SciKit GStat Documentation

Release 0.6.6

Mirko Mälicke

Jul 09, 2021

CONTENTS:

CHAPTER

ONE

WELCOME TO SCIKIT GSTAT

Download the docs as PDF

SciKit-Gstat is a scipy-styled analysis module for variogram analysis. The base class is called *Variogram*, which is probably the only import needed. However, several other classes exist:

- *DirectionalVariogram* for directional variograms
- *SpaceTimeVariogram* for spatio-temporal variograms
- *OrdinaryKriging* for interpolation
- *MetricSpace* for pre-computed spatial samples

The variogram classes have a similar interface and can compute experimental variograms and fit theoretical variogram model functions. The module makes use of a rich selection of semi-variance estimators, variogram model functions and sptial binning functions, while being extensible at the same time.

CHAPTER

TWO

HOW TO CITE

In case you use SciKit-GStat in other software or scientific publications, please reference this module. It is published and has a DOI. It can be cited as:

> **Mirko Mälicke, Egil Möller, Helge David Schneider, & Sebastian Müller. (2021, May 28).** mmaelicke/scikit-gstat: A scipy flavoured geostatistical variogram analysis toolbox (Version v0.6.0). Zenodo. http://doi.org/10.5281/zenodo.4835779

2.1 Installation

The package can be installed directly from the Python Package Index or GitHub. The version on GitHub might be more recent, as only stable versions are uploaded to the Python Package Index.

2.1.1 PyPI

The version from PyPI can directly be installed using pip

```
pip install scikit-gstat
```

2.1.2 GitHub

The most recent version from GitHub can be installed like:

```
git clone git@github.com:mmaelicke/scikit-gstat
cd scikit-gstat
pip install -e .
```

2.1.3 Conda-Forge

Since version *0.5.5*, SciKit-GStat is available on Conda-Forge. You can install it like:

```
conda install -c conda-forge scikit-gstat
```

2.1.4 Note

On Windows, you might run into problems installing all requirements in a clean Python environment, especially if C++ redistributables are missing. This can happen i.e. on *bare* VMs and the compilation of libraries required by scipy, numpy or numba package are the ones failing. In these cases, install the libraries first, and then SciKit-GStat or move to the conda-forge package

```
conda install numpy, scipy, numba
```

2.2 Getting Started

2.2.1 Load the class and data

The main class of scikit-gstat is the Variogram. It can directly be imported from the module, called skgstat. The main class can easily be demonstrated on the data module available with version >=*0.5.5*.

```
In [1]: import skgstat as skg

In [2]: import numpy as np

In [3]: import matplotlib.pyplot as plt

In [4]: plt.style.use('ggplot')

In [5]: data = skg.data.pancake(N=500, seed=42)

In [6]: print(data.get('origin'))
Image of a pancake with apparent spatial structure.
    Copyright Mirko Mälicke, 2020. If you use this data,
    cite SciKit-GStat: https://doi.org/10.5281/zenodo.1345584


In [7]: coordinates, values = data.get('sample')
```

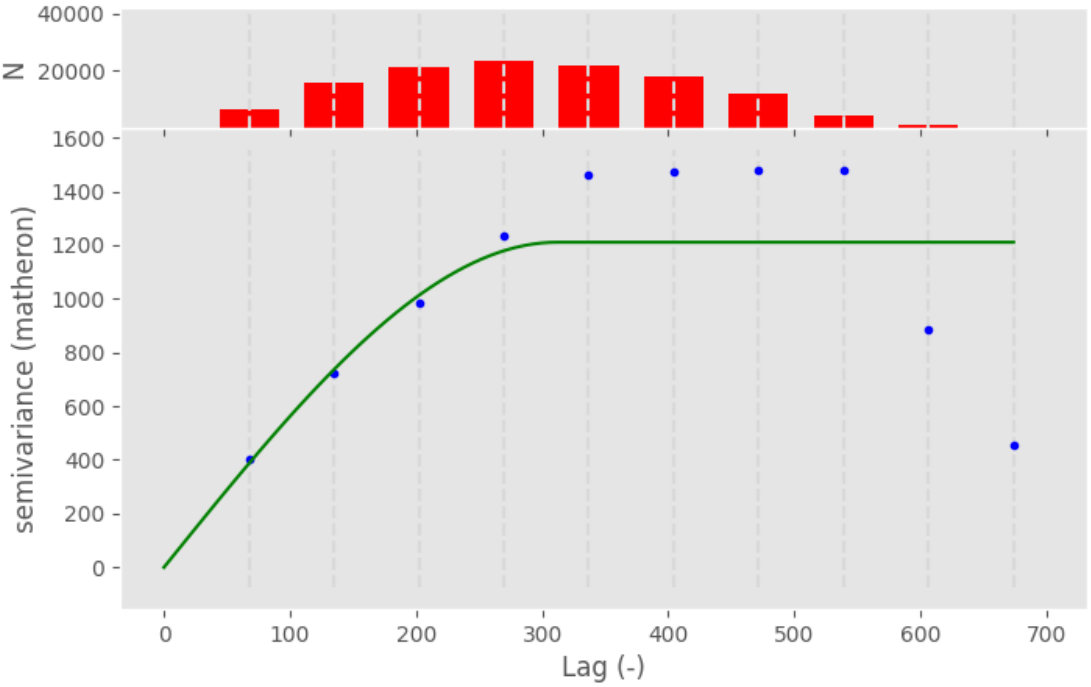
The Variogram needs at least an array of coordinates and an array of values on instantiation.

```
In [8]: V = skg.Variogram(coordinates=coordinates, values=values)

In [9]: print(V)
spherical Variogram
-------------------
Estimator:         matheron

Effective Range:   311.76

Sill:              1210.20

Nugget:            0.00
```

2.2.2 Plot

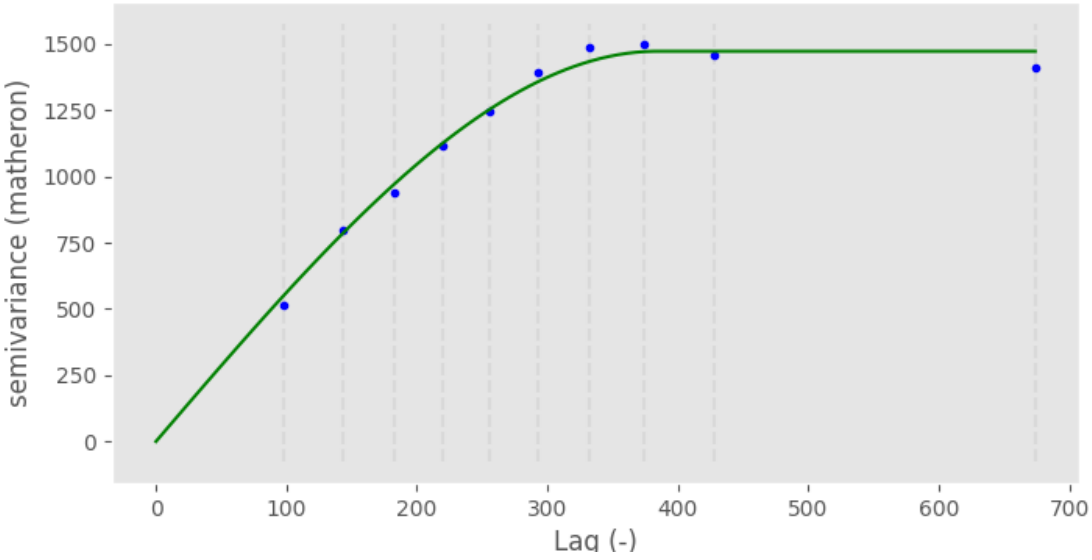
The Variogram class has its own plotting method.

```
In [10]: V.plot()
Out[10]: <Figure size 800x500 with 2 Axes>

In [11]: plt.close()
```



With version 0.2, the histogram plot can also be disabled. This is most useful, when the binning method for the lag classes is changed from *'even'* step classes to *'uniform'* distribution in the lag classes.

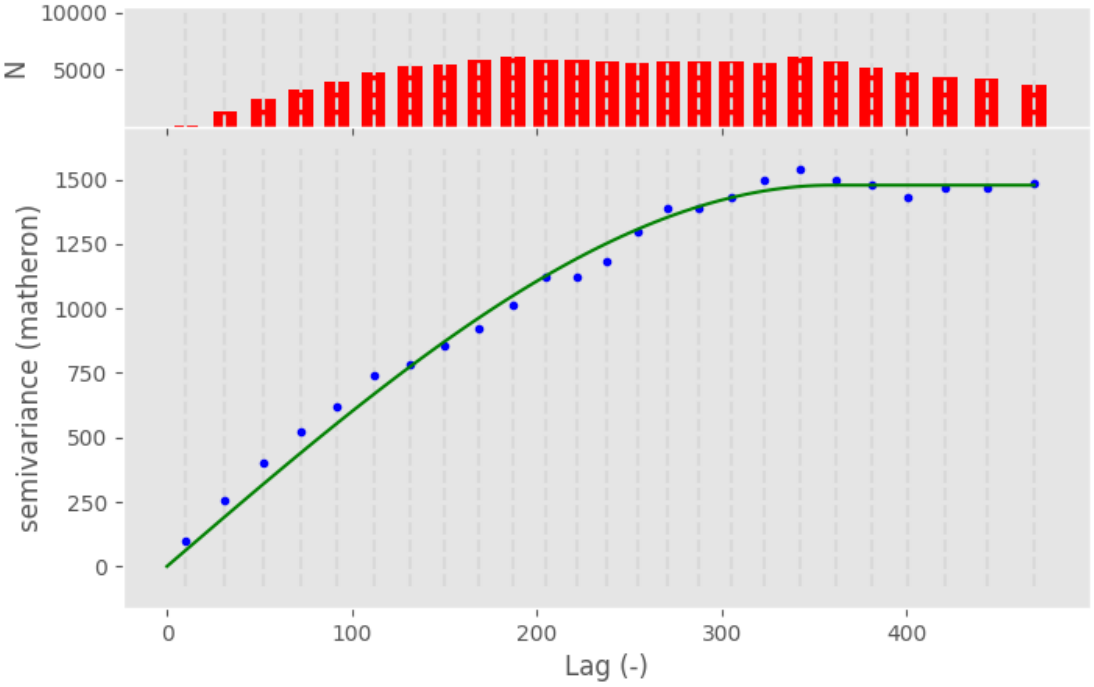
```
In [12]: V.set_bin_func('uniform')

In [13]: V.plot(hist=False)
Out[13]: <Figure size 800x400 with 1 Axes>

In [14]: plt.close()
```



2.2.3 Mutating

One of the main strenghs of *Variogram* is its ability to change arguments in place. Any dependent result or parameter will be invalidated and re-caluculated. You can i.e. increase the number of lag classes:

```
In [15]: V.n_lags = 25

In [16]: V.maxlag = 500

In [17]: V.bin_func = 'kmeans'

In [18]: V.plot()
Out[18]: <Figure size 800x500 with 2 Axes>

In [19]: plt.close()
```



Note, how the experimental variogram was updated and the model was fitted to the new data automatically.

2.3 User Guide

This user guide shall help you getting started with `scikit-gstat` package along with a more general introduction to variogram analysis.

2.3.1 Introduction

General

This user guide part of `scikit-gstat`'s documentation is meant to be an user guide to the functionality offered by the module along with a more general introduction to geostatistical concepts. The main use case is to hand this description to students learning geostatistics, whenever `scikit-gstat` is used. But before introducing variograms, the more general question what geostatistics actually are has to be answered.

Note: This user guide is meant to be an **introduction** to geostatistics. In case you are already familiar with the topic, you can skip this section.

What is geostatistics?

The basic idea of geostatistics is to describe and estimate spatial covariance, or correlation, in a set of point data. While the main tool, the semi-variogram, is quite easy to implement and use, a lot of important assumptions are underlying it. The typical application is geostatistics is an interpolation. Therefore, although using point data, a basic concept is to understand this point data as a sample of a (spatially) continuous variable that can be described as a random field rf, or to be more precise, a Gaussian random field in many cases. The most fundamental assumption in geostatistics is that any two values x_i and x_{i+h} are more similar, the smaller h is, which is a separating distance on the random field. In other words: *close observation points will show higher covariances than distant points*. In case this most fundamental conceptual assumption does not hold for a specific variable, geostatistics will not be the correct tool to analyse and interpolate this variable.

One of the most easiest approaches to interpolate point data is to use IDW (inverse distance weighting). This technique is implemented in almost any GIS software. The fundamental conceptual model can be described as:

$$Z_u = \frac{\sum_i^N w_i * Z(i)}{N}$$

where Z_u is the value of rf at a non-observed location with N observations around it. These observations get weighted by the weight w_i, which can be calculated like:

$$w_i = \frac{1}{||\overrightarrow{ux_i}||}$$

where u is the unobserved point and x_i is one of the sample points. Thus, $||\overrightarrow{ux_i}||$ is the 2-norm of the vector between the two points: the Euclidean distance in the coordinate space (which by no means has to be limited to the $\mathbb{R}^2$ case).

This basically describes a concept, where a value of the random field is estimated by a distance-weighted mean of the surrounding points. As close points shall have a higher impact, the inverse distance is used and thus the name of **inverse distance weighting**.

In the case of geostatistics this basic model still holds, but is extended. Instead of depending the weights exclusively on the separating distance, a weight will be derived from a variance over all values that are separated by a similar distance. This has the main advantage of incorporating the actual (co)variance found in the observations and basing the interpolation on this (co)variance, but comes at the cost of some strict assumptions about the statistical properties of the sample. Elaborating and assessing these assumptions is one of the main challenges of geostatistics.

Geostatistical Tools

Geostatistics is a wide field spanning a wide variety of disciplines, like geology, biology, hydrology or geomorphology. Each discipline defines their own set of tools, and apparently definitions, and progress is made until today. It is not the objective of `scikit-gstat` to be a comprehensive collection of all available tools. The objective is more to offer some common and also more sophisticated tools for variogram analysis. Thus, when useing `scikit-gstat`, you typically need another library for the actual application, like interpolation. In most cases that will be gstools. However, one can split geostatistics into three main fields, each of it with its own tools:

- **variography:** with the variogram being the main tool, the variography focuses on describing, visualizing and modelling covariance structures in space and time.
- **kriging:** is a family of interpolation methods, that utilize a variogram to estimate the kriging weights as sketched above.
- **geostatistical simulation:** is aiming on generate random fields that fit a given set of observations or a pre-defined variogram or covariance function.

Note: I am not planning to implement tools from all three fields. You can rather use one of the interfaces, like

Variogram.to_gstools to export a variogram to another library, that covers kriging and spatial random field generation in great detail.

How to use this guide

The main idea behind the user-guide is to introduce geostatistics at the example of SciKit-GStat. The module has a growing collection of data examples, that are used throughout the documentation. They can be loaded from the *data* submodule. Each function will return a dictionary of the actual sample and a brief description.

Note: Any data sample included has an origin and an owner. While they are all distributed under open licenses, you have to check the description for data ownership as all used licenses force you to attribute the owner.

```
In [1]: import skgstat as skg

In [2]: skg.data.aniso(N=20)
Out[2]:
{'sample': (array([[475, 386],
         [365, 358],
         [365, 100],
         [397, 487],
         [436, 419],
         [280, 380],
         [ 28,  64],
         [311,  44],
         [131, 327],
         [305, 256],
         [247, 216],
         [485,  42],
         [ 96, 225],
         [137, 429],
         [ 43,  47],
         [432, 367],
         [ 12, 393],
         [212, 219],
         [333, 348],
         [115, 263]]),
  array([168, 163, 148, 156, 161, 176, 125,  98, 198, 182, 172, 132,  99,
         212, 146, 169, 177, 204, 146,  68], dtype=uint8)),
 'origin': 'Random field greyscale image with geometric anisotropy.\n    The anisotropy␣
↪in North-East direction has a factor of 3. The random\n    field was generated using␣
↪gstools.\n    Copyright Mirko Mälicke, 2020. If you use this data,\n    cite SciKit-
↪GStat: https://doi.org/10.5281/zenodo.1345584\n    '}
```

These samples contain a coordinate and a value array.

2.3.2 Variography

The variogram

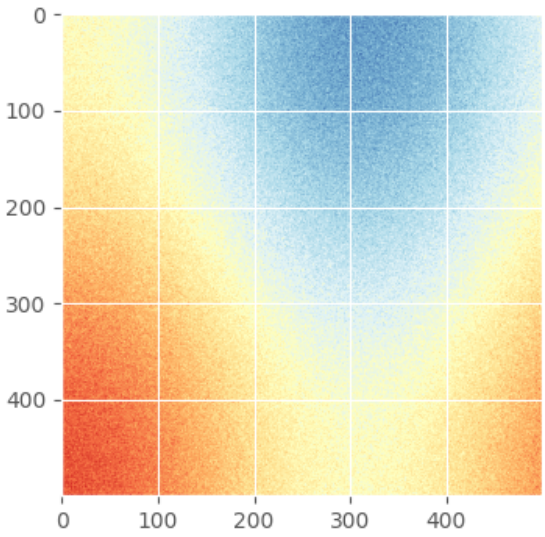
General

We start by constructing a random field and sample it. Without knowing about random field generators, an easy way to go is to stick two trigonometric functions together and add some noise. There should be clear spatial correlation apparent.

```
In [1]: import numpy as np

In [2]: import matplotlib.pyplot as plt

In [3]: plt.style.use('ggplot')

In [4]: from pprint import pprint
```

This field could look like

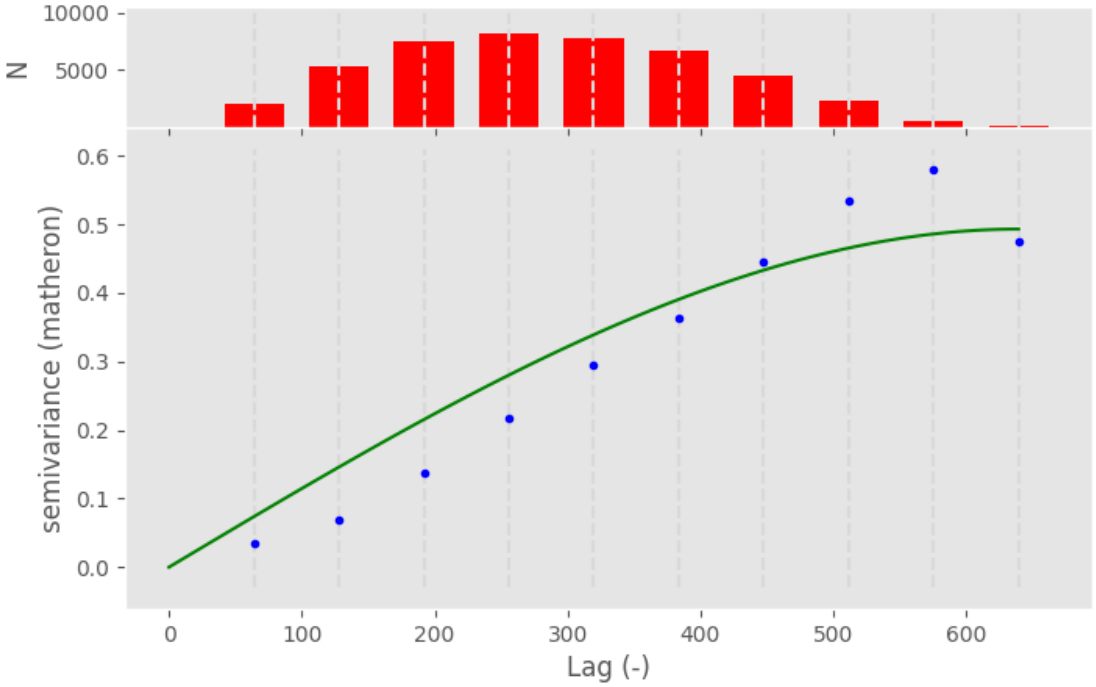
```
# apply the function to a meshgrid and add noise
In [5]: xx, yy = np.mgrid[0:0.5 * np.pi:500j, 0:0.8 * np.pi:500j]

In [6]: np.random.seed(42)

# generate a regular field
In [7]: _field = np.sin(xx)**2 + np.cos(yy)**2 + 10

# add noise
In [8]: np.random.seed(42)

In [9]: z = _field + np.random.normal(0, 0.15, (500,  500))

In [10]: plt.imshow(z, cmap='RdYlBu_r')
Out[10]: <matplotlib.image.AxesImage at 0x7f9db9eb4438>

In [11]: plt.close()
```



Using scikit-gstat

It's now easy and straightforward to calculate a variogram using `scikit-gstat`. We need to sample the field and pass the coordinates and value to the *`Variogram Class`*.

```
In [12]: import skgstat as skg

# random coordinates
In [13]: np.random.seed(42)

In [14]: coords = np.random.randint(0, 500, (300, 2))

In [15]: values = np.fromiter((z[c[0], c[1]] for c in coords), dtype=float)

In [16]: V = skg.Variogram(coords, values)

In [17]: V.plot()
Out[17]: <Figure size 800x500 with 2 Axes>
```



From my personal point of view, there are three main issues with this approach:

- If one is not an geostatistics expert, one has no idea what he actually did and can see in the presented figure.
- The figure includes an spatial model, one has no idea if this model is suitable and fits the observations (wherever they are in the figure) sufficiently.
- Refer to the `__init__` method of the Variogram class. There are 10+ arguments that can be set optionally. The default values will most likely not fit your data and requirements.

Therefore one will have to understand how the *Variogram Class* works along with some basic knowledge about variography in oder to be able to properly use `scikit-gstat`.

However, what we can discuss from the figure, is what a variogram actually is. At its core it relates a dependent variable to an independent variable and, in a second step, tries to describe this relationship with a statistical model. This model on its own describes some of the spatial properties of the random field and can further be utilized in an interpolation to select nearby points and weight them based on their statistical properties.

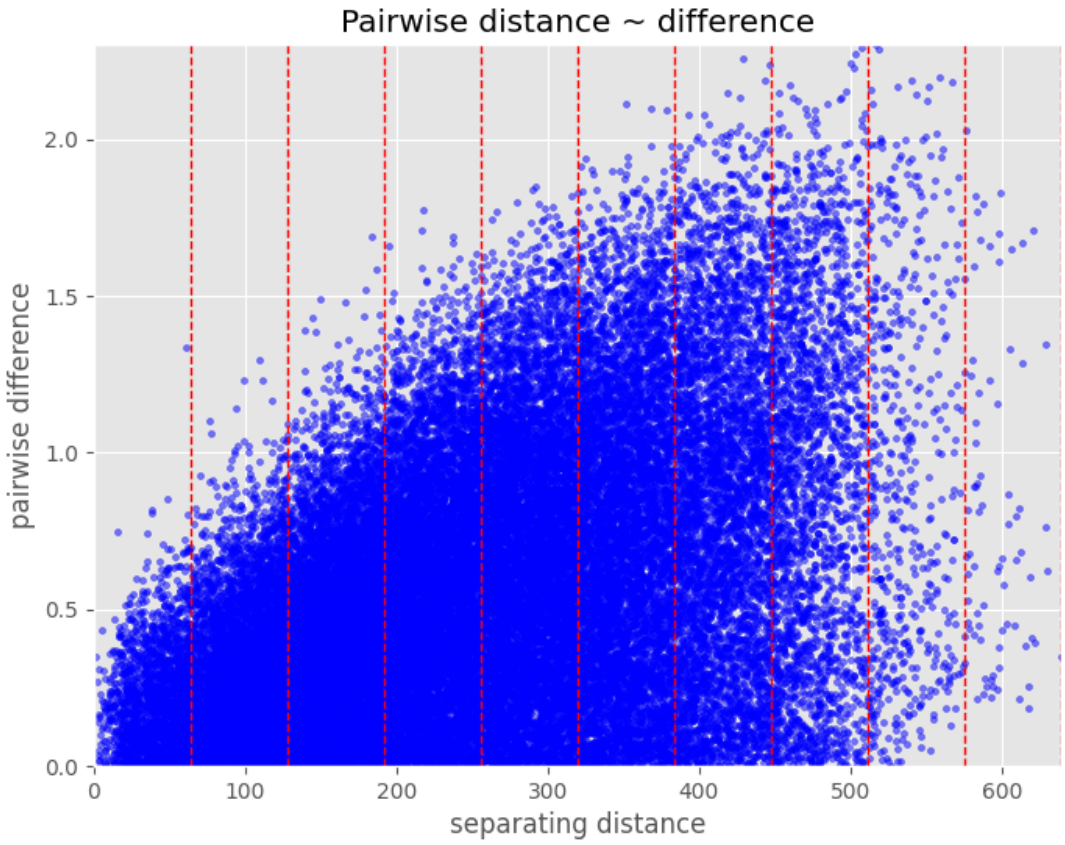
The variogram relates the separating distance between two observation points to a measure of observation similarity at that given distance. Our expectation is that variance is increasing with distance, what can basically be seen in the presented figure.

Distance

Consider the variogram figure from above, with which an *independent* and *dependent* variable was introduced. In statistics it is common to use *dependent* variable as an alias for *target variable*, because its value is dependent on the state of the independent variable. In the case of a variogram, this is the metric of variance on the y-axis. In geostatistics, the independent variable is usually a measure of Euclidean distance.

Consider observations taken in the environment, it is fairly unlikely to find two pairs of observations where the separating distance between the coordinates match exactly the same value. Therefore one has to group all point pairs at the same distance *lag* together into one group, or *bin*. Beside practicability, there is also another reason, why one would want to group point pairs at similar separating distances together into one bin. Consider the plot below, which shows the difference in value over the distance for all point pair combinations that can be formed for a given sample. The `Variogram Class` has a function for that: `distance_difference_plot`:

```
In [18]: V.distance_difference_plot()
Out[18]: <Figure size 800x600 with 1 Axes>

In [19]: plt.close()
```



While it is possible to see the increasing variability with increasing distance here quite nicely, it is not possible to guess meaningful moments for the distributions at different distances. Last but not least, to derive a simple model as presented in the variogram figure above by the green line, we have to be able to compress all values at a given distance lag to one estimation of variance. This would not be possible from the the figure above.

Note: There are also procedures that can fit a model directly based on unbinned data. As none of these methods is implemented into `scikit-gstat`, they will not be discussed here. If you need them, you are more than welcome to implement them.

Binning the separating distances into distance lags is therefore a crucial and most important task in variogram analysis. The final binning must discretizise the distance lag at a meaningful resolution at the scale of interest while still holding enough members in the bin to make valid estimations. Often this is a trade-off relationship and one has to find a suitable compromise.

Before diving into binning, we have to understand how the *Variogram Class* handles distance data. The distance calculation can be controlled by the `dist_func` argument, which takes either a string or a function. The default value is *'euclidean'*. This value is directly passed down to the `pdist` as the *metric* argument. Consequently, the distance data is stored as a distance matrix for all input locations passed to *Variogram* on creation. To be more precise, only the upper triangle is stored in a `array` with the distance values sorted row-wise. Consider this very straightforward set of locations:

```
In [20]: locations = [[0,0], [0,1], [1,1], [1,0]]

In [21]: V = skg.Variogram(locations, [0, 1, 2, 1], normalize=False)

In [22]: V.distance
Out[22]: array([1.   , 1.414, 1.   , 1.   , 1.414, 1.   ])

# turn into a 2D matrix again
In [23]: from scipy.spatial.distance import squareform

In [24]: print(squareform(V.distance))
[[0.    1.    1.414 1.   ]
 [1.    0.    1.    1.414]
 [1.414 1.    0.    1.   ]
 [1.    1.414 1.    0.   ]]
```

Binning

As already mentioned, in real world observation data, there won't be two observation location pairs at **exactly** the same distance. Thus, we need to group information about point pairs at **similar** distance together, to learn how similar their observed values are. With a *Variogram*, we will basically try to find and describe some systematic statistical behavior from these similarities. The process of grouping distance data together is called *binning*.

`scikit-gstat` has many different methods for binning distance data. They can be set using the *bin_func* attribute. You have to pass the name of the method. The available methods are:

- *even* - evenly spaced bins
- *uniform* - same sample sized bins
- *sturges* - derive number of bins by Sturge's rule
- *scott* - derive number of bins by Scotts's rule
- *sqrt* - derive number of bins by sqaureroot rule
- *doane* - derive number of bins by Doane's rule
- *fd* - derive number of bins by Freedmann-Diaconis estimator

- *kmeans* - derive bins by K-Means clustering
- *ward* - derive bins by hierachical clustering and Ward's criterion
- *stable_entropy* - derive bins from stable entropy setting

`['even', 'uniform', 'kmeans', 'ward', 'stable_entropy']` methods will use two parameters to calculate the bins from the distance matrix: *n_lags*, the amount of bins, and *maxlag*, the maximum distance lag to be considered. `['sturges', 'scott', 'sqrt', 'fd', 'doane']` will only use *maxlag* to derive *n_lags* from statistical properties of the distance matrix. The *even* method will then form *n_lags* bins from `0` to *maxlag* of same width. The *uniform* method will form the same amount of classes within the same range, using the same point pair count in each bin. The following example illustrates this:

```
In [25]: from skgstat.binning import even_width_lags, uniform_count_lags

In [26]: from scipy.spatial.distance import pdist

In [27]: loc = np.random.normal(50, 10, size=(30, 2))

In [28]: distances = pdist(loc)
```

Now, look at the different bin edges for the calculated dummy distance matrix:

```
In [29]: even_width_lags(distances, 10, 250)
Out[29]:
(array([ 4.405,  8.809, 13.214, 17.618, 22.023, 26.427, 30.832, 35.237,
        39.641, 44.046]),
 None)

In [30]: uniform_count_lags(distances, 10, 250)
Out[30]:
(array([ 7.198, 10.432, 12.34 , 15.147, 17.693, 20.381, 23.013, 26.278,
        30.641, 44.046]),
 None)
```

Using the *Variogram* you can see how the setting of different binning methods will update the *Variogram.bins* and eventually *n_lags*:

```
In [31]: test = skg.Variogram(
   ....:     *skg.data.pancake().get('sample'),  # use some sample data
   ....:     n_lags=25,                          # set 25 classes
   ....:     bin_func='even'
   ....: )
   ....:

In [32]: print(test.bins)
[ 26.955  53.91   80.865 107.82  134.775 161.73  188.684 215.639 242.594
 269.549 296.504 323.459 350.414 377.369 404.324 431.279 458.234 485.189
 512.144 539.099 566.053 593.008 619.963 646.918 673.873]
```

Now, we can easily switch to a method that will derive a new value for *n_lags*. That will auto-update *Variogram.bins* and *n_lags*.

```
# sqrt will very likely estimate way more bins
In [33]: test.bin_func = 'sqrt'
```

(continues on next page)

(continued from previous page)

```
In [34]: print(f'Auto-derived {test.n_lags} bins.')
Auto-derived 354 bins.

In [35]: print(V.bins)
[0.141 0.283 0.424 0.566 0.707 0.849 0.99  1.131 1.273 1.414]
```

Observation differences

By the term *observation differences*, the distance between the observed values are meant. As already layed out, the main idea of a variogram is to systematially relate similarity of observations to their spatial proximity. The spatial part was covered in the sections above, finalized with the calculation of a suitable binning of all distances. We want to relate exactly these bins to a measure of similarity of all observation point pairs that fall into this bin.

That's basically it. We need to do three more steps to come up with *one* value per bin, statistically describing the similarity at that distance.

1. Find all point pairs that fall into a bin
2. Calculate the *distance* (difference) of the observed values
3. Describe all differences by one number

Finding all pairs within a bin is straightforward. We already have the bin edges and all distances between all possible observation point combinations (stored in the distance matrix). Using the `squareform` function of scipy, we *could* turn the distance matrix into a 2D version. Then the row and column indices align with the values indices. However, *`Variogram`* implements a method for doing mapping a bit more efficiently.

A `array` of bin groups for each point pair that is indexed exactly like the `distance` array can be obtained by *`lag_groups`*.

This will be illustrated by some sample data from the *`data`* submodule.

```
In [36]: coords, vals = skg.data.pancake(N=200).get('sample')

In [37]: V = skg.Variogram(
   ....:          coords,
   ....:          vals,
   ....:          n_lags=25
   ....:      )
   ....:

In [38]: V.maxlag = 500
```

Then, you can compare the first 10 point pairs from the distance matrix to the first 10 elements returned by the *`lag_groups function`*.

```
# first 10 distances
In [39]: V.distance[:10].round(1)
Out[39]:
array([179.9,  51. , 151.2, 156.1,  12.8, 162.3, 142.4, 411.8, 156.5,
       257. ])

# first 10 groups
```

(continues on next page)

(continued from previous page)

```
In [40]: V.lag_groups()[:10]
Out[40]: array([ 8,  2,  7,  7,  0,  8,  7, 20,  7, 12])
```

Now, we need the actual *Variogram.bins* to verify the grouping.

```
In [41]: V.bins
Out[41]:
array([ 20.,  40.,  60.,  80., 100., 120., 140., 160., 180., 200., 220.,
       240., 260., 280., 300., 320., 340., 360., 380., 400., 420., 440.,
       460., 480., 500.])
```

The elements `[2, 3, 6, 8]``are grouped into group ``7`. Their distance values are `[151.2, 156.1, 142.4, 156.5]`. The grouping starts with `0`, therefore the corresponding upper bound of the bin is at index `7` and the lower at `6`. The bin edges are therefore `140. < x < 160.`. Consequently, the binning and grouping worked fine.

If you want to access all value pairs at a given group, it would of course be possible to use the machanism above to find the correct points. However, *Variogram* offers an iterator that already does that for you: *lag_classes*. This iterator will yield all pair-wise observation value differences for the bin of the actual iteration. The first iteration (index = 0, if you wish) will yield all differences of group id `0`.

Note: *lag_classes* will yield the difference in value of observation point pairs, not the pairs themselves.

```
In [42]: for i, group in enumerate(V.lag_classes()):
   ....:     print('[Group %d]: %.2f' % (i, np.mean(group)))
   ....:
[Group 0]: 10.36
[Group 1]: 19.10
[Group 2]: 24.20
[Group 3]: 29.70
[Group 4]: 31.24
[Group 5]: 35.21
[Group 6]: 36.84
[Group 7]: 40.44
[Group 8]: 39.32
[Group 9]: 38.65
[Group 10]: 38.36
[Group 11]: 43.47
[Group 12]: 42.45
[Group 13]: 44.85
[Group 14]: 45.26
[Group 15]: 43.83
[Group 16]: 43.36
[Group 17]: 43.98
[Group 18]: 43.80
[Group 19]: 43.74
[Group 20]: 46.80
[Group 21]: 44.70
[Group 22]: 44.65
[Group 23]: 44.49
[Group 24]: 50.13
```

The only thing that is missing for a variogram is that we will not use the arithmetic mean to describe the realtionship.

Experimental variograms

The last stage before a variogram function can be modeled is to define an experimental variogram, also known as *empirical variogram*, which will be used to parameterize a variogram model. However, the expermental variogram already contains a lot of information about spatial relationships in the data. Therefore, it's worth looking at more closely. Last but not least a poor expermental variogram will also affect the variogram model, which is ultimatively used to interpolate the input data.

Note: In geostatistical literature you can find the terms *experimental* and *empirical* variogram. Both refer to the varigoram estimated from a sample. In SciKit-GStat the term `experimental` variogram is used for the estimated semi-variances solely. Thus, this is a 1D structure (of length `n_lags`). The term *empirical* (`Variogram.get_empirical`) is used for the combination of `bins` and `experimental`, thus it is a tuple of two 1D arrays.

The previous sections summarized how distance is calculated and handeled by the `Variogram class`. The `lag_groups` function makes it possible to find corresponding observation value pairs for all distance lags. Finally the last step will be to use a more suitable estimator for the similarity of observation values at a specific lag. In geostatistics this estimator is called semi-variance and the the most popular estimator is called *Matheron estimator*. By default, the `Matheron` estimator will be used. It is defined as

$$\gamma(h) = \frac{1}{2N(h)} * \sum_{i=1}^{N(h)} (x)^2$$

with:

$$x = Z(x_i) - Z(x_{i+h})$$

where $Z(x_i)$ is the observation value at the i-th location x_i. h is the distance lag and $N(h)$ is the number of point pairs at that lag.

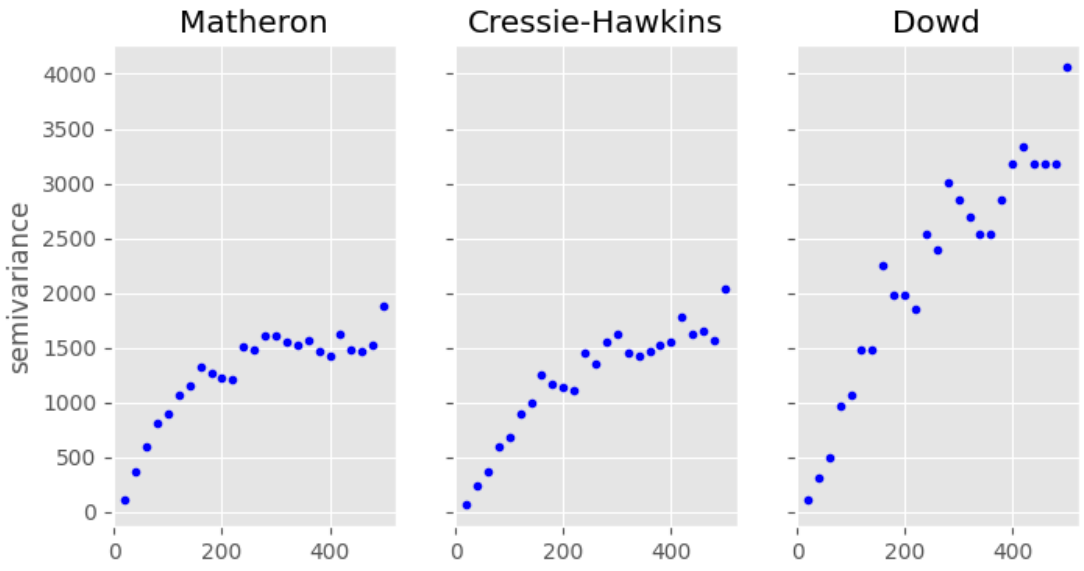
You will find more estimators in `skgstat.estimators`. There is the `Cressie-Hawkins`, which is more robust to extreme values. Other so called robust estimators are `Dowd` or `Genton`. The remaining are experimental estimators and should only be used with caution. Let's compare them directly. You could use the code from the last section to group the pair-wise value differencens into lag groups and apply the formula for each estimator. In the example below, we will iteratively change the `Variogram` instance used so far to achieve this:

```
In [43]: fig, _a = plt.subplots(1, 3, figsize=(8,4), sharey=True)

In [44]: axes = _a.flatten()

In [45]: axes[0].plot(V.bins, V.experimental, '.b')
Out[45]: [<matplotlib.lines.Line2D at 0x7f9dba00e198>]

In [46]: V.estimator = 'cressie'

In [47]: axes[1].plot(V.bins, V.experimental, '.b')
Out[47]: [<matplotlib.lines.Line2D at 0x7f9db9867f28>]

In [48]: V.estimator = 'dowd'

In [49]: axes[2].plot(V.bins, V.experimental, '.b')
Out[49]: [<matplotlib.lines.Line2D at 0x7f9db9867b00>]

In [50]: axes[0].set_ylabel('semivariance')
```

(continues on next page)

(continued from previous page)

```
Out[50]: Text(0, 0.5, 'semivariance')

In [51]: axes[0].set_title('Matheron')
Out[51]: Text(0.5, 1.0, 'Matheron')

In [52]: axes[1].set_title('Cressie-Hawkins')
Out[52]: Text(0.5, 1.0, 'Cressie-Hawkins')

In [53]: axes[2].set_title('Dowd')
Out[53]: Text(0.5, 1.0, 'Dowd')

In [54]: fig.show()
```



Note: With this example it is not a good idea to use the Gention estimator, as it takes a long time to calculate the experimental variogram.

Variogram models

The last step to describe the spatial pattern in a data set using variograms is to model the empirically observed and calculated experimental variogram with a proper mathematical function. Technically, this setp is straightforward. We need to define a function that takes a distance value and returns a semi-variance value. One big advantage of these models is, that we can assure different things, like positive definitenes. Most models are also monotonically increasing and approach an upper bound. Usually these models need three parameters to fit to the experimental variogram. All three parameters have a meaning and are usefull to learn something about the data. This upper bound a model approaches is called *sill*. The distance at which 95% of the sill are approached is called the *effective range*. That means, the range is the distance at which observation values do **not** become more dissimilar with increasing distance. They are statistically independent. That also means, it doesn't make any sense to further describe spatial relationships of observations further apart with means of geostatistics. The last parameter is the *nugget*. It is used to add semi-variance to all values. Graphically that means to *move the variogram up on the y-axis*. The nugget is the semi-variance modeled on the 0-distance lag. Compared to the sill it is the share of variance that cannot be described spatially.

Warning: There is a very important design decision underlying all models in SciKit-GStat. All models take the *effective range* as a parameter. If you look into literature, there is also the **model** parameter *range*. That can be very confusing, hence it was decided to fit models on the *effective range*. You can translate one into the other quite easily. Transformation factors are reported in literature, but not commonly the same ones are used. Finally, the transformation is always coded into SciKit-GStat's `models`, even if it's a 1:1 *transformation*.

The spherical model

The sperical model is the most commonly used variogram model. It is characterized by a very steep, exponential increase in semi-variance. That means it approaches the sill quite quickly. It can be used when observations show strong dependency on short distances. It is defined like:

$$\gamma = b + C_0 * \left(1.5 * \frac{h}{r} - 0.5 * \frac{h}{r}^3\right)$$

if h < r, and

$$\gamma = b + C_0$$

else. `b` is the nugget, C_0 is the sill, `h` is the input distance lag and `r` is the effective range. That is the range parameter described above, that describes the correlation length. Many other variogram model implementations might define the range parameter, which is a variogram parameter. This is a bit confusing, as the range parameter is specific to the used model. Therefore I decided to directly use the *effective range* as a parameter, as that makes more sense in my opinion.

As we already calculated an experimental variogram and find the spherical model in the `models` sub-module, we can utilize e.g. `curve_fit` from scipy to fit the model using a least squares approach.

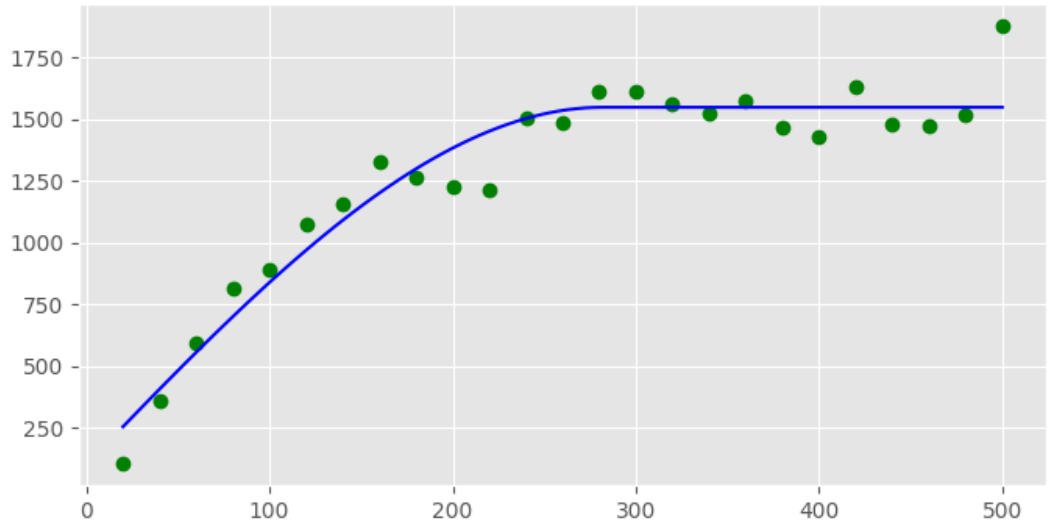
Note: With the given example, the default usage of `curve_fit` will use the Levenberg-Marquardt algorithm, without initial guess for the parameters. This will fail to find a suitable range parameter. Thus, for this example, you need to pass an initial guess to the method.

```
In [55]: from skgstat import models

# set estimator back
In [56]: V.estimator = 'matheron'

In [57]: V.model = 'spherical'

In [58]: xdata = V.bins

In [59]: ydata = V.experimental

In [60]: from scipy.optimize import curve_fit

# initial guess - otherwise lm will not find a range
In [61]: p0 = [np.mean(xdata), np.mean(ydata), 0]

In [62]: cof, cov =curve_fit(models.spherical, xdata, ydata, p0=p0)
```

Here, *cof* are now the coefficients found to fit the model to the data.

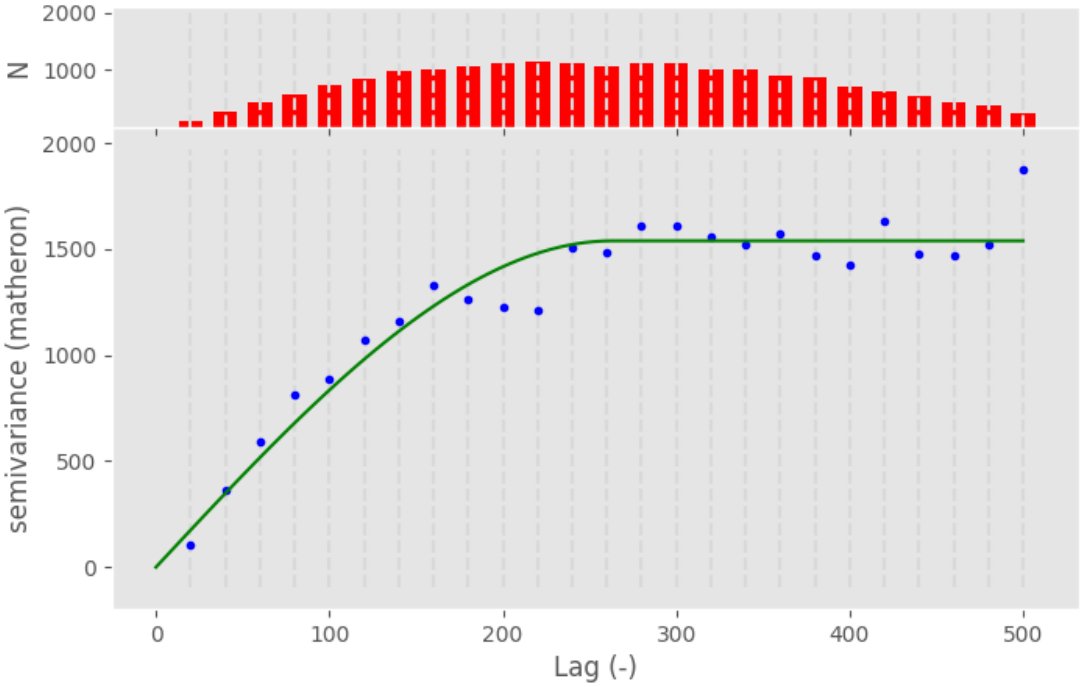
```
In [63]: print("range: %.2f   sill: %.f   nugget: %.2f" % (cof[0], cof[1], cof[2]))
range: 281.73   sill: 1446   nugget: 101.66
```

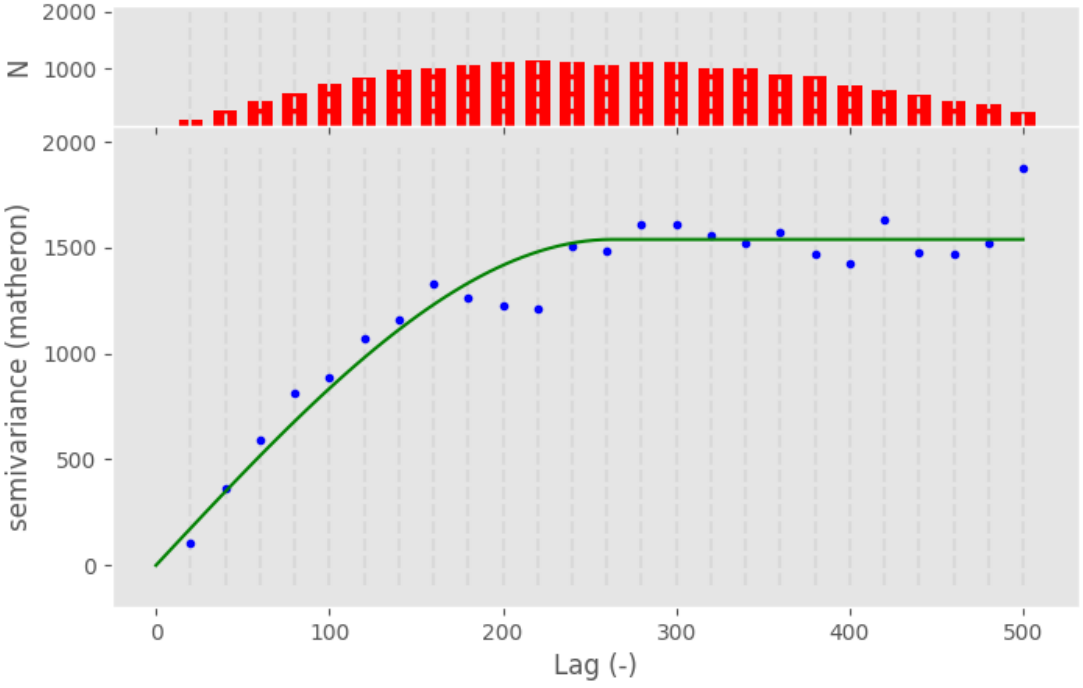
```
In [64]: xi =np.linspace(xdata[0], xdata[-1], 100)

In [65]: yi = [models.spherical(h, *cof) for h in xi]

In [66]: plt.plot(xdata, ydata, 'og')
Out[66]: [<matplotlib.lines.Line2D at 0x7f9db93a42e8>]

In [67]: plt.plot(xi, yi, '-b');
```



The *Variogram Class* does in principle the same thing. The only difference is that it tries to find a good initial guess for the parameters and limits the search space for parameters. That should make the fitting more robust. Technically, we used the Levenberg-Marquardt algorithm above. That's a commonly used, very fast least squares implementation. However, sometimes it fails to find good parameters, as it is unbounded and *searching* an invalid parameter space. The default for *Variogram* is Trust-Region Reflective (TRF), which is also the default for *Variogram*. It uses a valid parameter space as bounds and therefore won't fail in finding parameters. You can, hoever, switch to Levenberg-Marquardt by setting the *Variogram.fit_method* to 'lm'.

```
In [68]: V.fit_method ='trf'

In [69]: V.plot();

In [70]: pprint(V.parameters)
[263.12497861564043, 1539.0278111437954, 0]

In [71]: V.fit_method ='lm'

In [72]: V.plot();

In [73]: pprint(V.parameters)
[263.12045635155715, 1539.023039647012, 0]
```





Note: In this example, the fitting method does not make a difference at all. Generally, you can say that Levenberg-Marquardt is faster and TRF is more robust.

Exponential model

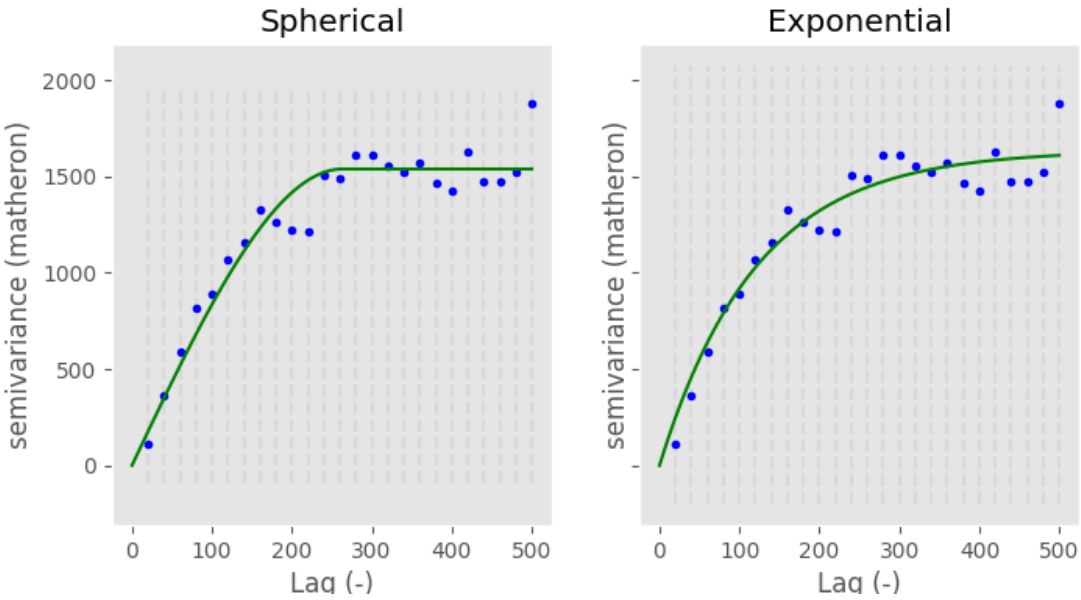
The exponential model is quite similar to the spherical one. It models semi-variance values to increase exponentially with distance, like the spherical. The main difference is that this increase is not as steep as for the spherical. That means, the effective range is larger for an exponential model, that was parameterized with the same range parameter.

Note: Remember that SciKit-GStat uses the *effective range* to overcome this confusing behaviour.

Consequently, the exponential can be used for data that shows a way too large spatial correlation extent for a spherical model to capture.

Applied to the data used so far, you can see the difference between the two models quite nicely:

```
In [74]: fig, axes = plt.subplots(1, 2, figsize=(8, 4), sharey=True)

In [75]: axes[0].set_title('Spherical')
Out[75]: Text(0.5, 1.0, 'Spherical')

In [76]: axes[1].set_title('Exponential')
Out[76]: Text(0.5, 1.0, 'Exponential')

In [77]: V.fit_method = 'trf'

In [78]: V.plot(axes=axes[0], hist=False)
Out[78]: <Figure size 800x400 with 2 Axes>

# switch the model
In [79]: V.model = 'exponential'

In [80]: V.plot(axes=axes[1], hist=False);
```



Keep in mind how important the theoretical model is. We will use it for interpolation later on and the quality of this interpolation will primarily rely on the fit of the model to the experimental data smaller than the effective range. From

the example above it is quite hard to tell, which is the correct one. Also, the goodness of fit is quite comparable:

```
# spherical
In [81]: V.model = 'spherical'

In [82]: rmse_sph = V.rmse

In [83]: r_sph = V.describe().get('effective_range')

# exponential
In [84]: V.model = 'exponential'

In [85]: rmse_exp = V.rmse

In [86]: r_exp = V.describe().get('effective_range')

In [87]: print('Spherical   RMSE: %.2f' % rmse_sph)
Spherical   RMSE: 114.70

In [88]: print('Exponential RMSE: %.2f' % rmse_exp)
Exponential RMSE: 104.83
```

But the difference in effective range is more pronounced:

```
In [89]: print('Spherical effective range:    %.1f' % r_sph)
Spherical effective range:    263.1

In [90]: print('Exponential effective range:  %.1f' % r_exp)
Exponential effective range:  363.3
```

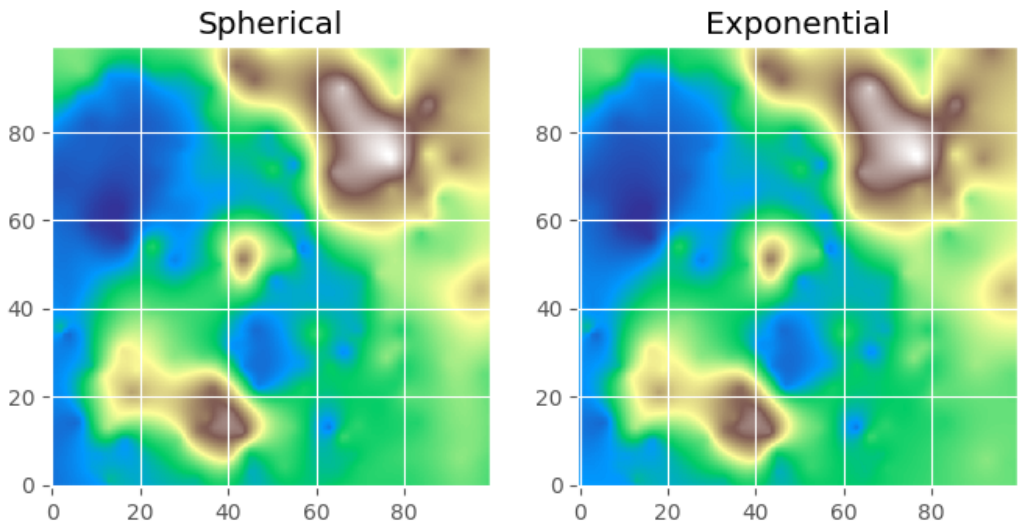
Finally, we can use both models to perform a Kriging interpolation.

```
In [91]: fig, axes = plt.subplots(1, 2, figsize=(8, 4))

In [92]: V.model = 'spherical'

In [93]: krige1 = V.to_gs_krige()

In [94]: V.model = 'exponential'

In [95]: krige2 = V.to_gs_krige()

# build a grid
In [96]: x = y = np.arange(0, 500, 5)

# apply
In [97]: field1, _ = krige1.structured((x, y))

In [98]: field2, _ = krige2.structured((x, y))

# use the same bounds
In [99]: vmin = np.min((field1, field2))

In [100]: vmax = np.max((field1, field2))
```

(continues on next page)

(continued from previous page)

```
# plot
In [101]: axes[0].set_title('Spherical')
Out[101]: Text(0.5, 1.0, 'Spherical')

In [102]: axes[1].set_title('Exponential')
Out[102]: Text(0.5, 1.0, 'Exponential')

In [103]: axes[0].imshow(field1, origin='lower', cmap='terrain_r', vmin=vmin, vmax=vmax)
Out[103]: <matplotlib.image.AxesImage at 0x7f9e1c9c7d68>

In [104]: axes[1].imshow(field2, origin='lower', cmap='terrain_r', vmin=vmin, vmax=vmax)
Out[104]: <matplotlib.image.AxesImage at 0x7f9dc32038d0>
```



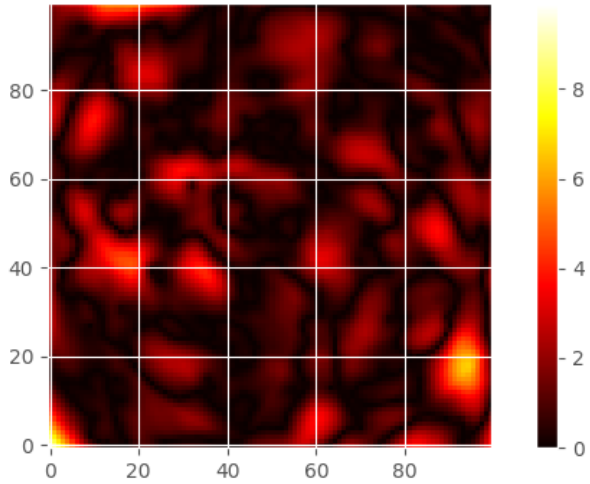
While the two final maps look alike, in the difference plot, you can spot some differences. While performing an analysis, with the model functions in mind, you should take these differences and add them as uncertainty cause by model choice to your final result.

```
# calculate the differences
In [105]: diff = np.abs(field2 - field1)

In [106]: print('Mean difference:     %.1f' % np.mean(diff))
Mean difference:     1.0

In [107]: print('3rd quartile diffs.: %.1f' % np.percentile(diff, 75))
3rd quartile diffs.: 1.4

In [108]: print('Max differences:     %.1f' % np.max(diff))
Max differences:     9.9

In [109]: plt.imshow(diff, origin='lower', cmap='hot')
Out[109]: <matplotlib.image.AxesImage at 0x7f9dba171e48>
```

(continues on next page)

(continued from previous page)

```
In [110]: plt.colorbar()
Out[110]: <matplotlib.colorbar.Colorbar at 0x7f9e01af2e80>
```

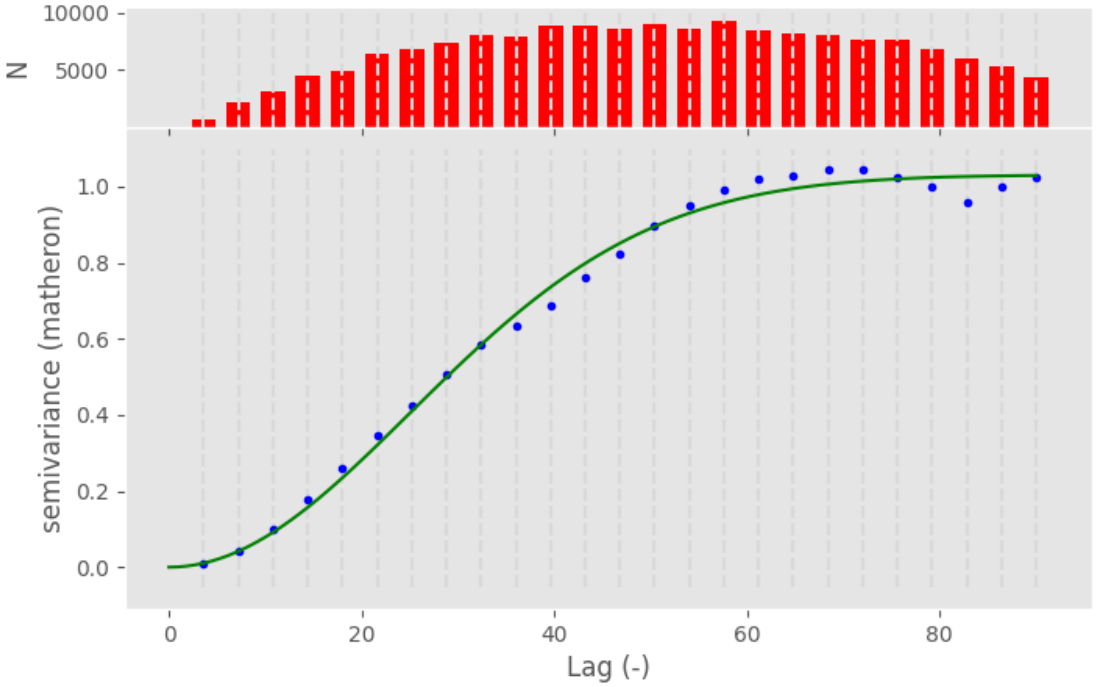


Gaussian model

The last fundamental variogram model is the Gaussian. Unlike the spherical and exponential it models a very different spatial relationship between semi-variance and distance. Following the Gaussian model, observations are assumed to be similar up to intermediate distances, showing just a gentle increase in semi-variance. Then, the semi-variance increases dramatically wihtin just a few distance units up to the sill, which is again approached asymtotically. The model can be used to simulate very sudden and sharp changes in the variable at a specific distance, while being very similar at smaller distances.

To show a typical Gaussian model, we will load another sample dataset, that actually shows a Gaussian experimental variogram.

```
In [111]: import pandas as pd

In [112]: data = pd.read_csv('data/sample_lr.csv')

In [113]: Vg = skg.Variogram(list(zip(data.x, data.y)), data.z.values,
   .....:     normalize=False, n_lags=25, maxlag=90, model='gaussian')
   .....:

In [114]: Vg.plot();
```



Matérn model

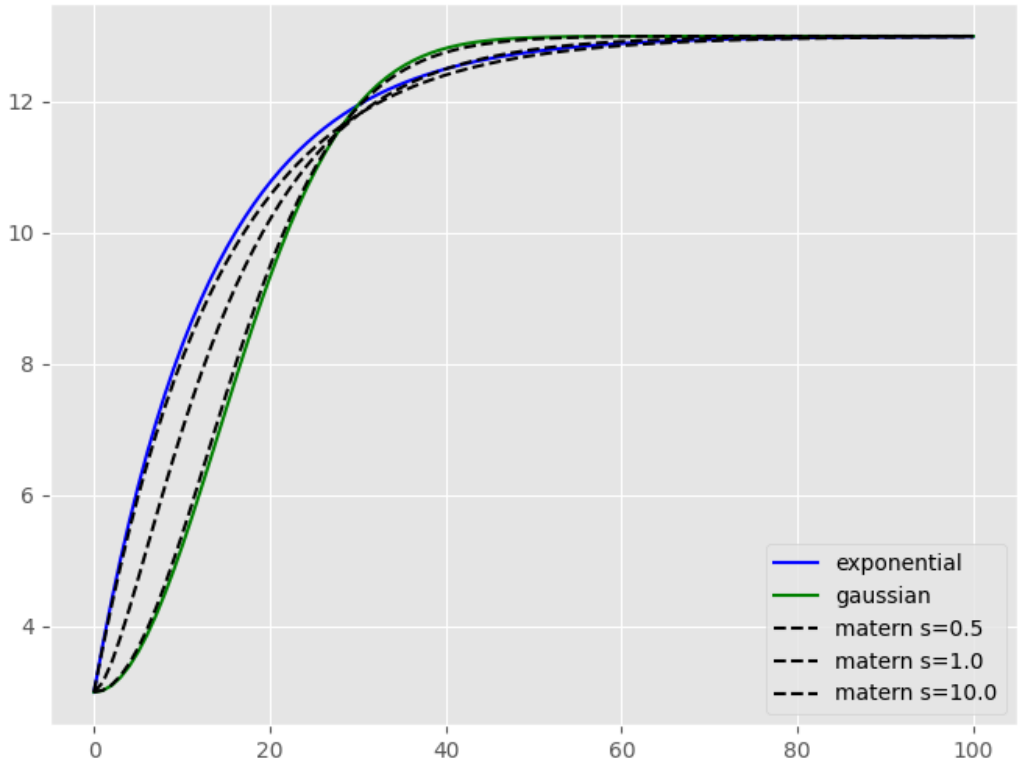
Another, quite powerful model is the Matérn model. Especially in cases where you cannot chose the appropiate model a priori so easily. The Matérn model takes an additional smoothness paramter, that can change the shape of the function in between an exponential model shape and a Gaussian one.

```
In [115]: xi = np.linspace(0, 100, 100)

# plot a exponential and a gaussian
In [116]: y_exp = [models.exponential(h, 40, 10, 3) for h in xi]

In [117]: y_gau = [models.gaussian(h, 40, 10, 3) for h in xi]

In [118]: fig, ax = plt.subplots(1, 1, figsize=(8,6))

In [119]: ax.plot(xi, y_exp, '-b', label='exponential')
Out[119]: [<matplotlib.lines.Line2D at 0x7f9db9ae4748>]

In [120]: ax.plot(xi, y_gau, '-g', label='gaussian')
Out[120]: [<matplotlib.lines.Line2D at 0x7f9db9694c50>]

In [121]: for s in (0.5, 1., 10.):
   .....:     y = [models.matern(h, 40, 10, s, 3) for h in xi]
   .....:     ax.plot(xi, y, '--k', label='matern s=%.1f' % s)
   .....:

In [122]: plt.legend(loc='lower right')
```

(continues on next page)

(continued from previous page)

```
Out[122]: <matplotlib.legend.Legend at 0x7f9db9694cc0>
```



This example illustrates really nicely, how the smoothness parameter adapts the Matérn model shape. Moreover, the smoothness parameter can be used to assess whether an experimental variogram is rather showing a Gaussian or exponential behavior.

Note: If you would like to export a Variogram instance to gstools, the smoothness parameter may not be smaller than `0.2`.

When direction matters

What is 'direction'?

The classic approach to calculate a variogram is based on the assumption that covariance between observations can be related to their separating distance. For this, point pairs of all observation points are formed and it is assumed that they can be formed without any restriction. The only paramter to be influenced is a limiting distance, beyond which a point pair does not make sense anymore.

This assumption might not always hold. Especially in landscapes, processes do not occur randomly, but in an organized manner. This organization is often directed, which can lead to stronger covariance in one direction than another. Therefore, another step has to be introduced before lag classes are formed.

The *direction* of a variogram is then a orientation, which two points need. If they are not oriented in the specified way, they will be ignored while calculating a semi-variance value for a given lag class. Usually, you will specify a orientation, which is called *azimuth*, and a *tolerance*, which is an offset from the given azimuth, at which a point pair will still be accepted.

Defining orientiation

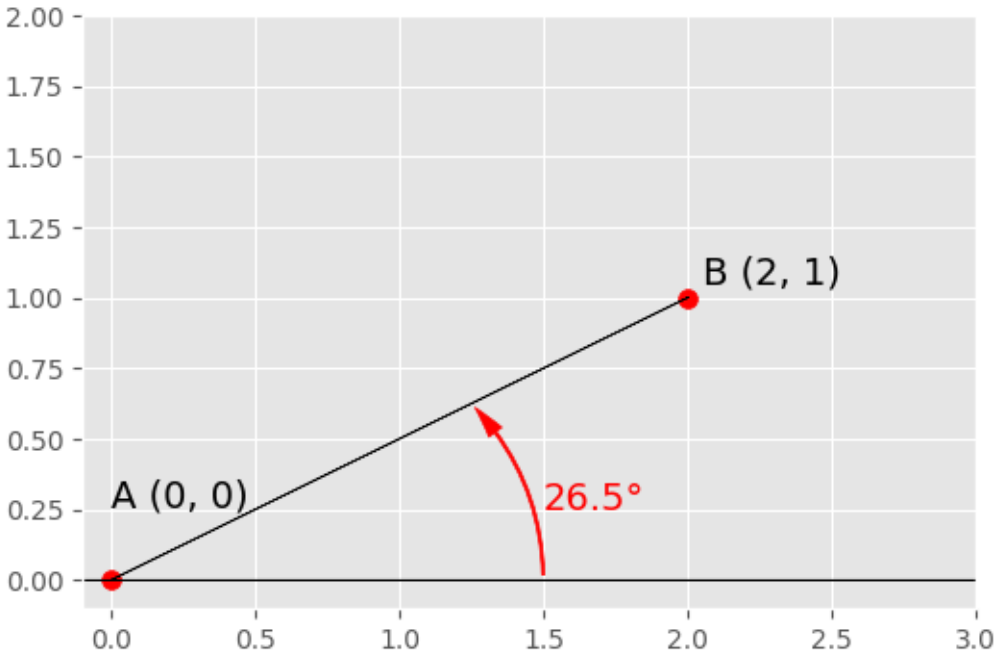
One has to decide how orientation of two points is determined. In scikit-gstat, orientation between two observation points is only defined in $\mathbb{R}^2$. We define the orientation as the **angle between the vector connecting two observation points with the x-axis**.

Thus, also the *azimuth* is defined as an angle of the azimutal vector to the x-axis, with an *tolerance* in degrees added to the exact azimutal orientation clockwise and counter clockwise.

The angle Φ between two vetors `u,v` is given like:

$$\Phi = cos^{-1}\left(\frac{u \circ v}{||u|| \cdot ||v||}\right)$$

```
In [123]: from matplotlib.patches import FancyArrowPatch as farrow

In [124]: fig, ax = plt.subplots(1, 1, figsize=(6,4))

In [125]: ax.arrow(0,0,2,1,color='k')
Out[125]: <matplotlib.patches.FancyArrow at 0x7f9db93df4a8>

In [126]: ax.arrow(-.1,0,3.1,0,color='k')
Out[126]: <matplotlib.patches.FancyArrow at 0x7f9db95f2a90>

In [127]: ax.set_xlim(-.1, 3)
Out[127]: (-0.1, 3.0)

In [128]: ax.set_ylim(-.1,2.)
Out[128]: (-0.1, 2.0)

In [129]: ax.scatter([0,2], [0,1], 50, c='r')
Out[129]: <matplotlib.collections.PathCollection at 0x7f9db95f2320>

In [130]: ax.annotate('A (0, 0)', (.0, .26), fontsize=14)
Out[130]: Text(0.0, 0.26, 'A (0, 0)')

In [131]: ax.annotate('B (2, 1)', (2.05,1.05), fontsize=14)
Out[131]: Text(2.05, 1.05, 'B (2, 1)')

In [132]: arrowstyle="Simple,head_width=6,head_length=12,tail_width=1"

In [133]: ar = farrow([1.5,0], [1.25, 0.625],  color='r', connectionstyle="arc3, rad=.2",
↪ arrowstyle=arrowstyle)

In [134]: ax.add_patch(ar)
Out[134]: <matplotlib.patches.FancyArrowPatch at 0x7f9db95f2d30>

In [135]: ax.annotate('26.5°', (1.5, 0.25), fontsize=14, color='r')
Out[135]: Text(1.5, 0.25, '26.5°')
```



The described definition of orientation is illustrated in the figure above. There are two observation points, $A(0, 0)$ and $B(2, 1)$. To decide wether to account for them when calculating the semi-variance at their separating distance lag, their orientation is used. Only if the direction of the varigram includes this orientation, the points are used. Imagine the azimuth and tolerance would be `45°`, then anything between `0°` (East) and `90°` orientation would be included. The given example shows the orientation angle $\Phi = 26.5$, which means the vector $\overrightarrow{AB}$ is included.

Calculating orientations

SciKit-GStat implements a slightly adaped version of the formula given in the last section. It makes use of symmetric search areas (tolerance is applied clockwise and counter clockwise) und therefore any calculated angle might be the result of calculating the orientation of $\overrightarrow{AB}$ or $\overrightarrow{BA}$. Mathematically, these two vectors have two different angles, but they are always both taken into account or omitted for a variagram at the same time. Thus, it does not make a difference for variography. However, it does make a difference when you try to use the orientation angles directly as the containing matrix can contain the inverse angles.

This can be demonstrated by an easy example. Let `c` be a set of points mirrored along the x-axis.

```
In [136]: c = np.array([[0,0], [2,1], [1,2], [2, -1], [1, -2]])

In [137]: east = np.array([1,0])
```

We can plug these two arrays into the the formula above:

```
In [138]: u = c[1:]   # omit the first one

In [139]: angles = np.degrees(np.arccos(u.dot(east) / np.sqrt(np.sum(u**2, axis=1))))
```

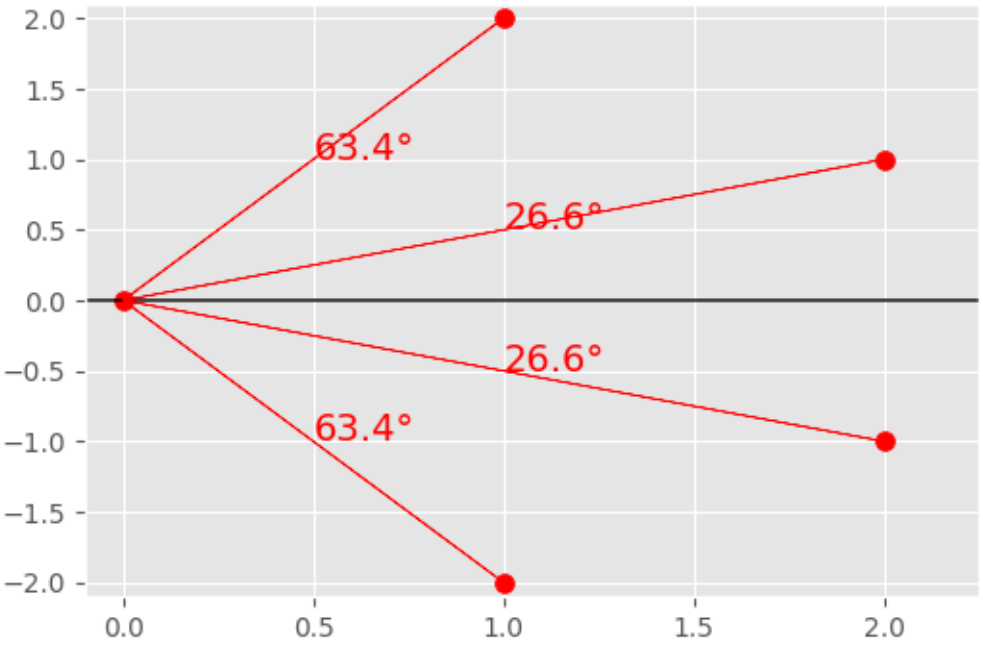
(continues on next page)

(continued from previous page)

```
In [140]: angles.round(1)
Out[140]: array([26.6, 63.4, 26.6, 63.4])
```

You can see, that the both points and their mirrored counterpart have the same angle to the x-axis, just like expected. This can be visualized by the plot below:

```
In [141]: fig, ax = plt.subplots(1, 1, figsize=(6,4))

In [142]: ax.set_xlim(-.1, 2.25)
Out[142]: (-0.1, 2.25)

In [143]: ax.set_ylim(-2.1,2.1)
Out[143]: (-2.1, 2.1)

In [144]: ax.arrow(-.1,0,3.1,0,color='k')
Out[144]: <matplotlib.patches.FancyArrow at 0x7f9db5d7f4a8>

In [145]: for i,p in enumerate(u):
   .....:     ax.arrow(0,0,p[0],p[1],color='r')
   .....:     ax.annotate('%.1f°' % angles[i], (p[0] / 2, p[1] / 2 ), fontsize=14, color=
↪'r')
   .....:

In [146]: ax.scatter(c[:,0], c[:,1], 50, c='r')
Out[146]: <matplotlib.collections.PathCollection at 0x7f9db5d7f898>
```



The main difference to the internal structure storing the orientation angles for a *DirectionalVariogram* instance will store different angles. To use the class on only five points, we need to prevent the class from fitting, as fitting on only

5 points will not work. But this does not affect the orientation calculations. Therefore, the `fit` mehtod is overwritten.

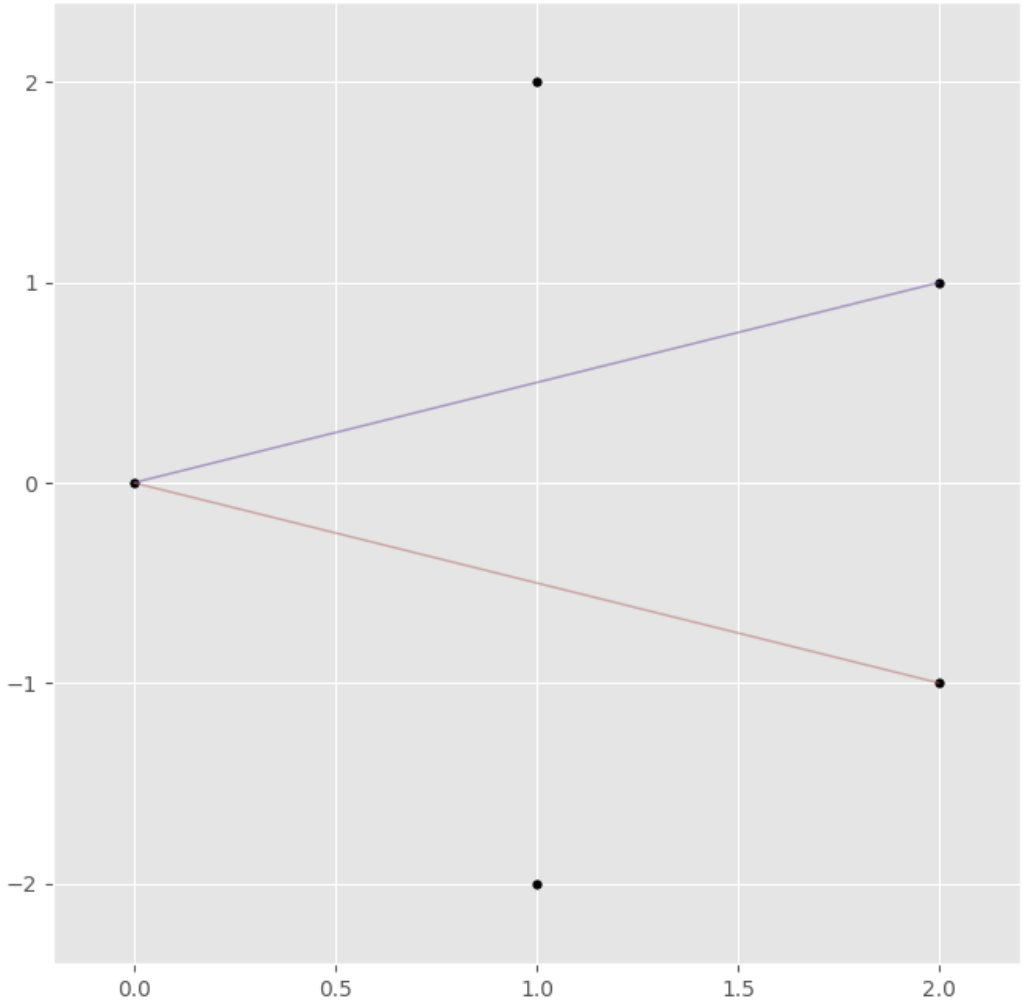
```
In [147]: class TestCls(skg.DirectionalVariogram):
   .....:     def fit(*args, **kwargs):
   .....:         pass
   .....:

In [148]: DV = TestCls(c, np.random.normal(0,1,len(c)))

In [149]: DV._calc_direction_mask_data()

In [150]: np.degrees(DV._angles + np.pi)[:len(c) - 1]
Out[150]: array([ 26.565,  63.435, 333.435, 296.565])
```

The first two points (with positive y-coordinate) show the same result. The other two, with negative y-coordinates, are also calculated counter clockwise:

```
In [151]: 360 - np.degrees(DV._angles + np.pi)[[2,3]]
Out[151]: array([26.565, 63.435])
```

The *DirectionalVariogram* class has a plotting function to show a network graph of all point pairs that are oriented in the variogram direction. But first we need to increase the tolerance as half tolerance (`45° / 2 = 22.5°` clockwise and counter clockwise) is smaller than both orientations.

```
In [152]: DV.tolerance = 90

In [153]: DV.pair_field()
Out[153]: <Figure size 800x800 with 1 Axes>
```



Directional variogram

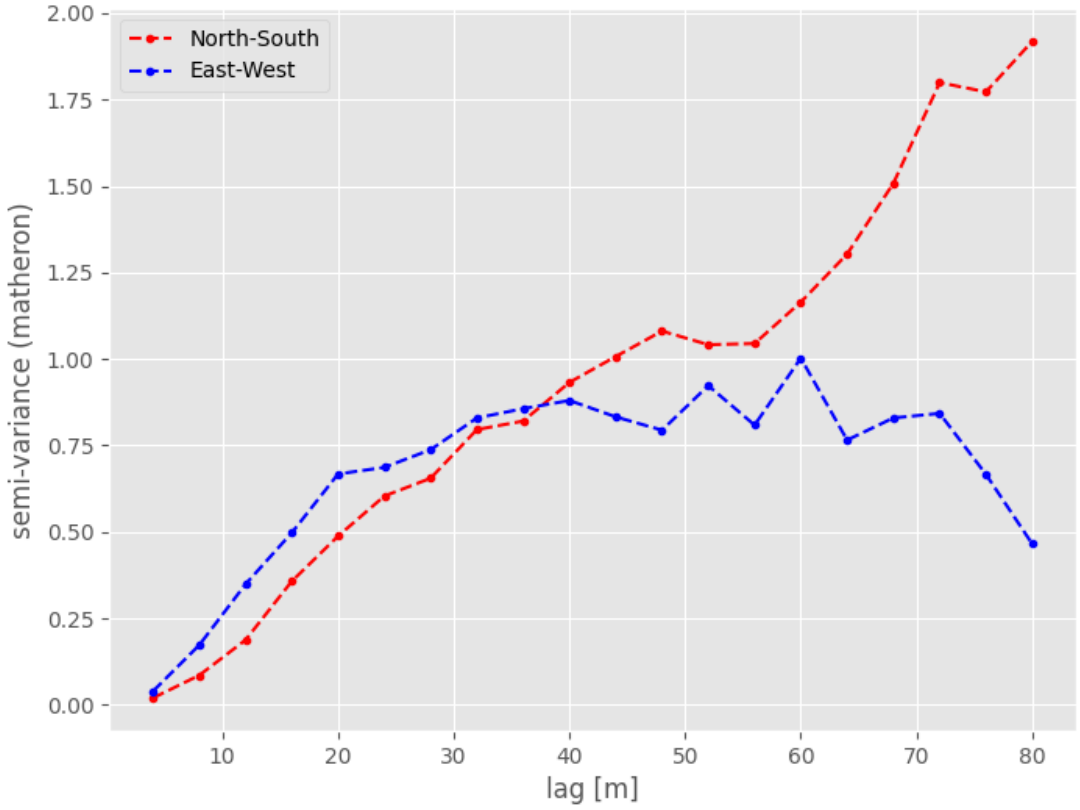
```
In [154]: field = np.loadtxt('data/aniso_x2.txt')

In [155]: np.random.seed(1312)

In [156]: coords = np.random.randint(100, size=(300,2))

In [157]: vals = [field[_[0], _[1]] for _ in coords]
```

The next step is to create two different variogram instances, which share the same parameters, but use a different azimuth angle. One oriented to North and the second one oriented to East.

```
In [158]: Vnorth = skg.DirectionalVariogram(coords, vals, azimuth=90, tolerance=90,␣
↪maxlag=80, n_lags=20)

In [159]: Veast = skg.DirectionalVariogram(coords, vals, azimuth=0, tolerance=90,␣
↪maxlag=80, n_lags=20)

In [160]: pd.DataFrame({'north':Vnorth.describe(), 'east': Veast.describe()})
Out[160]:
                                                                 north           ␣
↪                                    east
model                                                         spherical          ␣
↪                               spherical
estimator                                                      matheron          ␣
↪                                matheron
dist_func                                                     euclidean          ␣
↪                               euclidean
normalized_effective_range                                         6400          ␣
↪                                 2933.08
normalized_sill                                                 2.84452          ␣
↪                                0.810282
normalized_nugget                                                    0          ␣
↪                                       0
effective_range                                                     80          ␣
↪                                 36.6635
sill                                                             1.483          ␣
↪                                0.808277
nugget                                                               0          ␣
↪                                       0
params                       {'estimator': 'matheron', 'model': 'spherical'...  {
↪'estimator': 'matheron', 'model': 'spherical'...
kwargs                                                              {}          ␣
↪                                      {}
```

You can see, how the two are differing in effective range and also sill, only caused by the orientation. Let's look at the experimental variogram:

```
In [161]: fix, ax = plt.subplots(1,1,figsize=(8,6))

In [162]: ax.plot(Vnorth.bins, Vnorth.experimental, '.--r', label='North-South')
Out[162]: [<matplotlib.lines.Line2D at 0x7f9db5d06978>]

In [163]: ax.plot(Veast.bins, Veast.experimental, '.--b', label='East-West')
Out[163]: [<matplotlib.lines.Line2D at 0x7f9db5d24780>]

In [164]: ax.set_xlabel('lag [m]')
Out[164]: Text(0.5, 0, 'lag [m]')

In [165]: ax.set_ylabel('semi-variance (matheron)')
Out[165]: Text(0, 0.5, 'semi-variance (matheron)')

In [166]: plt.legend(loc='upper left')
Out[166]: <matplotlib.legend.Legend at 0x7f9db5d24a90>
```



The shape of both experimental variograms is very similar on the first 40 meters of distance. Within this range, the apparent anisotropy is not pronounced. The East-West oriented variograms also have an effective range of only about 40 meters, which means that in this direction the observations become statistically independent at larger distances. For the North-South variogram the effective range is way bigger and the variogram plot reveals much larger correlation lengths in that direction. The spatial dependency is thus directed in North-South direction.

To perform Kriging, you would now transform the data, especially in North-West direction, unitl both variograms look the same within the effective range. Finally, the Kriging result is back-transformed into the original coordinate system.

2.3.3 Interpolation

Spatial interpolation

In geostatistics the procedure of spatial interpolation is known as *Kriging*. That goes back to the inventor of Kriging, a South-African mining engineer called Dave Krige. He published the method in 1951. In many text books you will also find the term *prediction*, but be aware that Kriging is still based on the assumption that the variable is a random field. THerefore I prefer the term *estimation* and would label the Kriging method a *BLUE*, **B** est **L** inear **U** nbiased **E** stimator. In general terms, the objective is to estimate a variable at a location that was not observed using observations from close locations. Kriging is considered to be the **best** estimator, because we utilize the spatial structure described by a variogram to find suitable weights for averaging the observations at close locations.

Given a set of observation points s and observation values at these locations $Z(s)$, it can already be stated that the

estimation at an unobserved location $Z^*(s_0)$ is a weighted mean:

$$Z^*(s_0) = \sum_{i=0}^{N} \lambda_i Z(s_i)$$

where N is the size of s and λ is the array of weights. This is what we want to calculate from a fitted variogram model.

Assumed that λ had already been calculated, estimating the prediction is pretty straightforward:

```
In [1]: Z_s = np.array([4.2, 6.1, 0.2, 0.7, 5.2])

In [2]: lam = np.array([0.1, 0.3, 0.1, 0.1, 0.4])

# calculate the weighted mean
In [3]: np.sum(Z_s * lam)
Out[3]: 4.42
```

or shorter:

```
In [4]: Z_s.dot(lam)
Out[4]: 4.42
```

In the example above the weights were just made up. Now we need to understand how this array of weights can be calculated.

Using a spatial model

Instead of just making up weights, we will now learn how we can utilize a variogram model to calculate the weights. At its core a variogram describes how point observations become more dissimilar with distance. Point distances can easily be calculated, not only for observed locations, but also for unobserved locations. As the variogram is only a function of *distance*, we can easily calculate a semi-variance value for any possible combination of point pairs.

Assume we have five close observations for an unobserved location, like in the example above. Instead of making up weights, we can use the semi-variance value as a weight, as a first shot. What we still need are locations and a variogram model. For both, we can just make something up.

```
In [5]: x = np.array([4.0, 2.0, 4.1, 0.3, 2.0])

In [6]: y = np.array([5.5, 1.2, 3.7, 2.0, 2.5])

In [7]: z = np.array([4.2, 6.1, 0.2, 0.7, 5.2])

In [8]: s0 = [2., 2.]

In [9]: distance_matrix = pdist([s0] + list(zip(x,y)))

In [10]: squareform(distance_matrix)
Out[10]:
array([[0.   , 4.031, 0.8  , 2.702, 1.7  , 0.5  ],
       [4.031, 0.   , 4.742, 1.803, 5.093, 3.606],
       [0.8  , 4.742, 0.   , 3.265, 1.879, 1.3  ],
       [2.702, 1.803, 3.265, 0.   , 4.163, 2.419],
       [1.7  , 5.093, 1.879, 4.163, 0.   , 1.772],
       [0.5  , 3.606, 1.3  , 2.419, 1.772, 0.   ]])
```

Next, we build up a variogram model of spherical shape, that uses a effective range larger than the distances in the matrix. Otherwise, we would just calcualte the arithmetic mean.

```
In [11]: from skgstat.models import spherical

# range= 7. sill = 2. nugget = 0.
In [12]: model = lambda h: spherical(h, 7.0, 2.0, 0.0)
```

The distances to the first point *s0* are the first 5 elements in the distance matrix. Therefore the semi-variances are calculated straightforward.

```
In [13]: variances = model(distance_matrix[:5])

In [14]: assert len(variances) == 5
```

Of course we could now use the inverse of these semi-variances to weigh the observations, **but that would not be correct.** Remeber, that this array *variances* is what we want the target weights to incorporte. Whatever the weights are, these variances should be respected. At the same time, the five points among each other also have distances and therefore variances that should be respected. Or to put it differently. Take the first observation point s_1. The associated variances γ to the other four points need to match the one just calculated.

$$a_1 * \gamma(s_1, s_1) + a_2 * \gamma(s_1, s_2) + a_3 * \gamma(s_1, s_3) + a_4 * \gamma(s_1, s_4) + a_5 * \gamma(s_1, s_5) = \gamma(s_1, s_0)$$

Ok. First: $\gamma(s_1, s_1)$ is zero because the distance is obviously zero and the model does not have a nugget. All other distances have already been calculated. $a_1 ... a_5$ are factors. These are the weights used to satisfy all given semi-variances. This is what we need. Obviously, we cannot calculate 5 unknown variables from just one equation. Lukily we have four more observations. Writing the above equation for s_2, s_3, s_4, s_5. Additionally, we will write the linear equation system in matrix form as a dot product of the γ_i and the a_i part.

$$\begin{pmatrix} \gamma(s_1, s_1) & \gamma(s_1, s_2) & \gamma(s_1, s_3) & \gamma(s_1, s_4) & \gamma(s_1, s_5) \\ \gamma(s_2, s_1) & \gamma(s_2, s_2) & \gamma(s_2, s_3) & \gamma(s_2, s_4) & \gamma(s_2, s_5) \\ \gamma(s_3, s_1) & \gamma(s_3, s_2) & \gamma(s_3, s_3) & \gamma(s_3, s_4) & \gamma(s_3, s_5) \\ \gamma(s_4, s_1) & \gamma(s_4, s_2) & \gamma(s_4, s_3) & \gamma(s_4, s_4) & \gamma(s_4, s_5) \\ \gamma(s_5, s_1) & \gamma(s_5, s_2) & \gamma(s_5, s_3) & \gamma(s_5, s_4) & \gamma(s_5, s_5) \end{pmatrix} * \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{pmatrix} \gamma(s_0, s_1) \\ \gamma(s_0, s_2) \\ \gamma(s_0, s_3) \\ \gamma(s_0, s_4) \\ \gamma(s_0, s_5) \end{pmatrix}$$

That might look a bit complicated at first, but we have calculated almost everything. The last matrix are the *variances* that we calculated in the last step. The first matrix is of same shape as the sqaureform distance matrix calculated in the very begining. All we need to do is to map the variogram model on it and solve the system for the matrix of factors $a_1 \dots a_5$. In Python, there are several strategies how you could solve this problem. Let's at first build the matrix. We need a distance matrix without s_0 for that.

```
In [15]: dists = pdist(list(zip(x,y)))

In [16]: M = squareform(model(dists))

In [17]: pprint(M)
array([[0.   , 1.721, 0.756, 1.798, 1.409],
       [1.721, 0.   , 1.298, 0.786, 0.551],
       [0.756, 1.298, 0.   , 1.574, 0.995],
       [1.798, 0.786, 1.574, 0.   , 0.743],
       [1.409, 0.551, 0.995, 0.743, 0.   ]])

In [18]: pprint(variances)
array([1.537, 0.341, 1.1  , 0.714, 0.214])
```

And solve it:

```
In [19]: from scipy.linalg import solve

# solve for a
In [20]: a = solve(M, variances)

In [21]: pprint(a)
array([-0.022,  0.362,  0.018,  0.037,  0.593])

# calculate estimation
In [22]: Z_s.dot(a)
Out[22]: 5.226267185422778
```

That's it. Well, not really. We might have used the variogram and the spatial structure infered from the data for getting better results, but in fact our result is not **unbiased**. That means, the solver can choose any combination that satisfies the equation, even setting everything to zero except one weight. That means a could be biased. That would not be helpful.

```
In [23]: np.sum(a)
Out[23]: 0.9872744357166217
```

Kriging equation system

In the last section we came pretty close to the Kriging algorithm. The only thing missing is to assure unbiasedness. The weights sum up to almost one, but they are not one. We want to ensure, that they are always one. This is done by adding one more equation to the linear equation system. Also, we will rename the a array to λ, which is more frequently used for Kriging weights. The missing equation is:

$$\sum_{i=1}^{N} \lambda = 1$$

In matrix form this changes M to:

$$\begin{pmatrix} \gamma(s_1,s_1) & \gamma(s_1,s_2) & \gamma(s_1,s_3) & \gamma(s_1,s_4) & \gamma(s_1,s_5) & 1 \\ \gamma(s_2,s_1) & \gamma(s_2,s_2) & \gamma(s_2,s_3) & \gamma(s_2,s_4) & \gamma(s_2,s_5) & 1 \\ \gamma(s_3,s_1) & \gamma(s_3,s_2) & \gamma(s_3,s_3) & \gamma(s_3,s_4) & \gamma(s_3,s_5) & 1 \\ \gamma(s_4,s_1) & \gamma(s_4,s_2) & \gamma(s_4,s_3) & \gamma(s_4,s_4) & \gamma(s_4,s_5) & 1 \\ \gamma(s_5,s_1) & \gamma(s_5,s_2) & \gamma(s_5,s_3) & \gamma(s_5,s_4) & \gamma(s_5,s_5) & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} * \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \mu \end{bmatrix} = \begin{pmatrix} \gamma(s_0,s_1) \\ \gamma(s_0,s_2) \\ \gamma(s_0,s_3) \\ \gamma(s_0,s_4) \\ \gamma(s_0,s_5) \\ 1 \end{pmatrix}$$

This is the Kriging equation for Ordinary Kriging that can be found in text books. We added the ones to the result array and into the matrix of semivariances. μ is a Lagrangian multiplier that will be used to estimate the Kriging variance, which will be covered later. Ordinary Kriging still assumes the observation and their residuals to be normally distributed and second order stationarity.

Todo: Include the references to Kitanidis and Bardossy.

Applied in Python, this can be done like:

```
In [24]: B = np.concatenate((variances, [1]))

In [25]: M = np.concatenate((M, [[1, 1, 1, 1, 1]]), axis=0)
```

(continues on next page)

(continued from previous page)

```
In [26]: M = np.concatenate((M, [[1], [1], [1], [1], [1], [0]]), axis=1)

In [27]: weights = solve(M, B)

# see the weights
In [28]: print('Old weights:', a)
Old weights: [-0.022  0.362  0.018  0.037  0.593]

In [29]: print('New weights:', weights[:-1])
New weights: [-0.017  0.365  0.02   0.041  0.592]

In [30]: print('Old estimation:', Z_s.dot(a))
Old estimation: 5.226267185422778

In [31]: print('New estimation:', Z_s.dot(weights[:-1]))
New estimation: 5.2628805787423785

In [32]: print('Mean:', np.mean(Z_s))
Mean: 3.28
```

And the sum of weights:

```
In [33]: np.sum(weights[:-1])
Out[33]: 1.0
```

The estimation did not change a lot, but the weights perfectly sum up to one now.

Kriging error

In the last step, we introduced a factor μ. It was needed to solve the linear equation system while assuring that the weights sum up to one. This factor can in turn be added to the weighted target semi-variances used to build the equation system, to obtain the Kriging error.

```
In [34]: sum(B[:-1] * weights[:-1]) + weights[-1]
Out[34]: 0.262875753928683
```

This is really usefull when a whole map is interpolated. Using Kriging, you can also produce a map showing in which regions the interpolation is more certain.

Example

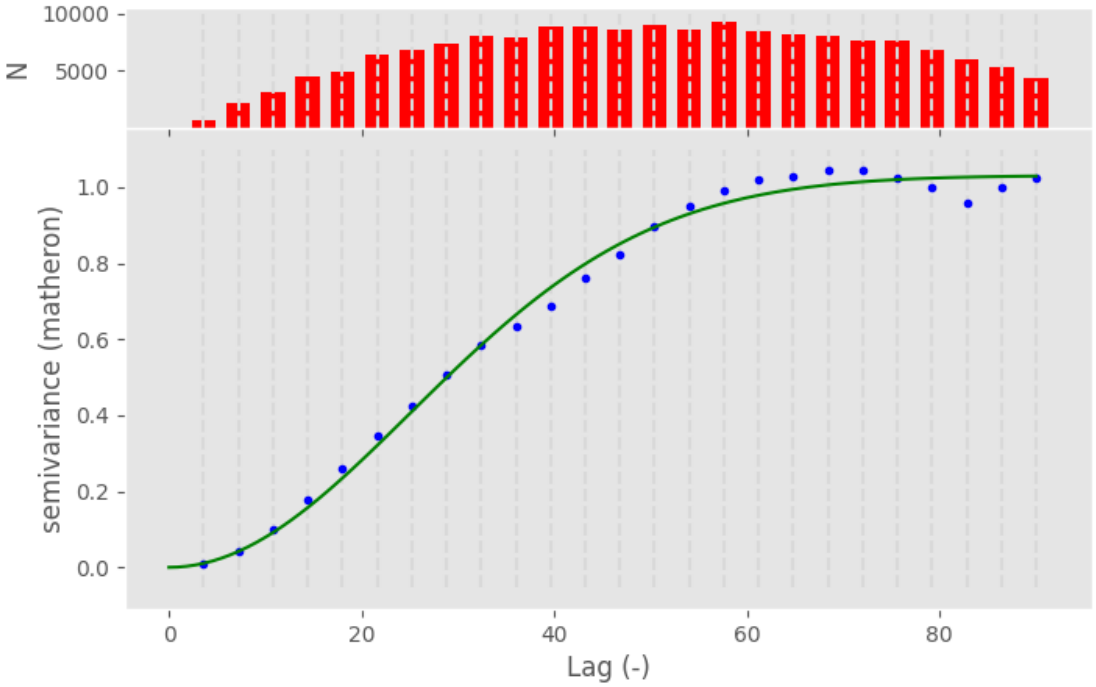
We can use the data shown in the variography section, to finally interpolate the field and check the Kriging error. You could either build a loop around the code shown in the previous section, or just use skgstat.

```
In [35]: data = pd.read_csv('data/sample_lr.csv')

In [36]: V = Variogram(data[['x', 'y']].values, data.z.values,
   ....:   maxlag=90, n_lags=25, model='gaussian', normalize=False)
   ....:
```

(continues on next page)

(continued from previous page)

```
In [37]: V.plot()
Out[37]: <Figure size 800x500 with 2 Axes>

In [38]: from skgstat import OrdinaryKriging

In [39]: ok = OrdinaryKriging(V, min_points=5, max_points=20, mode='exact')
```



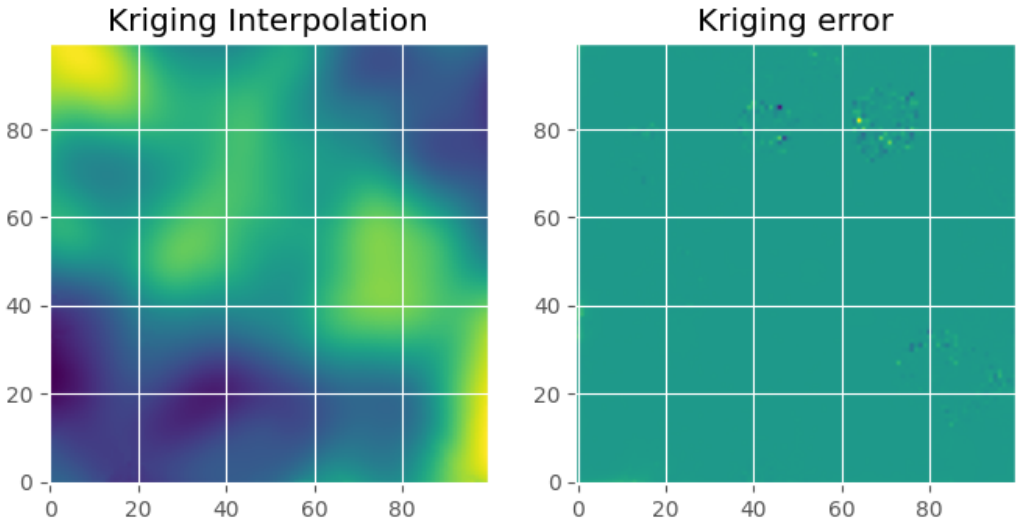
The *OrdinaryKriging* class need at least a fitted *Variogram* instance. Using *min_points* we can demand the Kriging equation system to be build upon at least 5 points to yield robust results. If not enough close observations are found within the effective range of the variogram, the estimation will not be calculated and a *np.NaN* value is estimated.

The *max_points* parameter will set the upper bound of the equation system by using in this case at last the 20 nearest points. Adding more will most likely not change the estimation, as more points will recieve small, if not negligible, weights. But it will increase the processing time, as each added point will increase the Kriging equation system dimensionality by one.

The *mode* parameter sets the method that will build up the equation system. There are two implemented: *mode='exact'* and *mode='estimate'*. Estimate is much faster, but if not used carefully, it can lead to numerical instability quite quickly. In the technical notes section of this userguide, you will find a whole section on the two modes.

Finally, we need the unobsered locations. The observations in the file were drawn from a *100x100* random field.

```
In [40]: xx, yy = np.mgrid[0:99:100j, 0:99:100j]

In [41]: field = ok.transform(xx.flatten(), yy.flatten()).reshape(xx.shape)

In [42]: s2 = ok.sigma.reshape(xx.shape)
```



2.4 Tutorials

The tutorials are designed to get you quickly started. It is assumed, that the tutorials are used together with the user guide. The user guide is way more texty, while the tutorials are focused on code.

2.4.1 1 - Getting Started

The main application for `scikit-gstat` is variogram analysis and Kriging. This Tutorial will guide you through the most basic functionality of `scikit-gstat`. There are other tutorials that will explain specific methods or attributes in `scikit-gstat` in more detail.

What you will learn in this tutorial

- How to instantiate `Variogram` and `OrdinaryKriging`
- How to read a variogram
- Perform an interpolation
- Most basic plotting

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pprint import pprint
plt.style.use('ggplot')
```

The `Variogram` and `OrdinaryKriging` classes can be loaded directly from `skgstat`. This is the name of the Python module.

[2]:
```
from skgstat import Variogram, OrdinaryKriging
```

At the current version, there are some deprecated attributes and method in the `Variogram` class. They do not raise `DeprecationWarnings`, but rather print a warning message to the screen. You can suppress this warning by adding an `SKG_SUPPRESS` environment variable

[3]:
```
%set_env SKG_SUPPRESS=true
```

```
env: SKG_SUPPRESS=true
```

1.1 Load data

You can find a prepared example data set in the `./data` subdirectory. This example is extracted from a generated Gaussian random field. We can expect the field to be stationary and show a nice spatial dependence, because it was created that way. We can load one of the examples and have a look at the data:

[4]:
```
data = pd.read_csv('./data/sample_sr.csv')
print("Loaded %d rows and %d columns" % data.shape)
data.head()
```

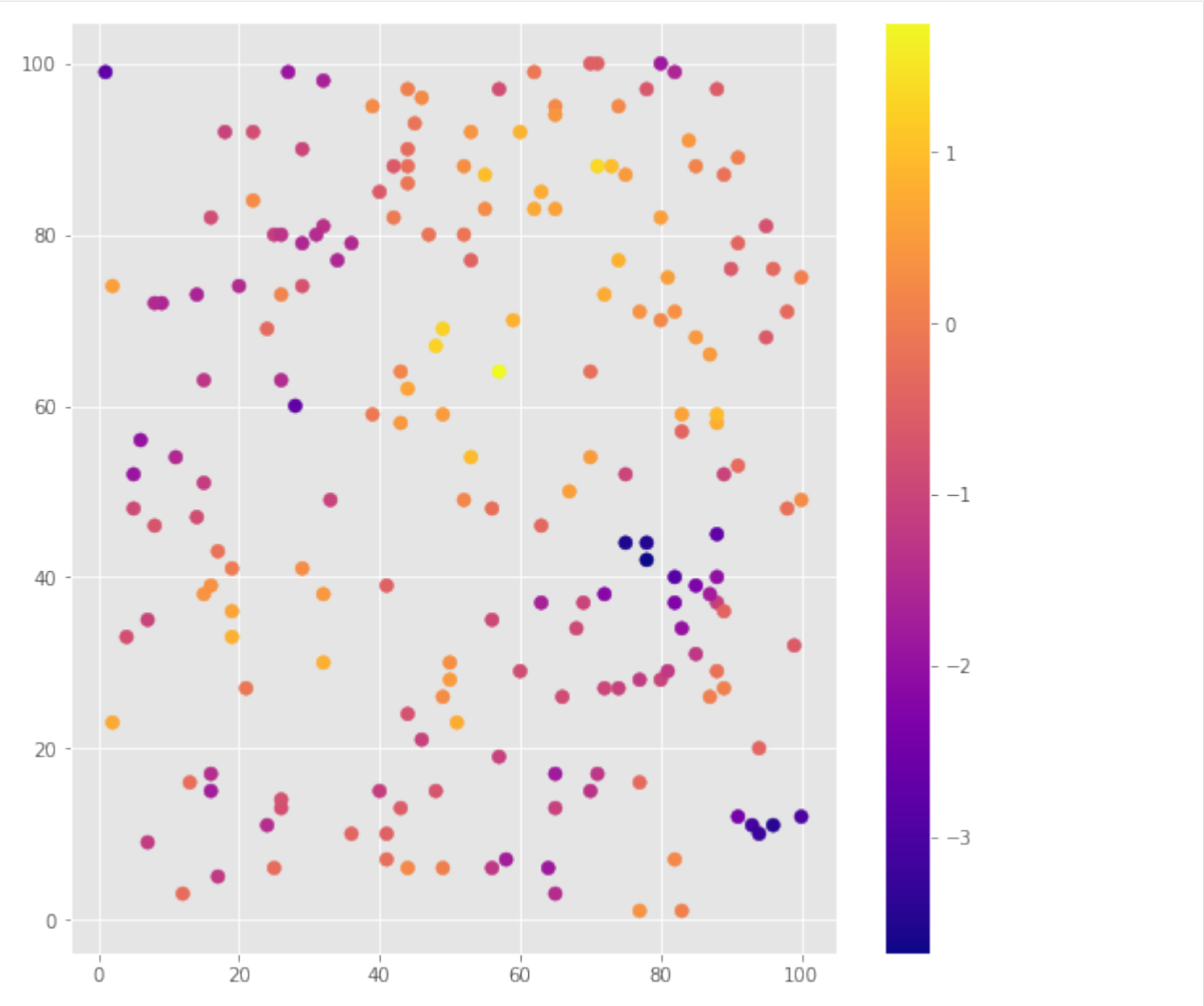
```
Loaded 200 rows and 3 columns
```

[4]:

	x	y	z
0	94	20	-0.394444
1	82	37	-2.283663
2	43	13	-0.546213
3	78	42	-3.681384
4	50	28	0.504538

Get a first overview of your data by plotting the `x` and `y` coordinates and visually inspect how the `z` spread out.

[5]:
```
fig, ax = plt.subplots(1, 1, figsize=(9, 9))
art = ax.scatter(data.x,data.y, s=50, c=data.z, cmap='plasma')
plt.colorbar(art);
```



We can already see a lot from here:

- The small values seem to concentrate on the upper left and lower right corner
- Larger values are arranged like a band from lower left to upper right corner
- To me, each of these blobs seem to have a diameter of something like 30 or 40 units.
- The distance between the minimum and maximum seems to be not more than 60 or 70 units.

These are already very important insights.

1.2 Build a Variogram

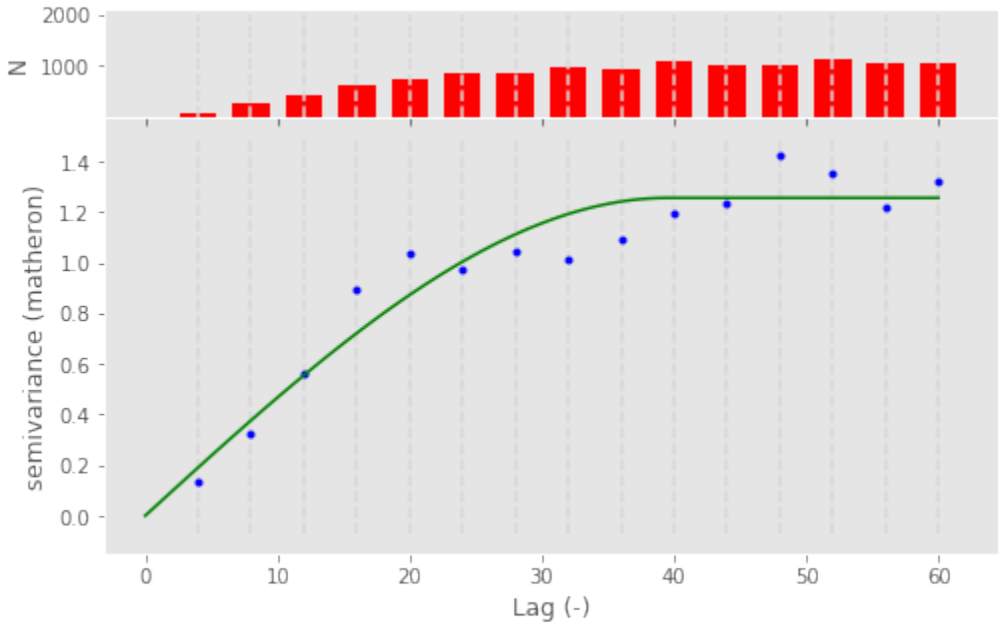
As a quick reminder, the variogram relates pair-wise separating distances of `coordinates` and relates them to the *semi-variance* of the corresponding `values` pairs. The default estimator used is the Matheron estimator:

$$\gamma(h) = \frac{1}{2N(h)} * \sum_{i=1}^{N(h)} (Z(x_i) - Z(x_{i+h}))^2$$

For more details, please refer to the User Guide

The `Variogram` class takes at least two arguments. The `coordinates` and the `values` observed at these locations. You should also at least set the `normalize` parameter to explicitly, as it changes it's default value in version `0.2.8` to `False`. This attribute affects only the plotting, not the variogram values. Additionally, the number of bins is set to 15, because we have fairly many observations and the default value of 10 is unnecessarily small. The `maxlag` set the maximum distance for the last bin. We know from the plot above, that more than 60 units is not really meaningful

[6]:
```
V = Variogram(data[['x', 'y']].values, data.z.values, normalize=False, maxlag=60, n_
↪lags=15)
fig = V.plot(show=False)
```



The upper subplot show the histogram for the count of point-pairs in each lag class. You can see various things here:

- As expected, there is a clear spatial dependency, because semi-variance increases with distance (blue dots)
- The default `spherical` variogram model is well fitted to the experimental data
- The shape of the dependency is **not** captured quite well, but fair enough for this example

The sill of the variogram should correspond with the field variance. The field is unknown, but we can compare the sill to the *sample* variance:

[7]:
```
print('Sample variance: %.2f   Variogram sill: %.2f' % (data.z.var(), V.describe()['sill
↪'']))
```

```
Sample variance: 1.10   Variogram sill: 1.26
```

The `describe` method will return the most important parameters as a dictionary. And we can simply print the variogram ob,ect to the screen, to see all parameters.

[8]:
```
pprint(V.describe())
```

```
{'effective_range': 39.50027313170537,
 'estimator': 'matheron',
 'name': 'spherical',
 'nugget': 0,
 'sill': 1.2553698556802062}
```

[9]:
```
print(V)
```

```
spherical Variogram
-------------------
Estimator:         matheron
Effective Range:   39.50
Sill:              1.26
Nugget:            0.00
```

1.3 Kriging

The Kriging class will now use the Variogram from above to estimate the Kriging weights for each grid cell. This is done by solving a linear equation system. For an unobserved location s_0, we can use the distances to 5 observation points and build the system like:

$$\begin{pmatrix} \gamma(s_1,s_1) & \gamma(s_1,s_2) & \gamma(s_1,s_3) & \gamma(s_1,s_4) & \gamma(s_1,s_5) & 1 \\ \gamma(s_2,s_1) & \gamma(s_2,s_2) & \gamma(s_2,s_3) & \gamma(s_2,s_4) & \gamma(s_2,s_5) & 1 \\ \gamma(s_3,s_1) & \gamma(s_3,s_2) & \gamma(s_3,s_3) & \gamma(s_3,s_4) & \gamma(s_3,s_5) & 1 \\ \gamma(s_4,s_1) & \gamma(s_4,s_2) & \gamma(s_4,s_3) & \gamma(s_4,s_4) & \gamma(s_4,s_5) & 1 \\ \gamma(s_5,s_1) & \gamma(s_5,s_2) & \gamma(s_5,s_3) & \gamma(s_5,s_4) & \gamma(s_5,s_5) & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} * \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \mu \end{bmatrix} = \begin{pmatrix} \gamma(s_0,s_1) \\ \gamma(s_0,s_2) \\ \gamma(s_0,s_3) \\ \gamma(s_0,s_4) \\ \gamma(s_0,s_5) \\ 1 \end{pmatrix}$$

For more information, please refer to the User Guide

Consequently, the `OrdinaryKriging` class needs a `Variogram` object as a mandatory attribute. Two very important optional attributes are `min_points` and `max_points`. They will limit the size of the Kriging equation system. As we have 200 observations, we can require at least 5 neighbors within the range. More than 15 will only unnecessarily slow down the computation. The `mode='exact'` attribute will advise the class to build and solve the system above for each location.

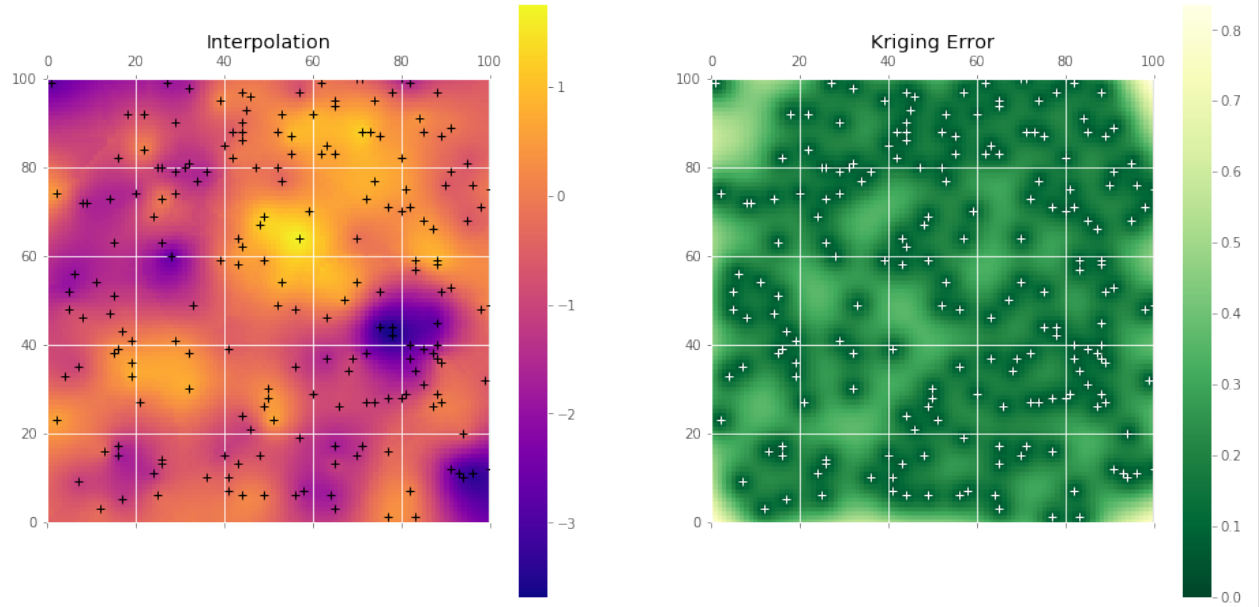
[10]:
```
ok = OrdinaryKriging(V, min_points=5, max_points=15, mode='exact')
```

The `transform` method will apply the interpolation for passed arrays of coordinates. It requires each dimension as a single 1D array. We can easily build a meshgrid of 100x100 coordinates and pass them to the interpolator. To recieve a 2D result, we can simply reshape the result. The Kriging error will be available as the `sigma` attribute of the interpolator.

[11]:
```
# build the target grid
xx, yy = np.mgrid[0:99:100j, 0:99:100j]
field = ok.transform(xx.flatten(), yy.flatten()).reshape(xx.shape)
s2 = ok.sigma.reshape(xx.shape)
```

And finally, we can plot the result.

[12]:
```
fig, axes = plt.subplots(1, 2, figsize=(16, 8))

art = axes[0].matshow(field.T, origin='lower', cmap='plasma')
axes[0].set_title('Interpolation')
axes[0].plot(data.x, data.y, '+k')
axes[0].set_xlim((0,100))
axes[0].set_ylim((0,100))
plt.colorbar(art, ax=axes[0])
art = axes[1].matshow(s2.T, origin='lower', cmap='YlGn_r')
axes[1].set_title('Kriging Error')
plt.colorbar(art, ax=axes[1])
axes[1].plot(data.x, data.y, '+w')
axes[1].set_xlim((0,100))
axes[1].set_ylim((0,100));
```



From the Kriging error map, you can see how the interpolation is very certain close to the observation points, but rather high in areas with only little coverage (like the upper left corner).

2.4.2 2 - Variogram Models

This tutorial will guide you through the theoretical variogram models available for the `Variogram` class.

In this tutorial you will learn:

- how to choose an appropiate model function
- how to judge fitting quality
- about sample size influence

[1]:
```
from skgstat import Variogram, OrdinaryKriging
import pandas as pd
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

[2]:
```
%env SKG_SUPPRESS = true
```

```
env: SKG_SUPPRESS=true
```

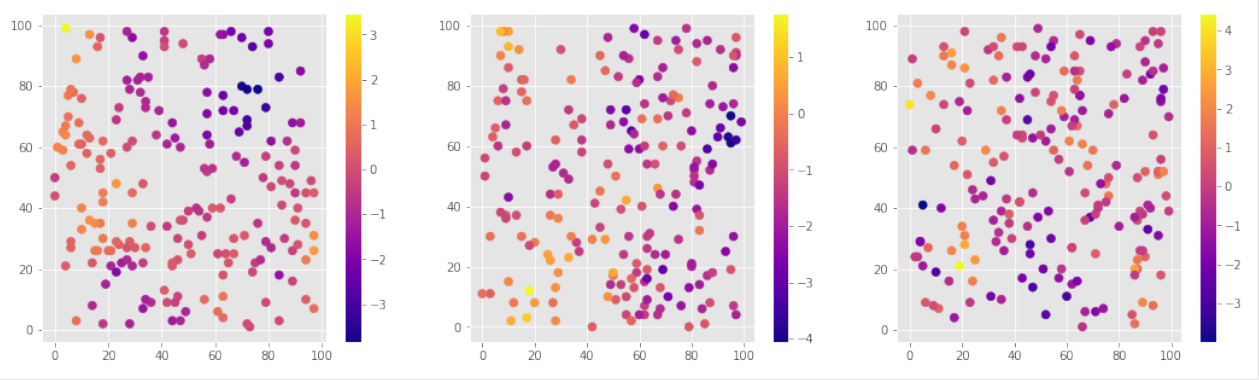
2.1 Load data

There are three prepared data sets in the `./data` folder. Each of them is a generated random field with different underlying spatial properties. We will use only the first one, but you can re-run all the examples with any of the other fields.

[3]:
```
data1 = pd.read_csv('data/sample_matern_15.csv')
data2 = pd.read_csv('data/sample_matern_40.csv')
data3 = pd.read_csv('data/sample_spherical_noise.csv')
```

[4]:
```
def plot_scatter(data, ax):
    art = ax.scatter(data.x, data.y, 50, c=data.z, cmap='plasma')
    plt.colorbar(art, ax=ax)
```

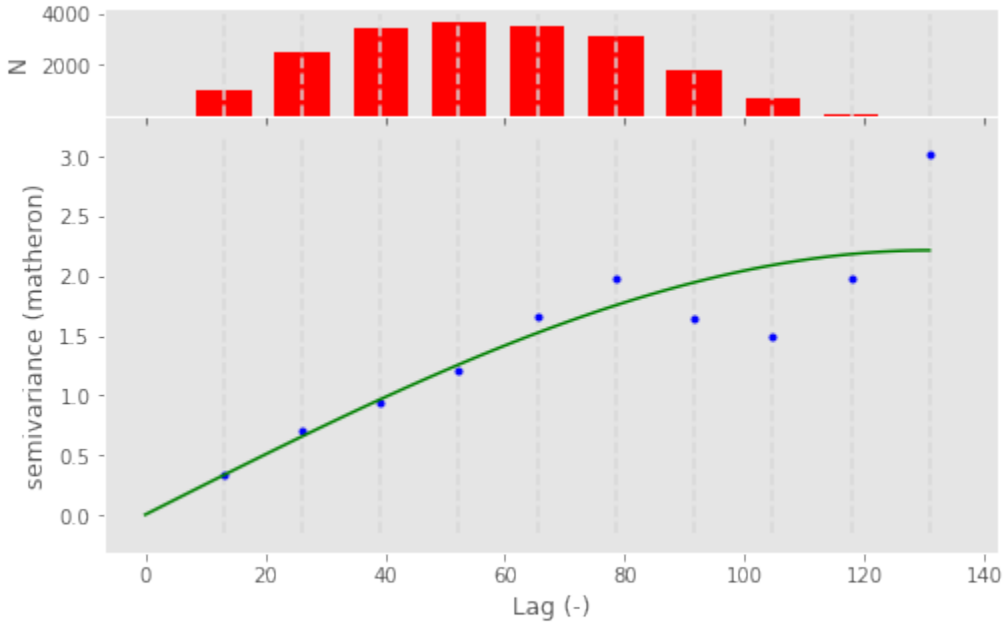
[5]:
```
fig, axes = plt.subplots(1, 3, figsize=(18, 5))
for data, ax in zip((data1, data2, data3), axes.flatten()):
    plot_scatter(data, ax)
```



2.2 Comparing theoretical models

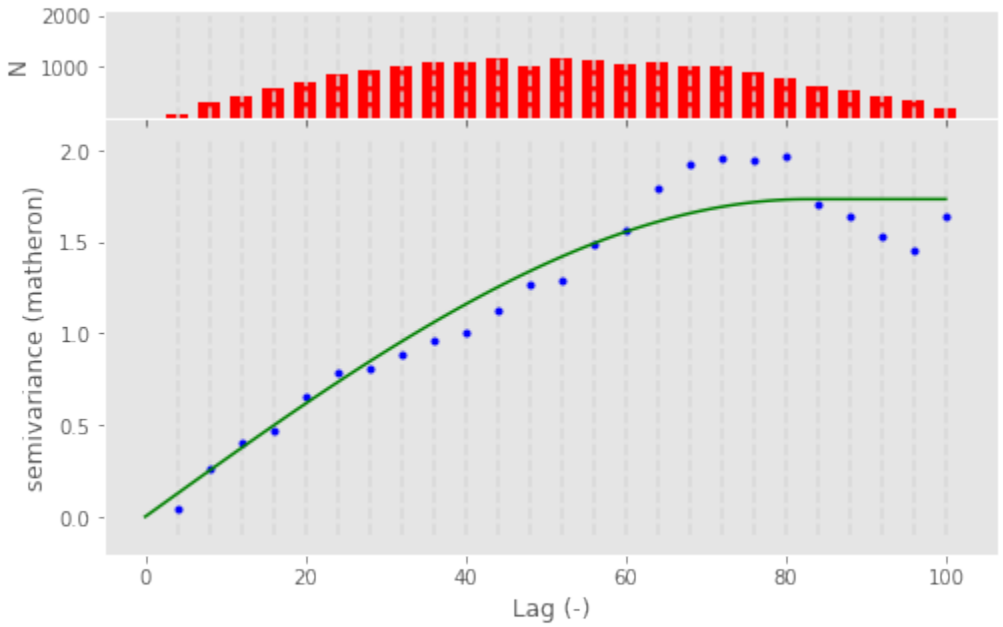
One of the features of `scikit-gstat` is the fact that it is programmed object oriented. That means, we can just instantiate a `Variogram` object and start changing arguments unitl it models spatial dependency in our observations well.

[6]:
```
V1 = Variogram(data1[['x', 'y']].values, data1.z.values, normalize=False)
V1.plot(show=False);
```



The data set includes 200 observations, consequently we can increase the number of lag classes. Additionally, the histogram shows, that the lags over 100 units are backed up by just a few observations. Thus, we can limit the lag classes to at least 100.

[7]:
```
V1.maxlag = 100
V1.n_lags = 25
V1.plot(show=False);
```

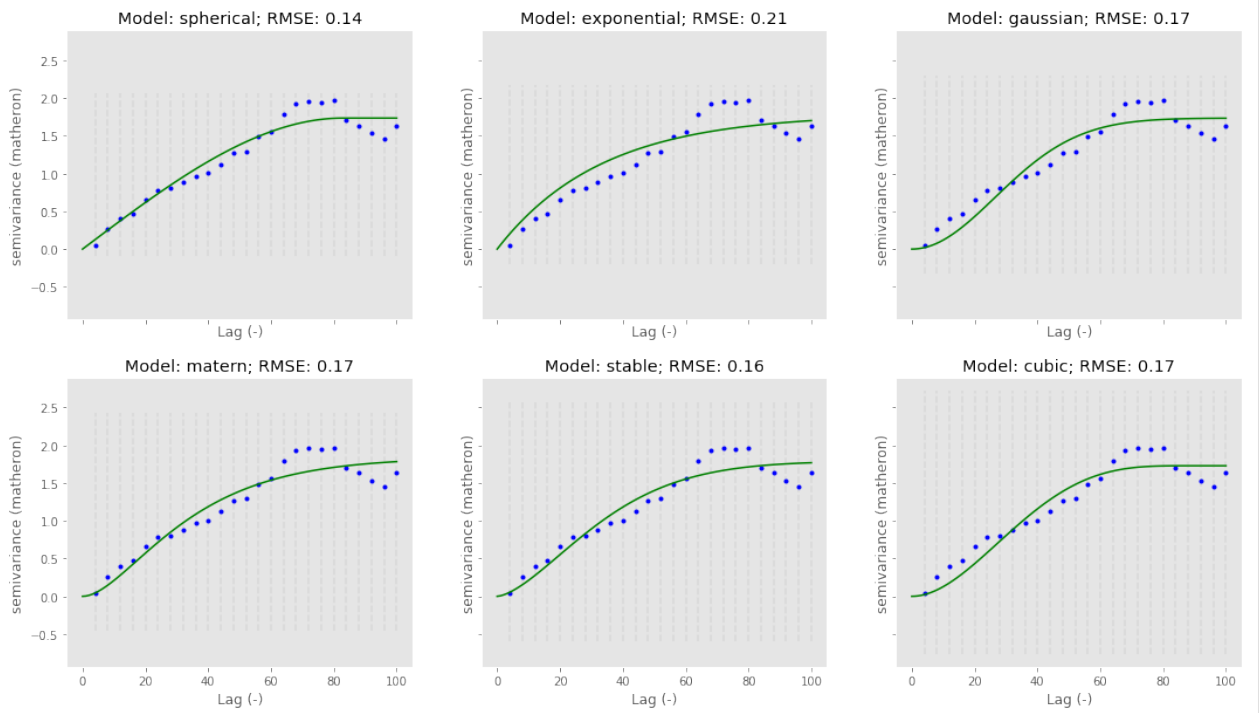


That's not too bad. Now, we can try different theoretical models. It is always a good idea to judge the fit visually, Especially, because we want it to fit to close bins more accurately than to distant bins, as they will ultimately determine the Kriging weights. Nevertheless, `Variogram` has a `rmse` and a `r2` property, that can be used as a quality measure for the fit. The `Variogram.plot` function also accepts one or two matplotlib subplot axes to plot the lag classes histogram

and variogram plot into them. The histogram can also be turned off.

[8]:
```
fig, _a = plt.subplots(2,3, figsize=(18, 10), sharex=True, sharey=True)
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
↪'cubic')):
    V1.model = model
    V1.plot(axes=axes[i], hist=False, show=False)
    axes[i].set_title('Model: %s; RMSE: %.2f' % (model, V1.rmse))
```

```
e:\dropbox\python\scikit-gstat\skgstat\models.py:17: RuntimeWarning: invalid value␣
↪encountered in double_scalars
  return func(*args, **kwargs)
```



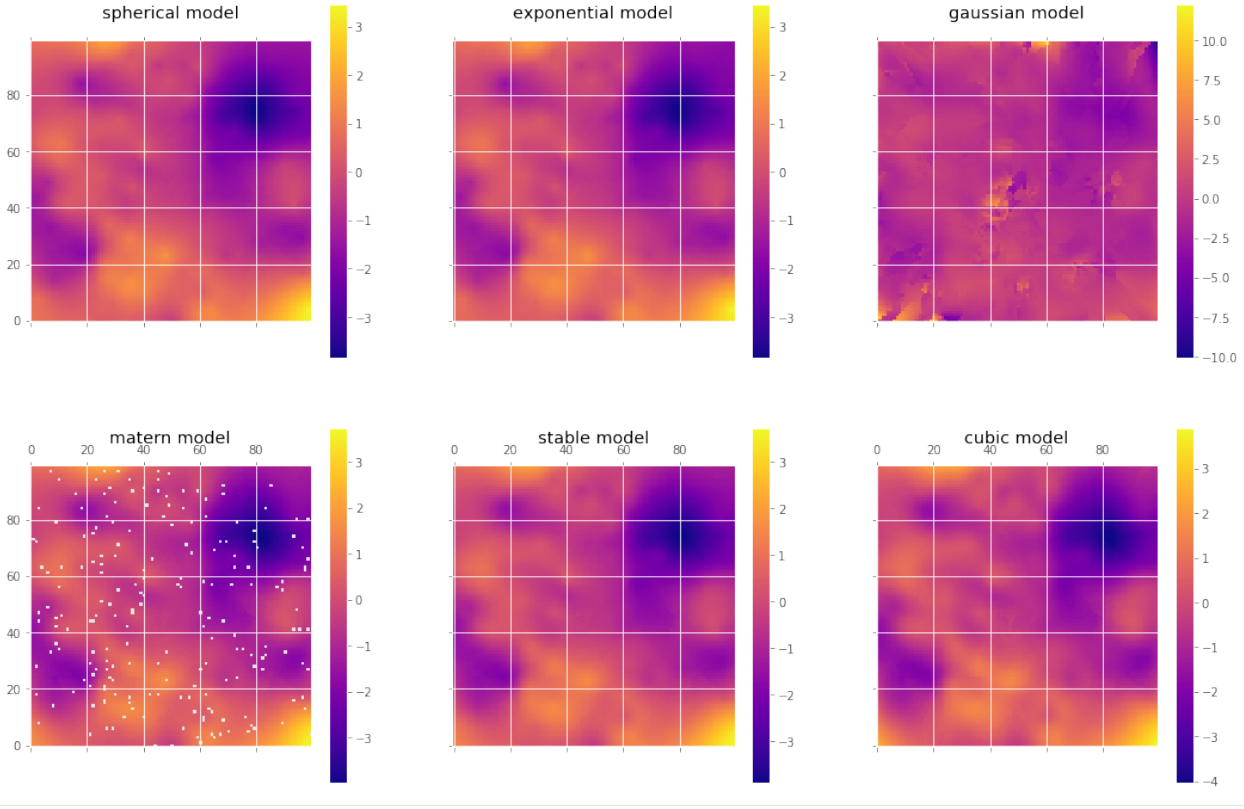
This is quite important. We find all 6 models to describe the experimental variogram equally well in terms of RMSE. However, the `cubic` and `gaussian` model are off the experimental values almost all the time. On short distances, the model is underestimating and on medium distances (up to the effective range) it is overestimating. The `exponential` model is overestimating all the time. The `spherical`, `matern` and `stable` model seem to be pretty good on short distances.

But what does this difference look like, when it comes to interpolation?

[9]:
```
def interpolate(V, ax):
    xx, yy = np.mgrid[0:99:100j, 0:99:100j]
    ok = OrdinaryKriging(V, min_points=5, max_points=15, mode='exact')
    field = ok.transform(xx.flatten(), yy.flatten()).reshape(xx.shape)
    art = ax.matshow(field, origin='lower', cmap='plasma')
    ax.set_title('%s model' % V.model.__name__)
    plt.colorbar(art, ax=ax)
    return field
```

[10]:
```
fields = []
fig, _a = plt.subplots(2,3, figsize=(18, 12), sharex=True, sharey=True)
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
↪'cubic')):
    V1.model = model
    fields.append(interpolate(V1, axes[i]))
```

```
e:\dropbox\python\scikit-gstat\skgstat\models.py:14: RuntimeWarning: invalid value␣
↪encountered in double_scalars
  mapping = map(lambda h: func(h, *new_args, **kwargs), args[0])
```



[11]:
```
pd.DataFrame({'spherical': fields[0].flatten(), 'exponential': fields[1].flatten(),
↪'gaussian': fields[2].flatten(),
              'matern': fields[3].flatten(), 'stable': fields[4].flatten(), 'cubic':␣
↪fields[5].flatten()}).describe()
```

[11]:

	spherical	exponential	gaussian	matern	stable	\
count	10000.000000	10000.000000	10000.000000	9800.000000	10000.000000	
mean	-0.422579	-0.427136	-0.376234	-0.422088	-0.419537	
std	1.106599	1.101533	1.546581	1.157061	1.140925	
min	-3.841342	-3.841342	-10.081703	-3.977538	-3.893159	
25%	-1.096673	-1.093462	-1.227488	-1.123188	-1.117248	
50%	-0.313185	-0.313439	-0.301236	-0.295343	-0.307810	
75%	0.301038	0.297955	0.502080	0.327033	0.322166	
max	3.467911	3.467911	12.305929	3.727784	3.718852	

	cubic

(continues on next page)

(continued from previous page)

```
count  10000.000000
mean      -0.412559
std        1.165185
min       -4.030550
25%       -1.088648
50%       -0.289786
75%        0.333818
max        3.887957
```

While most of the results look fairly similar there are a few things to notice:

1. Gaussian model is far off, producing estimations far outside the observed value ranges
2. All other models produce quite comparable numbers
3. The Matérn model fails, when recalculating the observations themselves

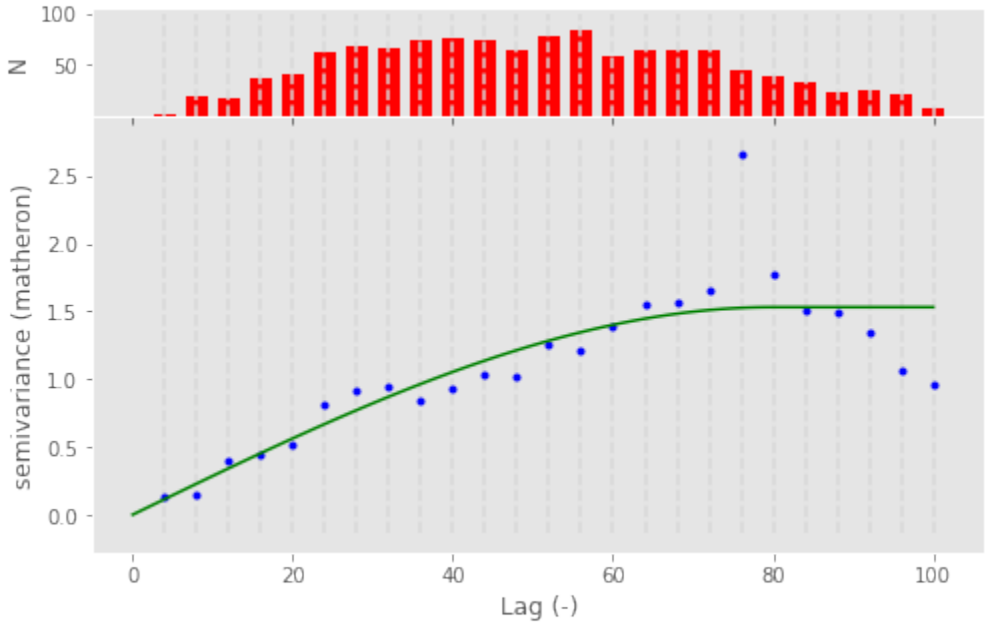
You have to remind that we had quite some observations. How does that look like, when the number of observations is decreased?

2.3 Using less observations

2.3.1 50% of all observations

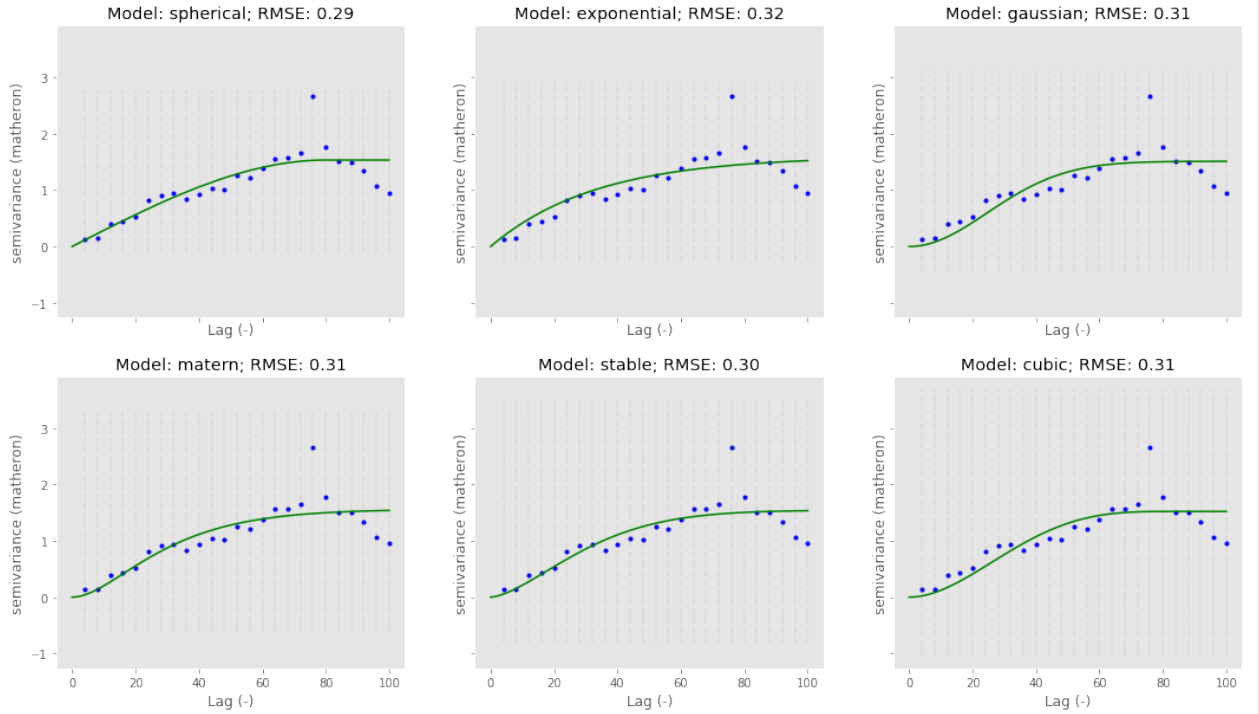
In this section we will run the same code, but on just a quarter and 10% of all available observations. First, we look into the variograms:

[12]:
```
subset1 = data1.iloc[:50]
V2 = Variogram(subset1[['x', 'y']].values, subset1.z.values, normalize=False, maxlag=100,
↪ n_lags=25)
V2.plot(show=False);
```

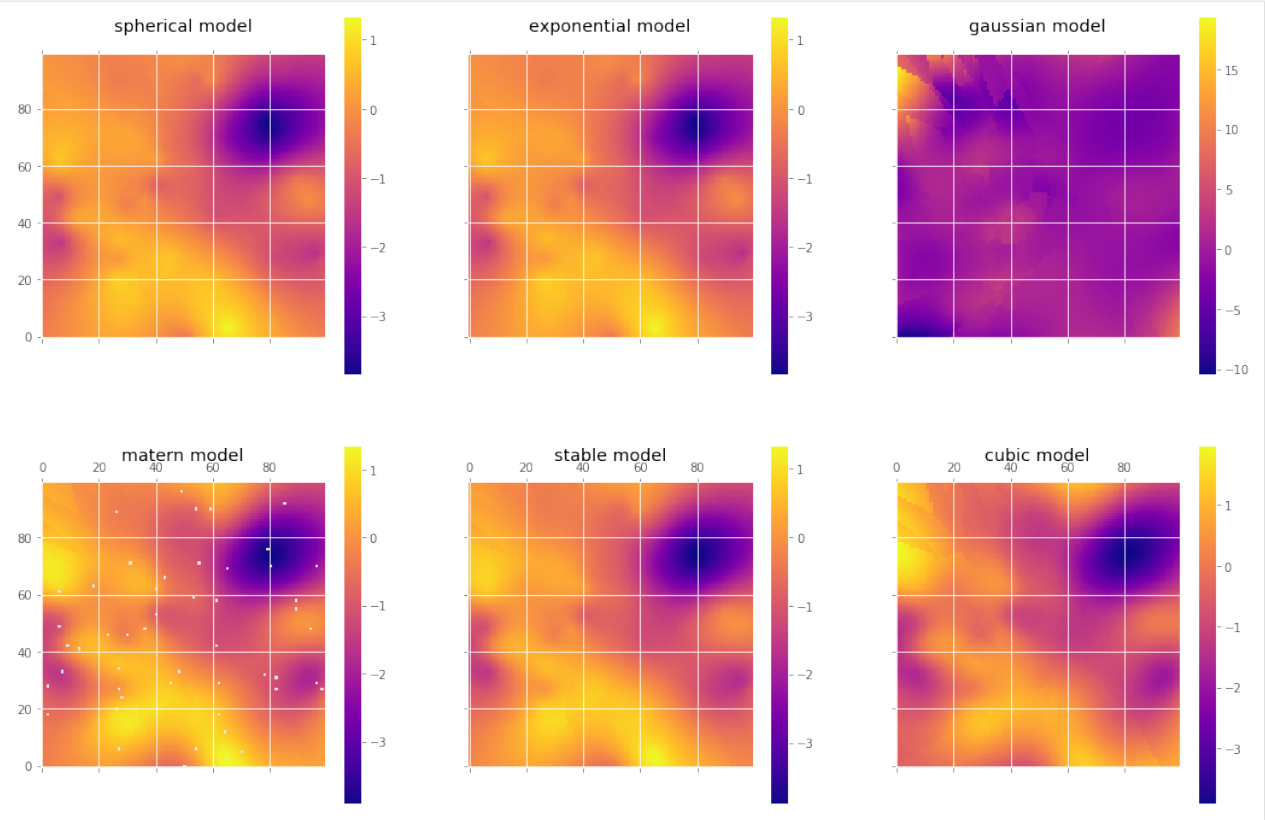


[13]:
```
fig, _a = plt.subplots(2,3, figsize=(18, 10), sharex=True, sharey=True)
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
↪'cubic')):
    V2.model = model
    V2.plot(axes=axes[i], hist=False, show=False)
    axes[i].set_title('Model: %s; RMSE: %.2f' % (model, V2.rmse))
```

```
e:\dropbox\python\scikit-gstat\skgstat\models.py:17: RuntimeWarning: invalid value␣
↪encountered in double_scalars
  return func(*args, **kwargs)
```



[14]:
```
fields = []
fig, _a = plt.subplots(2,3, figsize=(18, 12), sharex=True, sharey=True)
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
↪'cubic')):
    V2.model = model
    fields.append(interpolate(V2, axes[i]))
```



[15]:
```
pd.DataFrame({'spherical': fields[0].flatten(), 'exponential': fields[1].flatten(),
↪'gaussian': fields[2].flatten(),
              'matern': fields[3].flatten(), 'stable': fields[4].flatten(), 'cubic':␣
↪fields[5].flatten()}).describe()
```

[15]:

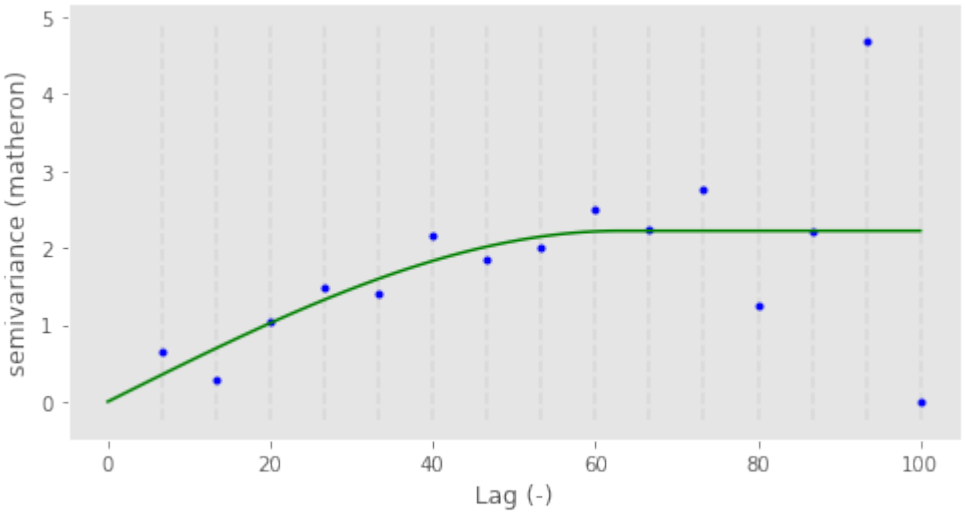
	spherical	exponential	gaussian	matern	stable	cubic
count	10000.000000	10000.000000	10000.000000	9950.000000	10000.000000	10000.000000
mean	-0.492584	-0.498393	-0.229285	-0.474786	-0.482607	-0.436488
std	0.933770	0.913543	2.596812	1.022044	0.987613	1.093714
min	-3.841342	-3.841342	-10.442055	-3.908236	-3.883908	-3.907538
25%	-0.968383	-0.961082	-1.519022	-1.008619	-0.981728	-1.026079
50%	-0.250866	-0.263681	-0.489828	-0.286338	-0.271219	-0.307099
75%	0.179633	0.159811	0.737149	0.265865	0.228785	0.352761
max	1.331839	1.331839	19.360127	1.346393	1.331839	1.973759

- The Gaussian model is of course still far off
- All other models struggle in hitting the high values. They are far off on the upper bound.

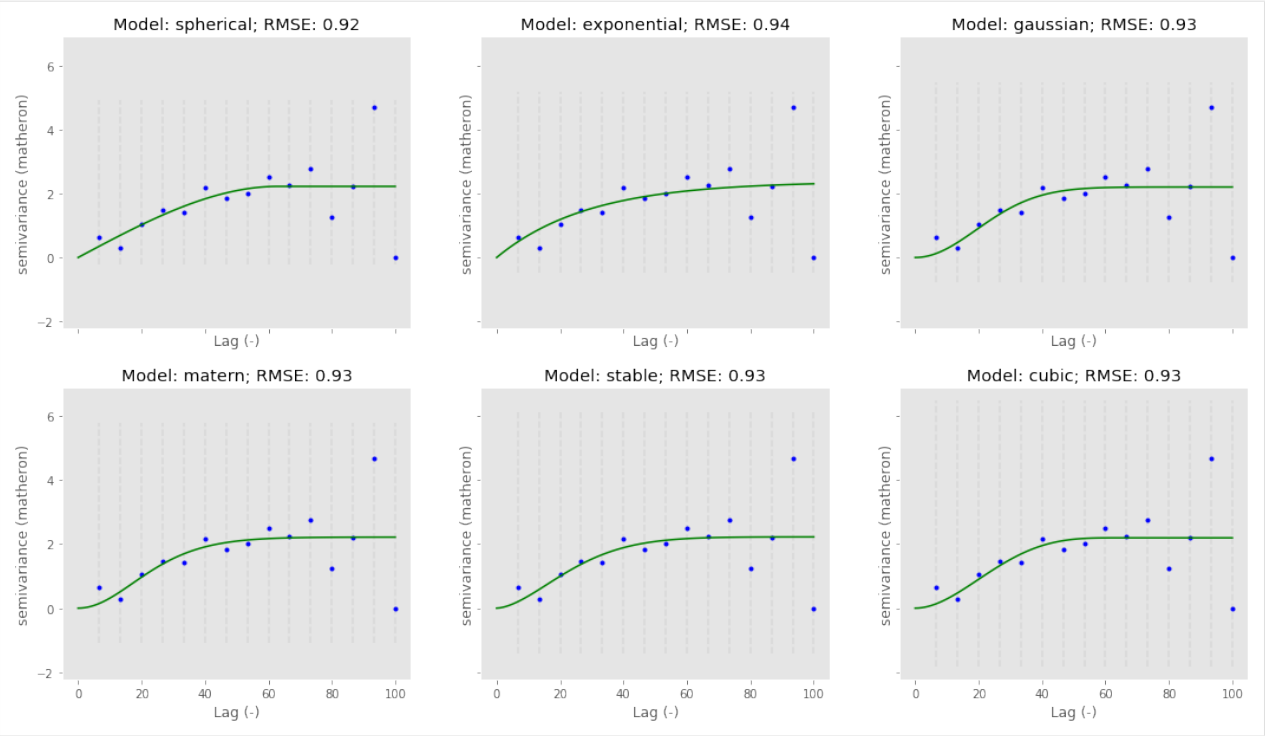
2.3.1 10% of all observations

In this section we will run the same code, but on just a quarter and 10% of all available observations. First, we look into the variograms:

[16]:
```
subset2 = data1.iloc[180:]
V3 = Variogram(subset2[['x', 'y']].values, subset2.z.values, normalize=False, maxlag=100,
↪ n_lags=15)
V3.plot(hist=False, show=False);
```



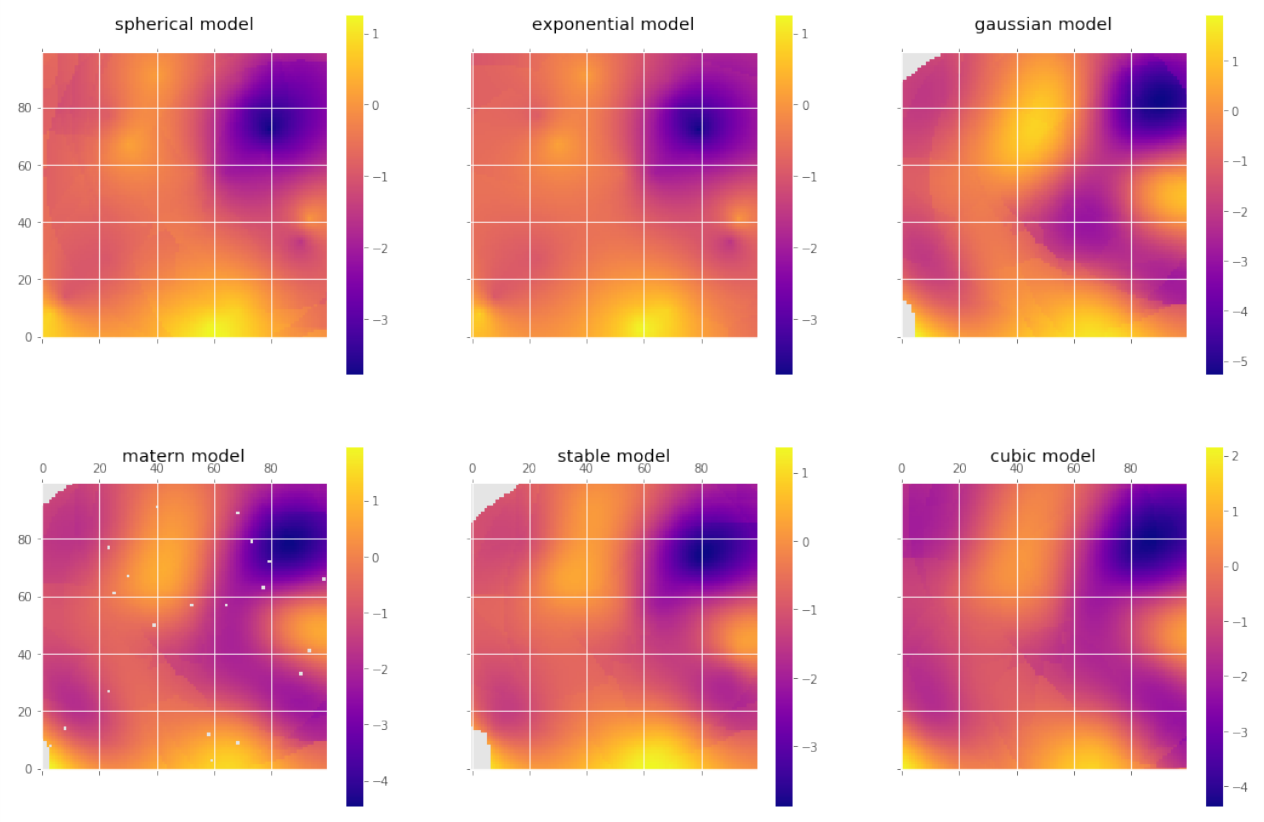
[17]:
```
fig, _a = plt.subplots(2,3, figsize=(18, 10), sharex=True, sharey=True)
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
↪'cubic')):
    V3.model = model
    V3.plot(axes=axes[i], hist=False, show=False)
    axes[i].set_title('Model: %s; RMSE: %.2f' % (model, V3.rmse))
```



In this example, we were basing the variogram analysis on only 20 observations. That is a number that could be considered to be the lower bound of geostatistics. The RMSE values are decreasing as the experimental variograms are more scattered. However, All six models seem to fit fairly well to the experimental data. It is hard to tell from just the figure above which is correct.

[18]:
```
fields = []
fig, _a = plt.subplots(2,3, figsize=(18, 12), sharex=True, sharey=True)
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
↪'cubic')):
    V3.model = model
    fields.append(interpolate(V3, axes[i]))
```

```
Warning: for 144 locations, not enough neighbors were found within the range.
Warning: for 76 locations, not enough neighbors were found within the range.
Warning: for 214 locations, not enough neighbors were found within the range.
```



[19]:
```
pd.DataFrame({'spherical': fields[0].flatten(), 'exponential': fields[1].flatten(),
↪'gaussian': fields[2].flatten(),
              'matern': fields[3].flatten(), 'stable': fields[4].flatten(), 'cubic':␣
↪fields[5].flatten()}).describe()
```

[19]:

	spherical	exponential	gaussian	matern	stable	cubic
count	10000.000000	10000.000000	9856.000000	9904.000000	9786.000000	10000.000000
mean	-0.808890	-0.793323	-1.094349	-0.994542	-0.899081	-1.058428
std	0.887960	0.820827	1.366513	1.127764	0.948683	1.124708
min	-3.785324	-3.785324	-5.277685	-4.458332	-3.880787	-4.364176
25%	-1.063349	-1.033391	-1.904575	-1.629145	-1.314486	-1.698163
50%	-0.671072	-0.621459	-0.963145	-0.899756	-0.801062	-1.004205
75%	-0.288246	-0.334686	-0.125231	-0.207966	-0.277770	-0.275732
max	1.259305	1.259305	1.914923	1.960789	1.380699	2.162505

Here, some interesting things happend:

1. The Gaussian model is performing well again.

2. There are substantial differences between the interpolation results
3. In many runs, NaN values were produced, because not enough neighbors could be found

We decreased the number of observations so far, that the `max_points` attribute came into effect. In the other cases the Kriging interpolator found so many close observations, that limiting them to 15 had the effect, that estimations were usually derived from obserations within a few units. Now, even if enough points are within the range, we use observations from medium distances. Here, the different shapes of the models come into effect, as could be seen from the last example.

2.4.3 3 - Semi-variance Estimators

This tutorial focuses on experimental variograms. It will guide you through the main semi-variance estimators available in `scikit-gstat`. Additionally, most of the parameters available for building an experimental variogram will be discussed.

In this tutorial you will learn:

- what estimators are available
- how they differ

[1]:
```
from skgstat import Variogram, OrdinaryKriging
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

[2]:
```
%env SKG_SUPPRESS=true
```

```
env: SKG_SUPPRESS=true
```

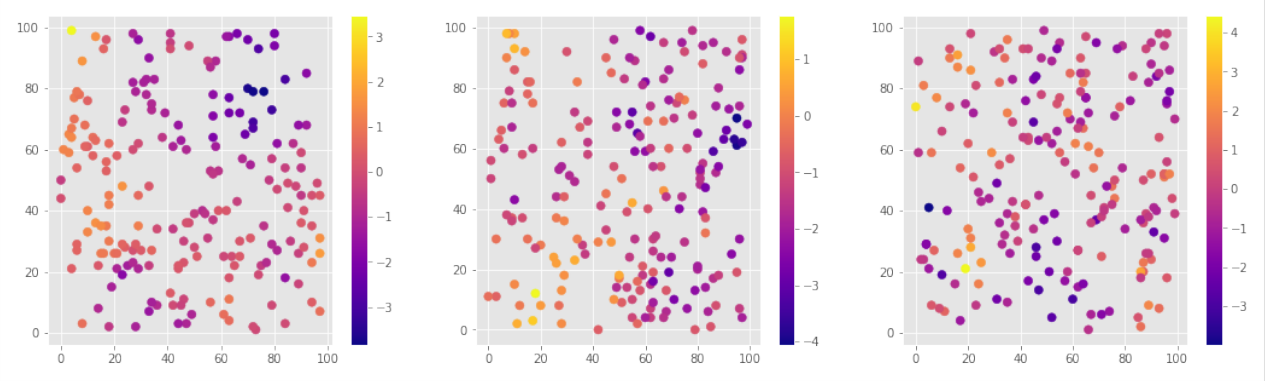
3.1 Load data

There are three prepared data sets in the `./data` folder. Each of them is a generated random field with different underlying spatial properties. We will use only the thrid one, but you can re-run all the examples with any of the other fields.

[3]:
```
data1 = pd.read_csv('data/sample_matern_15.csv')
data2 = pd.read_csv('data/sample_matern_40.csv')
data3 = pd.read_csv('data/sample_spherical_noise.csv')
```

[4]:
```
def plot_scatter(data, ax):
    art = ax.scatter(data.x, data.y, 50, c=data.z, cmap='plasma')
    plt.colorbar(art, ax=ax)
```

[5]:
```
fig, axes = plt.subplots(1, 3, figsize=(18, 5))
for data, ax in zip((data1, data2, data3), axes.flatten()):
    plot_scatter(data, ax)
```



3.2 Comparing estimators

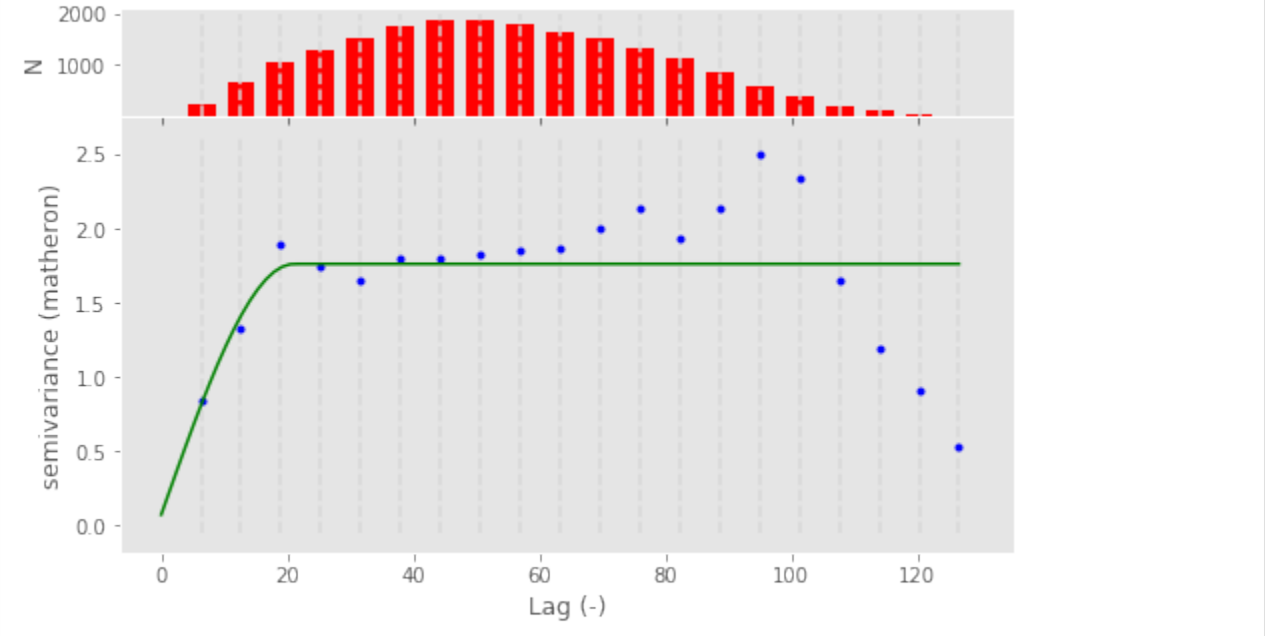
The default estimator configured in `Variogram` is the Mathéron estimator (Mathéron, 1963). It is defined like:

$$\gamma(h) = \frac{1}{2N(h)} * \sum_{i=1}^{N(h)} (Z(x_i) - Z(x_{i+h}))^2$$

where:

- h is the distance lag
- $N(h)$ is the number of observation pairs in h-lag class
- $Z(x_i)$ is the observation at the i-th location x

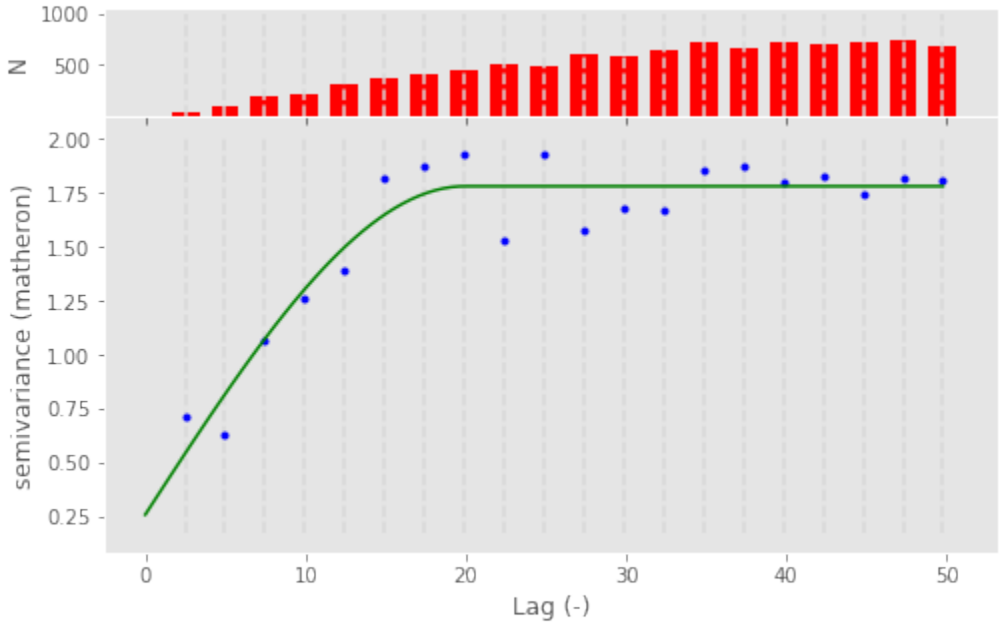
[6]:
```
V1 = Variogram(data3[['x', 'y']].values, data3.z.values, normalize=False, n_lags=20, use_
↪nugget=True)
V1.plot(show=False);
```



Following the histogram, we should set a `maxlag`. This property accepts a number $0 < maxlag < 1$ to set the `maxlag` to this ratio of the maximum separating distance. A number > 1 will use this at an absolute limit. You can also pass

'mean' or 'median'. This will calculate and set the mean or median of all distances in the distance matrix as maxlag.

[7]:
```
V1.maxlag = 'median'
V1.plot(show=False);
```



3.3 Alternative estimators

scikit-gstat implements more than only the Mathéron estimator. Setting estimator='cressie' will set the Cressie-Hawkins estimator. It is implemented as follows (Cressie and Hawkins, 1980):

$$2\gamma(h) = \frac{\left(\frac{1}{N(h)} \sum_{i=1}^{N(h)} |Z(x_i) - Z(x_{i+h})|^{0.5}\right)^4}{0.457 + \frac{0.494}{N(h)} + \frac{0.045}{N^2(h)}}$$

By setting estimator='dowd', the Dowd estimator (Dowd, 1984) will be used:

$$2\gamma(h) = 2.198 * median(Z(x_i) - Z(x_{i+h}))^2$$

Finally, estimator='genton' will set the Genton estimator (Genton, 1998):

$$\gamma(h) = 2.2191\{|V_i(h) - V_j(h)|; i < j\}_{(k)}$$

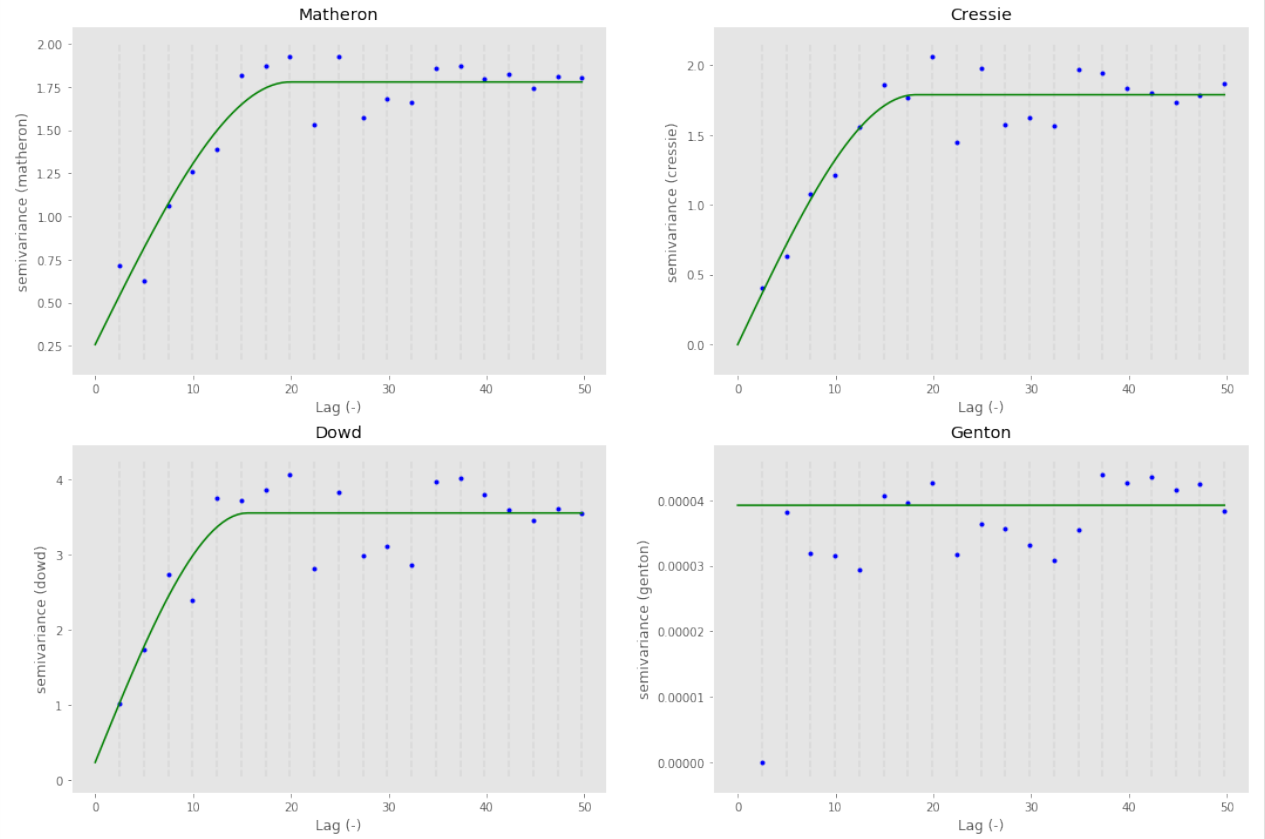
with:

$$k = \binom{[N_h/2] + 1}{2}$$

and:

$$q = \binom{N_h}{2}$$

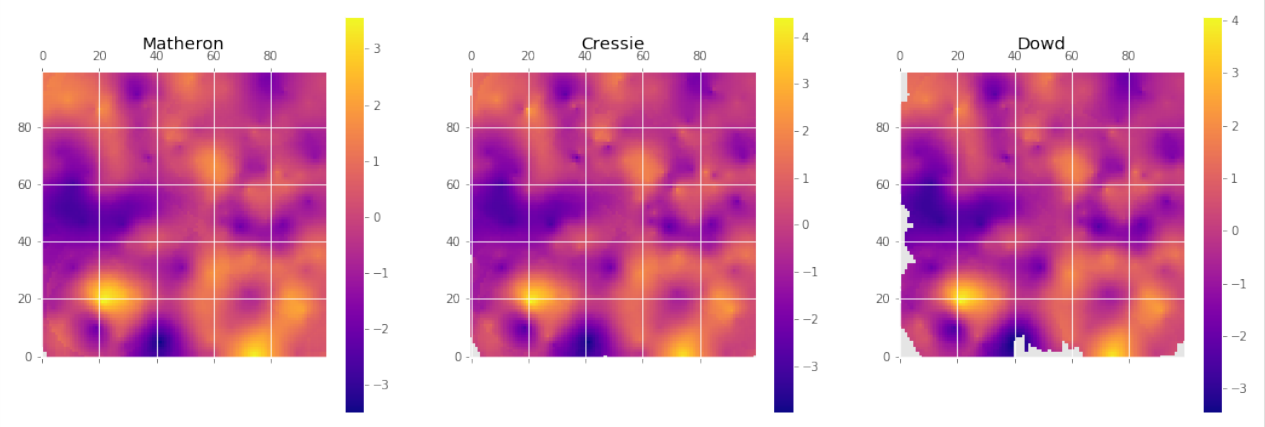
[8]:
```
fig, _a = plt.subplots(2, 2, figsize=(18, 12))
axes = _a.flatten()
for ax, est in zip(axes, ('matheron', 'cressie', 'dowd', 'genton')):
    V1.estimator = est
    V1.plot(axes=ax, hist=False, show=False)
    ax.set_title(est.capitalize())
```



For Kriging, the difference on the first few lag classes is important, as no points will be used for estimation, that lies outside the range.

[9]:
```
xx, yy = np.mgrid[0:99:100j, 0:99:100j]
fig, _a = plt.subplots(1, 3, figsize=(18, 6))
axes = _a.flatten()

for ax, est in zip(axes, ('matheron', 'cressie', 'dowd')):
    V1.estimator = est
    ok = OrdinaryKriging(V1, min_points=5, max_points=15, mode='exact')
    field = ok.transform(xx.flatten(), yy.flatten()).reshape(xx.shape)
    art = ax.matshow(field, origin='lower', cmap='plasma')
    plt.colorbar(art, ax=ax)
    ax.set_title(est.capitalize())
```

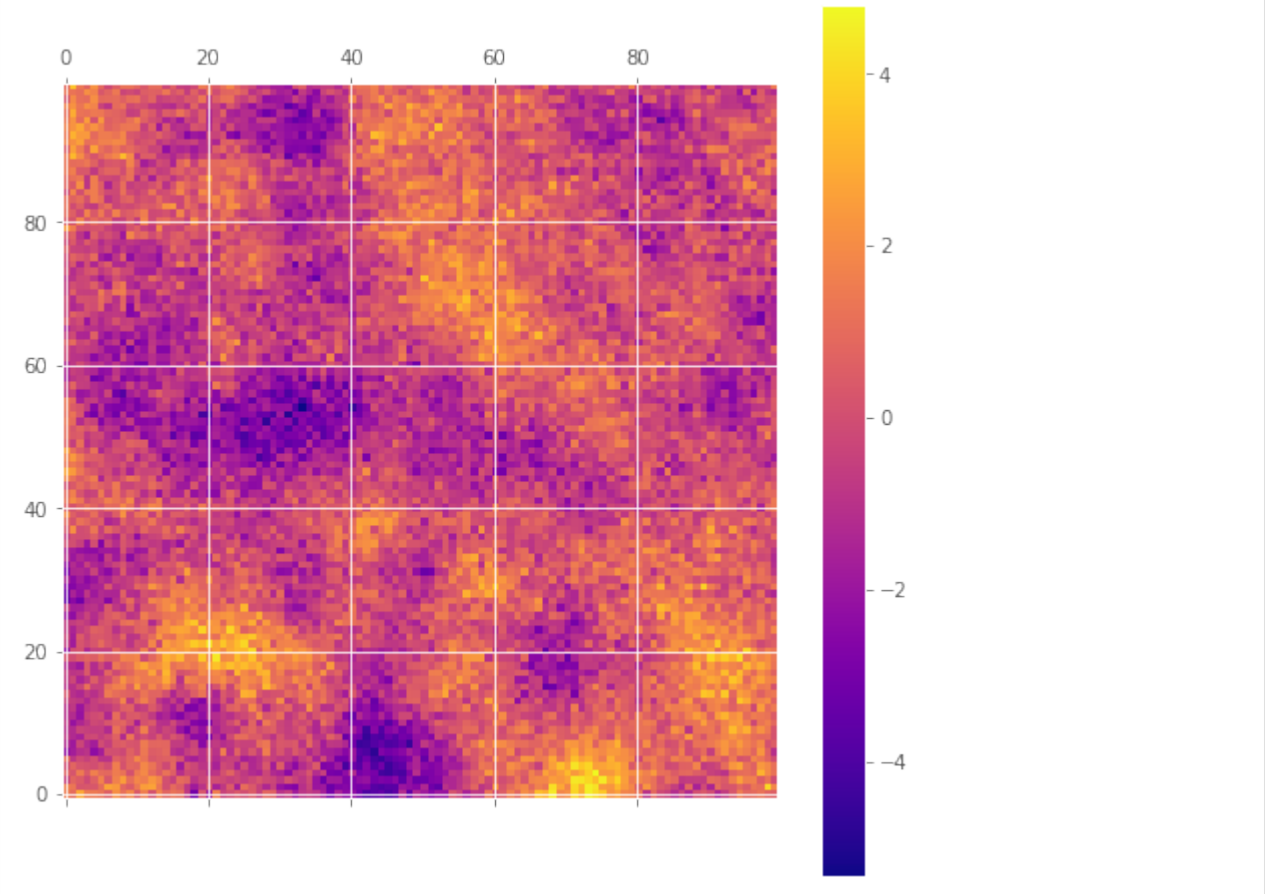
```
Warning: for 6 locations, not enough neighbors were found within the range.
Warning: for 27 locations, not enough neighbors were found within the range.
Warning: for 286 locations, not enough neighbors were found within the range.
```



You can see from these results that the Cressie and the Dowd estimator pronounce the extreme values more. The original field is also in the `./data` folder. We can load it for comparison.

[10]:
```
rf = np.loadtxt('data/rf_spherical_noise.txt')
fig, ax = plt.subplots(1, 1, figsize=(8,8))
art = ax.matshow(rf, origin='lower', cmap='plasma')
plt.colorbar(art, ax=ax)
```

[10]:
```
<matplotlib.colorbar.Colorbar at 0x2a79e168348>
```



All three variograms were not able to capture the random variability. But the Matheron estimator was also not able to

reconstruct the maxima from the random field.

3.4 References

Cressie, N., and D. Hawkins (1980): Robust estimation of the variogram. Math. Geol., 12, 115-125.

Dowd, P. A., (1984): The variogram and kriging: Robust and resistant estimators, in Geostatistics for Natural Resources Characterization. Edited by G. Verly et al., pp. 91 - 106, D. Reidel, Dordrecht.

Genton, M. G., (1998): Highly robust variogram estimation, Math. Geol., 30, 213 - 221.

Matheron, G. (1963). Principles of geostatistics. Economic Geology, 58(8), 1246–1266. https://doi.org/10.2113/gsecongeo.58.8.1246

2.4.4 4. Plotting

At the core of SciKit-GStat is a set of classes, that can be used interactively to perform variogram analysis. One important aspect of this analysis is a rich collection of plotting functions. These are directly available as class methods of the `Variogram`, `DirectionalVariogram` and `SpaceTimeVariogram` method. With version `0.3.3`, SciKit-GStat implements two different plotting backend: matplotlib and plotly. Generally speaking, matplotlib is great for creating publication ready figures in a variety of formats, including vector-graphic PDF files. Plotly, on the other hand, will translate the figure into their Javascript library and open a webbrowser with an interactive plot. This way you can obtain the same figure either for your publication as PDF, or as a HTML object that can be injected into a project report website.

With the newly introduced `skgstat.plotting` backend, you can easily read and change the backend with a single convenient function. The default backend is `'matplotlib'`. Please be aware, that `plotly` is only a soft dependency, meaning you need to take care of the installation yourself, to keep SciKit-GStat's dependency list shorter.

The data used to create the `Variogram` and `DirectionalVariogram` is from Thiesen et al. (2020). Here, the *short range* random field sample with noise, sampled with two hunded locations. The spatio-temporal data is derived from Fersch et al. (2020). From that data publication, the wireless sensor network data is used. The originaly published 15 minutes intervals soil temperature data at 20 cm depth was taken for all 55 stations and aggregated to mean hourly values. To further decrease the data size, only every 6th data point is used here. Estimating the full data set will take approx. 120GB RAM and processing took about 30 minutes. The results for the thinned data sample are very comparable.

Both data samples can either be obtained by the orignial publications, or from the SciKit-GStat documentation. Both samples are published under Creative Commons BY 4.0 license. Please cite the original publications if you use the data, and **not** SciKit-GStat.

References

Thiesen, Stephanie, et al. "Histogram via entropy reduction (HER): an information-theoretic alternative for geostatistics." Hydrology and Earth System Sciences 24.9 (2020): 4523-4540.

Fersch, Benjamin, et al. "A dense network of cosmic-ray neutron sensors for soil moisture observation in a pre-alpine headwater catchment in Germany." Earth System Science Data Discussions 2020 (2020): 1-35.

[1]:
```
import skgstat as skg
from skgstat.plotting import backend
import numpy as np
import pandas as pd
import json
import warnings
import plotly.graph_objects as go
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
from plotly.subplots import make_subplots
warnings.filterwarnings('ignore')
```

4.1 Load Data

Load the short-range and long range example from Thiesen et al. (2020). Only the short range example is used:

[2]:
```
sr = pd.read_csv('./data/sample_sr.csv')
lr = pd.read_csv('./data/sample_lr.csv')

c = sr.values[:,:2]
v = sr.values[:,2]
```

Estimate a variogram, with a few more lag classes, as there are enough observation points available.

[3]:
```
V = skg.Variogram(c,v, n_lags=25)
print(V)
```

```
spherical Variogram
-------------------
Estimator:         matheron
Effective Range:   38.69
Sill:              1.23
Nugget:            0.00
```

Load a artificial random field, generated from a Gaussian covariance function, with a 2x larger range in x-axis direction:

[4]:
```
an_field = np.loadtxt('./data/aniso_x2.txt')

np.random.seed(42)
c = np.random.randint(an_field.shape[0], size=(300, 2))
v = np.array([an_field[p[0], p[1]] for p in c])
```

Estimate the directional variogram with a few more lag classes and an azimuth of 90°. The tolerance is set rather low to illustrate the graphs better (fewer point connections.):

[5]:
```
DV = skg.DirectionalVariogram(c, v, n_lags=20, azimuth=40., tolerance=15.0)
print(DV)
```

```
spherical Variogram
-------------------
Estimator:         matheron
Effective Range:   41.20
Sill:              0.96
Nugget:            0.00
```

Load the TERENO soil temperature data from Fersch et al. (2020):

[6]:
```
with open('./data/tereno_fendt/tereno.json', 'r') as js:
    data_obj = json.load(js)
```

(continues on next page)

(continued from previous page)

```
coords = np.array(data_obj['coordinates'])
vals = np.array(data_obj['values'])
print(data_obj['description'])
```

```
Data derived from Fersch et al. (2020) https://doi.org/10.5194/essd-2020-48. Published␣
↪under CC BY 4.0.
 It is  From the WSN product, the T_a in 20cm depth is extracted
```

Estimate the spatio-temporal variogram with a product-sum model. Only every 6th hour is taken into account to decrease the memory footprint. If you use the full dataset, you need ~120 GiB RAM. The marginal variograms are kept as they are.

[7]:
```
STV = skg.SpaceTimeVariogram(coords, vals[:,::6], x_lags=20, t_lags=20, model='product-
↪sum')
print(STV)
```

```
<skgstat.SpaceTimeVariogram.SpaceTimeVariogram object at 0x7f1295593eb0>
```

4.2 Backend

You can switch to `plotly` as a plotting backend by calling the `plotting.backend` function and passing the name of the backend. Note that plotly is only a soft dependency and will not automatically be installed along with SciKit-GStat. You can install it like:

```
pip install plotly
```

4.3 Variogram

4.3.1 `Variogram.plot`

The `Variogram.plot` is the main plotting function in `scikit-gstat`. Before you use the variogram for further geostatistical methods, like kriging, or further analysis, make sure, that a suitable model was found and fitted to the experimental data. Further, you have to make sure that the statistical foundation of this estimation is sound, the lag classes are well designed and backed by a suiatable amount of data. Otherwise, any other geostatistical analysis or method will have to fail, no matter how nice the results might look like.

[8]:
```
# from skgstat.plotting import backend
backend('plotly')
```

Plotly

[9]:
```
fig = V.plot()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

A useful argument for `plot` is the `ax`, this takes a `matplotlib.AxesSubplot` for the `'matplotlib'` backend and a `plotly.Figure` for the `'plotly'` backend. You need to supply the correct amount of subplots (two). For convenience, the histogram in the upper subplot can be disabled.

[10]:
```
fig = make_subplots(rows=1, cols=1)
fig.update_layout(
    width=800,
    height=200,
    template='seaborn',
    showlegend=False,
    margin=dict(l=0, r=0, b=0, t=0)
)

V.plot(axes=fig, hist=False, show=False)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

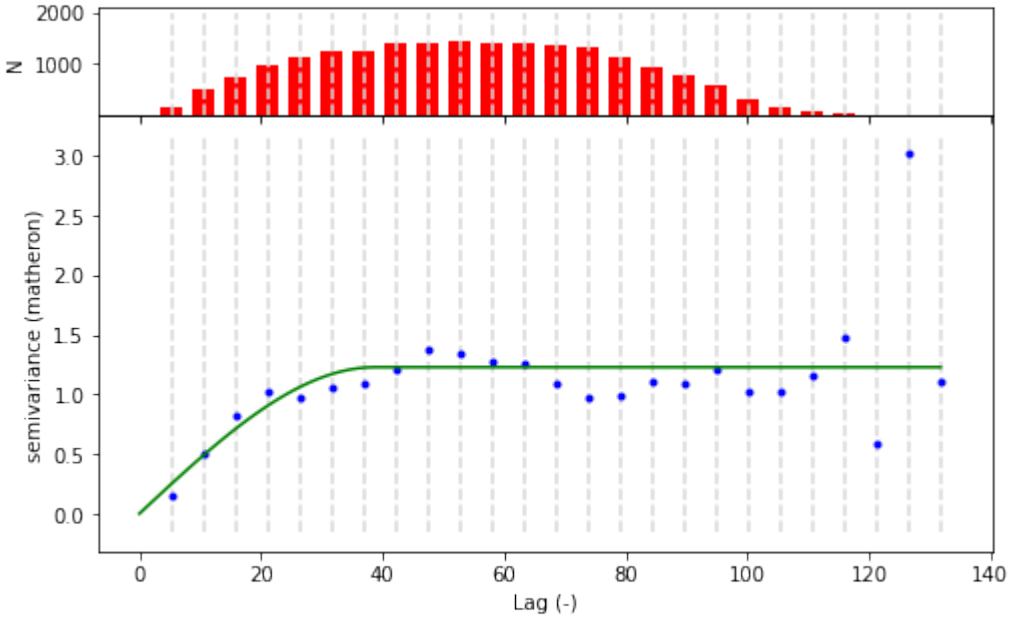
The `plot` functions is customizable and takes a lot of arguments. However, the same interface is used as for the `matplotlib` version of that function. Many matplotlib arguments are mapped to the corresponding plotly arguments. Beyond that, you can either try common plotly arguments, or update the figure afterwards:

[11]:
```
fig = V.plot(show=False)

fig.update_layout(
    legend=dict(x=0.05, y=1.1, xanchor='left', yanchor='top', orientation='h'),
    template='plotly_dark',
    annotations=[dict(
        text="AWESOME",
        xref="paper",
        yref="paper",
        x=0.5,
        y=0.5,
        font=dict(color="white", size=100),
        textangle=-30,
        opacity=.3
    )]
)
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

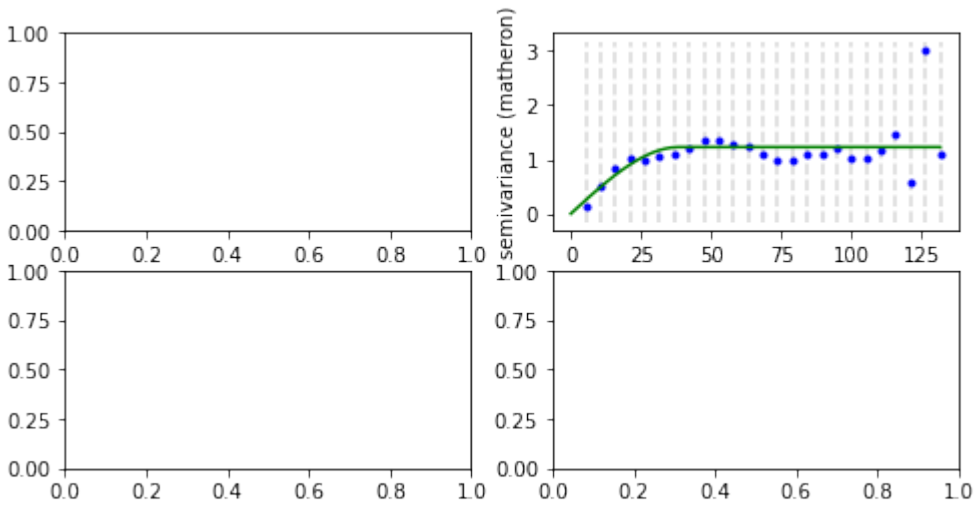
Matplotlib

[12]:
```
backend('matplotlib')

fig = V.plot()
```



With matplotlib, you can set any `matplotlib.AxesSubplot` as `axes` to plot on other figures. You can send two axes, for the variogram and the histogram, or only one and disable the histogram plotting.

[13]:
```
fig, ax = plt.subplots(2,2, figsize=(8, 4))

fig = V.plot(axes=ax.flatten()[1], hist=False)
```



4.3.2 Variogram.scattergram

You can plot a scattergram of all point pairs formed by the class. The pairs can be grouped by the lag classes, they were formed in. This way you can analyze how the two values of the point pait (head and tail) scatter and if this follows a pattern (i.e. anisotropy). It is recommended to use the `'plotly'` backend, as you can click on the legend entries to hide a specific class, or double-click to show only the selected lag class. This makes it much easier to inspect the classes.

Plotly

[14]:
```
backend('plotly')
fig = V.scattergram()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

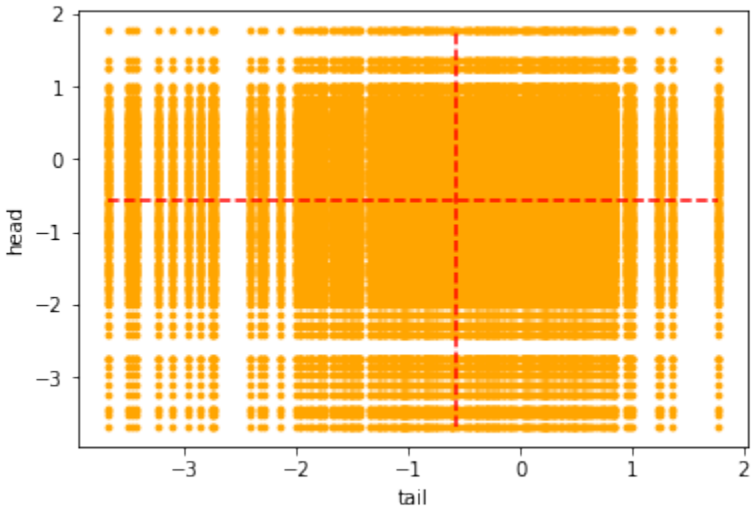
It is, however possible to re-create the plot that was used up to SciKit-GStat version `0.3.0` with only one color. This is still the default for the `'matplotlib'` backend.

[15]:
```
fig = V.scattergram(single_color=True)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Matplotlib

[16]:
```
backend('matplotlib')
fig = V.scattergram()
```



4.3.3 Variogram.location_trend

Another useful helper plot is the `location_trend`. This will plot the observation values related to their coordinate position, for each coordinate dimension separatedly. With the `'plotly'` backend, each dimension will appear as a coloured group in a single plot. By double-clicking the legend, you can inspect each group separately.

The `'plotly'` backend will automatically switch the used plot type from a ordinary scatter-plot to a WebGL backed scatter-plot, if there are more than 5000 observations. This will add some startup-overhead for the plot to appear, but the interactivity actions (like pan, zoom) are speed up by magnitudes.

Plotly

[17]:
```
backend('plotly')
fig = V.location_trend()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Since version `0.3.5` the `location_trend` function accepts a `add_trend_line` parameter, that defaults to `False`. If set to true, the class will fit linear models to each of the point clouds and output a trend line. It will also calculate the R^2, which you can use to either accept the input data as trend free or not (a high R^2 indicates a **linear** trend and hence you should decline using the input data).

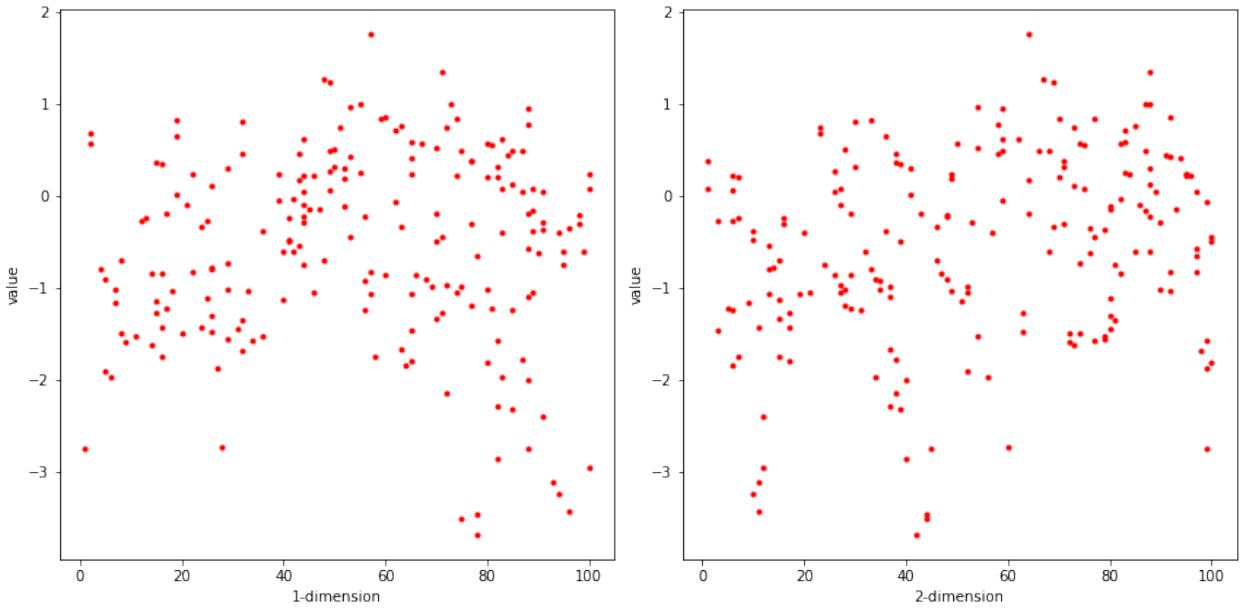
[18]:
```
fig = V.location_trend(add_trend_line=True)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Matplotlib

There is a difference between the `'matplotlib'` and `plotly'` backend in this plotting function. As Plotly utilizes the legend by default to show and hide traces on the plot, the user can conveniently switch between the coordinate dimensions. In Matplotlib, the figures are not interactive by default and therefore `scikit-gstat` will create one subplot for each coordinate dimension.

[19]:
```
backend('matplotlib')
fig = V.location_trend()
```



4.3.4 Variogram.distance_difference_plot

The final utility plot presented here is a scatter-plot that relates all pairwise-differences in value to the spatial distance of the respective point pairs. This can already be considered to be a variogram. For convenience, the plotting method will mark all upper lag class edges in the plot. This can already give you an idea, if the number of lag classes is chosen wisely, or if you need to adjust. To estimate valid, expressive variograms, this is maybe the most important preparation step. If your lag classes do not represent your data well, you will never find a useful variogram.

Plotly

[20]:
```
backend('plotly')
fig = V.distance_difference_plot()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

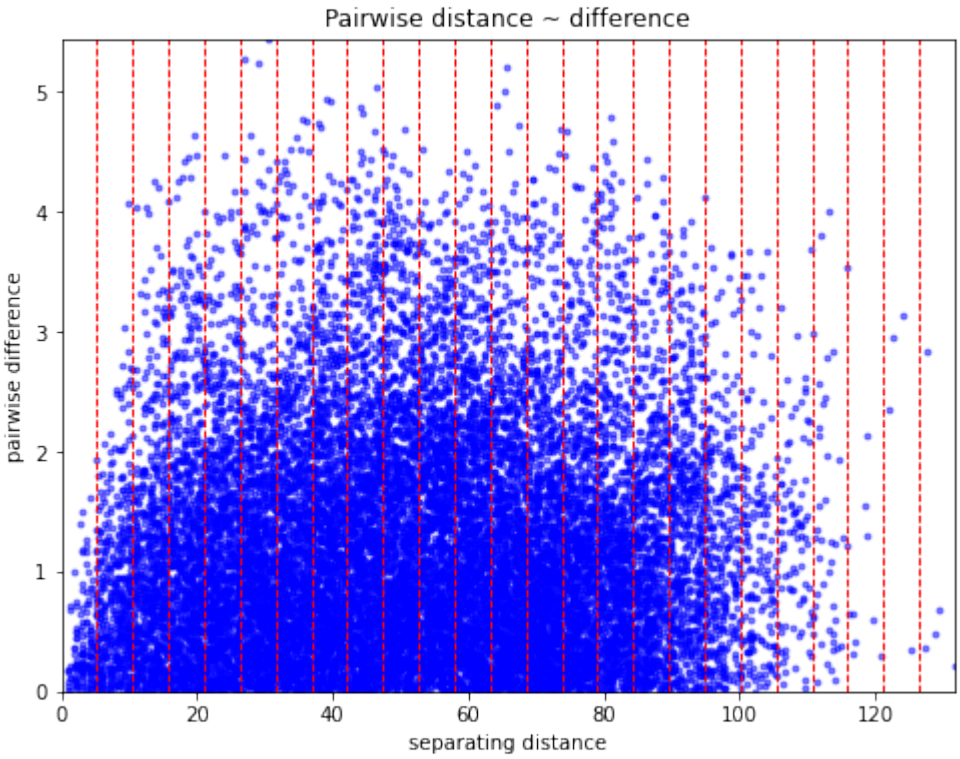
You might also consider to adapt the maximum lag distance using this plot, to exclude distances that are not well backed by data. Alternatively, the binning method can be changed. Or both

[21]:
```
Vcopy = V.clone()
Vcopy.bin_func = 'uniform'

fig = Vcopy.distance_difference_plot()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

Matplotlib

[22]:
```
backend('matplotlib')
fig = V.distance_difference_plot()
```



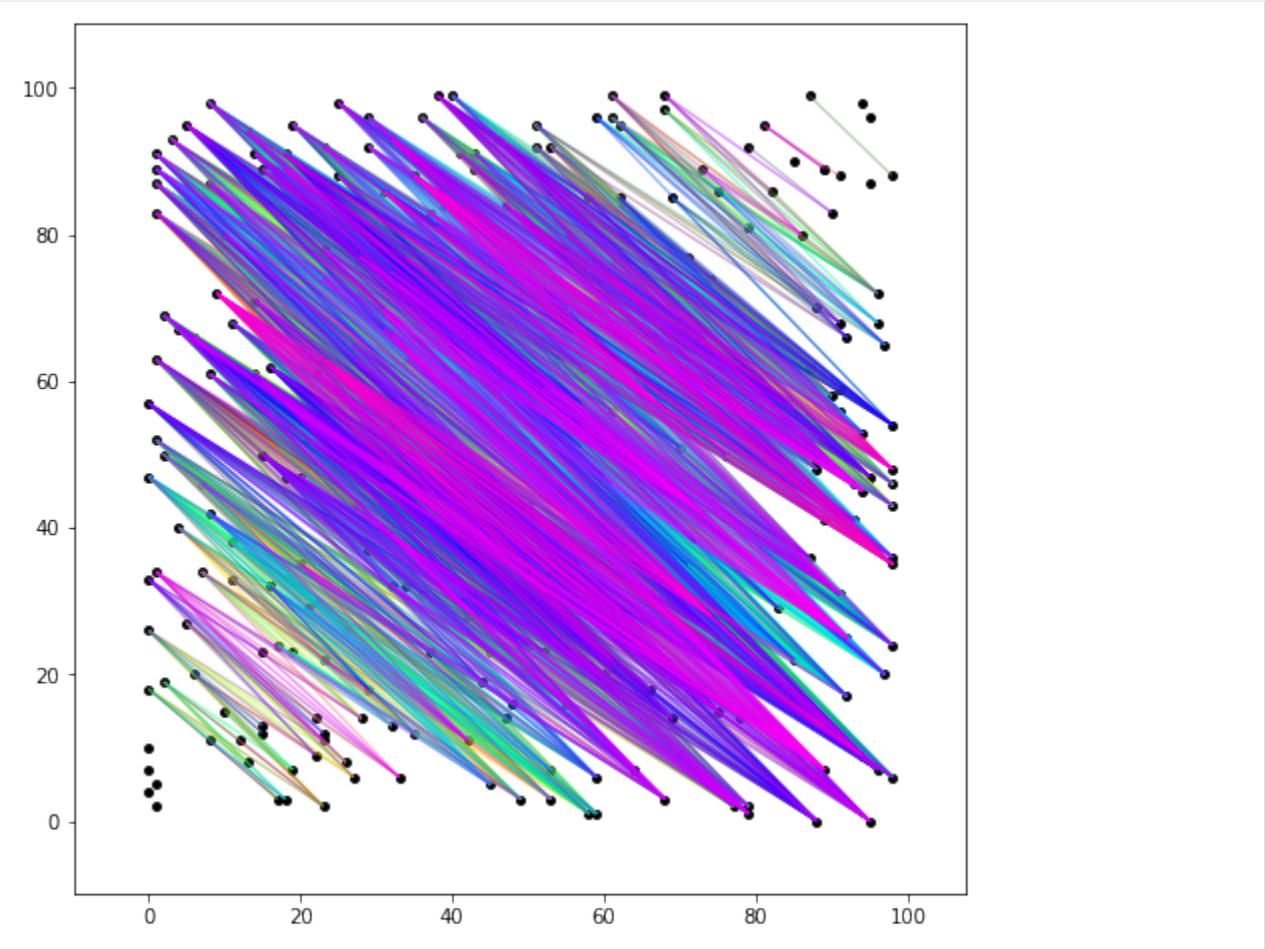
4.4 Directional Variogram

The `DirectionalVariogram` class is inheriting from `Variogram`. Therefore all plotting method shown above are available for directional variograms, as well. Additionally, there is one more plotting method, `DirectionalVariogram.pair_field`. This function will plot all coordinate locations and draw a line between all point pairs, that were not masked by the directional mask array and will, thus, be used for variogram estimation. By default, the method will draw all lines for all point pairs and you will see nothing on the plot. But there is also the possibility to draw these lines only for a subset of the coordinate locations

4.4.1 `pair_field`

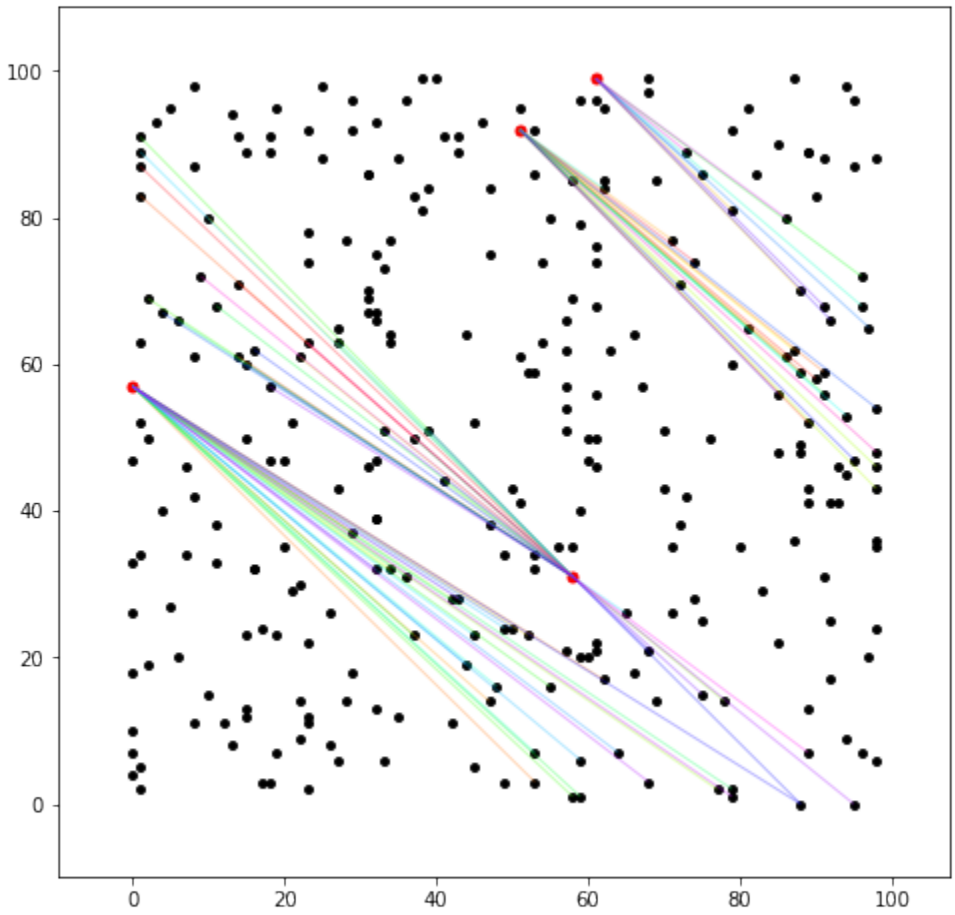
matplotlib

[23]:
```
backend('matplotlib')
fig = DV.pair_field()
```



Obviously, one can see the `azimuth` (40°) and narrow `tolerance` (15°) settings in the cone-like shapes of the connection lines, but the whole plot is not really instructive or helpful like this. Using the `points` keyword, you can show the lines only for a given set of coordinate locations. You have to pass a list of coordinate indices. With `add_points=True`, the seleceted points will be highlighted in red.

[24]:
```
fig = DV.pair_field(points=[0, 42, 104, 242], add_points=True)
```



plotly

Note: It is not recommended to plot the full `pair_field` with all points using plotly. Due to the implementation, that makes the plot really, really slow for rendering.

[25]:
```
backend('plotly')
```

[26]:
```
fig = DV.pair_field(points=[0,42, 104, 242], add_points=True)
fig.show()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

4.5 ST Variogram

The `SpaceTimeVariogram` does not inherit from the `Variogram`class and thus, its plotting methods are not available for space time variograms. However, the `SpaceTimeVariogram` has two properties, `SpaceTimeVariogram.XMarginal` and `SpaceTimeVariogram.TMarginal`, which are both instances of `Variogram` for the spatial and temporal marginal variogram. These instances in turn, have all plotting methods available, in addition to the plotting methods of `SpaceTimeVariogram` itself.

4.5.1 `plot(kind='scatter')`

The scatter plot can be used to inspect the experimental variogram data on a spatial and temporal axis, with the fitted spatio-temporal model fitted to the data.

plotly

[27]:
```
backend('plotly')
fig = STV.plot(kind='scatter')
fig
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

The method can also remove the model from the plot. This can be helpful in case the experimental data should be analyzed. Then, the model plane might be disturbing.

[28]:
```
STV.plot(kind='scatter', no_model=True)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

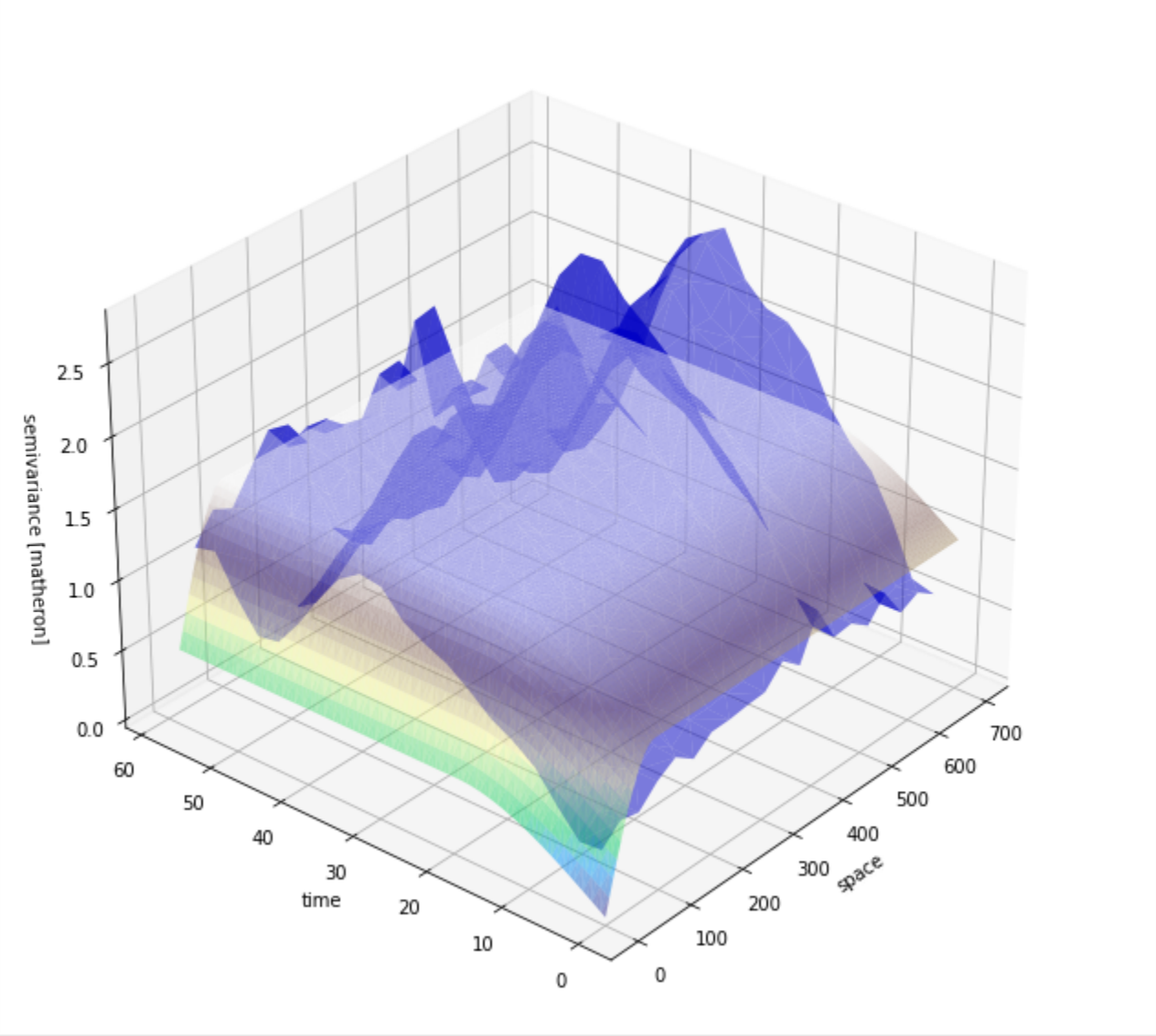
And finally, the experimental point data can be connected to a surface grid, to emphasize an apparent structure more easily in a 3D plot. This can be done by switching to `kind='surf'`.

[29]:
```
STV.plot(kind='surf')
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

matplotlib

[30]:
```
backend('matplotlib')
fig = STV.plot(kind='surf')
```



4.5.2 `contour`

3D plots are great for data exploration, especially if they are interactive. For publications, 3D plots are not that helpful. Additionally, it can be quite tricky sometimes to find a good angle to focus on the main message of a 3D plot. Hence, there are more plotting modes. They can either be used by setting `kind='contour'` or `kind='contourf'`. Alternatively, these two plotting types also have their own method. In both cases, the experimental semi-variance is plotted on a two dimensional plane. The spatial dimension can be found on the x-axis and the temporal dimension on the y-axis. The semi-variance itself is shown as a contour plot, that can either only plot the lines (`'contour'`) or filled areas for each contour (`'contourf'`).

plotly

[31]:
```
backend('plotly')
STV.contour()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

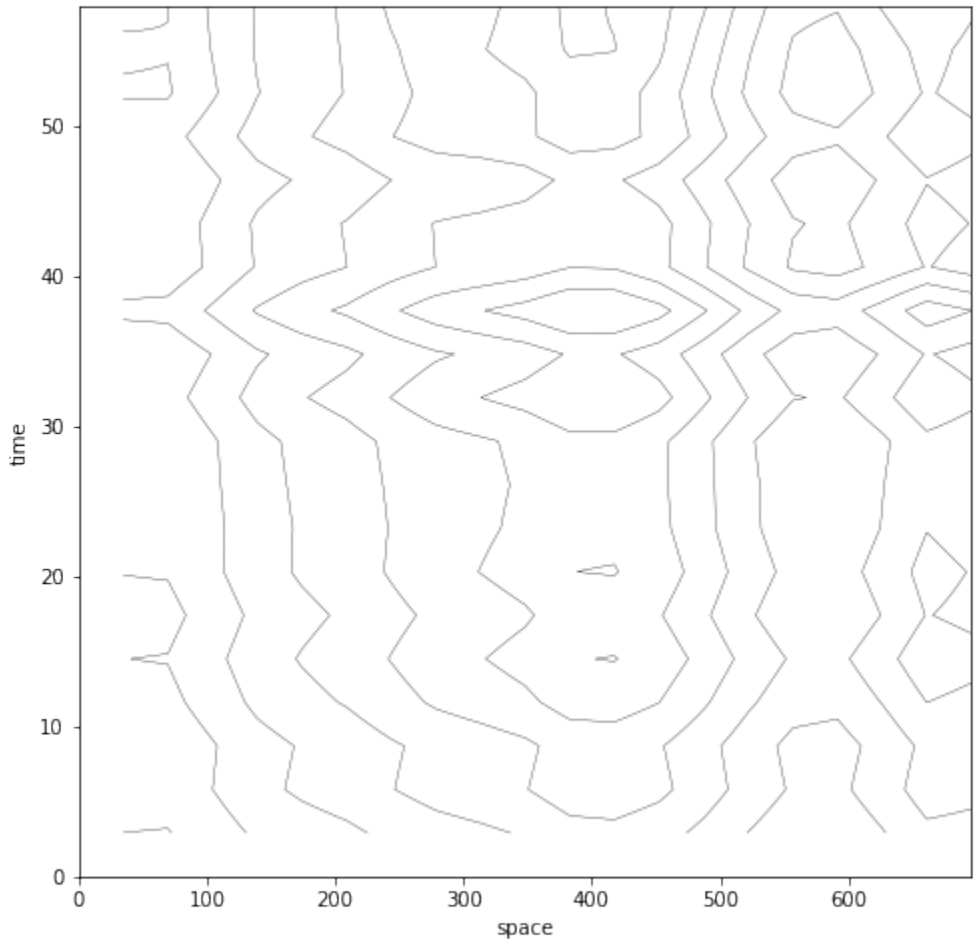
[32]:
```
STV.contourf()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

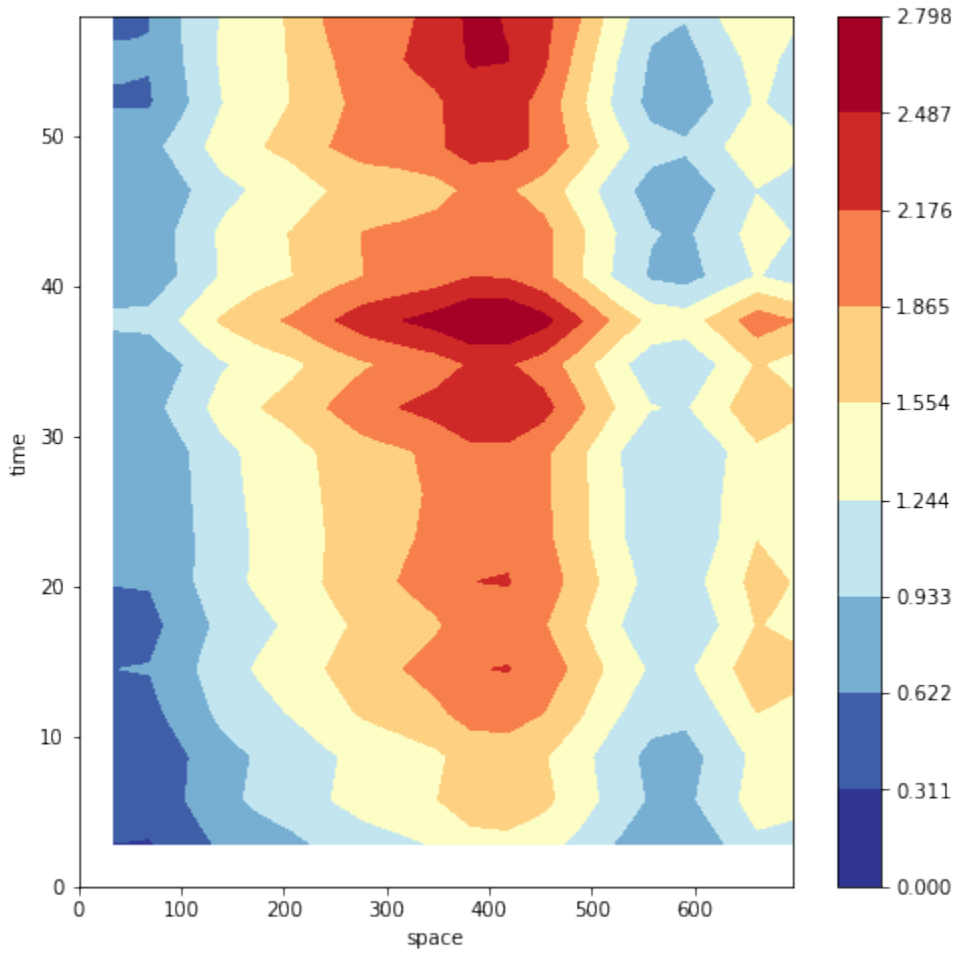
matplotlib

The matplotlib versions of the contour plots are not that sophisticated, but the returned figure can be adjusted to your needs.

[33]:
```
backend('matplotlib')
fig = STV.plot(kind='contour')
```



[34]:
```
fig = STV.plot(kind='contourf')
```



4.5.3 marginals

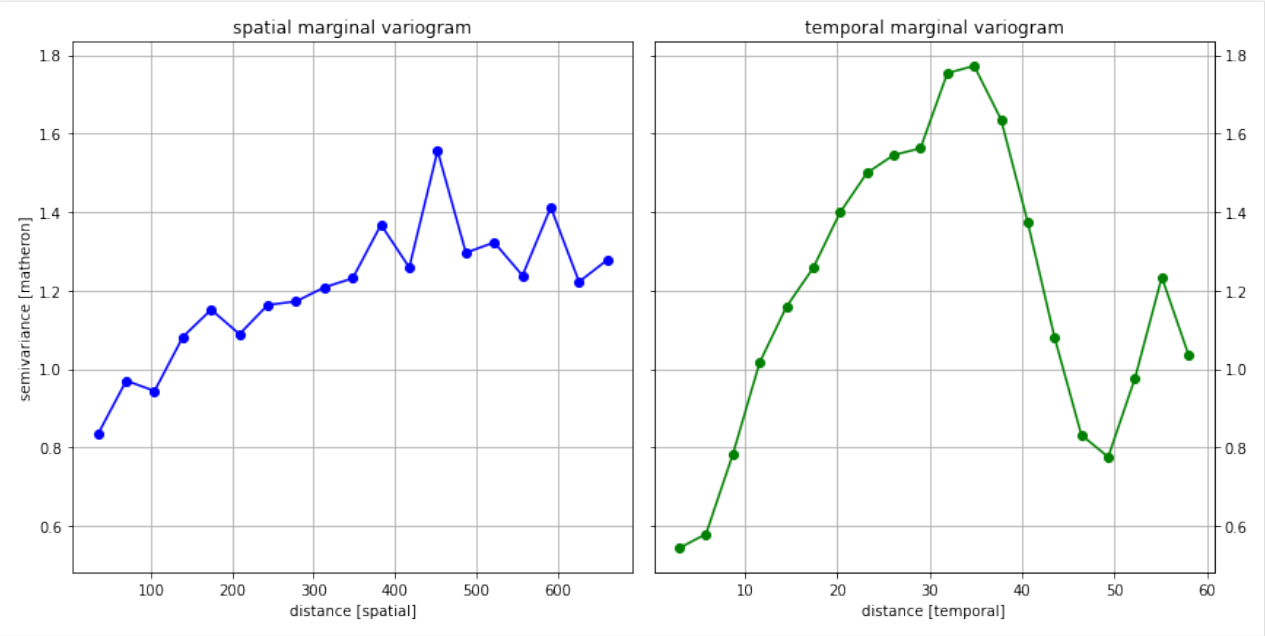
A very important step for the estimation of spatio-temporal variogram models, is the estimation of marginal models. While the marginal models are implemented as `Variogram` instances and can be changed and plotted like any other `Variogram` instance, it can come very handy to plot the marginal models side-by-side.

This can be done with the `SpaceTimeVariogram.marginals` method.

[35]:
```
backend('plotly')
STV.marginals()
```

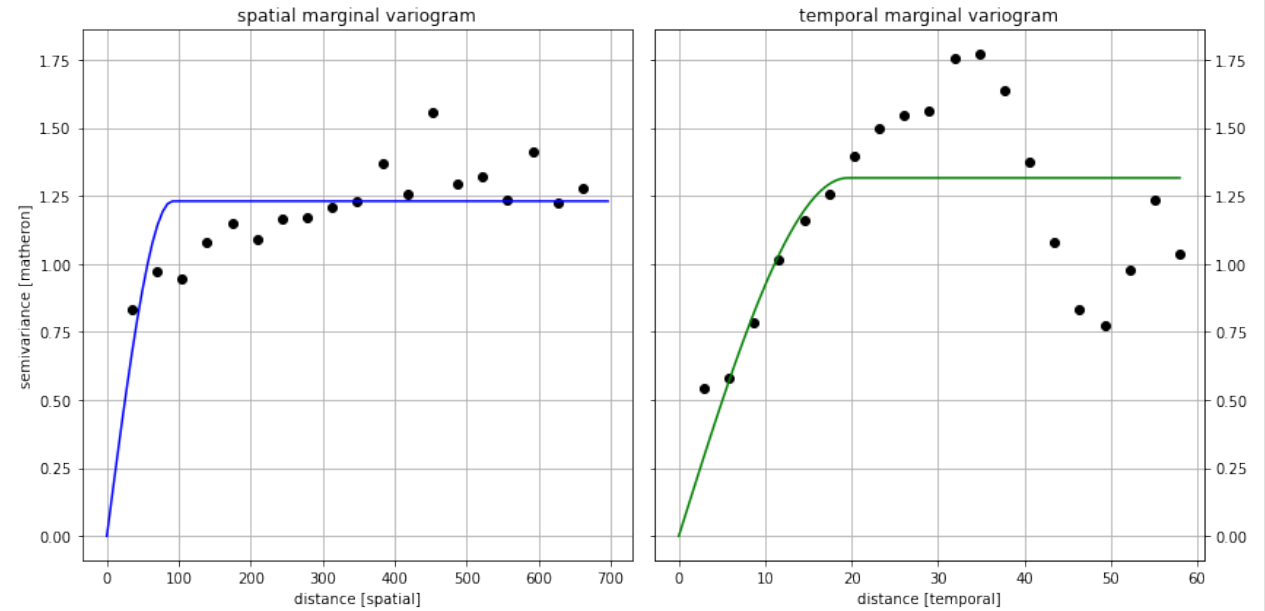
Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[36]:
```
backend('matplotlib')
fig = STV.marginals()
```



Additionally, the separated spatial and temporal models can be plotted into each sub-plot:

[37]:
```
fig = STV.marginals(include_model=True)
```



[38]:
```
backend('plotly')
STV.marginals(include_model=True)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

2.4.5 5 - Lag classes

This tutorial focuses the estimation of lag classes. It is one of the most important, maybe **the** most important step for estimating variograms. Usually, lag class generation, or binning, is not really focused in geostatistical literature. The main reason is, that usually, the same method is used. A user-set amount of equidistant lag classes is formed with `0` as lower bound and `maxlag` as upper bound. Maxlag is often set to the median or 60% percentile of all pairwise separating distances.

In SciKit-GStat this is also the default behavior, but only one of dozen of different implemented methods. Thus, we want to shed some light onto the other methods here. SciKit-GStat implements methods of two different kinds. The first kind are the methods, that take a fixed `N`, the number of lag classes, accessible through the `Variogram.n_lags` property. These methods are `['even', 'uniform', 'kmeans', 'ward']`. The other kind is often used in histogram estimation and will apply a (simple) rule to figure out a suitable `N` themself. Using one of these methods will overwrite the `Variogram.n_lags` property. THese methods are: `['sturges', 'scott', 'fd', 'sqrt', 'doane']`.

[1]:
```
import skgstat as skg
import numpy as np
import pandas as pd
from imageio import imread
import plotly.graph_objects as go
from plotly.subplots import make_subplots

skg.plotting.backend('plotly')
```

5.1 Sample data

Loads a data sample and draws `n_samples` from the field. For sampling the field, random samples from a gamma distribution with a fairly high scale are drawn, to ensure there are some outliers in the samle. The values are then re-scaled to the shape of the random field and the values are extracted from it.

[2]:
```
n_samples = 150
CHANNEL = 0

pan = imread('./data/pancake6_500x500.png')
r = pan[:,:,CHANNEL]

# sample coordinates from a gamma distribution
np.random.seed(1312)
c = np.random.gamma(10, 40, size=(n_samples,2))

# rescale to image size
c[::,0] = c[::,0] / np.max(c[::,0] * 1.1) + 0.02
c[::,1] = c[::,1] / np.max(c[::,1] * 1.1) + 0.02
coords = (c * pan.shape[0]).astype(int)

# sample red values
vals = [r[c[0], c[1]] for c in coords]
```

[3]:
```
fig = make_subplots(1,2,shared_xaxes=True, shared_yaxes=True)
fig.add_trace(
    go.Scatter(x=coords[:,0], y=coords[:,1], mode='markers', marker=dict(color=vals,
↪cmin=0, cmax=255), name='samples'),
```

(continues on next page)

(continued from previous page)

```
    row=1, col=1
)
fig.add_trace(go.Heatmap(z=r, name='field'), row=1, col=2)
fig.update_layout(width=900, height=450, template='plotly_white')
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.2 Lag class binning - fixed `N`

Apply different lag class binning methods and visualize their histograms. In this section, the distance matrix between all point pair combinations (`NxN`) is binned using each method. The plots visualize the histrogram of the distance matrix of the variogram, **not** the variogram lag classes themselves.

[4]:
```
N = 15

# use a nugget
V = skg.Variogram(coords, vals, n_lags=N, use_nugget=True)
```

5.2.1 default `'even'` lag classes

The default binning method will find `N` equidistant bins. This is the default behavior and used in almost all geostatistical publications. It should not be used without a `maxlag` (like done in the plot below).

[5]:
```
# apply binning
bins, _ = skg.binning.even_width_lags(V.distance, N, None)

# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'even'}~~binning$")
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.2.2 `'uniform'` lag classes

The histogram of the `uniform` method will adjust the lag class widths to have the same sample size for each lag class. This can be used, when there must not be any empty lag classes on small data samples, or comparable sample sizes are desireable for the semi-variance estimator.

[6]:
```
# apply binning
bins, _ = skg.binning.uniform_count_lags(V.distance, N, None)
```

(continues on next page)

(continued from previous page)

```
# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'uniform'}~~binning$")
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.2.3 'kmeans' lag classes

The distance matrix is clustered by a K-Means algorithm. The centroids, are taken as a good guess for lag class centers. Each lag class is then formed by taking half the distance to each sorted neighboring centroid as a bound. This will most likely result in non-equidistant lag classes.

One important note about K-Means clustering is, that it is not a deterministic method, as the starting points for clustering are taken randomly. Thus, the decision was made to seed the random start values. Therefore, the K-Means implementation in SciKit-GStat is deterministic and will always return the same lag classes for the same distance matrix.

[7]:
```
# apply binning
bins, _ = skg.binning.kmeans(V.distance, N, None)

# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'K-Means'}~~binning$")
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.2.4 'ward' lag classes

The other clustering algorithm is a hierarchical clustering algorithm. This algorithm groups values together based on their similarity, which is expressed by Ward's criterion. Agglomerative algorithms work iteratively and deterministic, as at first iteration each value forms a cluster on its own. Each cluster is then merged with the most similar other cluster, one at a time, until all clusters are merged, or the clustering is interrupted. Here, the clustering is interrupted as soon as the specified number of lag classes is reached. The lag classes are then formed similar to the K-Means method, either by taking the cluster mean or median as center.

Ward's criterion defines the one other cluster as the closest, that results in the smallest intra-cluster variance for the merged clusters. The main downside is the processing speed. You will see a significant difference for `'ward'` and should not use it on medium and large datasets.

[8]:
```
# apply binning
bins, _ = skg.binning.ward(V.distance, N, None)

# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'ward'}~~binning$")
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.3 Lag class binning - adjustable `N`

5.3.1 `'sturges'` lag classes

Sturge's rule is well known and pretty straightforward. It's the default method for histograms in R. The number of equidistant lag classes is defined like:

$$n = log_2(x + 1)$$

Sturge's rule works good for small, normal distributed datasets.

[9]:
```
# apply binning
bins, n = skg.binning.auto_derived_lags(V.distance, 'sturges', None)

# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'sturges'}~~binning~~%d~classes
↪$" % n)
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.3.2 `'scott'` lag classes

Scott's rule is another quite popular approach to estimate histograms. The rule is defined like:

$$h = \sigma \frac{24 * \sqrt{\pi}^{\frac{1}{3}}}{x}$$

Other than Sturge's rule, it will estimate the lag class width from the sample size standard deviation. Thus, it is also quite sensitive to outliers.

[10]:
```
# apply binning
bins, n = skg.binning.auto_derived_lags(V.distance, 'scott', None)

# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'scott'}~~binning~~%d~classes$
↪" % n)
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.3.3 'sqrt' lag classes

The only advantage of this method is its speed. The number of lag classes is simply defined like:

$$n = \sqrt{x}$$

Thus, it's usually not really a good choice, unless you have a lot of samples.

[11]:
```
# apply binning
bins, n = skg.binning.auto_derived_lags(V.distance, 'sqrt', None)

# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'sqrt'}~~binning~~%d~classes$"
↪% n)
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.3.4 'fd' lag classes

The Freedman-Diaconis estimator can be used to derive the number of lag classes again from an optimal lag class width like:

$$h = 2\frac{IQR}{x^{1/3}}$$

As it is based on the interquartile range (IQR), it is very robust to outlier. That makes it a suitable method to estimate lag classes on non-normal distance matrices. On the other side it usually over-estimates the n for small datasets. Thus it should only be used on medium to small datasets.

[12]:
```
# apply binning
bins, n = skg.binning.auto_derived_lags(V.distance, 'fd', None)

# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'fd'}~~binning~~%d~classes$" %␣
↪n)
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.3.5 'doane' lag classes

Doane's rule is an extension to Sturge's rule that takes the skewness of the distance matrix into account. It was found to be a very reasonable choice on most datasets where the other estimators didn't yield good results.

It is defined like:

$$n = 1 + \log_2(s) + \log_2\left(1 + \frac{|g|}{k}\right)$$

$$g = E\left[\left(\frac{x - \mu_g}{\sigma}\right)^3\right]$$

$$k = \sqrt{\frac{6(s-2)}{(s+1)(s+3)}}$$

[13]:
```
# apply binning
bins, n = skg.binning.auto_derived_lags(V.distance, 'doane', None)

# get the histogram
count, _ = np.histogram(V.distance, bins=bins)

go.Figure(
    go.Bar(x=bins, y=count),
    layout=dict(template='plotly_white', title=r"$\texttt{'doane'}~~binning~~%d~classes$
↪" % n)
)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4 Variograms

The following section will give an overview on the influence of the chosen binning method on the resulting variogram. All parameters will be the same for all variograms, so any change is due to the lag class binning. The variogram will use a maximum lag of `200` to get rid of the very thin last bins at large distances.

The `maxlag` is very close to the effective range of the variogram, thus you can only see differences in sill. But the variogram fitting is not at the focus of this tutorial. You can also change the parameter and fit a more suitable spatial model

[14]:
```
# use a exponential model
V.set_model('spherical')

# set the maxlag
V.maxlag = 200
```

5.4.1 `'even'` lag classes

[15]:
```
# set the new binning method
V.bin_func = 'even'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" - range: {np.round(V.cof[0], 1)} sill: {np.round(V.cof[1],␣
↪1)}')
fig.update_layout(template='plotly_white')
```

```
"even" - range: 200.0 sill: 627.1
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4.2 `'uniform'` lag classes

[16]:
```
# set the new binning method
V.bin_func = 'uniform'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" - range: {np.round(V.cof[0], 1)} sill: {np.round(V.cof[1],␣
↪1)}')
fig.update_layout(template='plotly_white')
```

```
"uniform" - range: 200.0 sill: 551.2
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4.3 'kmeans' lag classes

[17]:
```
# set the new binning method
V.bin_func = 'kmeans'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" - range: {np.round(V.cof[0], 1)} sill: {np.round(V.cof[1],␣
↪1)}')
fig.update_layout(template='plotly_white')
```

```
"kmeans" - range: 184.0 sill: 579.0
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4.4 'ward' lag classes

[18]:
```
# set the new binning method
V.bin_func = 'ward'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" - range: {np.round(V.cof[0], 1)} sill: {np.round(V.cof[1],␣
↪1)}')
fig.update_layout(template='plotly_white')
```

```
"ward" - range: 181.8 sill: 569.0
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4.5 'sturges' lag classes

[19]:
```
# set the new binning method
V.bin_func = 'sturges'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" adjusted {V.n_lags} lag classes - range: {np.round(V.cof[0],
↪ 1)} sill: {np.round(V.cof[1], 1)}')
fig.update_layout(template='plotly_white')
```

```
"sturges" adjusted 15 lag classes - range: 200.0 sill: 624.1
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4.6 'scott' lag classes

[20]:
```
# set the new binning method
V.bin_func = 'scott'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" adjusted {V.n_lags} lag classes - range: {np.round(V.cof[0],
↪ 1)} sill: {np.round(V.cof[1], 1)}')
fig.update_layout(template='plotly_white')
```

```
"scott" adjusted 27 lag classes - range: 200.0 sill: 621.1
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4.7 'fd' lag classes

[21]:
```
# set the new binning method
V.bin_func = 'fd'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" adjusted {V.n_lags} lag classes - range: {np.round(V.cof[0],
↪ 1)} sill: {np.round(V.cof[1], 1)}')
fig.update_layout(template='plotly_white')
```

```
"fd" adjusted 31 lag classes - range: 200.0 sill: 616.6
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4.8 'sqrt' lag classes

[22]:
```
# set the new binning method
V.bin_func = 'sqrt'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" adjusted {V.n_lags} lag classes - range: {np.round(V.cof[0],
↪ 1)} sill: {np.round(V.cof[1], 1)}')
fig.update_layout(template='plotly_white')
```

```
"sqrt" adjusted 100 lag classes - range: 200.0 sill: 614.9
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.4.9 'doane' lag classes

[23]:
```
# set the new binning method
V.bin_func = 'doane'

# plot
fig = V.plot(show=False)
print(f'"{V._bin_func_name}" adjusted {V.n_lags} lag classes - range: {np.round(V.cof[0],
↪ 1)} sill: {np.round(V.cof[1], 1)}')
fig.update_layout(template='plotly_white')
```

```
"doane" adjusted 18 lag classes - range: 200.0 sill: 618.7
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

2.4.6 6 - GSTools

With version `0.5` `scikit-gstat` offers an interface to the awesome gstools library. This way, you can use a `Variogram` estimated with `scikit-gstat` in gstools to perform random field generation, kriging and much, much more.

For a `Variogram` instance, there are three possibilities to export into gstools:

1. `Variogram.get_empirical(bin_center=True)` returns a pair of distance lag bins and experimental semi-variance values, like gstools.variogram.vario_estimate.
2. `Variogram.to_gstools` returns a parameterized `CovModel` derived from the Variogram.
3. `Variogram.to_gs_krige` returns a GSTools Krige instance based on the variogram

6.1 get_empirical

6.1.1 Reproducing the gstools example

You can reproduce the Getting Started example for variogram estimation from GSTools docs with `scikit-gstat`, and replace the calculation of the empirical variogram with `skg.Variogram`.

Note: This does only make sense if you want to use a distance metric, binning procedure or semi-variance estimator, that is not included in `gstools` or are bound to `scikit-gstat` for any other reason. `Variogram` will *always* perform a full model fitting cycle on instantiation, which could lead to some substantial overhead here. This behavior might change in a future version of `scikit-gstat`.

[1]:
```
# import
import skgstat as skg
import gstools as gs
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
# use the example from gstools
# generate a synthetic field with an exponential model
```

(continues on next page)

(continued from previous page)

```
x = np.random.RandomState(19970221).rand(1000) * 100.
y = np.random.RandomState(20011012).rand(1000) * 100.
model = gs.Exponential(dim=2, var=2, len_scale=8)
srf = gs.SRF(model, mean=0, seed=19970221)
field = srf((x, y))

# combine x and y for use in skgstat
coords = np.column_stack((x, y))
```

In the example, gstools.variogram.vario_estimate is used to estimate the empirical variogram:

```
# estimate the variogram of the field
bin_center, gamma = gs.vario_estimate((x, y), field)
```

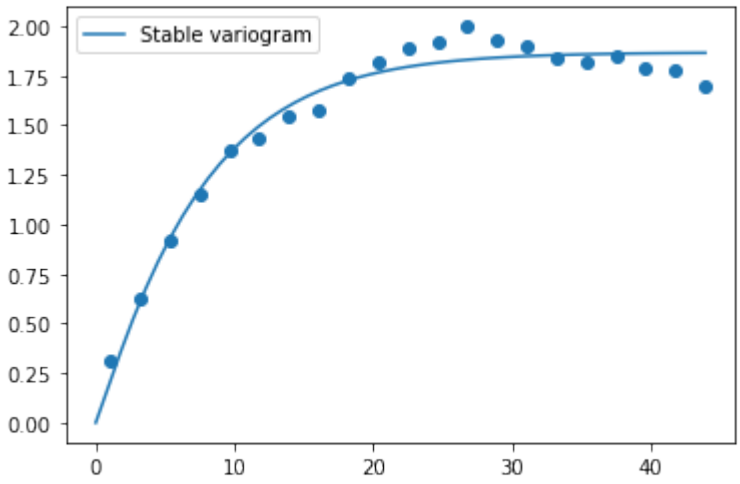
Here, we can use `skg.Variogram`. From the shown arguments, `estimator` and `bin_func` are using the default values:

[3]:
```
V = skg.Variogram(coords, field, n_lags=21, estimator='matheron', maxlag=45, bin_func=
↪'even')
bin_center, gamma = V.get_empirical(bin_center=True)
```

And finally, the exact same code from the GSTools docs can be called:

[4]:
```
%matplotlib inline
# fit the variogram with a stable model. (no nugget fitted)
fit_model = gs.Stable(dim=2)
fit_model.fit_variogram(bin_center, gamma, nugget=False)
# output
ax = fit_model.plot(x_max=max(bin_center))
ax.scatter(bin_center, gamma)
print(fit_model)
```

```
Stable(dim=2, var=1.87, len_scale=7.55, nugget=0.0, alpha=1.08)
```



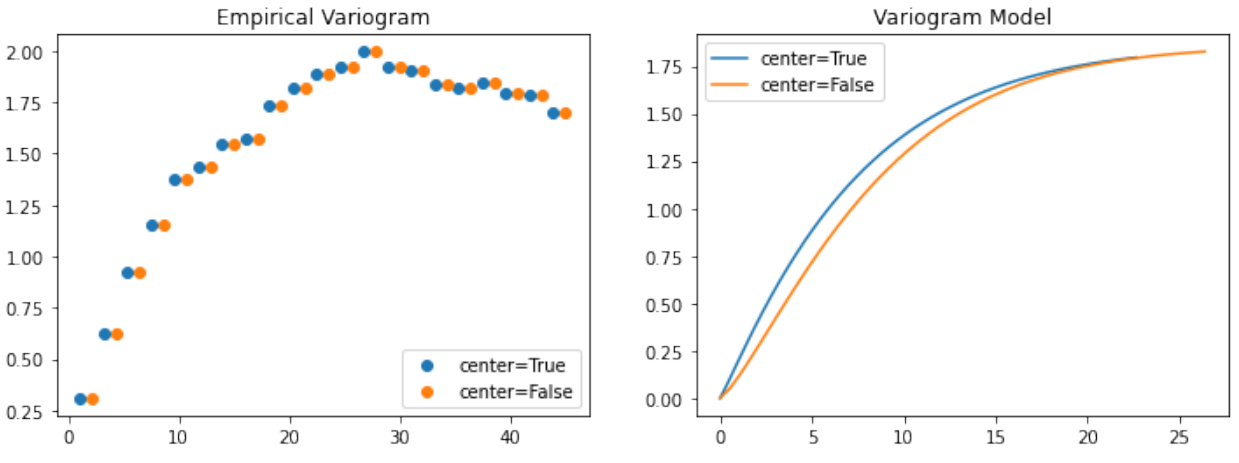
6.1.2 bin_center=False

It is important to understand, that `gstools` and `skgstat` are handling lag bins different. While `skgstat` uses the upper limit, `gstools` assumes the bin center. This can have implications, if a model is fitted. Consider the example below, in which only the `bin_center` setting is different.

[5]:
```
bin_edges, _ = V.get_empirical(bin_center=False)

# fit the variogram with a stable model. (no nugget fitted)
edge_model = gs.Stable(dim=2)
_  = edge_model.fit_variogram(bin_edges, gamma, nugget=False)
```

[6]:
```
fig, axes = plt.subplots(1,2, figsize=(12,4))

# plot first
fit_model.plot(ax=axes[1], label='center=True')
# plot second
edge_model.plot(ax=axes[1], label='center=False')

# bins
axes[0].scatter(bin_center, gamma, label='center=True')
axes[0].scatter(bin_edges, gamma, label='center=False')

axes[0].set_title('Empirical Variogram')
axes[1].set_title('Variogram Model')
axes[0].legend(loc='lower right')
print(fit_model)
print(edge_model)
```

```
Stable(dim=2, var=1.87, len_scale=7.55, nugget=0.0, alpha=1.08)
Stable(dim=2, var=1.86, len_scale=8.8, nugget=0.0, alpha=1.27)
```



Notice the considerable gap between the two model functions. This can already lead to seroius differences, i.e. in Kriging.

6.1.3 Using other arguments

Now, with the example from the GSTools docs working, we can start chaning the arguments to create quite different empirical variograms.

Note: This should just illustrate the available possibilities, the result is by no means producing a better estimate of the initially created Gaussian random field.

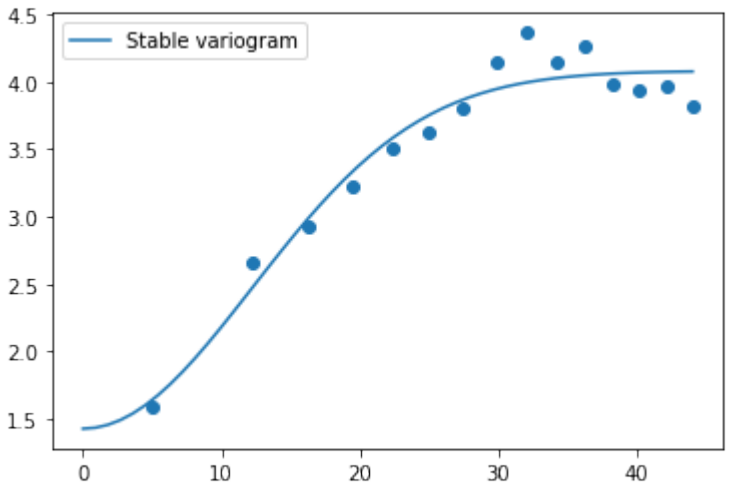
In this example different things will be changed:

- use only 15 lag classes, but distribute the point pairs equally. Note the differing widths of the classes. (`bin_func='uniform'`)
- The Dowd estimator is used. (`estimator='dowd'`)
- The Taxicab metric (aka. Manhattan metric or cityblock metric) is used over Euklidean for no obvious reason. (`dist_func='cityblock'`)

[7]:
```
V = skg.Variogram(coords, field, n_lags=15, estimator='dowd', maxlag=45, bin_func=
↪'uniform', dist_func='cityblock')
bin_center, gamma = V.get_empirical(bin_center=True)
```

[8]:
```
# fit the variogram with a stable model. (no nugget fitted)
fit_model = gs.Stable(dim=2)
fit_model.fit_variogram(bin_center, gamma, nugget=True)
# output
ax = fit_model.plot(x_max=max(bin_center))
ax.scatter(bin_center, gamma)
print(fit_model)
```

```
Stable(dim=2, var=2.66, len_scale=17.3, nugget=1.42, alpha=2.0)
```



If you fit the `gs.Stable` with a nugget, it fits quite well. But keep in mind that this does not necessarily describe the original field very well and was just fitted for demonstration.

6.2 to_gstools

The second possible interface to `gstools` is the `Variogram.to_gstools` function. This will return one of the classes listed in the gstools documentation. The variogram parameters are extracted and passed to gstools. You should be able to use it, just like any other `CovModel`.

However, there are a few things to consider:

- `skgstat` can only export isotropic models.
- The `'harmonize'` cannot be exported

6.2.1 exporting Variogram

In this example, the same Variogram from above is estimated, but we use the `'exponential'` model. An exponential covariance function was used in the first place to create the field that was sampled.

[9]:
```
skg.plotting.backend('plotly')
V = skg.Variogram(coords, field, n_lags=21, estimator='matheron', model='exponential',␣
↪maxlag=45, bin_func='even')
V.plot(show=False)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

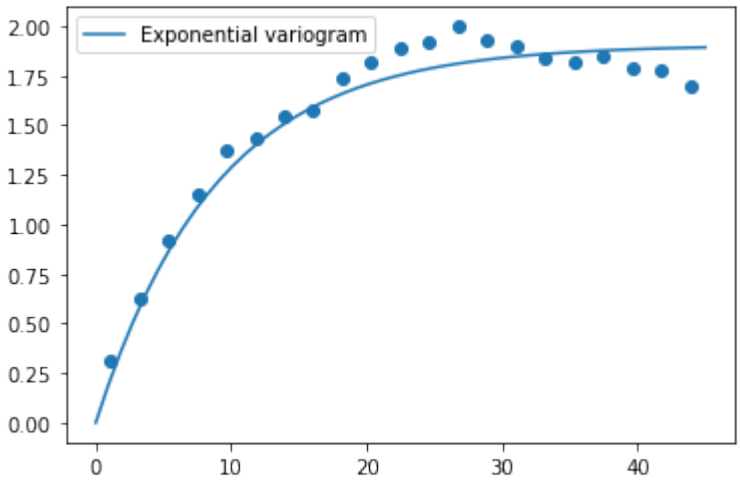
Now export the model to `gstools`:

[10]:
```
exp_model = V.to_gstools()
print(exp_model)
```

```
Exponential(dim=2, var=1.9, len_scale=26.6, nugget=0.0, rescale=3.0)
```

[11]:
```
# get the empirical for the plot as well
bins, gamma = V.get_empirical(bin_center=True)

ax = exp_model.plot(x_max=45)
ax.scatter(bins, gamma)
```

[11]:
```
<matplotlib.collections.PathCollection at 0x7fdc37337dc0>
```



Note: It is important to understand, that `skgstat` and `gstools` handle coordinates slightly different. If you export the `Variogram` to a `CovModel` and you want to use the `Variogram.coordinates`, you **must** transpose them.

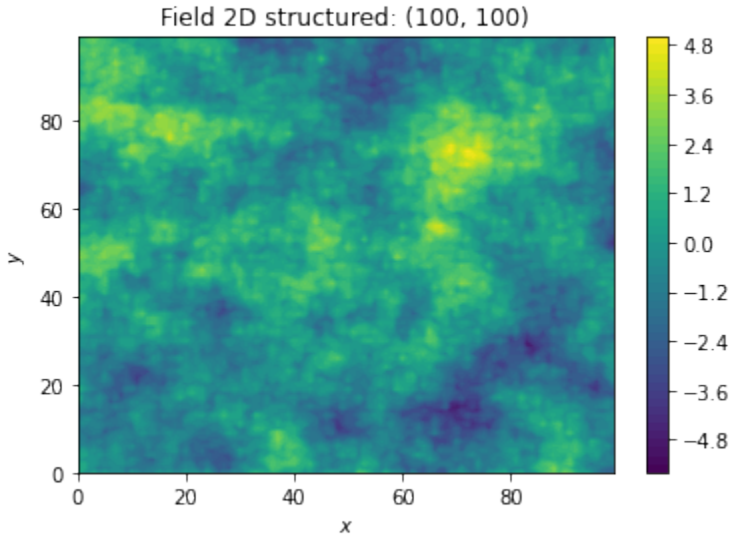
```
# variogram is a skgstat.Variogram instance
model = variogram.to_gstools()
cond_pos = variogram.coordinates.T

# use i.e. in Kriging
krige = gs.krige.Ordinary(model, cond_pos, variogram.values)
```

6.2.2 Spatial Random Field Generation

With a `CovModel`, we can use any of the great tools implemented in `gstools`. First, let's create another random field with the exponential model that we exported in the last section:

[12]:
```
x = y = range(100)
new_field = gs.SRF(exp_model, seed=13062018)
new_field.structured([x, y])
new_field.plot()
```

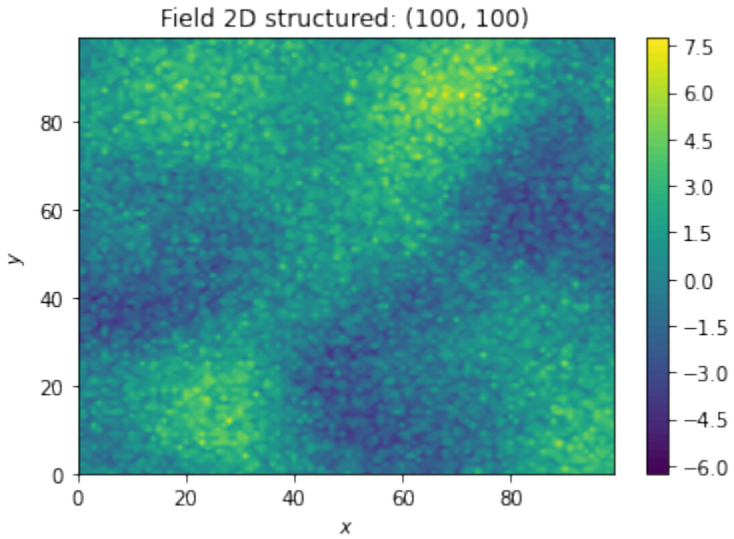
[12]:
```
<AxesSubplot:title={'center':'Field 2D structured: (100, 100)'}, xlabel='$x$', ylabel='$y
↪$'>
```



Keep in mind, that we did not call a Kriging procedure, but created **another** field. Of course, we can do the same thing with the more customized model, created in 6.1.3:

[13]:
```
malformed = gs.SRF(fit_model, seed=24092013)
malformed.structured([x,y])
malformed.plot()
```

[13]:
```
<AxesSubplot:title={'center':'Field 2D structured: (100, 100)'}, xlabel='$x$', ylabel='$y
↪$'>
```



Notice how the spatial properties as well as the value range has changed. That's why it is important to estimate `Variogram` or `CovModel` carefully and not let the GIS do that for you somewhere hidden in the dark.

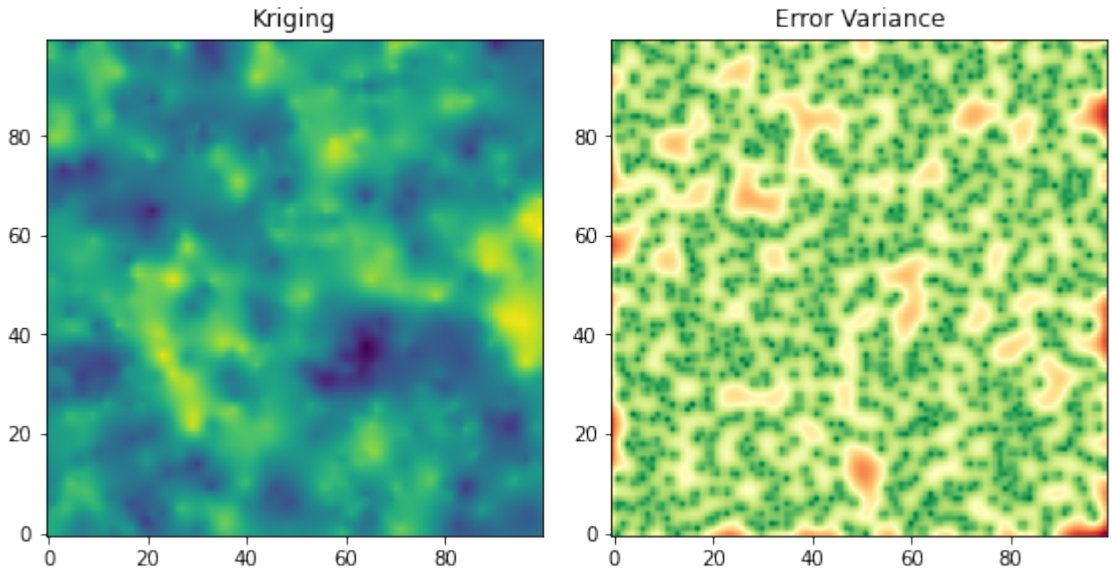
6.3 to_gs_krige

Finally, after carefully esitmating and fitting a variogram using SciKit-GStat, you can also export it directly into a GSTools Krige instance. We use the variogram as in the other sections:

[14]:
```
# export
krige = V.to_gs_krige(unbiased=True)  # will result in ordinary kriging
print(krige)

# create a regular grid
x = y = range(100)

# interpolate
result, sigma = krige.structured((x, y))
```

```
Krige(model=Exponential, cond_no=1000)
```

[15]:
```
fig, axes = plt.subplots(1, 2, figsize=(8, 4))

# plot
axes[0].imshow(result, origin='lower')
axes[1].imshow(sigma, origin='lower', cmap='RdYlGn_r')

# label
axes[0].set_title('Kriging')
axes[1].set_title('Error Variance')

plt.tight_layout()
```



2.5 Technical Notes

This chapter collects a number of technical advises for using scikit-gstat. These examples either give details on the implementation or guide a correct package usage. This are technical notes, no tutorials. The application of the shown examples might not make sense in every situation

2.5.1 Fitting a theoretical model

General

The fit function of Variogram relies as of this writing on the `scipy.optimize.curve_fit()` function. That function can be used by ust passing a function and a set of x and y values and hoping for the best. However, this will not always yield the best parameters. Especially not for fitting a theoretical variogram function. There are a few assumptions and simplifications, that we can state in order to utilize the function in a more meaningful way.

Default fit

The example below shows the performance of the fully unconstrained fit, performed by the Levenberg-Marquardt algorithm. In scikit-gstat, this can be used by setting the *fit_method* parameter to *lm*. However, this is not recommended.

```
In [1]: from scipy.optimize import curve_fit

In [2]: import matplotlib.pyplot as plt

In [3]: plt.style.use('ggplot')

In [4]: import numpy as np

In [5]: from skgstat.models import spherical
```

The fit of a spherical model will be illustrated with some made-up data representing an experimental variogram:

```
In [6]: y = [1,7,9,6,14,10,13,9,11,12,14,12,15,13]

In [7]: x = list(range(len(y)))

In [8]: xi = np.linspace(0, len(y), 100)
```

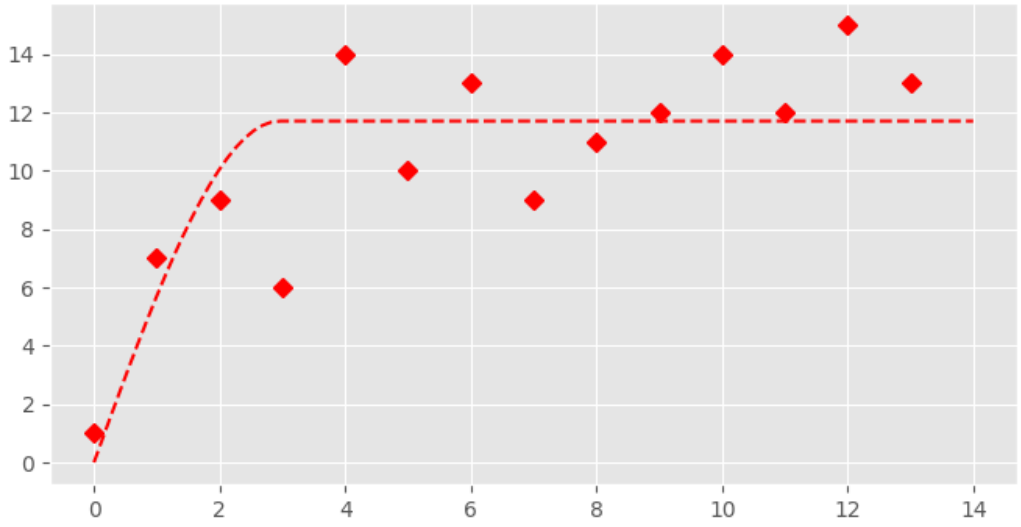
As the *spherical* function is compiled using numba, we wrap the function in order to let `curve_fit` correctly infer the parameters. Then, fitting is a straightforward task.

```
In [9]: def f(h, a, b):
   ...:     return spherical(h, a, b)
   ...:

In [10]: cof_u, cov = curve_fit(f, x, y)

In [11]: yi = list(map(lambda x: spherical(x, *cof_u), xi))

In [12]: plt.plot(x, y, 'rD')
Out[12]: [<matplotlib.lines.Line2D at 0x7f9db9ef48d0>]
```

(continues on next page)

(continued from previous page)

```
In [13]: plt.plot(xi, yi, '--r')
Out[13]: [<matplotlib.lines.Line2D at 0x7f9db9ef4470>]
```



In fact this looks quite good. But Levenberg-Marquardt is an unconstrained fitting algorithm and it could likely fail on finding a parameter set. The *fit* method can therefore also run a box constrained fitting algorithm. It is the Trust Region Reflective algorithm, that will find parameters within a given range (box). It is set by the *fit_method='tfr'* parameter and also the default setting.

Constrained fit

The constrained fitting case was chosen to be the default method in skgstat as the region can easily be specified. Furthermore it is possible to make a good guess on initial values. As we fit actual variogram parameters, namely the effective range, sill, nugget and in case of a stable or Matérn model an additional shape parameter, we know that these parameters cannot be zero. The semi-variance is defined to be always positive. Thus the lower bound of the region will be zero in any case. The upper limit can easily be inferred from the experimental variogram. There are some simple rules, that all theoretical functions follow:

- the sill, nugget and their sum cannot be larger than the maximum empirical semi-variance
- the range cannot be larger than maxlag, or if maxlag is None the maximum value in the distances

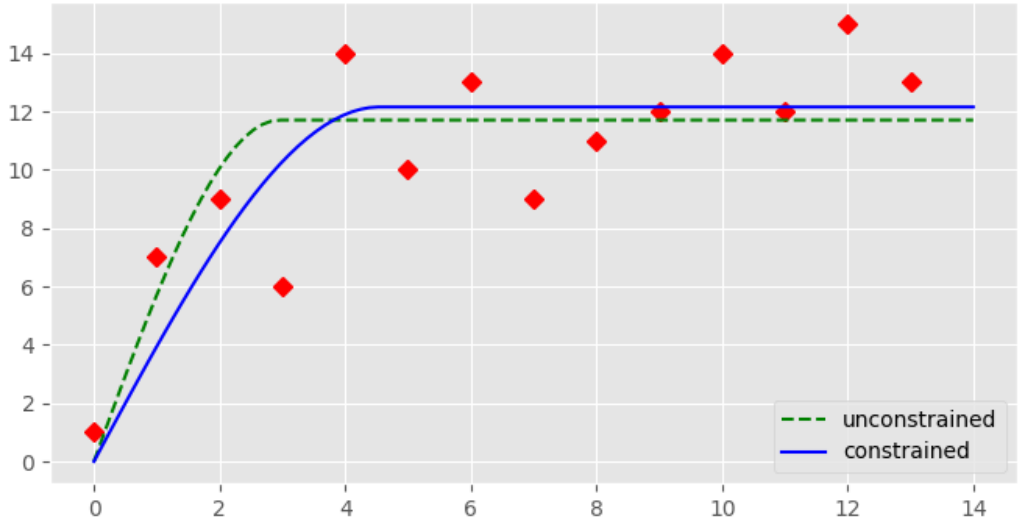
The *Variogram* class will set the bounds to exactly these values as default behaviour. As an initial guess, it will use the mean value of semi-variances for the sill, the mean separating distance as range and 0 for the nugget. In the presented empirical variogram, difference between Levenberg-Marquardt and Trust Region Reflective is illustrated in the example below.

```
# default plot
In [14]: plt.plot(x, y, 'rD')
Out[14]: [<matplotlib.lines.Line2D at 0x7f9db9e74358>]

In [15]: plt.plot(xi, yi, '--g', label='unconstrained')
Out[15]: [<matplotlib.lines.Line2D at 0x7f9db9e74978>]
```

(continues on next page)

(continued from previous page)

```
In [16]: cof, cov = curve_fit(f, x, y, p0=[3., 14.], bounds=(0, (np.max(x), np.max(y))))

In [17]: yi = list(map(lambda x: spherical(x, *cof), xi))

In [18]: plt.plot(xi, yi, '-b', label='constrained')
Out[18]: [<matplotlib.lines.Line2D at 0x7f9db9ff9c50>]

In [19]: plt.legend(loc='lower right')
Out[19]: <matplotlib.legend.Legend at 0x7f9db9e74320>
```



The constrained fit, represented by the solid blue line is significantly different from the unconstrained fit (dashed, green line). The fit is overall better as a quick RMSE calculation shows:

```
In [20]: rmse_u = np.sqrt(np.sum([(spherical(_, *cof_u) - _)**2 for _ in x]))

In [21]: rmse_c = np.sqrt(np.sum([(spherical(_, *cof) - _)**2 for _ in x]))

In [22]: print('RMSE unconstrained: %.2f' % rmse_u)
RMSE unconstrained: 18.65

In [23]: print('RMSE constrained:   %.2f' % rmse_c)
RMSE constrained:   17.42
```

The last note about fitting a theoretical function, is that both methods assume all lag classes to be equally important for the fit. In the specific case of a variogram this is not true.

Distance weighted fit

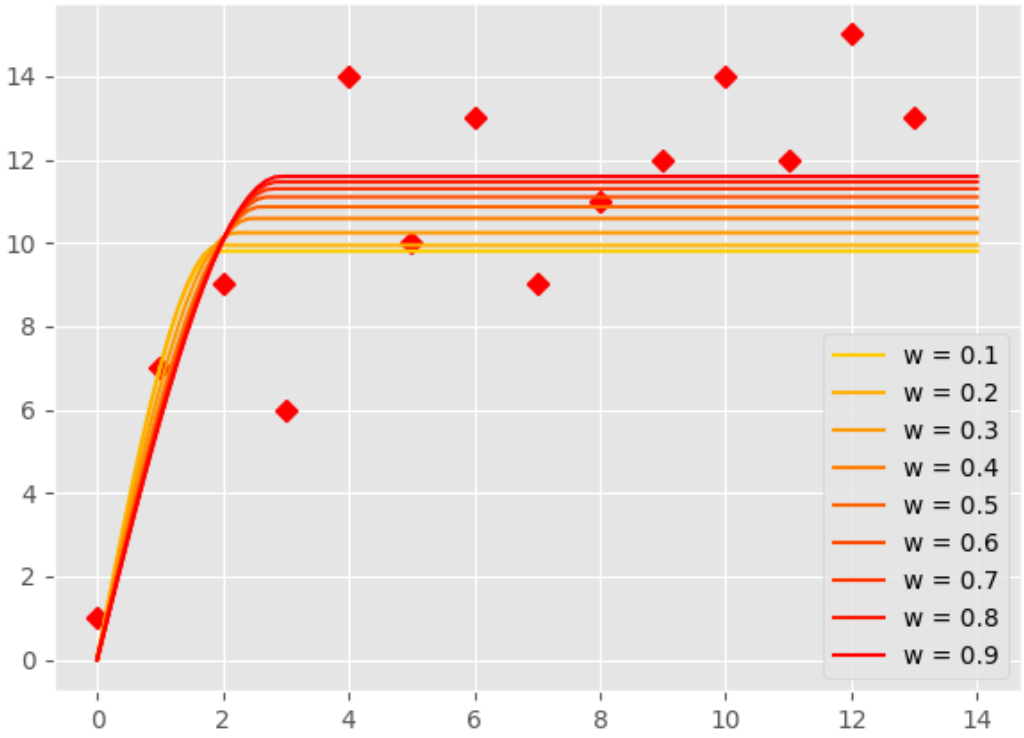
While the standard Levenberg-Marquardt and Trust Region Reflective algorithms are both based on the idea of least squares, they assume all observations to be equally important. In the specific case of a theoretical variogram function, this is not the case. The variogram describes a dependency of covariance in value on the separation distances of the observations. This model already implies that the dependency is stronger on small distances. Considering a kriging interpolation as the main application of the variogram model, points on close distances will get higher weights for the interpolated value of an unobserved location. The weight on large distances will be neglected anyway. Hence, a good fit on small separating distances is way more important. The `curve_fit` function does not have an option for weighting the squares of specific observations. At least it does not call it 'weights'. In terms of scipy, you can define a 'sigma', which is the uncertainty of the respective point. The uncertainty σ influences the least squares calculation as described by the equation:

$$\chi_{sq} = \sum \left(\frac{r}{\sigma}\right)^2$$

That means, the larger σ is, the *less* weight it will receive. That also means, we can almost ignore points, by assigning a ridiculous high σ to them. The following example should illustrate the effect. This time, the first 7 points will be weighted by a weight $\sigma = [0.1, 0.2, \ldots 0.9]$ and the remaining points will receive a $\sigma = 1$. In the case of $\sigma = 0.1$, this would change the least squares cost function to:

$$\chi_{sq;x_{1:7}} = \sum (10r)^2$$

```
In [24]: cm = plt.get_cmap('autumn_r')

In [25]: sigma = np.ones(len(x))

In [26]: fig, ax = plt.subplots(1, 1, figsize=(7, 5))

In [27]: ax.plot(x, y, 'rD')
Out[27]: [<matplotlib.lines.Line2D at 0x7f9dba16a588>]

In [28]: for w in np.arange(0.1, 1., 0.1):
   ....:     s = sigma.copy()
   ....:     s[:6] *= w
   ....:     cof, cov = curve_fit(f, x, y, sigma=s)
   ....:     yi = list(map(lambda x: spherical(x, *cof), xi))
   ....:     ax.plot(xi, yi, linestyle='-', color=cm(w + 0.1), label='w = %.1f' % w)
   ....:

In [29]: ax.legend(loc='lower right')
Out[29]: <matplotlib.legend.Legend at 0x7f9db9de7e10>
```



In the figure above, you can see how the last points get more and more ignored by the fitting. A smaller w value means more weight on the first 7 points. The more yellow lines have a smaller sill and range.

The *Variogram* class accepts lists like sigma from the code example above as *Variogram.fit_sigma* property. This way, the example from above could be implemented. However, *Variogram.fit_sigma* can also apply a function of distance to the lag classes to derive the σ values. There are several predefined functions. These are:

- sigma='linear': The residuals get weighted by the lag distance normalized to the maximum lag distance, denoted as w_n
- sigma='exp': The residuals get weighted by the function: $w = e^{1/w_n}$
- sigma='sqrt': The residuals get weighted by the function: $w = \sqrt(w_n)$
- sigma='sq': The residuals get weighted by the function: $w = w_n^2$

The example below illustrates their effect on the sample experimental variograms used so far.

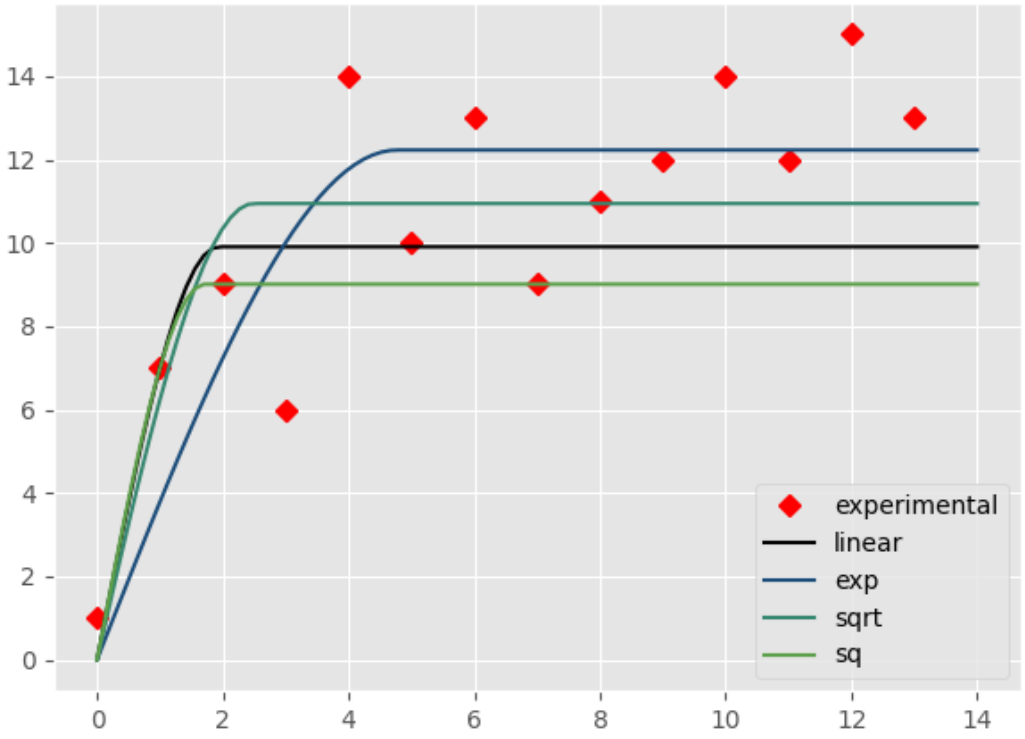
```
In [30]: cm = plt.get_cmap('gist_earth')

# increase the distance by one, to aviod zeros
In [31]: X = np.asarray([(_ + 1) for _ in x])

In [32]: s1 = X / np.max(X)

In [33]: s2 = np.exp(1. / X)

In [34]: s3 = np.sqrt(s1)
```

(continues on next page)

(continued from previous page)

```
In [35]: s4 = np.power(s1, 2)

In [36]: s = (s1, s2, s3, s4)

In [37]: labels = ('linear', 'exp', 'sqrt', 'sq')
```

```
In [38]: plt.plot(x, y, 'rD', label='experimental')
Out[38]: [<matplotlib.lines.Line2D at 0x7f9db9d58828>]

In [39]: for i in range(4):
   ....:     cof, cov = curve_fit(f, x, y, sigma=s[i], p0=(6.,14.), bounds=(0,(14,14)))
   ....:     yi = list(map(lambda x: spherical(x, *cof), xi))
   ....:     plt.plot(xi, yi, linestyle='-', color=cm((i/6)), label=labels[i])
   ....:

In [40]: plt.legend(loc='lower right')
Out[40]: <matplotlib.legend.Legend at 0x7f9db9d7ac50>
```



That's it.

2.5.2 Directional Variograms

General

With version 0.2.2, directional variograms have been introduced. A directional variogram is a variogram where point pairs are only included into the semivariance calculation if they fulfill a specified spatial relation. This relation is expressed as a *search area* that identifies all *directional* points for a given specific point. SciKit-GStat refers to this point as *poi* (point of interest). The implementation is done by the *DirectionalVariogram* class.

Understanding Search Area

Note: The *DirectionalVariogram* is in general capable of handling n-dimensional coordinates. The application of directional dependency is, however, only applied to the first two dimensions.

Understanding the search area of a directional is vital for using the *DirectionalVariogram* class. The search area is controlled by the `directional_model` property which determines the shape of the search area. The extend and orientation of this area is controlled by the parameters:

- *azimuth*
- *tolerance*
- *bandwidth*

As of this writing, SciKit-GStat supports three different search area shapes:

- *triangle* (*default*)
- `circle`
- *compass*

Additionally, the shape generation is controlled by the *tolerance* parameter (*triangle*, *compass*) and *bandwidth* parameter (*triangle*, `circle`). The *azimuth* is used to rotate the search area into a desired direction. An azimuth of 0° is heading East of the coordinate plane. Positive values for azimuth rotate the search area clockwise, negative values counter-clockwise. The *tolerance* specifies how far the angle (against 'x-axis') between two points can be off the azimuth to be still considered as a directional point pair. Based on this definition, two points at a larger distance would generally be allowed to differ more from azimuth in terms of coordinate distance. Therefore the *bandwidth* defines a maximum coordinate distance, a point can have from the azimuth line. The difference between the *triangle* and the *compass* search area is that the triangle uses the bandwidth and the compass does not.

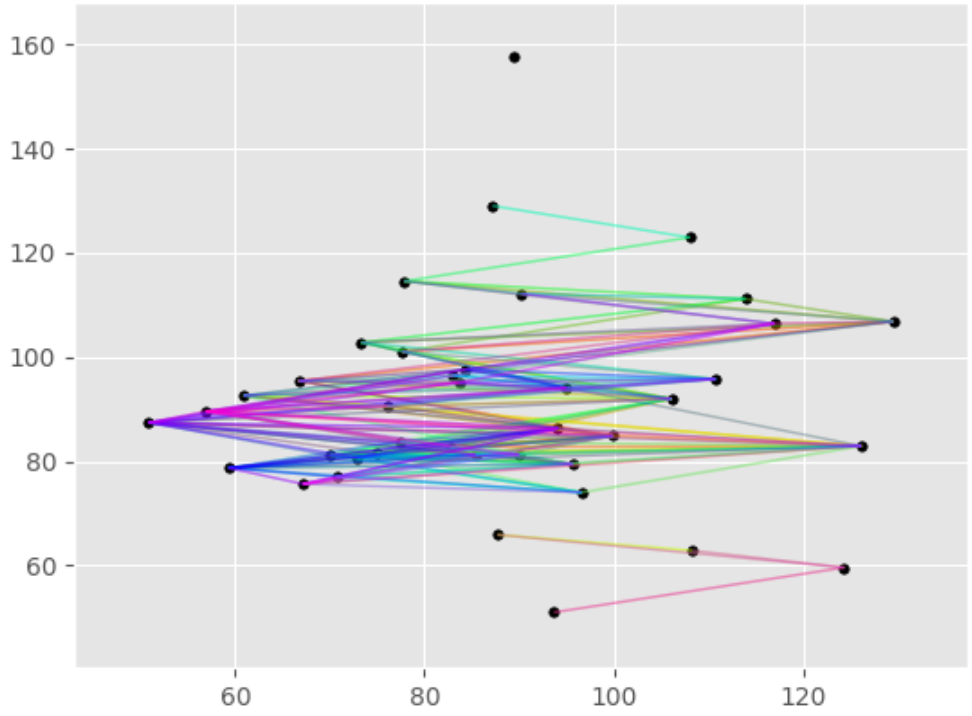
The *DirectionalVariogram* has a function to plot the effect of the search area. The method is called *pair_field*. Using random coordinates, the visualization is shown below.

```
In [1]: from skgstat import DirectionalVariogram

In [2]: import numpy as np

In [3]: import matplotlib.pyplot as plt

In [4]: plt.style.use('ggplot')

In [5]: np.random.seed(42)

In [6]: coords = np.random.gamma(15, 6, (40, 2))
```

(continues on next page)

(continued from previous page)

```
In [7]: np.random.seed(42)

In [8]: vals = np.random.normal(5,1, 40)

In [9]: DV = DirectionalVariogram(coords, vals,
   ...:     azimuth=0,
   ...:     tolerance=45,
   ...:     directional_model='triangle')
   ...:

In [10]: DV.pair_field(plt.gca())
Out[10]: <Figure size 640x480 with 1 Axes>
```



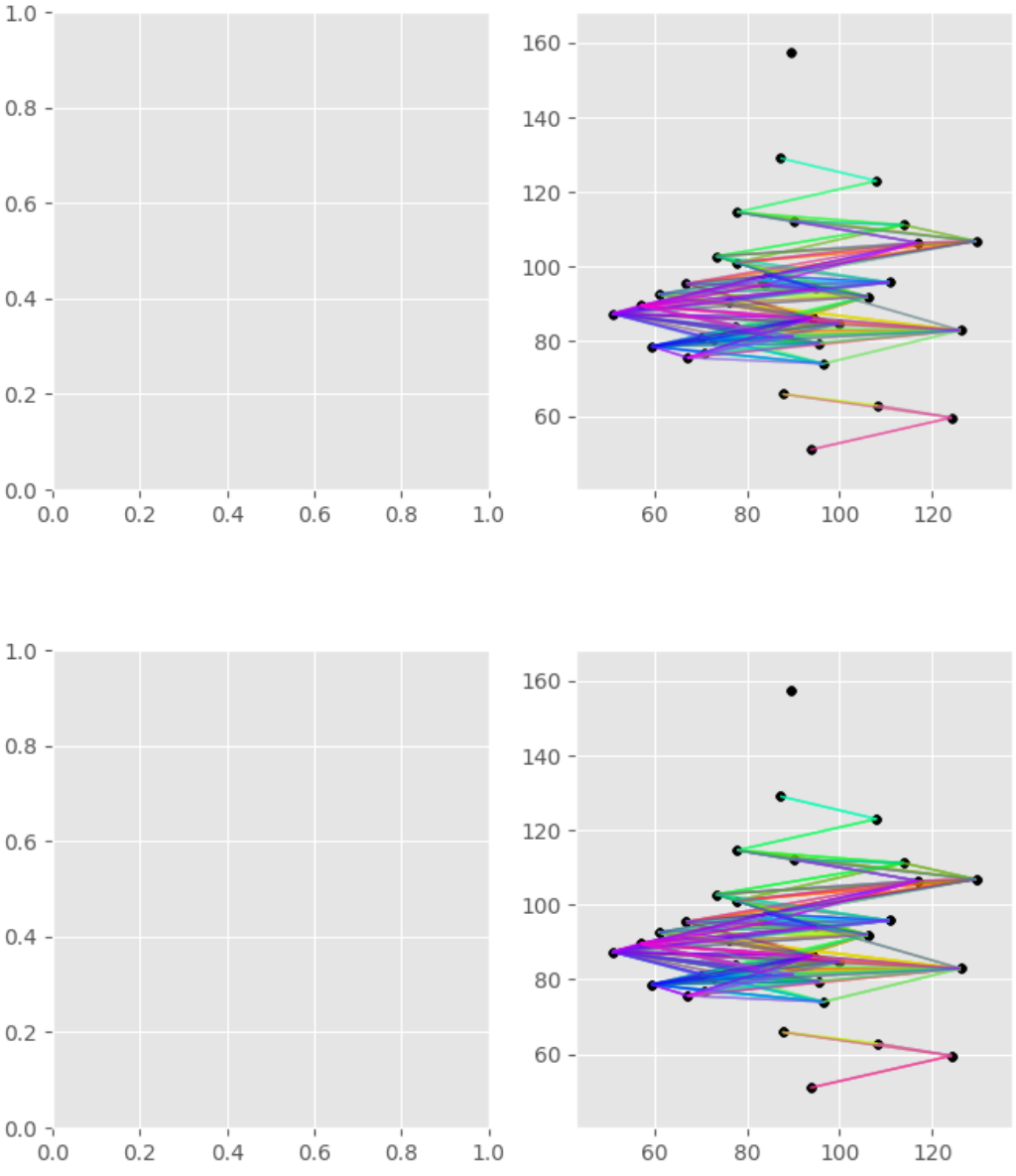
The model can easily be changed, using the *set_directional_model* function:

```
In [11]: fig, axes = plt.subplots(1, 2, figsize=(8, 4))

In [12]: DV.set_directional_model('triangle')

In [13]: DV.pair_field(plt.gca())
Out[13]: <Figure size 800x400 with 2 Axes>

In [14]: DV.pair_field(plt.gca())
```

(continues on next page)

(continued from previous page)

```
Out[14]: <Figure size 800x400 with 2 Axes>

In [15]: fig.show()

In [16]: DV.set_directional_model('compass')

In [17]: DV.pair_field(plt.gca())
Out[17]: <Figure size 800x400 with 2 Axes>

In [18]: fig.show()
```



Directional variograms

In principle, the *DirectionalVariogram* can be used just like the *Variogram* base class. In fact *DirectionalVariogram* inherits most of the behaviour. All the functionality described in the previous sections is added to the basic *Variogram*. All other methods and attributes can be used in the same way.

Warning: In order to implement the directional dependency, some methods have been rewritten in *DirectionalVariogram*. Thus the following methods do **not** show the same behaviour:

- *DirectionalVariogram.bins*
- DirectionalVariogram._calc_groups

2.5.3 Kriging estimate mode

General

Generally speaking, the kriging procedure for one unobserved point (poi) can be broken down into three different steps.

1. calculate the distance matrix between the poi and all observed locations to determine the in-range points and apply the minimum and maximum points to be used constraints.
2. build the kriging equation system by calculating the semi-variance for all distances left over from step 1. Formulate squareform matrix and add the Lagrange multipliers
3. Solve the kriging equation system, usually by matrix inversion.

Hereby, we try to optimize the step 2 for performance. The basic idea is to estimate the semivariances instead of calculating them on each iteration.

Why not calculate?

Calculating the semivariance for all elements in the kriging equation system gives us the best solution for the interpolation problem formulated by the respective variogram. The main point is that the distances for each unobserved location do differ at least slightly from all other unobserved locations in a kriging modeling application. The variogram parameters do not change, they are static within one modeling. This is what we want to use. The main advantage is, that the effective range is constant in this setting. If we can now specify a precision at which we want to resolute the range, we can pre-calculate the corresponding semivariance values. In the time-critical iterative formulation of the kriging equation system, one would use the pre-calculated values of the closest distance.

What about precision?

The precision is a hyperparameter. That means it is up to the user to decide how precise the estimation of the kriging itself can get given an estimated kriging equation system. The main advantage is, that the range and precision are constant values within the scope of a simulation and therefore the expected uncertainty can be calculated and the precision can be adjusted. This will take some effort fine-tune the kriging instance, but it can yield results, that are only numerically different while still increasing the calculation time one magnitude of order.

In terms of uncertainty, one can think of a variogram function, where the given lag distance is uncertain. This deviation can be calculated as:

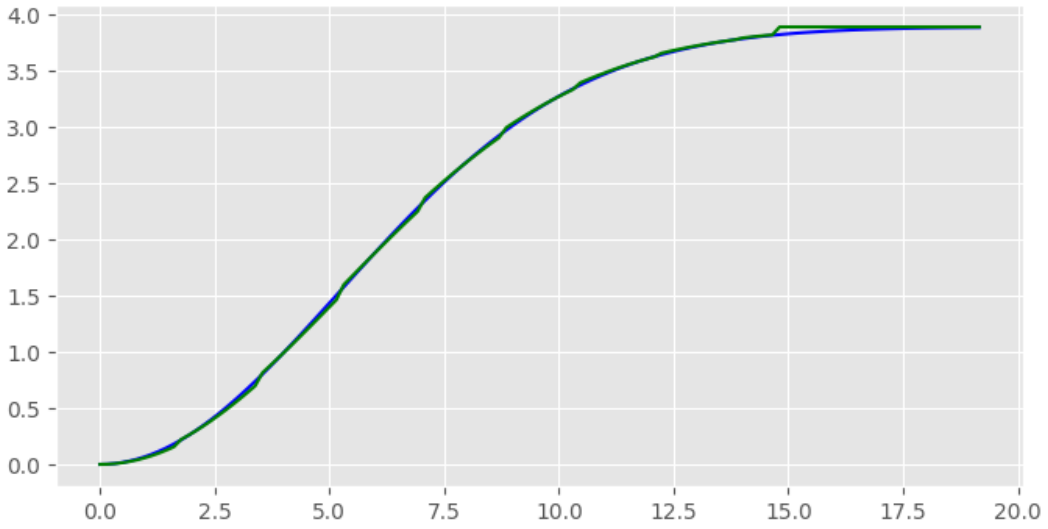
$$d = \frac{range}{precision}$$

and increasing the precision will obviously decrease the lag deviation.

Example

This example should illustrate the idea behind the estimation and show how the precision value can influence the result. An arbitrary variogram is created and then recalculated by the OrdinaryKriging routine to illustrate the precision.

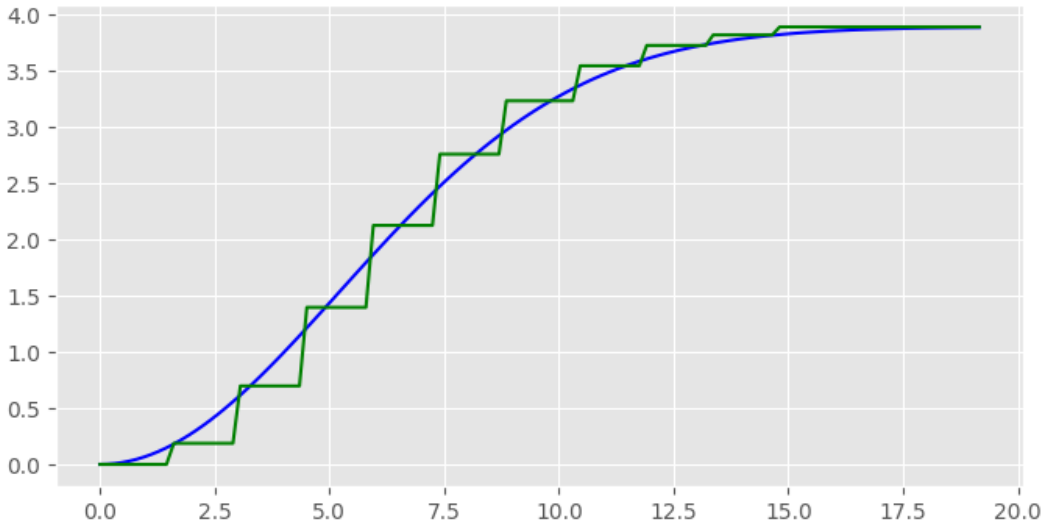
```
In [1]: import matplotlib.pyplot as plt

In [2]: from skgstat import Variogram, OrdinaryKriging

In [3]: import numpy as np

# create some random input
In [4]: np.random.seed(42)

In [5]: c = np.random.gamma(10, 4, size=(100,2))

In [6]: np.random.seed(42)

In [7]: v = np.random.normal(10, 2, size=100)

In [8]: V = Variogram(c, v, model='gaussian', normalize=False)

In [9]: ok = OrdinaryKriging(V, mode='exact')

# exact calculation
In [10]: x = np.linspace(0, ok.range * 1.3, 120)

In [11]: y_c = list(map(ok.gamma_model, x))

# estimation
In [12]: ok.mode = 'estimate'

In [13]: y_e = ok._estimate_matrix(x)

In [14]: plt.plot(x, y_c, '-b', label='exact variogram')
Out[14]: [<matplotlib.lines.Line2D at 0x7f9db9fa3128>]

In [15]: plt.plot(x, y_e, '-g', label='estimated variogram')
Out[15]: [<matplotlib.lines.Line2D at 0x7f9dc327e710>]

In [16]: plt.legend(loc='lower right')
Out[16]: <matplotlib.legend.Legend at 0x7f9dba0d2278>
```



There is almost no difference between the two lines and the result that can be expected will be very similar, as the kriging equation system will yield very similar weights to make the prediction.

If the precision is, however, chosen to coarse, there is a difference in the reconstructed variogram. This way, the idea behind the estimation becomes quite obvious.

```
# make precision really small
In [17]: ok.precision = 10

In [18]: y_e2 = ok._estimate_matrix(x)

In [19]: plt.plot(x, y_c, '-b')
Out[19]: [<matplotlib.lines.Line2D at 0x7f9db9f12cc0>]

In [20]: plt.plot(x, y_e2, '-g')
Out[20]: [<matplotlib.lines.Line2D at 0x7f9db9fbbcf8>]
```



2.6 Code Reference

2.6.1 Variogram Class

class skgstat.**Variogram**(*coordinates=None*, *values=None*, *estimator='matheron'*, *model='spherical'*, *dist_func='euclidean'*, *bin_func='even'*, *normalize=False*, *fit_method='trf'*, *fit_sigma=None*, *use_nugget=False*, *maxlag=None*, *samples=None*, *n_lags=10*, *verbose=False*, ***kwargs*)

Variogram Class

Calculates a variogram of the separating distances in the given coordinates and relates them to one of the semi-variance measures of the given dependent values.

__init__(*coordinates=None*, *values=None*, *estimator='matheron'*, *model='spherical'*, *dist_func='euclidean'*, *bin_func='even'*, *normalize=False*, *fit_method='trf'*, *fit_sigma=None*, *use_nugget=False*, *maxlag=None*, *samples=None*, *n_lags=10*, *verbose=False*, ***kwargs*)

Variogram Class

Parameters

- **coordinates** (*numpy.ndarray*, MetricSpace) – Changed in version 0.5.0: now accepts MetricSpace

  Array of shape (m, n). Will be used as m observation points of n-dimensions. This variogram can be calculated on 1 - n dimensional coordinates. In case a 1-dimensional array is passed, a second array of same length containing only zeros will be stacked to the passed one. For very large datasets, you can set maxlag to only calculate distances within the maximum lag in a sparse matrix. Alternatively you can supply a MetricSpace (optionally with a *max_dist* set for the same effect). This is useful if you're creating many different variograms for different measured parameters that are all measured at the same set of coordinates, as distances will only be calculated once, instead of once per variogram.

- **values** (*numpy.ndarray*) – Array of values observed at the given coordinates. The length of the values array has to match the m dimension of the coordinates array. Will be used to

calculate the dependent variable of the variogram.

- **estimator** (*str, callable*) – String identifying the semi-variance estimator to be used. Defaults to the Matheron estimator. Possible values are:
  - matheron [Matheron, default]
  - cressie [Cressie-Hawkins]
  - dowd [Dowd-Estimator]
  - genton [Genton]
  - minmax [MinMax Scaler]
  - entropy [Shannon Entropy]

  If a callable is passed, it has to accept an array of absoulte differences, aligned to the 1D distance matrix (flattened upper triangle) and return a scalar, that converges towards small values for similarity (high covariance).
- **model** (*str*) – String identifying the theoretical variogram function to be used to describe the experimental variogram. Can be one of:
  - spherical [Spherical, default]
  - exponential [Exponential]
  - gaussian [Gaussian]
  - cubic [Cubic]
  - stable [Stable model]
  - matern [Matérn model]
  - nugget [nugget effect variogram]
- **dist_func** (*str*) – String identifying the distance function. Defaults to 'euclidean'. Can be any metric accepted by scipy.spatial.distance.pdist. Additional parameters are not (yet) passed through to pdist. These are accepted by pdist for some of the metrics. In these cases the default values are used.
- **bin_func** (*str | Callable | Iterable*) – Changed in version 0.3.8: added 'fd', 'sturges', 'scott', 'sqrt', 'doane'

  Changed in version 0.3.9: added 'kmeans', 'ward'

  String identifying the binning function used to find lag class edges. All methods calculate bin edges on the interval [0, maxlag[. Possible values are:
  - *'even'* (default) finds *n_lags* same width bins
  - *'uniform'* forms *n_lags* bins of same data count
  - *'fd'* applies Freedman-Diaconis estimator to find *n_lags*
  - *'sturges'* applies Sturge's rule to find *n_lags*.
  - *'scott'* applies Scott's rule to find *n_lags*
  - *'doane'* applies Doane's extension to Sturge's rule to find *n_lags*
  - *'sqrt'* uses the square-root of `distance` as *n_lags*.
  - *'kmeans'* uses KMeans clustering to well supported bins
  - *'ward'* uses hierachical clustering to find minimum-variance clusters.

More details are given in the documentation for *set_bin_func*.

- **normalize** (*bool*) – Defaults to False. If True, the independent and dependent variable will be normalized to the range [0,1].
- **fit_method** (*str | None*) – Changed in version 0.3.10: Added 'ml' and 'custom'

  String identifying the method to be used for fitting the theoretical variogram function to the experimental. If None is passed, the fit does not run. More info is given in the Variogram.fit docs. Can be one of:

  - 'lm': Levenberg-Marquardt algorithm for unconstrained problems. This is the faster algorithm, yet is the fitting of a variogram not unconstrianed.
  - 'trf': Trust Region Reflective function for non-linear constrained problems. The class will set the boundaries itself. This is the default function.
  - 'ml': Maximum-Likelihood estimation. With the current implementation only the Nelder-Mead solver for unconstrained problems is implemented. This will estimate the variogram parameters from a Gaussian parameter space by minimizing the negative log-likelihood.
  - 'manual': Manual fitting. You can set the range, sill and nugget either directly to the *fit* function, or as *fit_* prefixed keyword arguments on Variogram instantiation.
- **fit_sigma** (*numpy.ndarray, str*) – Defaults to None. The sigma is used as measure of uncertainty during variogram fit. If fit_sigma is an array, it has to hold n_lags elements, giving the uncertainty for all lags classes. If fit_sigma is None (default), it will give no weight to any lag. Higher values indicate higher uncertainty and will lower the influcence of the corresponding lag class for the fit. If fit_sigma is a string, a pre-defined function of separating distance will be used to fill the array. Can be one of:

  - 'linear': Linear loss with distance. Small bins will have higher impact.
  - 'exp': The weights decrease by a e-function of distance
  - 'sqrt': The weights decrease by the squareroot of distance
  - 'sq': The weights decrease by the squared distance.

  More info is given in the Variogram.fit_sigma documentation.
- **use_nugget** (*bool*) – Defaults to False. If True, a nugget effet will be added to all Variogram.models as a third (or fourth) fitting parameter. A nugget is essentially the y-axis interception of the theoretical variogram function.
- **maxlag** (*float, str*) – Can specify the maximum lag distance directly by giving a value larger than 1. The binning function will not find any lag class with an edge larger than maxlag. If 0 < maxlag < 1, then maxlag is relative and maxlag * max(Variogram.distance) will be used. In case maxlag is a string it has to be one of 'median', 'mean'. Then the median or mean of all Variogram.distance will be used. Note maxlag=0.5 will use half the maximum separating distance, this is not the same as 'median', which is the median of all separating distances
- **samples** (*float, int*) – If set to a non-None value point pairs are sampled randomly. Two random subset of all points are chosen, and the distance matrix is calculated only between these two subsets. The size of each subset is set by *samples*: if < 1 it specifies a fraction of all points, if >= 1 it specifies the number of points in each subset.
- **n_lags** (*int*) – Specify the number of lag classes to be defined by the binning function.
- **verbose** (*bool*) – Set the Verbosity of the class. Not Implemented yet.

Keyword Arguments

- **entropy_bins** (*int, str*) – New in version 0.3.7.

  If the *estimator <skgstat.Variogram.estimator>* is set to *'entropy'* this argument sets the number of bins, that should be used for histogram calculation.

- **percentile** (*int*) – New in version 0.3.7.

  If the *estimator <skgstat.Variogram.estimator>* is set to *'entropy'* this argument sets the percentile to be used.

- **binning_random_state** (*int, None*) – New in version 0.3.9.

  If *bin_func* is *'kmeans'* this can overwrite the seed for the initial guess of the cluster centroids. Note, that K-Means is not deterministic and is therefore seeded to 42 here. You can pass *None* to disable this behavior, but use it with care, as you will get different results.

- **binning_agg_func** (*str*) – New in version 0.3.10.

  If *bin_func* is *'ward'* this keyword argument can switch from default mean aggregation to median aggregation for calculating the cluster centroids.

- **obs_sigma** (*int, float*) – New in version 0.6.0.

  If set, the Variogram will use this sigma as the standard deviation of the observations passed as values. Using a MonteCarlo simulation the uncertainties are propagated into the experimental variogram. If present, the plot will indicate the confidence interval as error bars around the experimental variogram.

property NS

Nash Sutcliffe efficiency of the fitted Variogram

Returns

property bin_func

Binning function

Returns an instance of the function used for binning the separating distances into the given amount of bins. Both functions use the same signature of func(distances, n, maxlag).

The setter of this property utilizes the Variogram.set_bin_func to set a new function.

Returns **binning_function**

Return type function

See also:

Variogram.set_bin_func

property bins

Distance lag bins

Independent variable of the the experimental variogram sample. The bins are the upper edges of all calculated distance lag classes. If you need bin centers, use *get_empirical*.

Returns **bins** – 1D array of the distance lag classes.

Return type numpy.ndarray

See also:

Variogram.get_empirical

clone()
Deep copy of self

Return a deep copy of self.

Returns

Return type *Variogram*

property coordinates
Coordinates property

Array of observation locations the variogram is build for. This property has no setter. If you want to change the coordinates, use a new Variogram instance.

Returns coordinates

Return type numpy.array

cross_validate(*method: str = 'jacknife'*, *n: Optional[int] = None*, *metric: str = 'rmse'*, *seed=None*) → float
Cross validation of the variogram model by means of Kriging. Right now, this function can only utilize a jacknife (leave-one-out) cross validation and will only use the builtin OrdinaryKriging method (not yet the to_gs_krige interface).

Parameters

- **method** (*str*) – Right now, 'jacknife' is the only possible input.
- **n** (*int*) – The number of points to be included into the cross-validation. If None (default), all points will be used.
- **metric** (*str*) – Metric used for cross-validation. Can be root mean square error (rmse), mean squared error (mse) or mean absolute error (mae).
- **seed** (*int*) – If n is not None, the random selection of input data for the cross-validation can be seeded.

Returns metric – The cross-validation result as specified above.

Return type float

See also:

skgstat.util.cross_validation.jacknife

data(*n=100*, *force=False*)
Theoretical variogram function

Calculate the experimental variogram and apply the binning. On success, the variogram model will be fitted and applied to n lag values. Returns the lags and the calculated semi-variance values. If force is True, a clean preprocessing and fitting run will be executed.

Parameters

- **n** (*integer*) – length of the lags array to be used for fitting. Defaults to 100, which will be fine for most plots
- **force** (*boolean*) – If True, the preprocessing and fitting will be executed as a clean run. This will force all intermediate results to be recalculated. Defaults to False

Returns variogram – first element is the created lags array second element are the calculated semi-variance values

Return type tuple

describe(*short=False*, *flat=False*)
Variogram parameters

Return a dictionary of the variogram parameters.

Changed in version 0.3.7: The describe now returns all init parameters in as the *describe()['params']* key and all keyword arguments as *describe()['kwargs']*. This output can be suppressed by setting *short=True*.

Parameters

- **short** (*bool*) – If *True*, the *'params'* and *'kwargs'* keys will be omitted. Defaults to *False*.
- **flat** (*bool*) – If *True*, the *'params'* and *'kwargs'* nested *dict`s will be distributed to the main `dict* to return a flat *dict*. Defaults to *False*

Returns **parameters** – Returns fitting parameters of the theoretical variogram model along with the init parameters of the *Variogram <skgstat.Variogram>* instance.

Return type dict

property dim
Input coordinates dimensionality.

distance_difference_plot(*ax=None*, *plot_bins=True*, *show=True*)
Raw distance plot

Plots all absoulte value differences of all point pair combinations over their separating distance, without sorting them into a lag.

Changed in version 0.4.0: This plot can be plotted with the plotly plotting backend

Parameters

- **ax** (*None, AxesSubplot*) – If None, a new matplotlib.Figure will be created. In case a Figure was already created, pass the Subplot to use as ax argument.
- **plot_bins** (*bool*) – If True (default) the bin edges will be included into the plot.
- **show** (*bool*) – If True (default), the show method of the Figure will be called before returning the Figure. Can be set to False, to avoid doubled figure rendering in Jupyter notebooks.

Returns

Return type matplotlib.pyplot.Figure

property experimental
Experimental Variogram

Array of experimental (empirical) semivariance values. The array length will be aligned to Variogram.bins. The current Variogram.estimator has been used to calculate the values. Depending on the setting of Variogram.harmonize (True | False), either Variogram._experimental or Variogram.isotonic will be returned.

Returns **vario** – Array of the experimental semi-variance values aligned to Variogram.bins.

Return type numpy.ndarray

See also:

`Variogram._experimental`, `Variogram.isotonic`

fit(*force=False*, *method=None*, *sigma=None*, ***kwargs*)
Fit the variogram

The fit function will fit the theoretical variogram function to the experimental. The preprocessed distance matrix, pairwise differences and binning will not be recalculated, if already done. This could be forced by setting the force parameter to true.

In case you call fit function directly, with method or sigma, the parameters set on Variogram object instantiation will get overwritten. All other keyword arguments will be passed to scipy.optimize.curve_fit function.

Changed in version 0.3.10: added 'ml' and 'custom' method.

Parameters

- **force** (*bool*) – If set to True, a clean preprocessing of the distance matrix, pairwise differences and the binning will be forced. Default is False.
- **method** (*string*) – A string identifying one of the implemented fitting procedures. Can be one of:
  - lm: Levenberg-Marquardt algorithms implemented in scipy.optimize.leastsq function.
  - trf: Trust Region Reflective algorithm implemented in scipy.optimize.least_squares(method='trf')
  - 'ml': Maximum-Likelihood estimation. With the current implementation only the Nelder-Mead solver for unconstrained problems is implemented. This will estimate the variogram parameters from a Gaussian parameter space by minimizing the negative log-likelihood.
  - 'manual': Manual fitting. You can set the range, sill and nugget either directly to the *fit* function, or as *fit_* prefixed keyword arguments on Variogram instantiation.

sigma [string, array] Uncertainty array for the bins. Has to have the same dimension as self.bins. Refer to Variogram.fit_sigma for more information.

Returns

Return type void

See also:

scipy.optimize.minimize, scipy.optimize.curve_fit, scipy.optimize.leastsq, scipy.optimize.least_squares

property fit_method

New in version 0.6.2.

Set the fit method to be used for this Variogram instance. Possible values are:

- `'trf'` - Trust-Region Reflective (default)
- `'lm'` - Levenberg-Marquardt
- `'ml'` - Maximum Likelihood estimation
- *'manual'* ` - Manual fitting by setting the parameters

Changed in version 0.6.6: Passing *None* will prevent the fitting procedure from running.

See also:

scipy.optimize.minimize, scipy.optimize.curve_fit, *Variogram.fit*

Notes

The default method (TRF) is a bounded least squares method, that sets constraints to the value space of all parameters. All methods use an initial guess for all used parameters. This is *max(bins)* for the range, *max(experimental)* for the sill, 20 for the Matérn smoothness, 2 for the stable model shape and 1 for the nugget if used.

property fit_sigma

Fitting Uncertainty

Set or calculate an array of observation uncertainties aligned to the Variogram.bins. These will be used to weight the observations in the cost function, which divides the residuals by their uncertainty.

When setting fit_sigma, the array of uncertainties itself can be given, or one of the strings: [‘linear’, ‘exp’, ‘sqrt’, ‘sq’, ‘entropy’]. The parameters described below refer to the setter of this property.

Changed in version 0.3.11: added the ‘entropy’ option.

Parameters **sigma** (*string, array*) – Sigma can either be an array of discrete uncertainty values, which have to align to the Variogram.bins, or of type string. Then, the weights for fitting are calculated as a function of (lag) distance.

- **sigma=’linear’**: The residuals get weighted by the lag distance normalized to the maximum lag distance, denoted as w_n
- **sigma=’exp’**: The residuals get weighted by the function: $w = e^{1/w_n}$
- **sigma=’sqrt’**: The residuals get weighted by the function: $w = \sqrt(w_n)$
- **sigma=’sq’**: The residuals get weighted by the function: $w = w_n^2$
- **sigma=’entropy’**: Calculates the Shannon Entropy as intrinsic uncertainty of each lag class.

Returns

Return type void

Notes

The cost function is defined as:

$$chisq = \sum \frac{r}{\sigma}^2$$

where r are the residuals between the experimental variogram and the modeled values for the same lag. Following this function, small values will increase the influence of that residual, while a very large sigma will cause the observation to be ignored.

See also:

scipy.optimize.curve_fit

property fitted_model

Fitted Model

Returns a callable that takes a distance value and returns a semivariance. This model is fitted to the current Variogram parameters. The function will be interpreted at return time with the parameters hard-coded into the function code.

Returns **model** – The current semivariance model fitted to the current Variogram model parameters.

Return type callable

`get_empirical`(*bin_center=False*)
Empirical variogram

Returns a tuple of dependent and independent sample values, this *Variogram* is estimated for. This is a tuple of the current *bins* and *experimental* semi-variance values. By default the upper bin edges are used. This can be set to bin center by the *bin_center* argument.

Parameters **bin_center** (*bool*) – If set to *True*, the center for each distance lag bin is used over the upper limit (default).

Returns

- **bins** (*numpy.ndarray*) – 1D array of *n_lags* distance lag bins.
- **experimental** (*numpy.ndarray*) – 1D array of *n_lags* experimental semi-variance values.

See also:

Variogram.bins, *Variogram.experimental*

`lag_classes`()
Iterate over the lag classes

Generates an iterator over all lag classes. Can be zipped with Variogram.bins to identify the lag.

Changed in version 0.3.6: yields an empty array for empty lag groups now

Returns

Return type iterable

`lag_groups`()
Lag class groups

Retuns a mask array with as many elements as self._diff has, identifying the lag class group for each pairwise difference. Can be used to extract all pairwise values within the same lag bin.

Returns

Return type numpy.ndarray

See also:

Variogram.lag_classes

`location_trend`(*axes=None*, *show=True*, ***kwargs*)
Location Trend plot

Plots the values over each dimension of the coordinates in a scatter plot. This will visually show correlations between the values and any of the coordinate dimension. If there is a value dependence on the location, this would violate the intrinsic hypothesis. This is a weaker form of stationarity of second order.

Changed in version 0.4.0: This plot can be plotted with the plotly plotting backend

Parameters

- **axes** (*list*) – Can be None (default) or a list of matplotlib.AxesSubplots. If a list is passed, the location trend plots will be plotted on the given instances. Note that then length of the list has to match the dimeonsionality of the coordinates array. In case 3D coordinates are used, three subplots have to be given.
- **show** (*boolean*) – If True (default), the *show* method of the Figure will be called. Can be set to False to prevent duplicated plots in some environments.

Keyword Arguments `add_trend_line` (*bool*) – New in version 0.3.5.

If set to *True*, the class will fit a linear model to each coordinate dimension and output the model along with a calculated R^2. With high R^2 values, you should consider rejecting the input data, or transforming it.

Note: Right now, this is only supported for `'plotly'` backend

Returns **fig** – The figure produced by the function. Dependends on the current backend.

Return type matplotlib.Figure, plotly.graph_objects.Figure

`property mae`

RMSE

Calculate the Mean absolute error between the experimental variogram and the theoretical model values at corresponding lags. Can be used as a fitting quality measure.

Returns

Return type float

See also:

Variogram.residuals

Notes

The MAE is implemented like:

$$MAE = \frac{\sum_{i=0}^{i=N(x)} |x - y|}{N(x)}$$

`property maxlag`

Maximum lag distance to be considered in this Variogram instance. You can limit the distance at which point pairs are calcualted. There are three possible ways how to do that, in absoulte lag units, which is a number larger one. Secondly, a number `0 < maxlag < 1` can be set, which will use this share of the maximum distance as maxlag. Lastly, a string can be set: `'mean'` and `'median'` for the mean or median value of the distance matrix.

Notes

This setting is largely flexible, but all options except the absolute limit in lag units need the full distance matrix to be calculated. Hence, it does **not** speed up the calculation of large distance matrices, just the estimation of the variogram. Thus, if you pre-calcualte the distance matrix using *MetricSpace*, only absoulte limits can be used.

`property mean_residual`

Mean Model residuals

Calculates the mean, absoulte deviations between the experimental variogram and theretical model values.

Returns

Return type float

property metric_space
New in version 0.5.6.

MetricSpace representation of the input coordinates. A *MetricSpace* can be used to pass pre-calculated coordinates to other *Variogram* instances.

Returns **metric_space**

Return type *skgstat.MetricSpace*

See also:

Variogram.coordinates coordinate representation

model_deviations()
Model Deviations

Calculate the deviations between the experimental variogram and the recalculated values for the same bins using the fitted theoretical variogram function. Can be utilized to calculate a quality measure for the variogram fit.

Returns **deviations** – first element is the experimental variogram second element are the corresponding values of the theoretical model.

Return type tuple

property mse
RMSE

Calculate the Mean squared error between the experimental variogram and the theoretical model values at corresponding lags. Can be used as a fitting quality measure.

Returns

Return type float

See also:

Variogram.residuals

Notes

The MSE is implemented like:

$$MSE = \frac{\sum_{i=0}^{i=N(x)} (x - y)^2}{N(x)}$$

property n_lags
Number of lag bins

Pass the number of lag bins to be used on this Variogram instance. This will reset the grouping index and fitting parameters

property nrmse
NRMSE

Calculate the normalized root mean squared error between the experimental variogram and the theoretical model values at corresponding lags. Can be used as a fitting quality measure

Returns

Return type float

See also:

Variogram.residuals, *Variogram.rmse*

Notes

The NRMSE is implemented as:

$$NRMSE = \frac{RMSE}{mean(y)}$$

where RMSE is Variogram.rmse and y is Variogram.experimental

property nrmse_r
NRMSE

Alternative normalized root mean squared error between the experimental variogram and the theoretical model values at corresponding lags. Can be used as a fitting quality measure.

Returns

Return type float

See also:

Variogram.rmse, *Variogram.nrmse*

Notes

Unlike Variogram.nrmse, nrmse_r is not normalized to the mean of y, but the differece of the maximum y to its mean:

$$NRMSE_r = \frac{RMSE}{max(y) - mean(y)}$$

property parameters
Extract just the variogram parameters range, sill and nugget from the *describe* output.

Returns **params** – [range, sill, nugget] for most models and [range, sill, shape, nugget] for matern and stable model.

Return type list

plot(*axes=None*, *grid=True*, *show=True*, *hist=True*)
Variogram Plot

Plot the experimental variogram, the fitted theoretical function and an histogram for the lag classes. The axes attribute can be used to pass a list of AxesSubplots or a single instance to the plot function. Then these Subplots will be used. If only a single instance is passed, the hist attribute will be ignored as only the variogram will be plotted anyway.

Changed in version 0.4.0: This plot can be plotted with the plotly plotting backend

Parameters

- **axes** (*list, tuple, array, AxesSubplot or None*) – If None, the plot function will create a new matplotlib figure. Otherwise a single instance or a list of AxesSubplots can be passed to be used. If a single instance is passed, the hist attribute will be ignored.
- **grid** (*bool*) – Defaults to True. If True a custom grid will be drawn through the lag class centers

- **show** (*bool*) – Defaults to True. If True, the show method of the passed or created matplotlib Figure will be called before returning the Figure. This should be set to False, when used in a Notebook, as a returned Figure object will be plotted anyway.
- **hist** (*bool*) – Defaults to True. If False, the creation of a histogram for the lag classes will be suppressed.

Returns

Return type matplotlib.Figure

preprocessing(*force=False*)
Preprocessing function

Prepares all input data for the fit and transform functions. Namely, the distances are calculated and the value differences. Then the binning is set up and bin edges are calculated. If any of the listed subsets are already prepared, their processing is skipped. This behaviour can be changed by the force parameter. This will cause a clean preprocessing.

Parameters **force** (*bool*) – If set to True, all preprocessing data sets will be deleted. Use it in case you need a clean preprocessing.

Returns

Return type void

property r
Pearson correlation of the fitted Variogram

Returns

property residuals
Model residuals

Calculate the model residuals defined as the differences between the experimental variogram and the theoretical model values at corresponding lag values

Returns

Return type numpy.ndarray

property rmse
RMSE

Calculate the Root Mean squared error between the experimental variogram and the theoretical model values at corresponding lags. Can be used as a fitting quality measure.

Returns

Return type float

See also:

Variogram.residuals

Notes

The RMSE is implemented like:

$$RMSE = \sqrt{\frac{\sum_{i=0}^{i=N(x)}(x-y)^2}{N(x)}}$$

scattergram(*ax=None*, *show=True*, ***kwargs*)

Scattergram plot

Groups the values by lags and plots the head and tail values of all point pairs within the groups against each other. This can be used to investigate the distribution of the value residuals.

Changed in version 0.4.0: This plot can be plotted with the plotly plotting backend

Parameters

- **ax** (*matplotlib.Axes, plotly.graph_objects.Figure*) – If None, a new plotting Figure will be created. If given, it has to be an instance of the used plotting backend, which will be used to plot on.
- **show** (*boolean*) – If True (default), the *show* method of the Figure will be called. Can be set to False to prevent duplicated plots in some environments.

Returns **fig** – Resulting figure, depending on the plotting backend

Return type matplotlib.Figure, plotly.graph_objects.Figure

set_bin_func(*bin_func: Union[Iterable, Callable[[numpy.ndarray, float, float], Tuple[numpy.ndarray, float]]]*)

Set binning function

Sets a new binning function to be used. The new binning method is set by either a string identifying the new function to be used, or an iterable containing the bin edges, or any function that can compute bins from the distances, number of lags and maximum lag. The string can be one of: [‘even’, ‘uniform’, ‘fd’,

‘sturges’, ‘scott’, ‘sqrt’, ‘doane’].

If the number of lag classes should be estimated automatically, it is recommended to use ‘ sturges’ for small, normal distributed locations and ‘fd’ or ‘scott’ for large datasets, where ‘fd’ is more robust to outliers. ‘sqrt’ is by far the fastest estimator. ‘doane’ is an extension of Sturge’s rule for non-normal distributed data.

Changed in version 0.3.8: added ‘fd’, ‘sturges’, ‘scott’, ‘sqrt’, ‘doane’

Changed in version 0.3.9: added ‘kmeans’, ‘ward’

Changed in version 0.4.0: added ‘stable_entropy’

Changed in version 0.4.1: refactored local wrapper function definition. The wrapper to pass kwargs to the binning functions is now implemented as a instance method, to make it pickleable.

Changed in version 0.6.5: added iterable and function as arguments to allow for custom bins.

Parameters **bin_func** (*str | Iterable | Callable*) – Can be one of:

- ’even’
- ’uniform’
- ’fd’
- ’sturges’
- ’scott’

- 'sqrt'
- 'doane'
- 'kmeans'
- 'ward'
- 'stable_entropy'

Returns

Return type void

Notes

``'even'``: Use skgstat.binning.even_width_lags for using n_lags lags of equal width up to maxlag.

``'uniform'``: Use skgstat.binning.uniform_count_lags for using n_lags lags up to maxlag in which the pairwise differences follow a uniform distribution.

``'sturges'``: estimates the number of evenly distributed lag classes (n) by Sturges rule[101]:

$$n = log_2 n + 1$$

``'scott'``: estimates the lag class widths (h) by Scott's rule[102]:

$$h = \sigma \frac{24 * \sqrt{\pi}^{\frac{1}{3}}}{n}$$

``'sqrt'``: estimates the number of lags (n) by the suare-root:

$$n = \sqrt{n}$$

``'fd'``: estimates the lag class widths (h) using the Freedman Diaconis estimator[103]:

$$h = 2\frac{IQR}{n^{1/3}}$$

``'doane'``: estimates the number of evenly distributed lag classes using Doane's extension to Sturge's rule[104]:

$$n = 1 + \log_2(s) + \log_2\left(1 + \frac{|g|}{k}\right) g = E\left[\left(\frac{x - \mu_g}{\sigma}\right)^3\right] k = \sqrt{\frac{6(s-2)}{(s+1)(s+3)}}$$

``'kmeans'``: This method will search for *n* clusters in the distance matrix. The cluster centroids are used to calculate the upper edges of the lag classes, by setting it to half of the distance between two neighboring clusters. Note: This does not necessarily result in even width bins.

``'ward'`` uses a hierachical culstering algorithm to iteratively merge pairs of clusters until there are only *n* remaining clusters. The merging is done by minimizing the variance for the merged cluster.

``'stable_entropy'`` will adjust *n* bin edges by minimizing the absolute differences between each lag's Shannon Entropy. This will lead to uneven bin widths. Each lag class value distribution will be of comparable intrinsic uncertainty from an information theoretic point of view, which makes the semi-variances quite

[101] Scott, D.W. (2009), Sturges' rule. WIREs Comp Stat, 1: 303-306. https://doi.org/10.1002/wics.35
[102] Scott, D.W. (2010), Scott's rule. WIREs Comp Stat, 2: 497-502. https://doi.org/10.1002/wics.103
[103] Freedman, David, and Persi Diaconis (1981), "On the histogram as a density estimator: L 2 theory." Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete 57.4: 453-476.
[104] Doane, D. P. (1976). Aesthetic frequency classifications. The American Statistician, 30(4), 181-183.

comparable. However, it is not guaranteed, that the binning makes any sense from a geostatistical point of view, as the first lags might be way too wide.

See also:

Variogram.bin_func, *skgstat.binning.uniform_count_lags*, *skgstat.binning.even_width_lags*, *skgstat.binning.auto_derived_lags*, *skgstat.binning.kmeans*, *skgstat.binning.ward*, sklearn.cluster.KMeans, sklearn.cluster.AgglomerativeClustering

References

set_dist_function(*func*)
Set distance function

Set the function used for distance calculation. func can either be a callable or a string. The ranked distance function is not implemented yet. strings will be forwarded to the scipy.spatial.distance.pdist function as the metric argument. If func is a callable, it has to return the upper triangle of the distance matrix as a flat array (Like the pdist function).

Parameters **func** (*string, callable*) –

Returns

Return type numpy.array

set_model(*model_name*)
Set model as the new theoretical variogram function.

set_values(*values*, *calc_diff=True*)
Set new values

Will set the passed array as new value array. This array has to be of same length as the first axis of the coordinates array. The Variogram class does only accept one dimensional arrays. On success all fitting parameters are deleted and the pairwise differences are recalculated. Raises :py:class:`ValueError`s on shape mismatches and a Warning

Parameters **values** (*numpy.ndarray*) –

Returns

Return type void

:raises ValueError : raised if the values array shape does not match the: coordinates array, or more than one dimension given :raises Warning : raised if all input values are the same:

See also:

Variogram.values

to_DataFrame(*n=100*, *force=False*)
Variogram DataFrame

Returns the fitted theoretical variogram as a pandas.DataFrame instance. The n and force parameter control the calaculation, refer to the data funciton for more info.

Parameters

- **n** (*integer*) – length of the lags array to be used for fitting. Defaults to 100, which will be fine for most plots
- **force** (*boolean*) – If True, the preprocessing and fitting will be executed as a clean run. This will force all intermediate results to be recalculated. Defaults to False

Returns

Return type pandas.DataFrame

See also:

Variogram.data

to_gs_krige(***kwargs*)
Instatiate a GSTools Krige class.

This can only export isotropic models. Note: the *fit_variogram* is always set to *False*

Parameters

- **variogram** (skgstat.Variogram) – Scikit-GStat Variogram instamce
- ****kwargs** – Keyword arguments forwarded to GSTools Krige. Refer to Krige to learn about all possible options. Note that the *fit_variogram* parameter will always be False.

Raises

- **ImportError** – When GSTools is not installed.
- **ValueError** – When GSTools version is not v1.3 or greater.
- **ValueError** – When given Variogram model is not supported ('harmonize').

Returns Instantiated GSTools Krige class.

Return type Krige

See also:

gstools.Krige

to_gstools(***kwargs*)
Instantiate a corresponding GSTools CovModel.

By default, this will be an isotropic model.

Parameters ****kwargs** – Keyword arguments forwarded to the instantiated GSTools CovModel. The default parameters 'dim', 'var', 'len_scale', 'nugget', 'rescale' and optional shape parameters will be extracted from the given Variogram but they can be overwritten here.

Raises

- **ImportError** – When GSTools is not installed.
- **ValueError** – When GSTools version is not v1.3 or greater.
- **ValueError** – When given Variogram model is not supported ('harmonize').

Returns Corresponding GSTools covmodel.

Return type CovModel

Note: In case you intend to use the *coordinates* in a GSTools workflow, you need to transpose the coordinate array like:

>> cond_pos Variogram.coordinates.T

---

transform(*x*)
Transform

Transform a given set of lag values to the theoretical variogram function using the actual fitting and preprocessing parameters in this Variogram instance

Parameters **x** (*numpy.array*) – Array of lag values to be used as model input for the fitted theoretical variogram model

Returns

Return type numpy.array

property triangular_distance_matrix

Like distance_matrix but with zeros below the diagonal… Only defined if distance_matrix is a sparse matrix

update_kwargs(***kwargs*)

New in version 0.3.7.

Update the keyword arguments of this Variogram instance. The keyword arguments will be validated first and the update the existing kwargs. That means, you can pass only the kwargs, which need to be updated.

Note: Updating the kwargs does not force a preprocessing circle. Any affected intermediate result, that might be cached internally, will not make use of updated kwargs. Make a call to *preprocessing(force=True)* to force a clean re-calculation of the Variogram instance.

---

property value_matrix

Value matrix

Returns a matrix of pairwise differences in absolute values. The matrix will have the shape (m, m) with m = len(Variogram.values). Note that Variogram.values holds the values themselves, while the value_matrix consists of their pairwise differences.

Returns **values** – Matrix of pairwise absolute differences of the values.

Return type numpy.matrix

See also:

Variogram._diff

property values

Values property

Array of observations, the variogram is build for. The setter of this property utilizes the Variogram.set_values function for setting new arrays.

Returns **values**

Return type numpy.ndarray

See also:

Variogram.set_values

2.6.2 DirectionalVariogram Class

class skgstat.**DirectionalVariogram**(*coordinates=None*, *values=None*, *estimator='matheron'*, *model='spherical'*, *dist_func='euclidean'*, *bin_func='even'*, *normalize=False*, *fit_method='trf'*, *fit_sigma=None*, *directional_model='triangle'*, *azimuth=0*, *tolerance=45.0*, *bandwidth='q33'*, *use_nugget=False*, *maxlag=None*, *n_lags=10*, *verbose=False*, ***kwargs*)

DirectionalVariogram Class

Calculates a variogram of the separating distances in the given coordinates and relates them to one of the semi-variance measures of the given dependent values.

The direcitonal version of a Variogram will only form paris of points that share a specified spatial relationship.

__init__(*coordinates=None*, *values=None*, *estimator='matheron'*, *model='spherical'*, *dist_func='euclidean'*, *bin_func='even'*, *normalize=False*, *fit_method='trf'*, *fit_sigma=None*, *directional_model='triangle'*, *azimuth=0*, *tolerance=45.0*, *bandwidth='q33'*, *use_nugget=False*, *maxlag=None*, *n_lags=10*, *verbose=False*, ***kwargs*)

Variogram Class

Directional Variogram. The calculation is not performant and not tested yet.

Parameters

- **coordinates** (*numpy.ndarray*) – Array of shape (m, n). Will be used as m observation points of n-dimensions. This variogram can be calculated on 1 - n dimensional coordinates. In case a 1-dimensional array is passed, a second array of same length containing only zeros will be stacked to the passed one.
- **values** (*numpy.ndarray*) – Array of values observed at the given coordinates. The length of the values array has to match the m dimension of the coordinates array. Will be used to calculate the dependent variable of the variogram.
- **estimator** (*str, callable*) – String identifying the semi-variance estimator to be used. Defaults to the Matheron estimator. Possible values are:
  - matheron [Matheron, default]
  - cressie [Cressie-Hawkins]
  - dowd [Dowd-Estimator]
  - genton [Genton]
  - minmax [MinMax Scaler]
  - entropy [Shannon Entropy]

  If a callable is passed, it has to accept an array of absoulte differences, aligned to the 1D distance matrix (flattened upper triangle) and return a scalar, that converges towards small values for similarity (high covariance).
- **model** (*str*) – String identifying the theoretical variogram function to be used to describe the experimental variogram. Can be one of:
  - spherical [Spherical, default]
  - exponential [Exponential]
  - gaussian [Gaussian]
  - cubic [Cubic]

  - stable [Stable model]
  - matern [Matérn model]
  - nugget [nugget effect variogram]

- **dist_func** (*str*) – String identifying the distance function. Defaults to 'euclidean'. Can be any metric accepted by scipy.spatial.distance.pdist. Additional parameters are not (yet) passed through to pdist. These are accepted by pdist for some of the metrics. In these cases the default values are used.
- **bin_func** (*str*) – Changed in version 0.3.8: added 'fd', 'sturges', 'scott', 'sqrt', 'doane'

  String identifying the binning function used to find lag class edges. All methods calculate bin edges on the interval [0, maxlag[. Possible values are:

  - *'even'* (default) finds *n_lags* same width bins
  - *'uniform'* forms *n_lags* bins of same data count
  - *'fd'* applies Freedman-Diaconis estimator to find *n_lags*
  - *'sturges'* applies Sturge's rule to find *n_lags*.
  - *'scott'* applies Scott's rule to find *n_lags*
  - *'doane'* applies Doane's extension to Sturge's rule to find *n_lags*
  - *'sqrt'* uses the square-root of `distance`. as *n_lags*.

  More details are given in the documentation for *set_bin_func*.
- **normalize** (*bool*) – Defaults to False. If True, the independent and dependent variable will be normalized to the range [0,1].
- **fit_method** (*str*) – String identifying the method to be used for fitting the theoretical variogram function to the experimental. More info is given in the Variogram.fit docs. Can be one of:

  - 'lm': Levenberg-Marquardt algorithm for unconstrained problems. This is the faster algorithm, yet is the fitting of a variogram not unconstrianed.
  - 'trf': Trust Region Reflective function for non-linear constrained problems. The class will set the boundaries itself. This is the default function.
- **fit_sigma** (*numpy.ndarray, str*) – Defaults to None. The sigma is used as measure of uncertainty during variogram fit. If fit_sigma is an array, it has to hold n_lags elements, giving the uncertainty for all lags classes. If fit_sigma is None (default), it will give no weight to any lag. Higher values indicate higher uncertainty and will lower the influcence of the corresponding lag class for the fit. If fit_sigma is a string, a pre-defined function of separating distance will be used to fill the array. Can be one of:

  - 'linear': Linear loss with distance. Small bins will have higher impact.
  - 'exp': The weights decrease by a e-function of distance
  - 'sqrt': The weights decrease by the squareroot of distance
  - 'sq': The weights decrease by the squared distance.

  More info is given in the Variogram.fit_sigma documentation.
- **directional_model** (*string, function*) – The model used for selecting all points fulfilling the directional constraint of the Variogram. A predefined model can be selected by passing the model name as string. Optionally a callable accepting the difference vectors between points in polar form as angles and distances and returning a mask array can be

passed. In this case, the azimuth, tolerance and bandwidth has to be incorporated by hand into the model.

- 'compass': includes points in the direction of the azimuth at given tolerance. The bandwidth parameter will be ignored.
- 'triangle': constructs a triangle with an angle of tolerance at the point of interest and union an rectangle parallel to azimuth, once the hypotenuse length reaches bandwidth.
- 'circle': constructs a half circle touching the point of interest, dislocating the center at the distance of bandwidth in the direction of azimuth. The half circle is union with an rectangle parallel to azimuth.

Visual representations, usage hints and implementation specifics are given in the documentation.

- **azimuth** (*float*) – The azimuth of the directional dependence for this Variogram, given as an angle in **degree**. The East of the coordinate plane is set to be at 0° and is counted clockwise to 180° and counter-clockwise to -180°. Only Points lying in the azimuth of a specific point will be used for forming point pairs.
- **tolerance** (*float*) – The tolerance is given as an angle in **degree**- Points being dislocated from the exact azimuth by half the tolerance will be accepted as well. It's half the tolerance as the point may be dislocated in the positive and negative direction from the azimuth.
- **bandwidth** (*float*) – Maximum tolerance acceptable in **coordinate units**, which is usually meter. Points at higher distances may be far dislocated from the azimuth in terms of coordinate distance, as the tolerance is defined as an angle. THe bandwidth defines a maximum width for the search window. It will be perpendicular to and bisected by the azimuth.
- **use_nugget** (*bool*) – Defaults to False. If True, a nugget effet will be added to all Variogram.models as a third (or fourth) fitting parameter. A nugget is essentially the y-axis interception of the theoretical variogram function.
- **maxlag** (*float, str*) – Can specify the maximum lag distance directly by giving a value larger than 1. The binning function will not find any lag class with an edge larger than maxlag. If 0 < maxlag < 1, then maxlag is relative and maxlag * max(Variogram.distance) will be used. In case maxlag is a string it has to be one of 'median', 'mean'. Then the median or mean of all Variogram.distance will be used. Note maxlag=0.5 will use half the maximum separating distance, this is not the same as 'median', which is the median of all separating distances
- **n_lags** (*int*) – Specify the number of lag classes to be defined by the binning function.
- **verbose** (*bool*) – Set the Verbosity of the class. Not Implemented yet.

Keyword Arguments

- **entropy_bins** (*int, str*) – New in version 0.3.7.

  If the *estimator <skgstat.Variogram.estimator>* is set to *'entropy'* this argument sets the number of bins, that should be used for histogram calculation.

- **percentile** (*int*) – New in version 0.3.7.

  If the *estimator <skgstat.Variogram.estimator>* is set to *'entropy'* this argument sets the percentile to be used.

_calc_direction_mask_data(*force=False*)

Calculate directional mask data. For this, the angle between the vector between the two points, and east (see comment about self.azimuth) is calculated. The result is stored in self._angles and contains the angle

of each point pair vector to the x-axis in radians.

Parameters **force** (*bool*) – If True, a new calculation of all angles is forced, even if they are already in the cache.

Notes

The masked data is in radias, while azimuth is given in degrees. For the Vector between a point pair A,B $\overrightarrow{AB} = u$ and the x-axis, represented by vector $\overrightarrow{e} = [1, 0]$, the angle Θ is calculated like:

$$cos(\Theta) = \frac{u \circ e}{|e| \cdot |[1, 0]|}$$

See also:

azimuth

_triangle(*angles*, *dists*)

Triangular Search Area

Construct a triangular bounded search area for building directional dependent point pairs. The Search Area will be located onto the current point of interest and the local x-axis is rotated onto the azimuth angle.

Parameters

- **angles** (*numpy.array*) – Vectors between point pairs in polar form (angle relative to east in radians, length in coordinate space units)
- **dists** (*numpy.array*) – Vectors between point pairs in polar form (angle relative to east in radians, length in coordinate space units)

Returns **mask** – Point pair mask, indexed as the results of scipy.spatial.distance.pdist are.

Return type numpy.array(bool)

Notes

```
    C
   /|\
a / | \ a
 /__|h_\
A   c   B
```

The point of interest is C and c is the bandwidth. The angle at C (gamma) is the tolerance. From this, a and then h can be calculated. When rotated into the local coordinate system, the two points needed to build the search area A,B are A := (h, 1/2 c) and B:= (h, -1/2 c)

a can be calculated like:

$$a = \frac{c}{2 * sin\left(\frac{\gamma}{2}\right)}$$

See also:

DirectionalVariogram._compass, DirectionalVariogram._circle

_compass(*angles*, *dists*)
Compass direction direction mask

Construct a search area for building directional dependent point pairs. The compass search area will **not** be bounded by the bandwidth. It will include all point pairs at the azimuth direction with a given tolerance. The Search Area will be located onto the current point of interest and the local x-axis is rotated onto the azimuth angle.

Parameters

- **angles** (*numpy.array*) – Vectors between point pairs in polar form (angle relative to east in radians, length in coordinate space units)
- **dists** (*numpy.array*) – Vectors between point pairs in polar form (angle relative to east in radians, length in coordinate space units)

Returns **mask** – Point pair mask, indexed as the results of scipy.spatial.distance.pdist are.

Return type numpy.array(bool)

See also:

DirectionalVariogram._triangle, DirectionalVariogram._circle

_direction_mask(*force=False*)
Directional Mask

Array aligned to self.distance masking all point pairs which shall be ignored for binning and grouping. The one dimensional array contains all row-wise point pair combinations from the upper or lower triangle of the distance matrix in case either of both is directional.

Returns **mask** – Array aligned to self.distance giving for each point pair combination a boolean value whether the point are directional or not.

Return type numpy.array

property azimuth
Direction azimuth

Main direction for te selection of points in the formation of point pairs. East of the coordinate plane is defined to be 0° and then the azimuth is set clockwise up to 180°and count-clockwise to -180°.

Parameters **angle** (*float*) – New azimuth angle in **degree**.

:raises ValueError : in case angle < -180° or angle > 180:

property bandwidth
Tolerance bandwidth

New bandwidth parameter. As the tolerance from azimuth is given as an angle, point pairs at high distances can be far off the azimuth in coordinate distance. The bandwidth limits this distance and has the unnit of the coordinate system.

Parameters **width** (*float*) – Positive coordinate distance.

:raises ValueError : in case width is negative:

property bins
Distance lag bins

Independent variable of the the experimental variogram sample. The bins are the upper edges of all calculated distance lag classes. If you need bin centers, use *get_empirical*.

Returns **bins** – 1D array of the distance lag classes.

Return type numpy.ndarray

See also:

Variogram.get_empirical

pair_field(*ax=None*, *cmap='gist_rainbow'*, *points='all'*, *add_points=True*, *alpha=0.3*, ***kwargs*)
Plot a pair field.

Plot a network graph for all point pairs that fulfill the direction filter and lie within each others search area.

Parameters

- **ax** (*matplotlib.Subplot*) – A matplotlib Axes object to plot the pair field onto. If `None`, a new new matplotlib figure will be created.
- **cmap** (*string*) – Any color-map name that is supported by matplotlib
- **points** (*'all', int, list*) – If not `'all'`, only the given coordinate (int) or list of coordinates (list) will be plotted. Recommended, if the input data is quite large.
- **add_points** (*bool*) – If True (default) The coordinates will be added as black points.
- **alpha** (*float*) – Alpha value for the colors to make overlapping vertices visualize better. Defaults to `0.3`.

preprocessing(*force=False*)
Preprocessing function

Prepares all input data for the fit and transform functions. Namely, the distances are calculated and the value differences. Then the binning is set up and bin edges are calculated. If any of the listed subsets are already prepared, their processing is skipped. This behaviour can be changed by the force parameter. This will cause a clean preprocessing.

Parameters **force** (*bool*) – If set to True, all preprocessing data sets will be deleted. Use it in case you need a clean preprocessing.

Returns

Return type void

set_directional_model(*model_name*)
Set new directional model

The model used for selecting all points fulfilling the directional constraint of the Variogram. A predefined model can be selected by passing the model name as string. Optionally a callable accepting the difference vectors between points in polar form as angles and distances and returning a mask array can be passed. In this case, the azimuth, tolerance and bandwidth has to be incorporated by hand into the model. The predefined options are:

- 'compass': includes points in the direction of the azimuth at given tolerance. The bandwidth parameter will be ignored.
- 'triangle': constructs a triangle with an angle of tolerance at the point of interest and union an rectangle parallel to azimuth, once the hypotenuse length reaches bandwidth.
- 'circle': constructs a half circle touching the point of interest, dislocating the center at the distance of bandwidth in the direction of azimuth. The half circle is union with an rectangle parallel to azimuth.

Visual representations, usage hints and implementation specifics are given in the documentation.

Parameters **model_name** (*string, callable*) – The name of the predefined model (string) or a function that accepts angle and distance arrays and returns a mask array.

to_gstools(**args*, ***kwargs*)
Instantiate a corresponding GSTools CovModel.

By default, this will be an isotropic model.

Parameters ****kwargs** – Keyword arguments forwarded to the instantiated GSTools CovModel. The default parameters 'dim', 'var', 'len_scale', 'nugget', 'rescale' and optional shape parameters will be extracted from the given Variogram but they can be overwritten here.

Raises

- **ImportError** – When GSTools is not installed.
- **ValueError** – When GSTools version is not v1.3 or greater.
- **ValueError** – When given Variogram model is not supported ('harmonize').

Returns Corresponding GSTools covmodel.

Return type CovModel

Note: In case you intend to use the *coordinates* in a GSTools workflow, you need to transpose the coordinate array like:

>> cond_pos Variogram.coordinates.T

---

property tolerance

Azimuth tolerance

Tolerance angle of how far a point can be off the azimuth for being still counted as directional. A tolerance angle will be applied to the azimuth angle symmetrically.

Parameters **angle** (*float*) – New tolerance angle in **degree**. Has to meet 0 <= angle <= 360.

:raises ValueError : in case angle < 0 or angle > 360:

2.6.3 SpaceTimeVariogram class

class skgstat.**SpaceTimeVariogram**(*coordinates*, *values*, *xdist_func='euclidean'*, *tdist_func='euclidean'*, *x_lags=10*, *t_lags='max'*, *maxlag=None*, *xbins='even'*, *tbins='even'*, *estimator='matheron'*, *use_nugget=False*, *model='product-sum'*, *verbose=False*)

__init__(*coordinates*, *values*, *xdist_func='euclidean'*, *tdist_func='euclidean'*, *x_lags=10*, *t_lags='max'*, *maxlag=None*, *xbins='even'*, *tbins='even'*, *estimator='matheron'*, *use_nugget=False*, *model='product-sum'*, *verbose=False*)

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

contour(*ax=None*, *zoom_factor=100.0*, *levels=10*, *colors='k'*, *linewidths=0.3*, *method='fast'*, ***kwargs*)

Variogram 2D contour plot

Plot a 2D contour plot of the experimental variogram. The experimental semi-variance values are spanned over a space - time lag meshgrid. This grid is (linear) interpolated onto the given resolution for visual reasons. Then, contour lines are caluclated from the denser grid. Their number can be specified by *levels*.

Parameters

- **ax** (*matplotlib.AxesSubplot, None*) – If None a new matplotlib.Figure will be created, otherwise the plot will be rendered into the given subplot.
- **zoom_factor** (*float*) – The experimental variogram will be interpolated onto a regular grid for visual reasons. The density of this plot can be set by zoom_factor. A factor of 10

will enlarge each of the axes by 10. Higher zoom_factors result in smoother contours, but are expansive in calculation time.

- **levels** (*int*) – Number of levels to be formed for finding contour lines. More levels result in more detailed plots, but are expansive in terms of calculation time.
- **colors** (*str, list*) – Will be passed down to matplotlib.pyplot.contour as *c* parameter.
- **linewidths** (*float, list*) – Will be passed down to matplotlib.pyplot.contour as *linewidths* parameter.
- **method** (*str*) – The method used for densifying the meshgrid. Can be one of 'fast' or 'precise'. Fast will use the scipy.ndimage.zoom method to incresae the node density. This is fast, but cannot interpolate *behind* any NaN occurance. 'Precise' performs an actual linear interpolation between the nodes using scipy.interpolate.griddata. This takes more time, but the result is less smoothed out.
- **kwargs** (*dict*) – Other arguments that can be specific to *contour* or *contourf* type. Accepts *xlabel*, *ylabel*, *xlim* and *ylim* as of this writing.

Returns **fig** – The Figure object used for rendering the contour plot.

Return type matplotlib.Figure

See also:

SpaceTimeVariogram.contourf

contourf(*ax=None*, *zoom_factor=100.0*, *levels=10*, *cmap='RdYlBu_r'*, *method='fast'*, ***kwargs*)
Variogram 2D filled contour plot

Plot a 2D filled contour plot of the experimental variogram. The experimental semi-variance values are spanned over a space - time lag meshgrid. This grid is (linear) interpolated onto the given resolution for visual reasons. Then, contour lines are caluclated from the denser grid. Their number can be specified by *levels*. Finally, each contour region is filled with a color supplied by the specified *cmap*.

Parameters

- **ax** (*matplotlib.AxesSubplot, None*) – If None a new matplotlib.Figure will be created, otherwise the plot will be rendered into the given subplot.
- **zoom_factor** (*float*) – The experimental variogram will be interpolated onto a regular grid for visual reasons. The density of this plot can be set by zoom_factor. A factor of 10 will enlarge each of the axes by 10. Higher zoom_factors result in smoother contours, but are expansive in calculation time.
- **levels** (*int*) – Number of levels to be formed for finding contour lines. More levels result in more detailed plots, but are expansive in terms of calculation time.
- **cmap** (*str*) – Will be passed down to matplotlib.pyplot.contourf as *cmap* parameter. Can be any valid color range supported by matplotlib.
- **method** (*str*) – The method used for densifying the meshgrid. Can be one of 'fast' or 'precise'. Fast will use the scipy.ndimage.zoom method to incresae the node density. This is fast, but cannot interpolate *behind* any NaN occurance. 'Precise' performs an actual linear interpolation between the nodes using scipy.interpolate.griddata. This takes more time, but the result is less smoothed out.
- **kwargs** (*dict*) – Other arguments that can be specific to *contour* or *contourf* type. Accepts *xlabel*, *ylabel*, *xlim* and *ylim* as of this writing.

Returns **fig** – The Figure object used for rendering the contour plot.

Return type matplotlib.Figure

See also:

SpaceTimeVariogram.contour

create_TMarginal()
Create an instance of skgstat.Variogram for the time marginal variogram by arranging the coordinates and values and infer parameters from this SpaceTimeVariogram instance.

create_XMarginal()
Create an instance of skgstat.Variogram for the space marginal variogram by arranging the coordinates and values and infer parameters from this SpaceTimeVariogram instance.

property distance
Distance matrices

Returns both the space and time distance matrix. This property is equivalent to two separate calls of *xdistance* and *tdistance*.

Returns distance matrices – Returns a tuple of the two distance matrices in space and time. Each distance matrix is a flattened upper triangle of the distance matrix squareform in row orientation.

Return type (numpy.array, numpy.array)

property experimental
Experimental Variogram

Returns an experimental variogram for the given data. The semivariances are arranged over the spatial binning as defined in SpaceTimeVariogram.xbins and temporal binning defined in SpaceTimeVariogram.tbins.

Returns variogram – Returns an two dimensional array of semivariances over space on the first axis and time over the second axis.

Return type numpy.ndarray

property fitted_model

get_marginal(*axis*, *lag=0*)
Marginal Variogram

Returns the marginal experimental variogram of axis for the given lag on the other axis. Axis can either be 'space' or 'time'. The parameter lag specifies the index of the desired lag class on the other axis.

Parameters

- **axis** (*str*) – The axis a marginal variogram shall be calculated for. Can either be ' space' or 'time'.
- **lag** (*int*) – Index of the lag class group on the other axis to be used. In case this is 0, this is often considered to be *the* marginal variogram of the axis.

Returns variogram – Marginal variogram of the given axis

Return type numpy.array

lag_classes()
Iterator over all lag classes

Returns an iterator over all lag classes by aligning all time lags over all space lags. This means that it will yield all time lag groups for a space lag of index 0 at first and then iterate the space lags.

Returns

Return type iterator

`lag_groups`(*axis*)
Lag class group mask array

Returns a mask array for the given axis (either 'space' or 'time'). It will have as amany elements as the respective distance matrices. **Unlike the base Variogram class, it does not mask the array of pairwise differences.**. It will mask the distance matrix of the respective axis.

Parameters **axis** (*str*) – Can either be 'space' or 'time'. Specifies the axis the mask array shall be returned for.

Returns **masK_array** – mask array that identifies the lag class group index for each pair of points on the given axis.

Return type numpy.array

`marginals`(*plot=True*, *axes=None*, *sharey=True*, *include_model=False*, ***kwargs*)
Plot marginal variograms

Plots the two marginal variograms into a new or existing figure. The space marginal variogram is defined to be the variogram of temporal lag class 0, while the time marginal variogram uses only spatial lag class 0. In case the expected variability is not of same magnitude, the sharey parameter should be set to `False` in order to use separated y-axes.

Parameters

- **plot** (*bool*) – Deprecated since version 0.4: With version 0.4, this parameter will be removed

  If set to False, no matplotlib.Figure will be returned. Instead a tuple of the two marginal experimental variogram values is returned.
- **axes** (*list*) – Is either `None` to create a new matplotlib.Figure. Otherwise it has to be a list of two matplotlib.AxesSubplot instances, which will then be used for plotting.
- **sharey** (*bool*) – If True (default), the two marginal variograms will share their y-axis to increase comparability. Should be set to False in the variances are of different magnitude.
- **include_model** (*bool*) – If True, the marginal variogram models fitted to the respective axis are included into the plot.
- **kwargs** (*dict*) – Only kwargs accepted is `figsize`, if ax is None. Anything else will be ignored.

Returns

- **variograms** (*tuple*) – If plot is False, a tuple of numpy.arrays are returned. These are the two experimental marginal variograms.
- **plots** (*matplotlib.Figure*) – If plot is True, the matplotlib.Figure will be returned.

`plot`(*kind='scatter'*, *ax=None*, ***kwargs*)
Plot the experimental variogram

At the current version the SpaceTimeVariogram class is not capable of modeling a spe-time variogram function, therefore all plots will only show the experimental variogram. As the experimental space-time semivariance is depending on a space and a time lag, one would basically need a 3D scatter plot, which is the default plot. However, 3D plots can be, especially for scientific usage, a bit problematic. Therefore the plot function can plot a variety of 3D and 2D plots.

Parameters

- **kind** (*str*) – Has to be one of:
  - **scatter**

- **surface**
- **contour**
- **contourf**
- **matrix**
- **marginals**

- **ax** (*matplotlib.AxesSubplot, mpl_toolkits.mplot3d.Axes3D, None*) – If None, the function will create a new figure and suitable Axes. Else, the Axes object can be passed to plot the variogram into an existing figure. In this case, one has to pass the correct type of Axes, whether it's a 3D or 2D kind of a plot.
- **kwargs** (*dict*) – All keyword arguments are passed down to the actual plotting function. Refer to their documentation for a more detailed description.

Returns **fig**

Return type matplotlib.Figure

See also:

SpaceTimeVariogram.scatter, *SpaceTimeVariogram.surface*, *SpaceTimeVariogram.marginals*

preprocessing(*force=False*)
Preprocessing

Start all necessary calculation jobs needed to derive an experimental variogram. This hasto be present before the model fitting can be done. The force parameter will make all calculation functions to delete all cached intermediate results and make a clean calculation.

Parameters **force** (*bool*) – If True, all cached intermediate results will be deleted and a clean calculation will be done.

scatter(*ax=None*, *elev=30*, *azim=220*, *c='blue'*, *depthshade=True*, ***kwargs*)
3D Scatter Variogram

Plot the experimental variogram into a 3D matplotlib.Figure. The two variogram axis (space, time) will span a meshgrid over the x and y axis and the semivariance will be plotted as z value over the respective space and time lag coordinate.

Parameters

- **ax** (*mpl_toolkits.mplot3d.Axes3D, None*) – If ax is None (default), a new Figure and Axes instance will be created. If ax is given, this instance will be used for the plot.
- **elev** (*int*) – The elevation of the 3D plot, which is a rotation over the xy-plane.
- **azim** (*int*) – The azimuth of the 3D plot, which is a rotation over the z-axis.
- **c** (*str*) – Color of the scatter points, will be passed to the matplotlib `c` argument. The function also accepts `color` as an alias.
- **depthshade** (*bool*) – If True, the scatter points will change their color according to the distance from the viewport for illustration reasons.
- **kwargs** (*dict*) – Other kwargs accepted are only `color` as an alias for `c` and `figsize`, if ax is None. Anything else will be ignored.

Returns **fig**

Return type matplotlib.Figure

Examples

In case an ax shall be passed to the function, note that this plot requires an AxesSubplot, that is capable of creating a 3D plot. This can be done like:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# STV is an instance of SpaceTimeVariogram
STV.scatter(ax=ax)
```

See also:

SpaceTimeVariogram.surface

set_bin_func(*bin_func*, *axis*)
Set binning function

Set a new binning function to either the space or time axis. Both axes support the methods: [‘even’, ‘uniform’]:

- **‘even’**, create even width bins
- **‘uniform’**, create bins of uniform distribution

Parameters

- **bin_func** (*str*) – Sepcifies the function to be loaded. Can be either ‘even’ or ‘uniform’.
- **axis** (*str*) – Specifies the axis to be used for binning. Can be either ‘space’ or ‘time’, or one of the two shortcuts ‘s’ and ‘t’

See also:

skgstat.binning.even_width_lags, *skgstat.binning.uniform_count_lags*

set_model(*model_name*)
Set space-time model

Set a new space-time model. It has to be either a callable of correct signature or a string identifying one of the predefined models

Parameters **model_name** (*str, callable*) – Either a callable of correct signature or a valid model name. Valid names are:

- sum
- product
- product-sum

set_tdist_func(*func_name*)
Set new space distance function

Set a new function for calculating the distance matrix in the space dimension. At the moment only strings are supported. Will be passed to scipy.spatical.distance.pdist as ‘metric’ attribute.

Parameters **func_name** (*str*) – The name of the function used to calculate the pairwise distances. Will be passed to scipy.spatial.distance.pdist as the ‘metric’ attribute.

:raises ValueError : in case a non-string argument is passed.:

set_values(*values*)
Set new values

The values should be an (m, n) array with m matching the size of coordinates first dimension and n is the time dimension.

:raises ValueError : in case n <= 1 or values are not an array of correct: dimensionality :raises AttributeError : in case values cannot be converted to a numpy.array:

set_xdist_func(*func_name*)
Set new space distance function

Set a new function for calculating the distance matrix in the space dimension. At the moment only strings are supported. Will be passed to scipy.spatical.distance.pdist as 'metric' attribute.

Parameters **func_name** (*str*) – The name of the function used to calculate the pairwise distances. Will be passed to scipy.spatial.distance.pdist as the 'metric' attribute.

:raises ValueError : in case a non-string argument is passed.:

surface(*ax=None*, *elev=30*, *azim=220*, *color='blue'*, *alpha=0.5*, ***kwargs*)
3D Scatter Variogram

Plot the experimental variogram into a 3D matplotlib.Figure. The two variogram axis (space, time) will span a meshgrid over the x and y axis and the semivariance will be plotted as z value over the respective space and time lag coordinate. Unlike *scatter* the semivariance will not be scattered as points but rather as a surface plot. The surface is approximated by (Delauney) triangulation of the z-axis.

Parameters

- **ax** (*mpl_toolkits.mplot3d.Axes3D, None*) – If ax is None (default), a new Figure and Axes instance will be created. If ax is given, this instance will be used for the plot.
- **elev** (*int*) – The elevation of the 3D plot, which is a rotation over the xy-plane.
- **azim** (*int*) – The azimuth of the 3D plot, which is a rotation over the z-axis.
- **color** (*str*) – Color of the scatter points, will be passed to the matplotlib `color` argument. The function also accepts `c` as an alias.
- **alpha** (*float*) – Sets the transparency of the surface as 0 <= alpha <= 1, with 0 being completely transparent.
- **kwargs** (*dict*) – Other kwargs accepted are only `color` as an alias for `c` and `figsize`, if ax is None. Anything else will be ignored.

Returns **fig**

Return type matplotlib.Figure

Notes

In case an ax shall be passed to the function, note that this plot requires an AxesSubplot, that is capable of creating a 3D plot. This can be done like:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

(continues on next page)

(continued from previous page)

```
# STV is an instance of SpaceTimeVariogram
STV.surface(ax=ax)
```

See also:

SpaceTimeVariogram.scatter

property tbins
Temporal binning

Returns the bin edges over the temporal axis. These can be used to align the temporal lag class grouping to actual time lags. The length of the array matches the number of temporal lag classes.

Returns **bins** – Returns the edges of the current temporal binning.

Return type numpy.array

property tdistance
Time distance

Returns a distance matrix containing the distance of all observation points in time. The time 'coordiantes' are created from the values multidimensional array, where the second dimension is assumed to be time. The unit will be time steps.

Returns **tdistance** – 1D-array of the upper triangle of a squareform representation of the distance matrix.

Return type numpy.array

property values
Values

The SpaceTimeVariogram stores (and needs) the observations as a two dimensional array. The first axis (rows) need to match the coordinate array, but instead of containing one value for each location, the values shall contain a time series per location.

Returns **values** – Returns a two dimensional array of all observations. The first dimension (rows) matches the coordinate array and the second axis contains the time series for each observation point.

Return type numpy.array

property xbins
Spatial binning

Returns the bin edges over the spatial axis. These can be used to align the spatial lag class grouping to actual distance lags. The length of the array matches the number of spatial lag classes.

Returns **bins** – Returns the edges of the current spatial binning.

Return type numpy.array

property xdistance
Distance matrix (space)

Return the upper triangle of the squareform pairwise distance matrix.

Returns **xdistance** – 1D-array of the upper triangle of a squareform representation of the distance matrix.

Return type numpy.array

2.6.4 Binning functions

SciKit-GStat implements a large amount of binning functions, which can be used to spatially aggregate the distance matrix into lag classes, or bins. There are a number of functions available, which usually accept more than one method identifier:

skgstat.binning.**even_width_lags**(*distances*, *n*, *maxlag*)
Even lag edges

Calculate the lag edges for a given amount of bins using the same lag step width for all bins.

Changed in version 0.3.8: Function returns *None* as second value to indicate that The number of lag classes was not changed

Parameters

- **distances** (*numpy.array*) – Flat numpy array representing the upper triangle of the distance matrix.
- **n** (*integer*) – Amount of lag classes to find
- **maxlag** (*integer, float*) – Limit the last lag class to this separating distance.

Returns **bin_edges** – The **upper** bin edges of the lag classes

Return type numpy.ndarray

skgstat.binning.**uniform_count_lags**(*distances*, *n*, *maxlag*)
Uniform lag counts

Calculate the lag edges for a given amount of bins with the same amount of observations in each lag class. The lag step width will be variable.

Changed in version 0.3.8: Function returns *None* as second value to indicate that The number of lag classes was not changed

Parameters

- **distances** (*numpy.array*) – Flat numpy array representing the upper triangle of the distance matrix.
- **n** (*integer*) – Amount of lag classes to find
- **maxlag** (*integer, float*) – Limit the last lag class to this separating distance.

Returns **bin_edges** – The **upper** bin edges of the lag classes

Return type numpy.ndarray

skgstat.binning.**auto_derived_lags**(*distances*, *method_name*, *maxlag*)
Derive bins automatically .. versionadded:: 0.3.8

Uses *histogram_bin_edges <numpy.histogram_bin_edges>* to derive the lag classes automatically. Supports any method supported by *histogram_bin_edges <numpy.histogram_bin_edges>*. It is recommended to use *'sturges'*, *'doane'* or *'fd'*.

Parameters

- **distances** (*numpy.array*) – Flat numpy array representing the upper triangle of the distance matrix.
- **maxlag** (*integer, float*) – Limit the last lag class to this separating distance.
- **method_name** (*str*) – Any method supported by *histogram_bin_edges <numpy.histogram_bin_edges>*

Returns **bin_edges** – The **upper** bin edges of the lag classes

Return type numpy.ndarray

See also:

numpy.histogram_bin_edges

skgstat.binning.**kmeans**(*distances*, *n*, *maxlag*, *binning_random_state=42*, ***kwargs*)

New in version 0.3.9.

Clustering of pairwise separating distances between locations up to maxlag. The lag class edges are formed equidistant from each cluster center. Note: this does not necessarily result in equidistance lag classes.

Parameters

- **distances** (*numpy.array*) – Flat numpy array representing the upper triangle of the distance matrix.
- **n** (*integer*) – Amount of lag classes to find
- **maxlag** (*integer, float*) – Limit the last lag class to this separating distance.

Returns **bin_edges** – The **upper** bin edges of the lag classes

Return type numpy.ndarray

See also:

sklearn.cluster.KMeans

Note: The KMeans that is used under the hood is not a deterministic algorithm, as the starting cluster centroids are seeded randomly. This can yield slightly different results on reach run. Thus, for this application, the random_state on KMeans is fixed to a specific value. You can change the seed by passing another seed to *Variogram* as *binning_random_state*.

---

skgstat.binning.**ward**(*distances*, *n*, *maxlag*, ***kwargs*)

New in version 0.3.9.

Clustering of pairwise separating distances between locations up to maxlag. The lag class edges are formed equidistant from each cluster center. Note: this does not necessarily result in equidistance lag classes.

The clustering is done by merging pairs of clusters that minimize the variance for the merged clusters, unitl *n* clusters are found.

Parameters

- **distances** (*numpy.array*) – Flat numpy array representing the upper triangle of the distance matrix.
- **n** (*integer*) – Amount of lag classes to find
- **maxlag** (*integer, float*) – Limit the last lag class to this separating distance.

Returns **bin_edges** – The **upper** bin edges of the lag classes

Return type numpy.ndarray

See also:

sklearn.clsuter.AgglomerativeClustering

skgstat.binning.**stable_entropy_lags**(*distances*, *n*, *maxlag*, ***kwargs*)

Optimizes the lag class edges for *n* lag classes. The algorithm minimizes the difference between Shannon Entropy for each lag class. Consequently, the final lag classes should be of comparable uncertainty.

Parameters

- **distances** (*numpy.array*) – Flat numpy array representing the upper triangle of the distance matrix.
- **n** (*integer*) – Amount of lag classes to find
- **maxlag** (*integer, float*) – Limit the last lag class to this separating distance.

Keyword Arguments

- **binning_maxiter** (*int*) – Maximum iterations before the optimization is stopped, if the lag edges do not converge.
- **binning_entropy_bins** (*int, str*) – Binning method for calculating the shannon entropy on each iteration.

Returns **bin_edges** – The **upper** bin edges of the lag classes

Return type numpy.ndarray

2.6.5 Estimator Functions

Scikit-GStat implements various semi-variance estimators. These fucntions can be found in the skgstat.estimators submodule. Each of these functions can be used independently from Variogram class. In this case the estimator is expecting an array of pairwise differences to calculate the semi-variance. Not the values themselves.

Matheron

skgstat.estimators.**matheron**(*x*)

Matheron Semi-Variance

Calculates the Matheron Semi-Variance from an array of pairwise differences. Returns the semi-variance for the whole array. In case a semi-variance is needed for multiple groups, this function has to be mapped on each group. That is the typical use case in geostatistics.

Parameters **x** (*numpy.ndarray*) – Array of pairwise differences. These values should be the distances between pairwise observations in value space. If xi and x[i+h] fall into the h separating distance class, x should contain abs(xi - x[i+h]) as an element.

Returns

Return type numpy.float64

Notes

This implementation follows the original publication[1] and the notes on their application[2]. Following the 1962 publication[?], the semi-variance is calculated as:

$$\gamma(h) = \frac{1}{2N(h)} * \sum_{i=1}^{N(h)} (x)^2$$

with:

$$x = Z(x_i) - Z(x_{i+h})$$

where x is exactly the input array x.

[1] Matheron, G. (1962): Traité de Géostatistique Appliqué, Tonne 1. Memoires de Bureau de Recherches Géologiques et Miniéres, Paris.
[2] Matheron, G. (1965): Les variables regionalisées et leur estimation. Editions Masson et Cie, 212 S., Paris.

References

Cressie

skgstat.estimators.**cressie**(x)

Cressie-Hawkins Semi-Variance

Calculates the Cressie-Hawkins Semi-Variance from an array of pairwise differences. Returns the semi-variance for the whole array. In case a semi-variance is needed for multiple groups, this function has to be mapped on each group. That is the typical use case in geostatistics.

Parameters **x** (*numpy.ndarray*) – Array of pairwise differences. These values should be the distances between pairwise observations in value space. If xi and x[i+h] fall into the h separating distance class, x should contain abs(xi - x[i+h]) as an element.

Returns

Return type numpy.float64

Notes

This implementation is done after the publication by Cressie and Hawkins from 1980[3]:

$$2\gamma(h) = \frac{(\frac{1}{N(h)} \sum_{i=1}^{N(h)} |x|^{0.5})^4}{0.457 + \frac{0.494}{N(h)} + \frac{0.045}{N^2(h)}}$$

with:

$$x = Z(x_i) - Z(x_{i+h})$$

where x is exactly the input array x.

References

Dowd

skgstat.estimators.**dowd**(x)

Dowd semi-variance

Calculates the Dowd semi-variance from an array of pairwise differences. Returns the semi-variance for the whole array. In case a semi-variance is needed for multiple groups, this function has to be mapped on each group. That is the typical use case in geostatistics.

Parameters **x** (*numpy.ndarray*) – Array of pairwise differences. These values should be the distances between pairwise observations in value space. If xi and x[i+h] fall into the h separating distance class, x should contain abs(xi - x[i+h]) as an element.

Returns

Return type numpy.float64

[3] Cressie, N., and D. Hawkins (1980): Robust estimation of the variogram. Math. Geol., 12, 115-125.

Notes

The Dowd estimator is based on the median of all pairwise differences in each lag class and is therefore robust to exteme values at the cost of variability. This implementation follows Dowd's publication[4]:

$$2\gamma(h) = 2.198 * median(x)^2$$

with:

$$x = Z(x_i) - Z(x_{i+h})$$

where x is exactly the input array x.

References

Genton

skgstat.estimators.**genton**(*x*)

Genton robust semi-variance estimator

Return the Genton semi-variance of the given sample x. Genton is a highly robust varigram estimator, that is designed to be location free and robust on extreme values in x. Genton is based on calculating kth order statistics and will for large data sets be close or equal to the 25% quartile of all ordered point pairs in X.

Parameters **x** (*numpy.ndarray*) – Array of pairwise differences. These values should be the distances between pairwise observations in value space. If xi and x[i+h] fall into the h separating distance class, x should contain abs(xi - x[i+h]) as an element.

Returns

Return type numpy.float64

Notes

The Genton estimator is described in great detail in the original publication[5] and is defined as:

$$Q_{N_h} = 2.2191\{|V_i(h) - V_j(h)|; i < j\}_{(k)}$$

and

$$k = \binom{[N_h/2] + 1}{2}$$

and

$$q = \binom{N_h}{2}$$

where k is the kth quantile of all q point pairs. For large N (k/q) will be close to 0.25. For N >= 500, (k/q) is close to 0.25 by two decimals and will therefore be set to 0.5 and the two binomial coefficients k, q are not calculated.

[4] Dowd, P. A., (1984): The variogram and kriging: Robust and resistant estimators, in Geostatistics for Natural Resources Characterization. Edited by G. Verly et al., pp. 91 - 106, D. Reidel, Dordrecht.

[5] Genton, M. G., (1998): Highly robust variogram estimation, Math. Geol., 30, 213 - 221.

References

Shannon Entropy

skgstat.estimators.**entropy**(*x*, *bins=None*)
Shannon Entropy estimator

Calculates the Shannon Entropy H as a variogram estimator. It is highly recommended to calculate the bins and explicitly set them as a list. In case this function is called for more than one lag class in a variogram, setting bins to None would result in different bin edges in each lag class. This would be very difficult to interpret.

Parameters

- **x** (*numpy.ndarray*) – Array of pairwise differences. These values should be the distances between pairwise observations in value space. If xi and x[i+h] fall into the h separating distance class, x should contain abs(xi - x[i+h]) as an element.
- **bins** (*int*, *list*, *str*) – list of the bin edges used to calculate the empirical distribution of x. If bins is a list, these values are used directly. In case bins is a integer, as many even width bins will be calculated between the minimum and maximum value of x. In case bins is a string, it will be passed as bins argument to numpy.histograms function.

Returns **entropy** – Shannon entropy of the given pairwise differences.

Return type numpy.float64

Notes

MinMax

Warning: This is an experimental semi-variance estimator. It is heavily influenced by extreme values and outliers. That behaviour is usually not desired in geostatistics.

skgstat.estimators.**minmax**(*x*)
Minimum - Maximum Estimator

Returns a custom value. This estimator is the difference of maximum and minimum pairwise differences, normalized by the mean. MinMax will be very sensitive to extreme values.

Do only use this estimator, in case you know what you are doing. It is experimental and might change its behaviour in a future version.

Parameters **x** (*numpy.ndarray*) – Array of pairwise differences. These values should be the distances between pairwise observations in value space. If xi and x[i+h] fall into the h separating distance class, x should contain abs(xi - x[i+h]) as an element.

Returns

Return type numpy.float64

Percentile

Warning: This is an experimental semi-variance estimator. It uses just a percentile of the given pairwise differences and does not bear any information about their variance.

skgstat.estimators.**percentile**(*x*, *p=50*)

Percentile estimator

Returns a given percentile as semi-variance. Do only use this estimator, in case you know what you are doing.

Do only use this estimator, in case you know what you are doing. It is experimental and might change its behaviour in a future version.

Parameters

- **x** (*numpy.ndarray*) – Array of pairwise differences. These values should be the distances between pairwise observations in value space. If xi and x[i+h] fall into the h separating distance class, x should contain abs(xi - x[i+h]) as an element.
- **p** (*int*) – Desired percentile. Should be given as whole numbers 0 < p < 100.

Returns

Return type np.float64

2.6.6 Variogram models

Scikit-GStat implements different theoretical variogram functions. These model functions expect a single lag value or an array of lag values as input data. Each function has at least a parameter *a* for the effective range and a parameter *c0* for the sill. The nugget parameter *b* is optinal and will be set to $b := 0$ if not given.

Spherical model

skgstat.models.**spherical**(*h*, *r*, *c0*, *b=0*)

Spherical Variogram function

Implementation of the spherical variogram function. Calculates the dependent variable for a given lag (h). The nugget (b) defaults to be 0.

Parameters

- **h** (*float*) – Specifies the lag of separating distances that the dependent variable shall be calculated for. It has to be a positive real number.
- **r** (*float*) – The effective range. Note this is not the range parameter! However, for the spherical variogram the range and effective range are the same.
- **c0** (*float*) – The sill of the variogram, where it will flatten out. The function will not return a value higher than C0 + b.
- **b** (*float*) – The nugget of the variogram. This is the value of independent variable at the distance of zero. This is usually attributed to non-spatial variance.

Returns **gamma** – Unlike in most variogram function formulas, which define the function for $2 * \gamma$, this function will return γ only.

Return type numpy.float64

Notes

The implementation follows[6]:

$$\gamma = b + C_0 * \left(1.5 * \frac{h}{a} - 0.5 * \frac{h}{a}^3\right)$$

if $h < r$, and

$$\gamma = b + C_0$$

else. r is the effective range, which is in case of the spherical variogram just a.

References

Exponential model

skgstat.models.**exponential**(*h*, *r*, *c0*, *b=0*)

Exponential Variogram function

Implementation of the exponential variogram function. Calculates the dependent variable for a given lag (h). The nugget (b) defaults to be 0.

Parameters

- **h** (*float*) – Specifies the lag of separating distances that the dependent variable shall be calculated for. It has to be a positive real number.
- **r** (*float*) – The effective range. Note this is not the range parameter! For the exponential variogram function the range parameter a is defined to be $a = \frac{r}{3}$. The effective range is the lag where 95% of the sill are exceeded. This is needed as the sill is only approached asymptotically by an exponential function.
- **c0** (*float*) – The sill of the variogram, where it will flatten out. The function will not return a value higher than C0 + b.
- **b** (*float*) – The nugget of the variogram. This is the value of independent variable at the distance of zero. This is usually attributed to non-spatial variance.

Returns **gamma** – Unlike in most variogram function formulas, which define the function for $2 * \gamma$, this function will return γ only.

Return type numpy.float64

Notes

The implementation following[7],[9] and[8] is as:

$$\gamma = b + C_0 * \left(1 - e^{-\frac{h}{a}}\right)$$

a is the range parameter, that can be calculated from the effective range r as: $a = \frac{r}{3}$.

[6] Burgess, T. M., & Webster, R. (1980). Optimal interpolation and isarithmic mapping of soil properties. I.The semi-variogram and punctual kriging. Journal of Soil and Science, 31(2), 315–331, http://doi.org/10.1111/j.1365-2389.1980.tb02084.x

[7] Cressie, N. (1993): Statistics for spatial data. Wiley Interscience.

[9] Journel, A G, and Huijbregts, C J. Mining geostatistics. United Kingdom: N. p., 1976.

[8] Chiles, J.P., Delfiner, P. (1999). Geostatistics. Modeling Spatial Uncertainty. Wiley Interscience.

References

Gaussian model

skgstat.models.**gaussian**(*h*, *r*, *c0*, *b=0*)
Gaussian Variogram function

Implementation of the Gaussian variogram function. Calculates the dependent variable for a given lag (h). The nugget (b) defaults to be 0.

Parameters

- **h** (*float*) – Specifies the lag of separating distances that the dependent variable shall be calculated for. It has to be a positive real number.
- **r** (*float*) – The effective range. Note this is not the range parameter! For the exponential variogram function the range parameter a is defined to be $a = \frac{r}{3}$. The effetive range is the lag where 95% of the sill are exceeded. This is needed as the sill is only approached asymptotically by an exponential function.
- **c0** (*float*) – The sill of the variogram, where it will flatten out. The function will not return a value higher than C0 + b.
- **b** (*float*) – The nugget of the variogram. This is the value of independent variable at the distance of zero. This is usually attributed to non-spatial variance.

Returns **gamma** – Unlike in most variogram function formulas, which define the function for $2 * \gamma$, this function will return γ only.

Return type numpy.float64

Notes

This implementation follows[10] and[11]:

$$\gamma = b + c_0 * \left(1 - e^{-\frac{h^2}{a^2}}\right)$$

a is the range parameter, that can be calculated from the effective range r as:

$$a = \frac{r}{2}$$

References

Cubic model

skgstat.models.**cubic**(*h*, *r*, *c0*, *b=0*)
Cubic Variogram function

Implementation of the Cubic variogram function. Calculates the dependent variable for a given lag (h). The nugget (b) defaults to be 0.

Parameters

- **h** (*float*) – Specifies the lag of separating distances that the dependent variable shall be calculated for. It has to be a positive real number.

[10] Chiles, J.P., Delfiner, P. (1999). Geostatistics. Modeling Spatial Uncertainty. Wiley Interscience.
[11] Journel, A G, and Huijbregts, C J. Mining geostatistics. United Kingdom: N. p., 1976.

- **r** (*float*) – The effective range. Note this is not the range parameter! However, for the cubic variogram the range and effective range are the same.
- **c0** (*float*) – The sill of the variogram, where it will flatten out. The function will not return a value higher than C0 + b.
- **b** (*float*) – The nugget of the variogram. This is the value of independent variable at the distance of zero. This is usually attributed to non-spatial variance.

Returns **gamma** – Unlike in most variogram function formulas, which define the function for $2 * \gamma$, this function will return γ only.

Return type numpy.float64

Notes

This implementation is taken from [12]_:

$$\gamma = b + C_0 * \left[7 * \left(\frac{h^2}{a^2}\right) - \frac{35}{4} * \left(\frac{h^3}{a^3}\right) + \frac{7}{2} * \left(\frac{h^5}{a^5}\right) - \frac{3}{4} * \left(\frac{h^7}{a^7}\right)\right]$$

a is the range parameter. For the cubic function, the effective range and range parameter are the same.

References

Stable model

skgstat.models.**stable**(*h*, *r*, *c0*, *s*, *b=0*)

Stable Variogram function

Implementation of the stable variogram function. Calculates the dependent variable for a given lag (h). The nugget (b) defaults to be 0.

Changed in version 0.4.4: Now returns the nugget at lag 0

Parameters

- **h** (*float*) – Specifies the lag of separating distances that the dependent variable shall be calculated for. It has to be a positive real number.
- **r** (*float*) – The effective range. Note this is not the range parameter! For the stable variogram function the range parameter a is defined to be $a = \frac{r}{\sqrt[s]{3}}$. The effective range is the lag where 95% of the sill are exceeded. This is needed as the sill is only approached asymptotically by the e-function part of the stable model.
- **c0** (*float*) – The sill of the variogram, where it will flatten out. The function will not return a value higher than C0 + b.
- **s** (*float*) – Shape parameter. For s <= 2 the model will be shaped more like a exponential or spherical model, for s > 2 it will be shaped most like a Gaussian function.
- **b** (*float*) – The nugget of the variogram. This is the value of independent variable at the distance of zero. This is usually attributed to non-spatial variance.

Returns **gamma** – Unlike in most variogram function formulas, which define the function for $2 * \gamma$, this function will return γ only.

Return type numpy.float64

Notes

The implementation is taken from [12]_:

$$\gamma = b + C_0 * \left(1. - e^{-\frac{h}{a}^s}\right)$$

a is the range parameter and is calculated from the effective range r as:

$$a = \frac{r}{3^{\frac{1}{s}}}$$

References

Matérn model

skgstat.models.**matern**(*h*, *r*, *c0*, *s*, *b=0*)

Matérn Variogram function

Implementation of the Matérn variogram function. Calculates the dependent variable for a given lag (h). The nugget (b) defaults to be 0.

Changed in version 0.4.4: now returns the nugget instead of NaN for lag 0.

Parameters

- **h** (*float*) – Specifies the lag of separating distances that the dependent variable shall be calculated for. It has to be a positive real number.
- **r** (*float*) – The effective range. Note this is not the range parameter! For the Matérn variogram function the range parameter a is defined to be $a = \frac{r}{2}$ and $a = \frac{r}{3}$ if s is smaller than 0.5 or larger than 10. The effective range is the lag where 95% of the sill are exceeded. This is needed as the sill is only approached asymptotically by Matérn model.
- **c0** (*float*) – The sill of the variogram, where it will flatten out. The function will not return a value higher than C0 + b.
- **s** (*float*) – Smoothness parameter. The smoothness parameter can shape a smooth or rough variogram function. A value of 0.5 will yield the exponential function, while a smoothness of +inf is exactly the Gaussian model. Typically a value of 10 is close enough to Gaussian shape to simulate its behaviour. Low values are considered to be 'smooth', while larger values are considered to describe a 'rough' random field.
- **b** (*float*) – The nugget of the variogram. This is the value of independent variable at the distance of zero. This is usually attributed to non-spatial variance.

Returns **gamma** – Unlike in most variogram function formulas, which define the function for $2 * \gamma$, this function will return γ only.

Return type numpy.float64

Notes

The implementation is taken from[13]:

$$\gamma(h) = b + C_0 \left(1 - \frac{1}{2^{\upsilon-1}\Gamma(\upsilon)} \left(\frac{h}{a}\right)^{\upsilon} K_{\upsilon}\left(\frac{h}{a}\right)\right)$$

a is the range parameter and is calculated from the effective range r as:

$$a = \frac{r}{2}$$

References

2.6.7 Kriging Class

class skgstat.**OrdinaryKriging**(*variogram*, *min_points=5*, *max_points=15*, *mode='exact'*, *precision=100*, *solver='inv'*, *n_jobs=1*, *perf=False*, *sparse=False*, *coordinates=None*, *values=None*)

__init__(*variogram*, *min_points=5*, *max_points=15*, *mode='exact'*, *precision=100*, *solver='inv'*, *n_jobs=1*, *perf=False*, *sparse=False*, *coordinates=None*, *values=None*)

Ordinary Kriging routine

Ordinary kriging estimator derived from the given *Variogram <skgstat.Variogram>* class. To calculate estimations for unobserved locations, an instance of this class can either be called, or the *OrdinaryKriging.transform* method can be used.

Parameters

- **variogram** (Variogram) – Variogram used to build the kriging matrix. Make sure that this instance is describing the spatial dependence in the data well, otherwise the kriging estimation will most likely produce bad estimations.
- **min_points** (*int*) – Minimum amount of points, that have to lie within the variogram's range. In case not enough points are available, the estimation will be rejected and a null value will be estimated.
- **max_points** (*int*) – Maximum amount of points, that will be considered for the estimation of one unobserved location. In case more points are available within the variogram's range, only the *max_points* closest will be used for estimation. Note that the kriging matrix will be an max_points x max_points matrix and large numbers do significantly increase the calculation time.
- **mode** (*str*) – Has to be one of 'exact' or 'estimate'. In exact mode (default) the variogram matrix will be calculated from scratch in each iteration. This gives an exact solution, but it is also slower. In estimate mode, a set of semivariances is pre-calculated and the closest value will be used. This is significantly faster, but the estimation quality is dependent on the given precision.
- **precision** (*int*) – Only needed if *mode='estimate'*. This is the number of pre-calculated in-range semivariances. If chosen too low, the estimation will be off, if too high the performance gain is limited.
- **solver** (*str*) – Do not change this argument

[13] Zimmermann, B., Zehe, E., Hartmann, N. K., & Elsenbeer, H. (2008). Analyzing spatial data: An assessment of assumptions, new methods, and uncertainty using soil hydraulic data. Water Resources Research, 44(10), 1–18. https://doi.org/10.1029/2007WR006604

- **n_jobs** (*int*) – Number of processes to be started in multiprocessing.
- **perf** (*bool*) – If True, the different parts of the algorithm will record their processing time. This is meant to be used for optimization and will be removed in a future version. Do not rely on this argument.
- **sparse** (*bool*) –
- **coordinates** (*numpy.ndarray*, MetricSpace) –
- **values** (*numpy.ndarray*) –

transform(**x*)
Kriging

returns an estimation of the observable for the given unobserved locations. Each coordinate dimension should be a 1D array.

Changed in version 0.6.4: sigma array is now initialized with NaN, instead of empty.

Parameters **x** (*numpy.array*, MetricSpace) – One 1D array for each coordinate dimension. Typically two or three array, x, y, (z) are passed for 2D and 3D Kriging

Returns **Z** – Array of estimates

Return type numpy.array

2.6.8 Example data sets

Datasets

skgstat.data.**aniso**(*N=500*, *seed=42*)
Sample of the *ansio_field*. By default the greyscale image is sampled at 500 random locations.

Parameters

- **N** (*int*) – Number of sample points to use.
- **seed** (*int*) – By default a seed is set to always return the same sample for same N and band input

Returns **result** – Dictionary of the sample and a citation information. The sample is a tuple of two numpy arrays.

Return type dict

See also:

skgstat.data._loader.field field loader

aniso_field Return the full field

Notes

This image was created using `gstools.SRF`. The spatial random field was created using a Gaussian model and has a size of 500x500 pixel. The created field was normalized and rescaled to the value range of a `uint8`. The spatial model includes a small nugget (~ 1/25 of the value range). If you use this example somewhere else, please cite SciKit-GStat **[501]**_, as it is distributed with the library.

References

skgstat.data.**aniso_field**()

Image of a greyscale image with geometric anisotropy. The anisotropy has a North-Easth orientation and has a approx. 3 times larger correlation length than in the perpendicular orientation.

Returns **result** – Dictionary of the sample and a citation information. The sample a numpy array repesenting the image.

Return type dict

See also:

skgstat.data._loader.field field loader

aniso Return a sample

Notes

This image was created using `gstools.SRF`. The spatial random field was created using a Gaussian model and has a size of 500x500 pixel. The created field was normalized and rescaled to the value range of a `uint8`. The spatial model includes a small nugget (~ 1/25 of the value range). If you use this example somewhere else, please cite SciKit-GStat **[501]**_, as it is distributed with the library.

References

skgstat.data.**pancake**(*N=500*, *band=0*, *seed=42*)

Sample of the *pancake_field*. By default, the Red band is sampled at 500 random location without replacement.

Parameters

- **N** (*int*) – Number of sample points to use.
- **band** (*int*) – can be 0 (Red), 1 (Green), 2 (Blue) or `'mean'`, which will average all three RGB bands
- **seed** (*int*) – By default a seed is set to always return the same sample for same N and band input

Returns **result** – Dictionary of the sample and a citation information. The sample is a tuple of two numpy arrays.

Return type dict

See also:

`get_sample` *pancake_field*

Notes

The image originates from a photograph of an actual pancake. The image was cropped to an 500x500 pixel extent keeping the center of the original photograph. If you use this example somewhere else, please cite SciKit-GStat [501]_, as it is distributed with the library.

References

skgstat.data.**pancake_field**(*band=0*)

Image of a pancake with apparent spatial structure. The pankcake has three RGB bands.

Parameters **band** (*int*) – can be 0 (Red), 1 (Green), 2 (Blue) or `'mean'`, which will average all three RGB bands

Returns **result** – Dictionary of the sample and a citation information. The sample is 2D numpy array of the field.

Return type dict

See also:

`skgstat.data._loader.field`, *skgstat.data.pancake*

Notes

The image originates from a photograph of an actual pancake. The image was cropped to an 500x500 pixel extent keeping the center of the original photograph. If you use this example somewhere else, please cite SciKit-GStat [501]_, as it is distributed with the library.

References

Utility Functions

..automodule:: skgstat.data._loader

members field, get_sample

2.6.9 MetricSpace - Coordinate representation

MetricSpace

class skgstat.**MetricSpace**(*coords*, *dist_metric='euclidean'*, *max_dist=None*)

A MetricSpace represents a point cloud together with a distance metric and possibly a maximum distance. It efficiently provides the distances between each point pair (when shorter than the maximum distance).

Note: If a max_dist is specified a sparse matrix representation is used for the distances, which saves space and calculation time for large datasets, especially where max_dist << the size of the point cloud in space. However, it slows things down for small datasets.

__init__(*coords*, *dist_metric='euclidean'*, *max_dist=None*)

ProbabalisticMetricSpace class

Parameters

- **coords** (*numpy.ndarray*) – Coordinate array of shape (Npoints, Ndim)

- **dist_metric** (*str*) – Distance metric names as used by scipy.spatial.distance.pdist
- **max_dist** (*float*) – Maximum distance between points after which the distance is considered infinite and not calculated.

find_closest(*idx*, *max_dist=None*, *N=None*)
find neighbors Find the (N) closest points (in the right set) to the point with index idx (in the left set).

Parameters

- **idx** (*int*) – Index of the point that the N closest neighbors are searched for.
- **max_dist** (*float*) – Maximum distance at which other points are searched
- **N** (*int*) – Number of points searched.

Returns **ridx** – Indices of the N closeset points to idx

Return type numpy.ndarray

diagonal(*idx=None*)
Return a diagonal matrix (as per `squareform`), optionally for a subset of the points

Parameters **idx** (*list*) – list of indices that the diagonal matrix is calculated for.

Returns **diagonal** – squareform matrix of the subset of coordinates

Return type numpy.ndarray

property dists
A distance matrix of all point pairs. If *self.max_dist* is not *None* and *self.dist_metric* is set to *euclidean*, a *scipy.sparse.csr_matrix* sparse matrix is returned.

property tree
If *self.dist_metric* is *euclidean*, a *scipy.spatial.cKDTree* instance of *self.coords*. Undefined otherwise.

MetricSpacePair

class skgstat.**MetricSpacePair**(*ms1*, *ms2*)
A MetricSpacePair represents a set of point clouds (MetricSpaces). It efficiently provides the distances between each point in one point cloud and each point in the other point cloud (when shorter than the maximum distance). The two point clouds are required to have the same distance metric as well as maximum distance.

__init__(*ms1*, *ms2*)

Parameters

- **ms1** (MetricSpace) –
- **ms2** (MetricSpace) –
- **Note** (*ms1* and *ms2* need to have the same *max_dist* and) –
- **distance_metric.** –

find_closest(*idx*, *max_dist=None*, *N=None*)
find neighbors Find the (N) closest points (in the right set) to the point with index idx (in the left set).

Parameters

- **idx** (*int*) – Index of the point that the N closest neighbors are searched for.
- **max_dist** (*float*) – Maximum distance at which other points are searched
- **N** (*int*) – Number of points searched.

Returns **ridx** – Indices of the N closeset points to idx

Return type numpy.ndarray

property dists
A distance matrix of all point pairs. If *self.max_dist* is not *None* and *self.dist_metric* is set to *euclidean*, a *scipy.sparse.csr_matrix* sparse matrix is returned.

2.6.10 Utility Functions

Shannon Entropy

skgstat.util.shannon.**shannon_entropy**(*x*, *bins*)
Shannon Entropy

Calculates the Shannon Entropy, which is the most basic metric in information theory. It can be used to calculate the information content of discrete distributions. This can be used to estimate the intrinsic uncertainty of a sample,independent of the value range or variance, which makes it more comparable.

Parameters

- **x** (*numpy.ndarray*) – flat 1D array of the observations
- **bins** (*list, int*) – upper edges of the bins used to calculate the histogram of x.

Returns **h** – Shannon Entropy of x, given bins.

Return type float

Cross Validation

skgstat.util.cross_validation.**jacknife**(*variogram*, *n: Optional[int] = None*, *metric: str = 'rmse'*, *seed=None*) → float
Leave-one-out cross validation of the given variogram model using the OrdinaryKriging instance. This method can be called using *Variogram.cross_validate*.

Parameters

- **variogram** (skgstat.Variogram) – The variogram isnstance to be validated
- **n** (*int*) – Number of points that should be used for cross validation. If None is given, all points are used (default).
- **metric** (*str*) – Metric used for cross validation. Can be one of [‘rmse’, ‘mse’, ‘mae’]

Returns **metric** – Cross-validation result The value is given in the selected metric.

Return type float

Uncertainty Propagation

skgstat.util.uncertainty.**propagate**(*variogram: Optional[*skgstat.Variogram.Variogram*] = None*, *source: Union[str, List[str]] = 'values'*, *sigma: Union[float, List[float]] = 5*, *evalf: Union[str, List[str]] = 'experimental'*, *verbose: bool = False*, *use_bounds: bool = False*, ***kwargs*)

Uncertainty propagation for the variogram. For a given *Variogram* instance a source of error and scale of error distribution can be specified. The function will propagate the uncertainty into different parts of the *Variogram* and return the confidence intervals or error bounds.

Parameters

- **variogram** (skgstat.Variogram) – The base variogram. The variogram parameters will be used as fixed arguments for the Monte Carlo simulation.
- **source** (*list*) – Source of uncertainty. This has to be an attribute of *Variogram*. Right now only `'values'` is really supported, anything else is untested.
- **sigma** (*list*) – Standard deviation of the error distribution.
- **evalf** (*list*) – Evaluation function. This specifies, which part of the *Variogram* should be used to be evaluated. Possible values are `'experimental'` for the experimental variogram, `'model'` for the fitted model and `parameter'` for the variogram parameters
- **verbose** (*bool*) – If True, the uncertainty_framework package used under the hood will print a progress bar to the console. Defaults to False.
- **use_bounds** (*bool*) – Shortcut to set the confidence interval bounds to the minimum and maximum value and thus return the error margins over a confidence interval.

Keyword Arguments

- **distribution** (*str*) – Any valid numpy.random distribution function, that takes the scale as argument. Defaults to `'normal'`.
- **q** (*int*) – Width (percentile) of the confidence interval. Has to be a number between 0 and 100. 0 will result in the minimum and maximum value as bounds. 100 turns both bounds into the median value. Defaults to `10`
- **num_iter** (*int*) – Number of iterations used in the Monte Carlo simulation. Defaults to `500`.
- **eval_at** (*int*) – If evalf is set to model, the theoretical model get evaluated at this many evenly spaced lags up to maximum lag. Defaults to `100`.
- **n_jobs** (*int*) – The evaluation can be performed in parallel. This will specify how many processes may be spawned in parallel. None will spwan only one (default).

Note: This is an untested experimental feature.

Returns **conf_interval** – Confidence interval of the uncertainty propagation as [lower, median, upper]. If more than one evalf is given, a list of ndarrays will be returned. See notes for more details.

Return type numpy.ndarray

Notes

For each member of the evaluated property, the lower and upper bound along with the median value is retuned as `[low, median, up]`. Thus the returned array has the shape `(N, 3)`. N is the lengh of evaluated property, which is `n_lags` <`skgstat.Variogram.n_lags()` for `'experimental'`, either `3` for `'parameter'` or `4` if `Variogram.model = 'stable' | 'matern'` and `100` for `'model'` as the model gets evaluated at 100 evenly spaced lags up to the maximum lag class. This amount can be changed using the eval_at parameter.

If more than one evalf parameter is given, the Variogram will be evaluated at multiple steps and each one will be returned as a confidence interval. Thus if `len(evalf) == 2`, a list containing two confidence interval matrices will be returned. The order is [experimental, parameter, model].

2.7 Changelog

2.7.1 Version 0.6.4

- [Kriging] *OrdinaryKriging.sigma* <*skgstat.OrdinaryKriging*> is now initialized as a *NaN*-filled array.
- [Kriging] *OrdinaryKriging._estimator* <*skgstat.OrdinaryKriging*> handles the error variance matrix index now correctly. On error during kriging, the index was not incremented, which lead to malformed error variance field output.

2.7.2 Version 0.6.3

- [interfaces] If any of the gstools interfaces are used, the Variogram will call *`fit`* without forcing a full preprocessing cycle. This fixes edge cases, where a parameter was mutated, but the fitting not performed before the instance was exported. This should only have happended in very rare occasions.
- [data] added the meuse dataset from the R-package `'sp'`

2.7.3 Version 0.6.2

- [Variogram] the fitting method is now implemented as *`Variogram.fit_method`* property. It will drop fitting parameters if the fit method is changed to something else than `'manual'`.
- [Variogram] If an invalid *`Variogram.fit_method`* is set, an `AttributeError` will instantly be raised. Beforehand it was only raised on the next call of *`fit`*

2.7.4 Version 0.6.1

- The Dockerfile was completely rewritten. A user can now specify the used Python version at build time of the docker image.
- The Dockerfile is now part of the python package

2.7.5 Version 0.6.0

- The util and data submodule are now always loaded at top-level
- fixed a potential circular import
- added uncertainty tools to util. This is not yet finished and may change the signature before it gets stable with Version 1.0 or 1.1

Note: The current implementation of uncertainty propagation is not stable. It will be changed until version 0.7. The entry-point *obs_sigma* will stay stable and persist, but currently the uncertainty propagation will not be updated and invalidated as the Variogram instance changes.

2.7.6 Version 0.5.6

- [Variogram] the interal *MetricSpace* instance used to calculate the distance matrix is now available as the *Variogram.metric_space* property.
- [Variogram] *Variogram.metric_space* is now read-only.
- [unittest] two unittests are changed (linting, not functionality)

2.7.7 Version 0.5.5

- [data] new submodule *data* contains sample random fields and methods for sampling these fields in a reproducible way at random locations and different sample sizes.

2.7.8 Version 0.5.4

- [util] added a new *cross_validation* utility module to cross-validate variograms with leave-one-out Kriging cross validations.

2.7.9 Version 0.5.3

- [MetricSpace] new class `ProbabilisticMetricSpace` that extends the metric space by a stochastic element to draw samples from the input data, instead of using the full dataset.

2.7.10 Version 0.5.2

- [interface] new interface function added: *to_gs_krige*. This interface will return a `gs.Krige` instance from the fitted variogram.
- some typos were corrected
- some code refactored (mainly linting errors)

2.7.11 Version 0.5.1

- [plotting] the spatio-temporal 2D and 3D plots now label the axis correctly.
- [plotting] fixed swapped plotting axes for spatio-temporal plots.

2.7.12 Version 0.5.0

- [MetricSpace] A new class *MetricSpace* was introduced. This class can be passed to any class that accepted coordinates so far. This wrapper can be used to pre-calculate large distance matrices and pass it to a lot of Variograms.
- [MetricSpacePair] A new class *MetricSpacePair* was introduced. This is a pair of two *MetricSpaces* and pre-calculates all distances between the two spaces. This is i.e. used in Kriging to pre-calcualte all distance between the input coordinates and the interpolation grid only once.

2.7.13 Version 0.4.4

- [models] the changes to *matern* introduced in *0.3.2* are reversed. The Matérn model does not adapt the smoothness scaling to effective range anymore, as the behavior was too inconsistent.
- [interface] minor bugfix of circular import in *variogram_estimator* interface
- [models] *matern(0, ...)* now returns the nugget instead of *numpy.NaN*
- [models] *stable(0, ...)* now returns the nugget instead of *numpy.NaN* or a *ZeroDivisionError*.

2.7.14 Version 0.4.3

- [Variogram] *dim* now returns the spatial dimensionality of the input data.
- [Variogram] fixed a numpy depreaction warning in *_calc_distances*

2.7.15 Version 0.4.2

- [Variogram] *bins* now cases manual setted bin edges automatically to a `numpy.array()`.
- [Variogram] *get_empirical* returns the empirical variogram. That is a tuple of the current *bins* and *experimental* arrays, with the option to move the bin to the lag classes centers.

2.7.16 Version 0.4.1

- [Variogram] moved the bin function setting into a wrapper instance method, which was an anonymous lambda before. This makes the Variogram serializable again.
- [Variogram] a list of pylint errors were solved. Still enough left.

2.7.17 Version 0.4.0

- [binning] added *'stable_entropy'* option that will optimize the lag class edges to be of comparable Shannon Entropy.

2.7.18 Version 0.3.11

- [Variogram] A new method is introduced to calculate fitting weights. Works for all but the manual fit method. By setting `fit_sigma='entropy'`, the fitting weights will be adjusted according to the lag classes' Shannon entropy. That will ignore lag classes of high uncertainty and emphasize lags of low uncertainty.

2.7.19 Version 0.3.10

- [binning] added a median aggregation option to `ward`. This can be enabled by setting *binning_agg_func* to *'median'*. The cluster centroids will be derived from the members median value, instead of mean value.
- [Variogram] added `fit_method='ml'` - a maximum likelihood fitting procedure to fit the theoretical variogram to the experimental
- [Variogram] added `fit_method='manual'`. This is a manual fitting method that takes the variogram parameters either at instantiation prefixed by *fit_*, or as keyword arguments by `fit`.
- [Variogram] the manual fitting method will preseve the previous parameters, if the Variogram was fitted before and the fitting parameters are not manually overwritten.

2.7.20 Version 0.3.9

- [binning] added `kmeans` and `ward` for forming non-equidistant lag classes based on a distance matrix clustering
- [Kriging] Kriging now stores the last interpolated field as *z*. This is the first of a few changes in future releases, which will ultimately add some plotting methods to Kriging.

2.7.21 Version 0.3.8

- [plotting] minor bugfixes in plotting routines (wrong arguments, pltting issues)
- [docs] added a tutorial about plotting
- [binning] added `auto_derived_lags` for a variety of different methods that find a good estimate for either the number of lag classes or the lag class width. These can be used by passing the method name as `bin_func` parameter: Freedman-Diaconis (*'fd'*), Sturge's rule (*'sturges'*), Scott's rule (*'scott'*) and Doane's extension to Sturge's rule (*'doane'*). Uses *histogram_bin_edges <numpy.histogram_bin_edges>* internally.

2.7.22 Version 0.3.7

- [Variogram] now accepts arbitary kwargs. These can be used to further specify functional behavior of the class. As of Version *0.3.7* this is used to pass arguments down to the `entropy` and `percentile` estimators.
- [Variogram] the `describe` now adds the `init` arguments by default to the output. The method can output the init params as a nested dict inside the output or flatten the output dict.

2.7.23 Version 0.3.6

Warning: There is some potential breaking behaviour

- [Variogram] some internal code cleanup. Removed some unnecessary loops
- [Variogram] setting the `n_lags` property now correctly forces a recalculation of the lag groupings. So far they were kept untouches, which might result in old experimental variogram values for the changed instance. **This is a potential breaking change**.
- [Variogram] The `lag_classes` generator now yields empty arrays for unoccupied lag classes. This will result in `NaN` values for the semi-variance. This is actually a bug-fix. **This is a potential breaking change**

2.7.24 Version 0.3.5

- [plotting] The `location_trend` can now add trend model lines to the scatter plot for the *'plotly'* backend and calculate the R^2 for the trend model.
- [Variogram] the *internal* attribute holding the name of the current distance function was renamed from *_dict_func* to *_dist_func_name*

2.7.25 Version 0.3.4

- [plotting] The `scattergram` functions color the plotted points with respect to the lag bin they are originating from. For *matplotlib*, this coloring is suppressed, but can activated by passing the argument `scattergram(single_color=False)`.

2.7.26 Version 0.3.3

- [plotting] a new submodule is introduced: `skgstat.plotting`. This contains all plotting functions. The plotting behavior is not changed, but using `skgstat.plotting.backend()`, the used plotting library can be switched from *matplotlib* to *plotly*
- [stmodels] some code cleanup
- [SpaceTimeVariogram] finally can fit the product-sum model to the experimental variogram

2.7.27 Version 0.3.2

- [models] Matérn model now adapts effective range to smoothness parameter
- [models] Matérn model documentation updated
- [models] some minor updates to references in the docs

2.7.28 Version 0.3.1

- [Variogram] - internal distance calculations were refactored, to speed things up
- [Kriging] - internal distance calculations were refactored, to speed things up

2.7.29 Version 0.3.0

- [Variogram] some internal calculations were changed.
- [DirectionalVariogram] - the circular search are is removed and raises a NotImplementedError
- [DirectionalVariogram] - direction mask data is calculated way faster and without shapely involved.
- shapely is not a dependency anymore
- [unittests] - more unittests were added.

2.7.30 Version 0.2.8

- [Variogram] is now `pickle.dump()`-able, by removing `lambda` usage (thanks to @redhog!)
- [Variogram] now raises a *Warning* if all input values are the same
- [DOCS] Tutorial added and Dockerfile finalized
- [Variogram] *normalize* default value changed to *normalize=False*
- [Variogram] *harmonize* parameter is removed
- [Variogram] Monotonization (old harmonize par) is available as a new theoretical model function. Can be used by setting *model='harmonize'*
- [interfaces] gstools interface implemented. `gstools_cov_model` takes a *`skgstat.Variogram`* instance and returns a **fitted** *gstools.CovModel*.

2.7.31 Version 0.2.7

- [Kriging] Little performance gains due to code cleanup.
- [Variogram] The *normalize=True* default in *__init__* will change to *normalize=False* in a future version. A DeprecationWarning was included.
- [tests] The Variogram class fitting unit tests are now explicitly setting the normalize parameter to handle the future deprecation.
- [tests] More unittests were added to increase coverage
- [interfaces] The new submodule *skgstat.interfaces* is introduced. This submodule collects interfacing classes to use skgstat classes with other Python modules.
- [interfaces] The first interfacing class is the `VariogramEstimator`. This is a scikit-learn compatible *Estimator* class that can wrap a *Variogram*. The intended usage is to find variogram hyper-parameters using *GridSearchCV*. This is also the only usecase covered in the unit tests.
- [interfaces] Implemented `pykrige_as_kwargs`. Pass a *`Variogram`* object and a dict of parameters is returned that can be passed to pykrige Kriging classes using the double star operator.
- Added Dockerfile. You can now build a docker container with scikit-gstat installed in a miniconda environment. On run, a jupyter server is exposed on Port 8888. In a future release, this server will serve tutorial notebooks.

- [stmodels] small bugfix in product model
- [stmodels] removed variogram wrapper and added stvariogram wrapper to correctly detect space and time lags

2.7.32 Version 0.2.6

- [OrdinaryKriging]: widely enhanced the class in terms of performance, code coverage and handling.
  - added *mode* property: The class can derive exact solutions or estimate the kriging matrix for high performance gains
  - multiprocessing is supported now
  - the *solver* property can be used to choose from 3 different solver for the kriging matrix.
- [OrdinaryKriging]: calculates the kriging variance along with the estimation itself. The Kriging variance can be accessed after a call to *`OrdinaryKriging.transform`* and can be accessed through the *OrdinaryKriging.sigma* attribute.
- [Variogram] deprecated `Variogram.compiled_model`. Use *`Variogram.fitted_model`* instead.
- [Variogram] added a new and much faster version of the parameterized model: *`Variogram.fitted_model`*
- [Variogram] minor change in the cubic model. This made the adaption of the associated unit test necessary.

2.7.33 Version 0.2.5

- added *`OrdinaryKriging`* for using a *`Variogram`* to perform an interpolation.

2.7.34 Version 0.2.4

- added *`SpaceTimeVariogram`* for calculating dispersion functions depending on a space and a time lag.

2.7.35 Version 0.2.3

- **[severe bug]** A severe bug was in `Variogram.__vdiff_indexer` was found and fixed. The iterator was indexing the `Variogram._diff` array different from `Variogram.distance`. **This lead to wrong semivariance values for all versions > 0.1.8!**. Fixed now.
- [Variogram] added unit tests for parameter setting
- [Variogram] fixed `fit_sigma` setting of `'exp'`: changed the formula from $e^{\left(\frac{1}{x}\right)}$ to $1. - e^{\left(\frac{1}{x}\right)}$ in order to increase with distance and, thus, give less weight to distant lag classes during fitting.

2.7.36 Version 0.2.2

- added DirectionalVariogram class for direction-dependent variograms
- [Variogram] changed default values for *estimator* and *model* from function to string

2.7.37 Version 0.2.1

- added various unittests

2.7.38 Version 0.2.0

- completely rewritten Variogram class compared to v0.1.8

PYTHON MODULE INDEX