framework/PyKrige/pull/23) by [Malte Ziebarth](https://github.com/mjziebarth)
• Kriging parameter tuning with scikit-learn. PR [#24](https://github.com/GeoStat-Framework/PyKrige/pull/24) by [Sudipta Basaks](https://github.com/basaks).
• Variogram model parameters can be specified using a list or a dict. Allows for directly feeding in the partial sill rather than the full sill. PR [#47](https://github.com/GeoStat-Framework/PyKrige/pull/47) by [Benjamin Murphy](https://github.com/bsmurphy).

Enhancements

• Improved memory usage in variogram calculations. PR [#42](https://github.com/GeoStat-Framework/PyKrige/pull/42) by [Sudipta Basaks](https://github.com/basaks).
• Added benchmark scripts. PR [#36](https://github.com/GeoStat-Framework/PyKrige/pull/36) by [Roman Yurchak](https://github.com/rth)
• Added an extensive example using the meusegrids dataset. PR [#28](https://github.com/GeoStat-Framework/PyKrige/pull/28) by [kvanlombeek](https://github.com/kvanlombeek).

Bug fixes

• Statistics calculations in 3D kriging. PR [#45](https://github.com/GeoStat-Framework/PyKrige/pull/45) by [Will Chang](https://github.com/whdc).
• Automatic variogram estimation robustified. PR [#47](https://github.com/GeoStat-Framework/PyKrige/pull/47) by [Benjamin Murphy](https://github.com/bsmurphy).
5.5 Version 1.3.1

December 10, 2016

• More robust setup for building Cython extensions

5.6 Version 1.3.0

October 23, 2015

• Added support for Python 3.

• Updated the setup script to handle problems with trying to build the Cython extensions. If the appropriate compiler hasn’t been installed on Windows, then the extensions won’t work (see [this discussion of using Cython extensions on Windows] for how to deal with this problem). The setup script now attempts to build the Cython extensions and automatically falls back to pure Python if the build fails. **NOTE that the Cython extensions currently are not set up to work in Python 3** (see [discussion in issue #10]), so they are not built when installing with Python 3. This will be changed in the future.

• [closed issue #2]: https://github.com/GeoStat-Framework/PyKrige/issues/2

• [this discussion of using Cython extensions on Windows]: https://github.com/cython/cython/wiki/CythonExtensionsOnWindows

• [discussion in issue #10]: https://github.com/GeoStat-Framework/PyKrige/issues/10

5.7 Version 1.2.0

August 1, 2015

• Updated the execution portion of each class to streamline processing and reduce redundancy in the code.

• Integrated kriging with a moving window for two-dimensional ordinary kriging. Thanks to Roman Yurchak for this addition. This can be very useful for working with very large datasets, as it limits the size of the kriging matrix system. However, note that this approach can also produce unexpected oddities if the spatial covariance of the data does not decay quickly or if the window is too small. (See Kitanidis 1997 for a discussion of potential problems in kriging with a moving window; also see [closed issue #2] for a brief note about important considerations when kriging with a moving window.)

• Integrated a Cython backend for two-dimensional ordinary kriging. Again, thanks to Roman Yurchak for this addition. Note that currently the Cython backend is only implemented for two-dimensional ordinary kriging; it is not implemented in any of the other kriging classes. (I’ll gladly accept any pull requests to extend the Cython backend to the other classes.)

• Implemented two new generic drift capabilities that should allow for use of arbitrary user-designed drifts. These generic drifts are referred to as ‘specified’ and ‘functional’ in the code. They are available for both two-dimensional and three-dimensional universal kriging (see below). With the ‘specified’ drift capability, the user specifies the values of the drift term at every data point and every point at which the kriging system is to be evaluated. With the ‘functional’ drift capability, the user provides callable function(s) of the two or three spatial coordinates that define the drift term(s). The functions must only take the spatial coordinates as arguments. An arbitrary number of ‘specified’ or ‘functional’ drift terms may be used. See UniversalKriging.__doc__ or UniversalKriging3D.__doc__ for more information.

• Made a few changes to how the drift terms are implemented when the problem is anisotropic. The regional linear drift is applied in the adjusted coordinate frame. For the point logarithmic drift, the point coordinates are
transformed into the adjusted coordinate frame and the drift values are calculated in the transformed frame. The external scalar drift values are extracted using the original (i.e., unadjusted) coordinates. Any functions that are used with the ‘functional’ drift capability are evaluated in the adjusted coordinate frame. Specified drift values are not adjusted as they are taken to be for the exact points provided.

- Added support for three-dimensional universal kriging. The previous three-dimensional kriging class has been renamed OrdinaryKriging3D within module ok3d, and the new class is called UniversalKriging3D within module uk3d. See UniversalKriging3D.__doc__ for usage information. A regional linear drift (‘regional_linear’) is the only code-internal drift that is currently supported, but the ‘specified’ and ‘functional’ generic drift capabilities are also implemented here (see above). The regional linear drift is applied in all three spatial dimensions.

5.8 Version 1.1.0

May 25, 2015

- Added support for two different approaches to solving the entire kriging problem. One approach solves for the specified grid or set of points in a single vectorized operation; this method is default. The other approach loops through the specified points and solves the kriging system at each point. In both of these techniques, the kriging matrix is set up and inverted only once. In the vectorized approach, the rest of the kriging system (i.e., the RHS matrix) is set up as a single large array, and the whole system is solved with a single call to numpy.dot(). This approach is faster, but it can consume a lot of RAM for large datasets and/or large grids. In the looping approach, the rest of the kriging system (the RHS matrix) is set up at each point, and the kriging system at that point is solved with a call to numpy.dot(). This approach is slower, but it does not take as much memory. The approach can be specified by using the backend kwarg in the execute() method: ‘vectorized’ (default) for the vectorized approach, ‘loop’ for the looping approach. Thanks to Roman Yurchak for these changes and optimizations.

- Added support for implementing custom variogram models. To do so, set variogram_model to ‘custom’. You must then also specify variogram_parameters as well as variogram_function, which must be a callable object that takes only two arguments, first a list of function parameters and then the distances at which to evaluate the variogram model. Note that currently the code will not automatically fit the custom variogram model to the data. You must provide the variogram_parameters, which will be passed to the callable variogram_function as the first argument.

- Modified anisotropy rotation so that coordinate system is rotated CCW by specified angle. The sense of rotation for 2D kriging is now the opposite of what it was before.

- Added support for 3D kriging. This is now available as class Krige3D in pykrige.k3d. The usage is essentially the same as with the two-dimensional kriging classes, except for a few extra arguments that must be passed during instantiation and when calling Krige3D.execute(). See Krige3D.__doc__ for more information.

5.9 Version 1.0.3

February 15, 2015

- Fixed a problem with the tests that are performed to see if the kriging system is to be solved at a data point. (Tests are completed in order to determine whether to force the kriging solution to converge to the true data value.)

- Changed setup script.

5.10 Version 1.0

January 25, 2015
• Changed license to New BSD.
• Added support for point-specific and masked-grid kriging. Note that the arguments for the `OrdinaryKriging.execute()` and `UniversalKriging.execute()` methods have changed.
• Changed semivariogram binning procedure.
• Boosted execution speed by almost an order of magnitude.
• Fixed some problems with the external drift capabilities.
• Added more comprehensive testing script.
• Fixed slight problem with `read_asc_grid()` function in `kriging_tools`. Also made some code improvements to both the `write_asc_grid()` and `read_asc_grid()` functions in `kriging_tools`.

5.11 Version 0.2.0

November 23, 2014

• Consolidated backbone functions into a single module in order to reduce redundancy in the code. `OrdinaryKriging` and `UniversalKriging` classes now import and call the `core` module for the standard functions.
• Fixed a few glaring mistakes in the code.
• Added more documentation.

5.12 Version 0.1.2

October 27, 2014

• First complete release.
Symbols

__init__() (pykrige.ok.OrdinaryKriging method), 9
__init__() (pykrige.ok3d.OrdinaryKriging3D method), 20
__init__() (pykrige.rk.Krige method), 31
__init__() (pykrige.rk.RegressionKriging method), 29
__init__() (pykrige.uk.UniversalKriging method), 15
__init__() (pykrige.uk3d.UniversalKriging3D method), 25

display_variogram_model()
(pykrige.ok.OrdinaryKriging method), 10
display_variogram_model()
(pykrige.ok3d.OrdinaryKriging3D method), 20
display_variogram_model()
(pykrige.uk.UniversalKriging method), 15
display_variogram_model()
(pykrige.uk3d.UniversalKriging3D method), 26

eexecute() (pykrige.ok.OrdinaryKriging method), 10
execute() (pykrige.ok3d.OrdinaryKriging3D method), 20
execute() (pykrige.rk.Krige method), 31
execute() (pykrige.uk.UniversalKriging method), 15
execute() (pykrige.uk3d.UniversalKriging3D method), 26

f

fit() (pykrige.rk.Krige method), 32
fit() (pykrige.rk.RegressionKriging method), 29

G

get_epsilon_residuals()
(pykrige.ok.OrdinaryKriging method), 11
get_epsilon_residuals()
(pykrige.ok3d.OrdinaryKriging3D method), 21
get_epsilon_residuals()
(pykrige.uk.UniversalKriging method), 16
get_epsilon_residuals()
(pykrige.uk3d.UniversalKriging3D method), 27
get_params() (pykrige.rk.Krige method), 32
get_statistics() (pykrige.ok.OrdinaryKriging method), 11
get_statistics() (pykrige.ok3d.OrdinaryKriging3D method), 21
get_statistics() (pykrige.uk.UniversalKriging method), 16
get_statistics() (pykrige.uk3d.UniversalKriging3D method), 27
get_variogram_points()
(pykrige.ok.UniversalKriging method), 11
get_variogram_points()
(pykrige.uk.UniversalKriging method), 16

K

Krive (class in pykrige.rk), 30
krige_residual() (pykrige.rk.RegressionKriging method), 30

O

OrdinaryKriging (class in pykrige.ok), 7
OrdinaryKriging3D (class in pykrige.ok3d), 17

P

plot_epsilon_residuals()
(pykrige.ok.OrdinaryKriging method), 11
plot_epsilon_residuals()
(pykrige.ok3d.OrdinaryKriging3D method), 21
plot_epsilon_residuals()
(pykrige.uk.UniversalKriging method), 16
plot_epsilon_residuals()
(pykrige.uk3d.UniversalKriging3D method), 27
predict() (pykrige.rk.Krige method), 32
predict() (pykrige.rk.RegressionKriging method), 30
print_statistics() (pykrige.ok.OrdinaryKriging method), 11
print_statistics()
(pykrige.ok3d.OrdinaryKriging3D method), 21
print_statistics() (pykrige.uk.UniversalKriging method), 16
print_statistics()
(pykrige.uk3d.UniversalKriging3D method), 27

R
read_asc_grid() (in module pykrige.kriging_tools), 33
RegressionKriging (class in pykrige.rk), 28

S
score() (pykrige.rk.Krige method), 32
score() (pykrige.rk.RegressionKriging method), 30
set_params() (pykrige.rk.Krige method), 33
switch_plotting() (pykrige.ok.OrdinaryKriging method), 11
switch_plotting()
(pykrige.ok3d.OrdinaryKriging3D method), 22
switch_plotting() (pykrige.uk.UniversalKriging method), 17
switch_plotting()
(pykrige.uk3d.UniversalKriging3D method), 27
switch_verbose() (pykrige.ok.OrdinaryKriging method), 11
switch_verbose() (pykrige.ok3d.OrdinaryKriging3D method), 22
switch_verbose() (pykrige.uk.UniversalKriging method), 17
switch_verbose() (pykrige.uk3d.UniversalKriging3D method), 27

U
UniversalKriging (class in pykrige.uk), 12
UniversalKriging3D (class in pykrige.uk3d), 23
update_variogram_model()
(pykrige.ok.OrdinaryKriging method), 11
update_variogram_model()
(pykrige.ok3d.OrdinaryKriging3D method), 22
update_variogram_model()
(pykrige.uk.UniversalKriging method), 17
update_variogram_model()
(pykrige.uk3d.UniversalKriging3D method), 27

W
write_asc_grid() (in module pykrige.kriging_tools), 33