
SciKit GStat Documentation

Release 0.2.8

Mirko Mälicke

Feb 04, 2020

CONTENTS:

WELCOME TO SCIKIT GSTAT'S DOCUMENTATION!

SciKit-Gstat is a scipy-styled analysis module for geostatistics. It includes two base classes *Variogram* and *DirectionalVariogram*. Both have a very similar interface and can compute experimental variograms and model variograms. The module makes use of a rich selection of semi-variance estimators and variogram model functions, while being extensible at the same time.

With version 0.2.4, the class *SpaceTimeVariogram* has been added. It computes space-time experimental variogram. However, space-time modeling is not implemented yet.

With version 0.2.5, the class *OrdinaryKriging* has been added. It is working and can be used. However, it is not documented, the arguments might still change, multiprocessing is not implemented and the kriging algorithm is not yet very efficient.

Note: Scikit-gstat was rewritten in major parts. Most of the changes are internal, but the attributes and behaviour of the *Variogram* has also changed substantially. A detailed description of the new versions usage will follow. The last version of the old *Variogram* class, 0.1.8, is kept in the *version-0.1.8* branch on GitHub, but not developed any further. It is not compatible to the current version.

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, & Helge David Schneider. (2019, November 7). Scikit-GStat 0.2.6: A scipy flavoured geostatistical analysis toolbox written in Python. (Version v0.2.6). Zenodo. <http://doi.org/10.5281/zenodo.3531816>

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 https://github.com/mmaelicke/scikit-gstat.git
cd scikit-gstat
pip install -r requirements.txt
python setup.py install
```

2.1.3 Note

Depending on you OS, you might run into problems installing all requirements in a clean Python environment. These problems are usually caused by the scipy and numba package, which might need to be compiled. From our experience, no problems should occur, when an environment manager like anaconda is used. Then, the requirements can be installed like:

```
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 random data.

```
In [1]: from skgstat import Variogram
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]: coordinates = np.random.gamma(20, 5, (50,2))
In [7]: np.random.seed(42)
In [8]: values = np.random.normal(20, 5, 50)
```

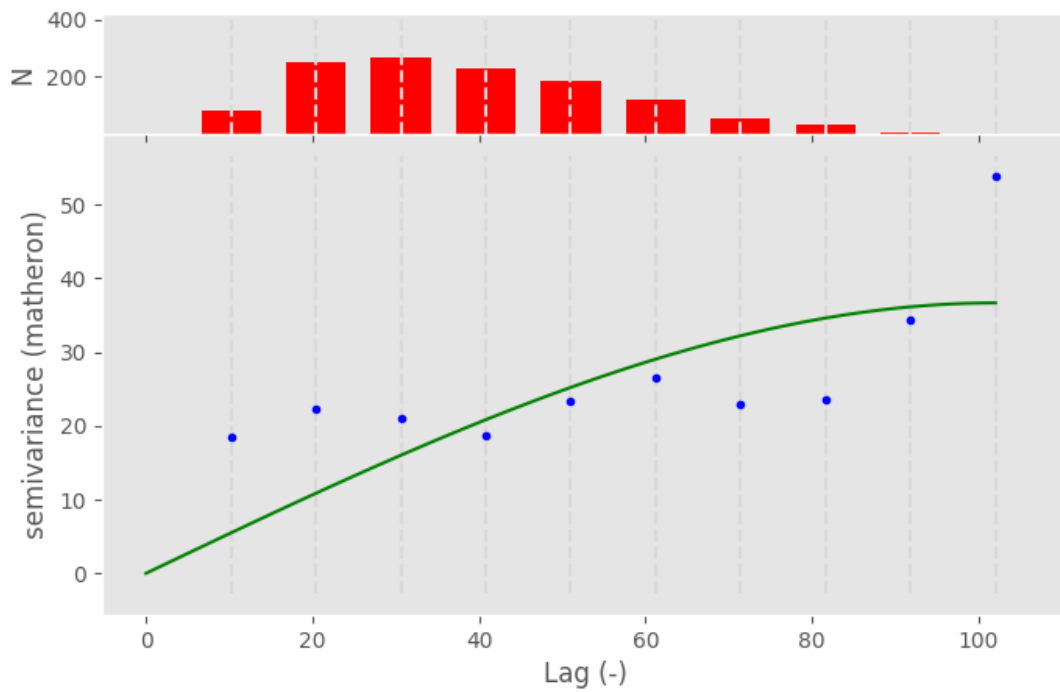
The Variogram needs at least an array of coordinates and an array of values on instantiation.

```
In [9]: V = Variogram(coordinates=coordinates, values=values)
In [10]: print(V)
spherical Variogram
-----
Estimator:          matheron
Effective Range:    102.05
Sill:               36.65
Nugget:             0.00
```

2.2.2 Plot

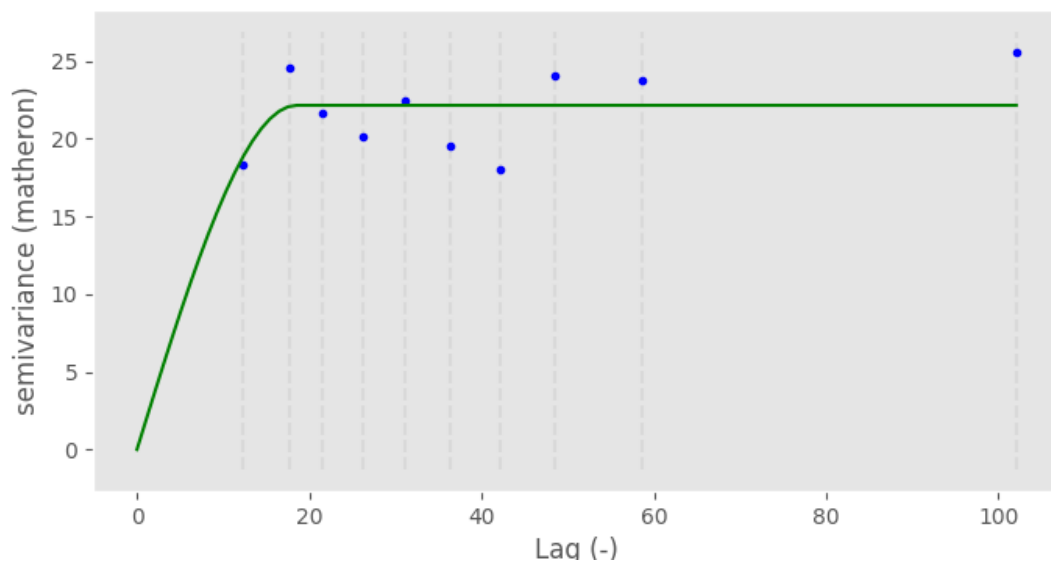
The Variogram class has its own plotting method.

```
In [11]: V.plot()
Out[11]: <Figure size 800x500 with 2 Axes>
```

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>
```



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 correlations in a set of point data. While the main tool, the variogram, is quite easy to implement and use, a lot of assumptions are underlying it. The typical application is geostatistics is an interpolation. Therefore, although using point data, a basic concept is to understand these 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 not observed 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. That would only be possible if professionals from each discipline contribute to the project. The objective is more to offer some common tools and simplify the process of geostatistical analysis and tool development thereby. However, you 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 an interpolation method, that utilizes a variogram to find the estimate for weights as shown in the section above.
- **geostatistical simulation:** is aiming on generate random fields that fit a given set of observations or a pre-defined variogram.

Note: I am planning to implement common tools from all three fields. However, up to now, I am only focusing on variograms and no field generators or kriging procedures are available.

How to use this Guide

Write something about code examples and stuff

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
```

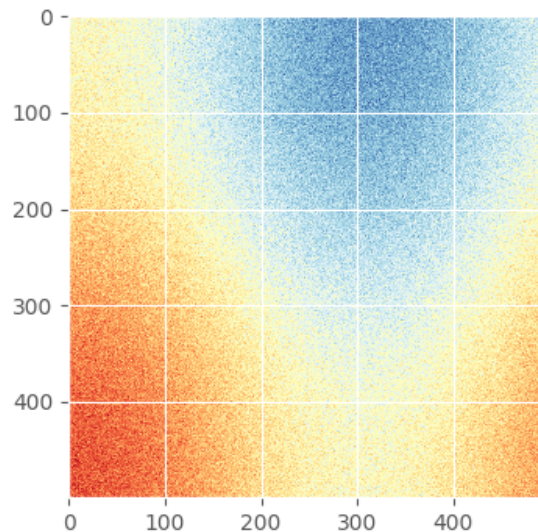
(continues on next page)

(continued from previous page)

```
# 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 0x7fa915742cc0>
```



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 [11]: from skgstat import Variogram

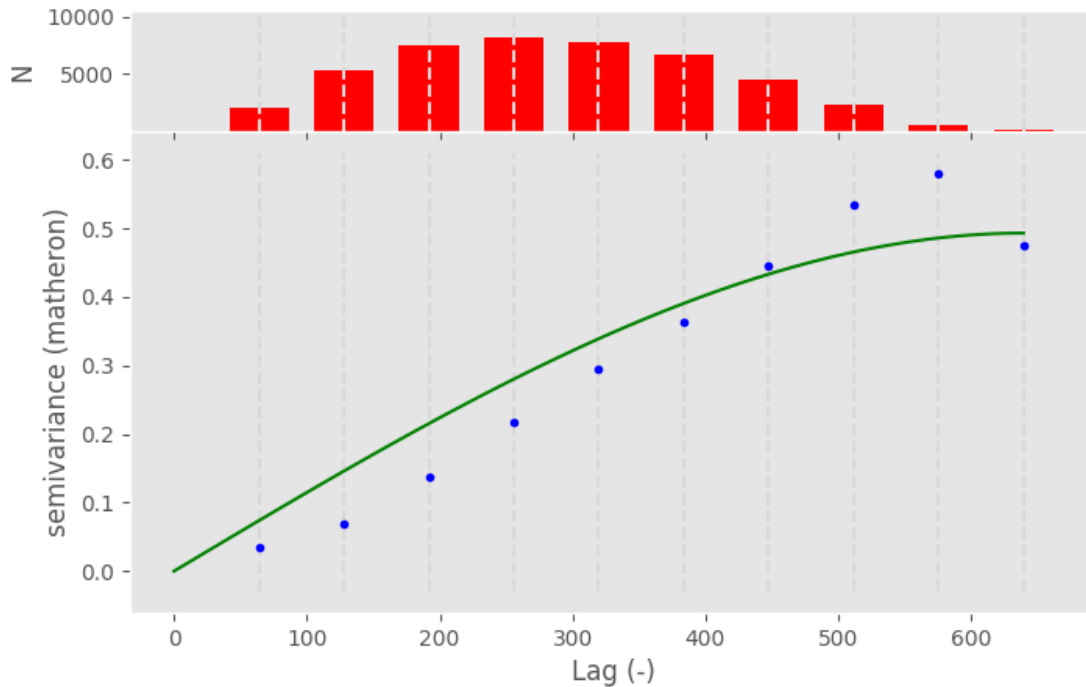
# random coordinates
In [12]: np.random.seed(42)

In [13]: coords = np.random.randint(0, 500, (300, 2))

In [14]: values = np.fromiter((z[c[0], c[1]] for c in coords), dtype=float)

In [15]: V = Variogram(coords, values)

In [16]: V.plot()
Out[16]: <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 order 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.

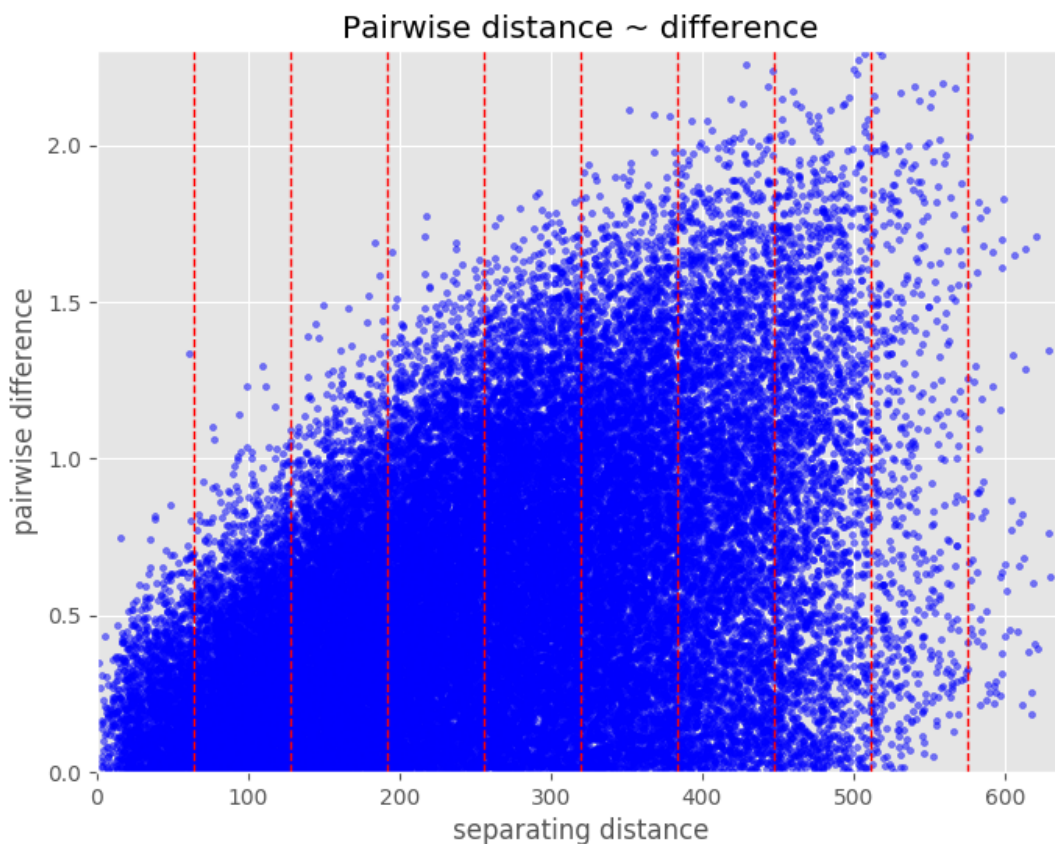
The variogram relates the separating distance between two observation points to a measure of variability of values at that given distance. Our expectation is that variance is increasing with distance, what can basically be seen in the presented figure.

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. The independent variable is a measure of (usually) 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 it is useful 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. This becomes obvious, when one plots 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 [17]: V.distance_difference_plot()
Out [17]: <Figure size 800x600 with 1 Axes>
```



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 within the bins. 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. Else you'll have to wait until I did that.

Binning the separating distances into distance lags is therefore a crucial and most important task in variogram analysis. The final binning must discretize 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 instantiation. To be more precise, only the upper triangle is stored in an `array` with the distance values sorted row-wise. Consider this very straightforward set of locations:

```
In [18]: locations = [[0,0], [0,1], [1,1], [1,0]]

In [19]: V = Variogram(locations, [0, 1, 2, 1], normalize=False)

In [20]: V.distance
Out[20]: array([1.    , 1.414, 1.    , 1.    , 1.414, 1.    ])

# turn into a 2D matrix again
In [21]: from scipy.spatial.distance import squareform

In [22]: 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 will hardly 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 two different methods for binning distance data. They can be set using the `bin_func` attribute. You have to pass the name of the method. This has to be one of `['even', 'uniform']` to use one of the predefined binning functions. Both methods will use two parameters to calculate the bins from the distance matrix: `n`, the amount of bins, and `maxlag`, the maximum distance lag to be considered. You can choose both parameters during *Variogram* instantiation as `n_lags` and `maxlag`. The `'even'` method will then form `n` bins from 0 to `maxlag` of same width. The `'uniform'` method will form `n` bins from 0 to `maxlag` with the same value count in each bin. The following example should illustrate this:

```
In [23]: from skgstat.binning import even_width_lags, uniform_count_lags

In [24]: from scipy.spatial.distance import pdist

In [25]: loc = np.random.normal(50, 10, size=(30, 2))

In [26]: distances = pdist(loc)
```

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

```
In [27]: even_width_lags(distances, 10, 250)
Out[27]:
array([ 4.405,  8.809, 13.214, 17.618, 22.023, 26.427, 30.832, 35.237,
        39.641, 44.046])

In [28]: uniform_count_lags(distances, 10, 250)
Out[28]:
array([ 7.198, 10.432, 12.34 , 15.147, 17.693, 20.381, 23.013, 26.278,
        30.641, 44.046])
```

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 systematically 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, the *Variogram Class* implements a method for doing mapping a bit more efficiently.

Note: As of this writing, the actual iterator that yields the group number for each point is written in a plain Python loop. This is not very fast and in fact the main bottleneck of this class. I am evaluating `numba`, `cython` or a `numpy` based solution at the moment to gain better performance.

A `array` of bin groups for each point pair that is indexed exactly like the distance `<skgstat.Variogram.distance() array` can be obtained by `lag_groups`.

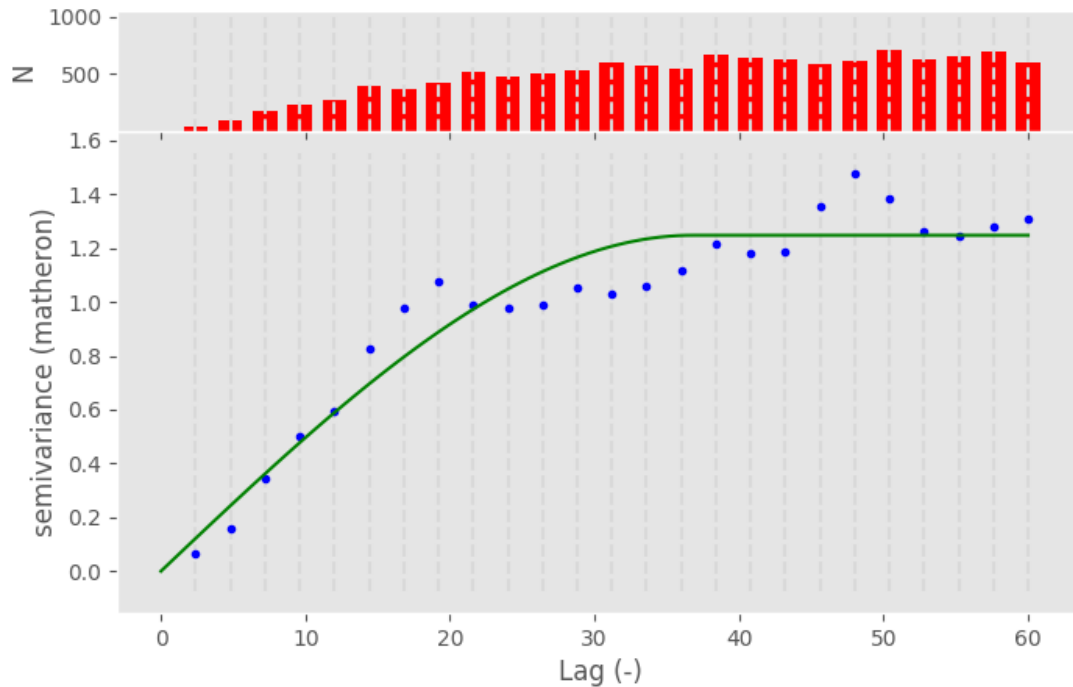
This will be illustrated by some sample data (you can find the CSV file in the github repository of SciKit-GStat). You can easily read the data using `pandas`.

```
In [29]: import pandas as pd

In [30]: data = pd.read_csv('data/sample_sr.csv')

In [31]: V = Variogram(list(zip(data.x, data.y)), data.z,
.....:     normalize=False, n_lags=25, maxlag=60)
.....:

In [32]: V.plot()
Out[32]: <Figure size 800x500 with 2 Axes>
```

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 [33]: V.distance[:10]
Out[33]:
array([20.809, 51.478, 27.203, 44.721, 91.608, 71.784, 87.693, 85.446,
       18.788, 23.195])

# first 10 groups
In [34]: V.lag_groups()[:10]
Out[34]: array([ 8, 21, 11, 18, -1, -1, -1, -1,  7,  9])
```

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

```
In [35]: V.bins
Out[35]:
array([ 2.4,  4.8,  7.2,  9.6, 12. , 14.4, 16.8, 19.2, 21.6, 24. , 26.4,
       28.8, 31.2, 33.6, 36. , 38.4, 40.8, 43.2, 45.6, 48. , 50.4, 52.8,
       55.2, 57.6, 60. ])
```

The first and 9th element are grouped into group 3. Their values are 20.8 and 18.8. The grouping starts with 0, therefore the corresponding upper bound of the bin is at index 3 and the lower at 2. The bin edges are therefore $15.8 < x < 21.07$. 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 mechanism above to find the correct points. However, *Variogram* class 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 [36]: for i, group in enumerate(V.lag_classes()):
        ....:     print('[Group %d]: %.2f' % (i, np.mean(group)))
        ....:
[Group 0]: 0.27
[Group 1]: 0.43
[Group 2]: 0.65
[Group 3]: 0.80
[Group 4]: 0.84
[Group 5]: 1.00
[Group 6]: 1.11
[Group 7]: 1.14
[Group 8]: 1.10
[Group 9]: 1.11
[Group 10]: 1.13
[Group 11]: 1.15
[Group 12]: 1.12
[Group 13]: 1.14
[Group 14]: 1.18
[Group 15]: 1.25
[Group 16]: 1.23
[Group 17]: 1.24
[Group 18]: 1.31
[Group 19]: 1.38
[Group 20]: 1.33
[Group 21]: 1.29
[Group 22]: 1.27
[Group 23]: 1.31
[Group 24]: 1.32
```

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

Experimental variograms

The last stage before a variogram function can be modeled is to define an empirical variogram, also known as *experimental variogram*, which will be used to parameterize a variogram model. However, the experimental 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 experimental variogram will also affect the variogram model, which is ultimately used to interpolate the input data.

The previous sections summarized how distance is calculated and handled 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 most popular estimator is called *Matheron estimator*. In case the estimator used is not further specified, Matheron was 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.

```
In [37]: fig, _a = plt.subplots(2, 2, figsize=(8,8))

In [38]: axes = _a.flatten()

In [39]: V.plot(axes=axes[0], hist=False)
Out[39]: <Figure size 800x800 with 4 Axes>

In [40]: V.estimator = 'cressie'

In [41]: V.plot(axes=axes[1], hist=False)
Out[41]: <Figure size 800x800 with 4 Axes>

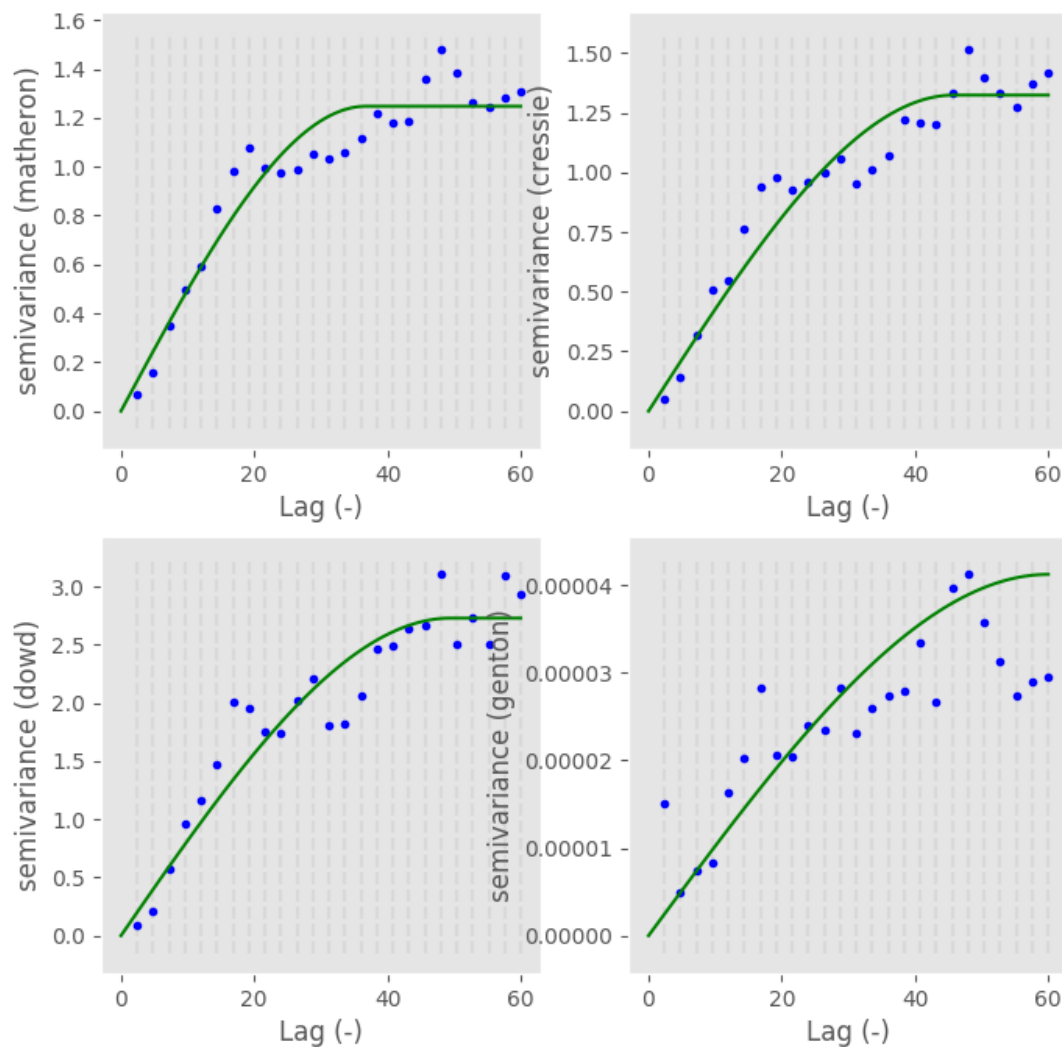
In [42]: V.estimator = 'dowd'

In [43]: V.plot(axes=axes[2], hist=False)
Out[43]: <Figure size 800x800 with 4 Axes>

In [44]: V.estimator = 'genton'

In [45]: V.plot(axes=axes[3], hist=False)
Out[45]: <Figure size 800x800 with 4 Axes>

In [46]: fig.show()
```



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 step is straightforward. We need to define a function that takes a distance value (not a lag) and returns a semi-variance value. One big advantage of these models is, that we can assure different things, like positive definiteness. 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 useful 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 *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 can not be described spatially.

The spherical model

The spherical 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:

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

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 `skgstat.models` sub-module, we can utilize e.g. `curve_fit` from `scipy` to fit the model using a least squares approach.

```
In [47]: from skgstat import models

# set estimator back
In [48]: V.estimate = 'matheron'

In [49]: V.model = 'spherical'

In [50]: xdata = V.bins

In [51]: ydata = V.experimental

In [52]: from scipy.optimize import curve_fit

In [53]: cof, cov = curve_fit(models.spherical, xdata, ydata)
```

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

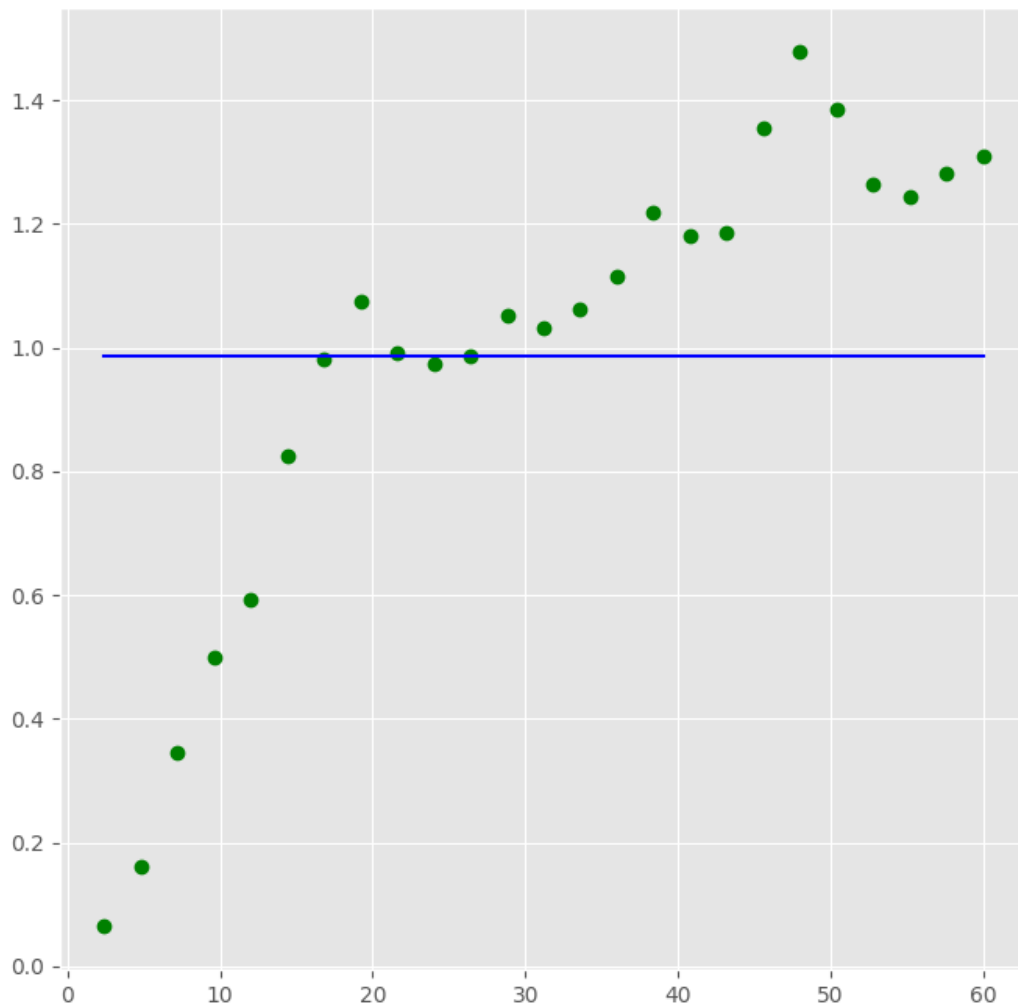
```
In [54]: pprint("range: %.2f\nsill: %.f\nnugget: %.2f" % (cof[0], cof[1], cof[2]))
'range: 1.00\nsill: 102\nnugget: -100.87'
```

```
In [55]: xi = np.linspace(xdata[0], xdata[-1], 100)

In [56]: yi = [models.spherical(h, *cof) for h in xi]

In [57]: plt.plot(xdata, ydata, 'og')
Out[57]: [<matplotlib.lines.Line2D at 0x7fa917c96dd8>]

In [58]: 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. *Variogram* can be forced to use the same by setting the `Variogram.fit_method` to 'lm'. The default, however, is 'trf', which is the *Trust Region Reflective* algorithm, the bounded fit with initial guesses described above. You can use it like:

```
In [59]: V.fit_method = 'trf'

In [60]: V.plot();

In [61]: pprint(V.describe())
{'effective_range': 36.78585313528298,
 'estimator': 'matheron',
 'name': 'spherical',
```

(continues on next page)

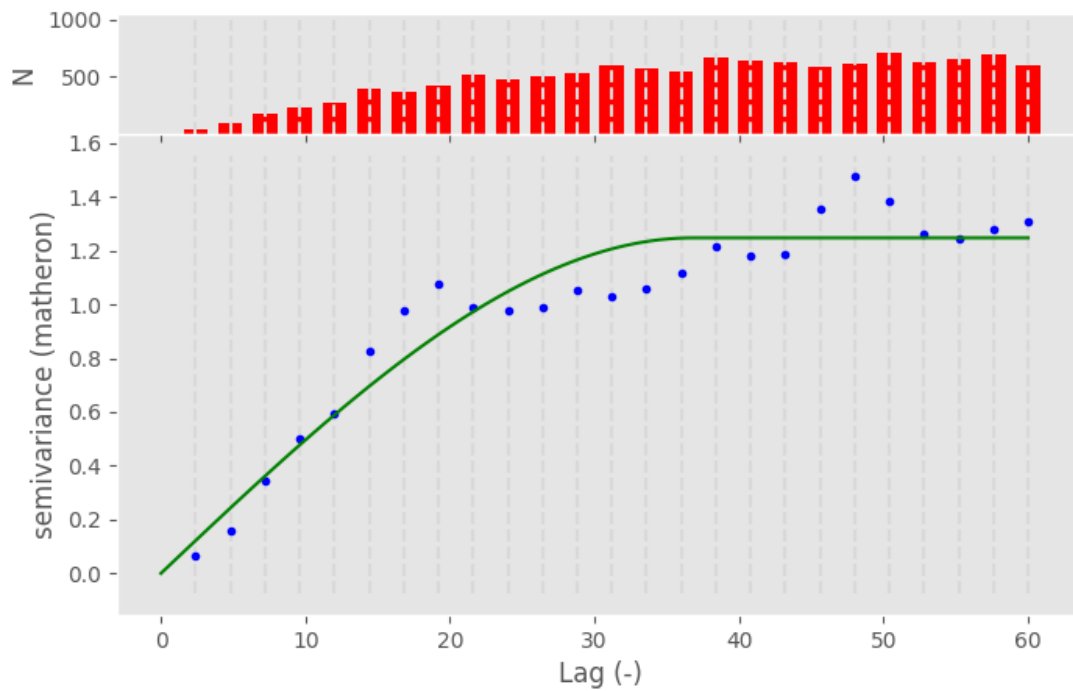
(continued from previous page)

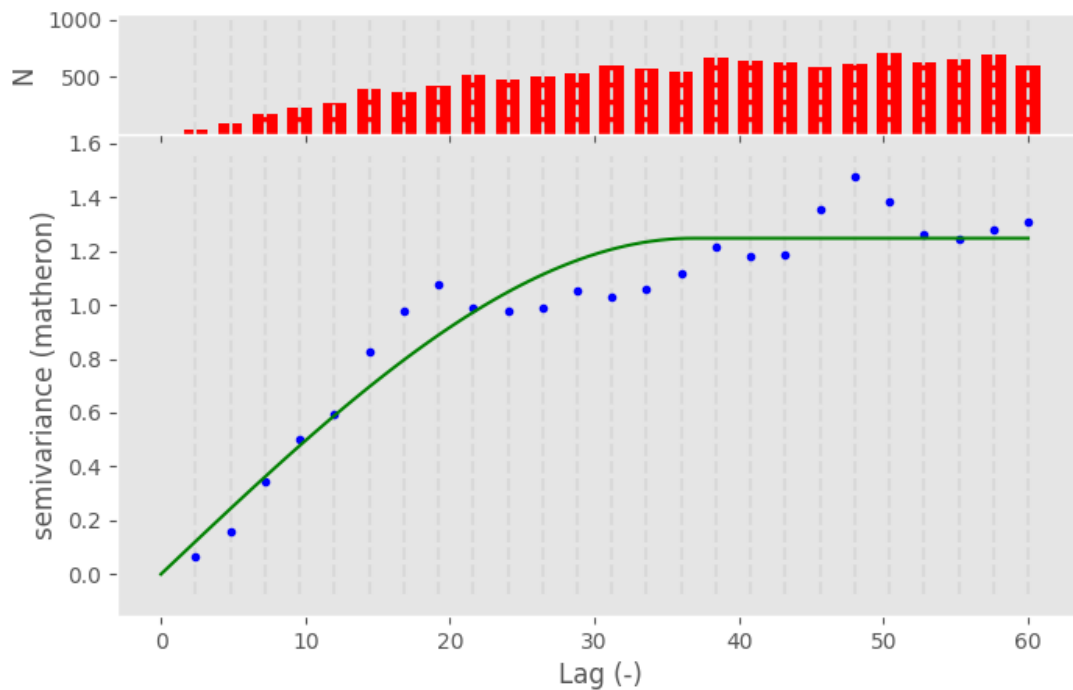
```
'nugget': 0,
'sill': 1.2473949087006804}

In [62]: V.fit_method = 'lm'

In [63]: V.plot();

In [64]: pprint(V.describe())
{'effective_range': 36.78585313528298,
 'estimator': 'matheron',
 'name': 'spherical',
 'nugget': 0,
 'sill': 1.2473949087006804}
```





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

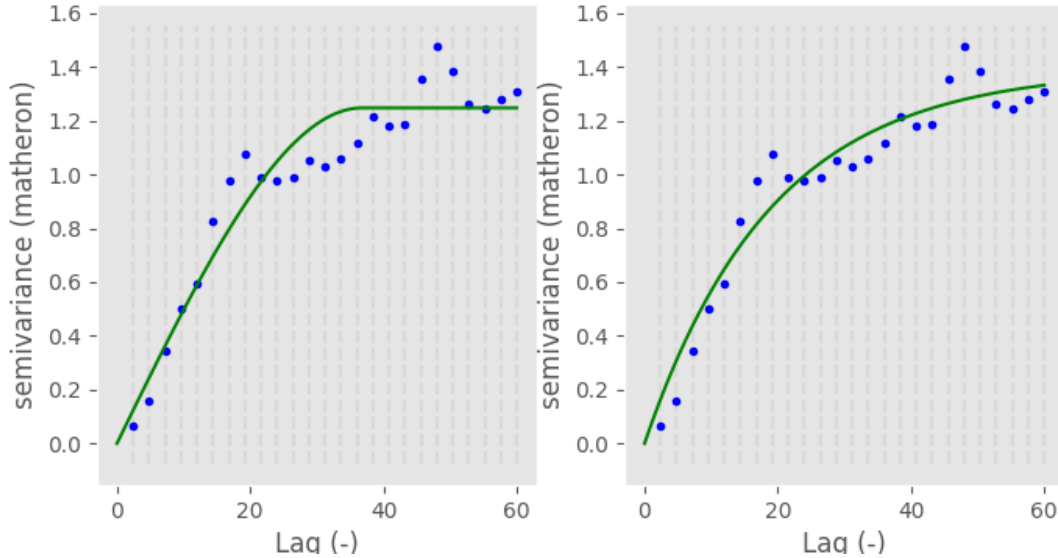
The exponential model is quite similar to the spherical one. It models semi-variance values to increase exponentially with distance, like the spherical. The main difference is that this increase is not as steep as for the spherical. That means, the effective range is larger for an exponential model,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

```
In [65]: fig, axes = plt.subplots(1, 2, figsize=(8, 4))
In [66]: V.fit_method = 'trf'
In [67]: V.plot(axes=axes[0], hist=False)
Out[67]: <Figure size 800x400 with 2 Axes>
In [68]: V.model = 'exponential'
In [69]: V.plot(axes=axes[1], hist=False);
```

Gaussian model

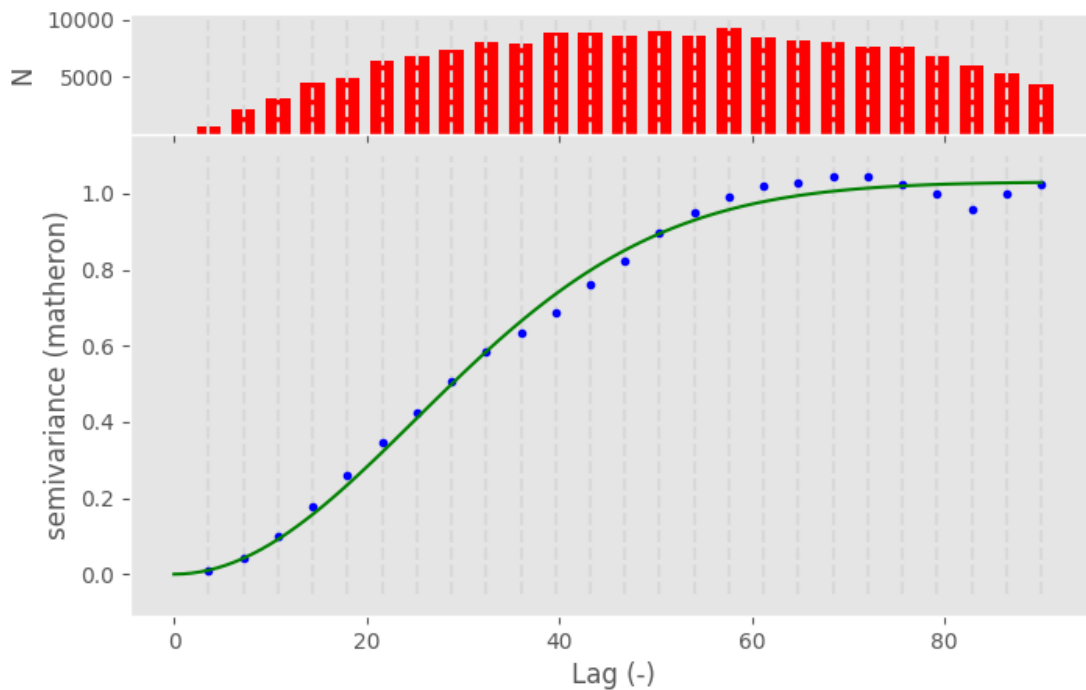
The last fundamental variogram model is the Gaussian. Unlike the spherical and exponential models a very different spatial relationship between semi-variance and distance. Following the Gaussian model, observations are assumed to be similar up to intermediate distances, showing just a gentle increase in semi-variance. Then, the semi-variance increases dramatically within just a few distance units up to the sill, which is again approached asymptotically. 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.

```
In [70]: data = pd.read_csv('data/sample_lr.csv')

In [71]: Vg = Variogram(list(zip(data.x, data.y)), data.z.values,
.....:                  normalize=False, n_lags=25, maxlag=90, model='gaussian')
.....:

In [72]: Vg.plot();
```



Matérn model

One of the not so commonly used models is the Matérn model. It is nevertheless implemented into scikit-gstat as it is one of the most powerful models. Especially in cases where you cannot choose the appropriate model a priori so easily. The Matérn model takes an additional smoothness parameter, that can change the shape of the function in between an exponential model shape and a Gaussian one.

When direction matters

What is 'direction'?

Space-time variography

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 calculate 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 s_0 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**. Remember, that this array *variances* is what we want the target weights to incorporate. 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. $a_1 \dots a_5$ 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. Luckily 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 squareform distance matrix calculated in the very beginning. 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
```

(continues on next page)

(continued from previous page)

```
# 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 inferred 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)

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

(continues on next page)

(continued from previous page)

```
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: 5.262880578742379

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 useful 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`.

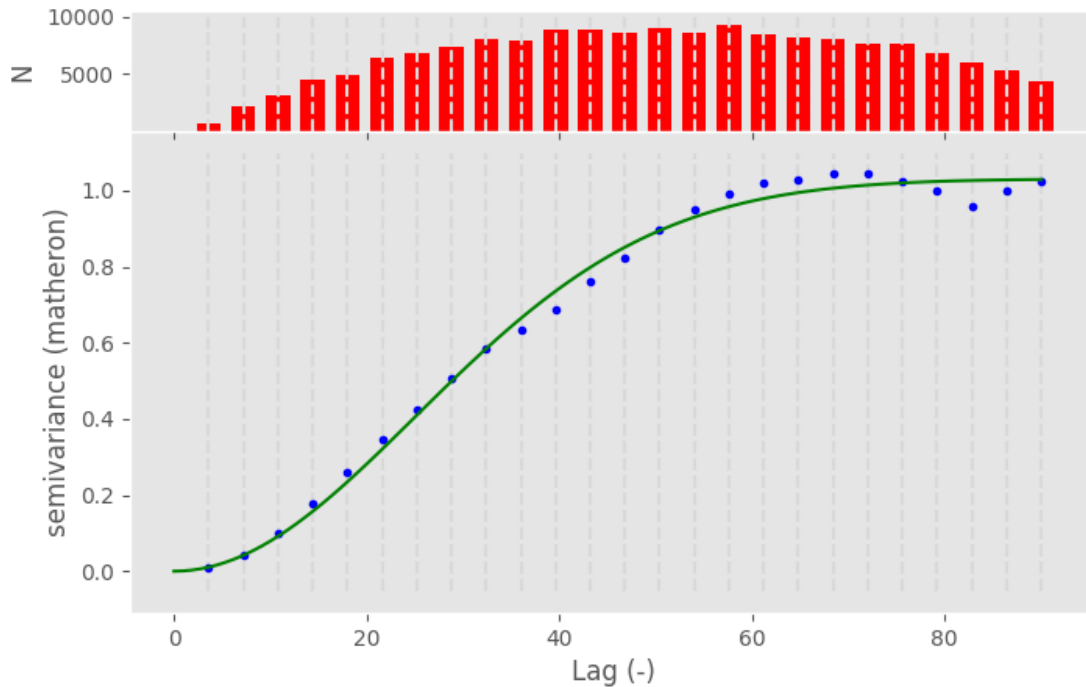
```
In [35]: data = pd.read_csv('data/sample_lr.csv')

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

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')
```



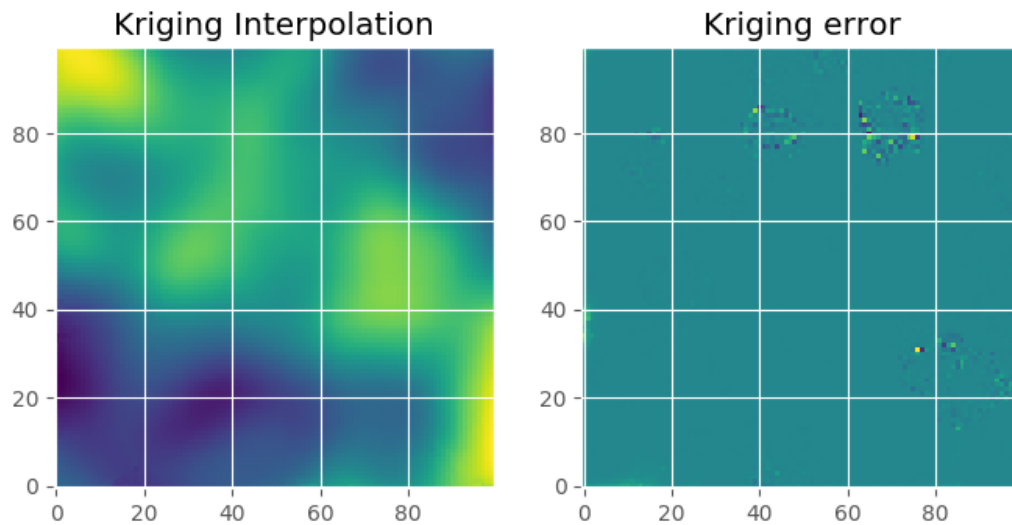
The *OrdinaryKriging* class need at least a fitted *Variogram* instance. Using *min_points* we can demand the Kriging equation system to be build upon at least 5 points to yield robust results. If not enough close observations are found within the effective range of the variogram, the estimation will 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 unobserved 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 01 - 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:

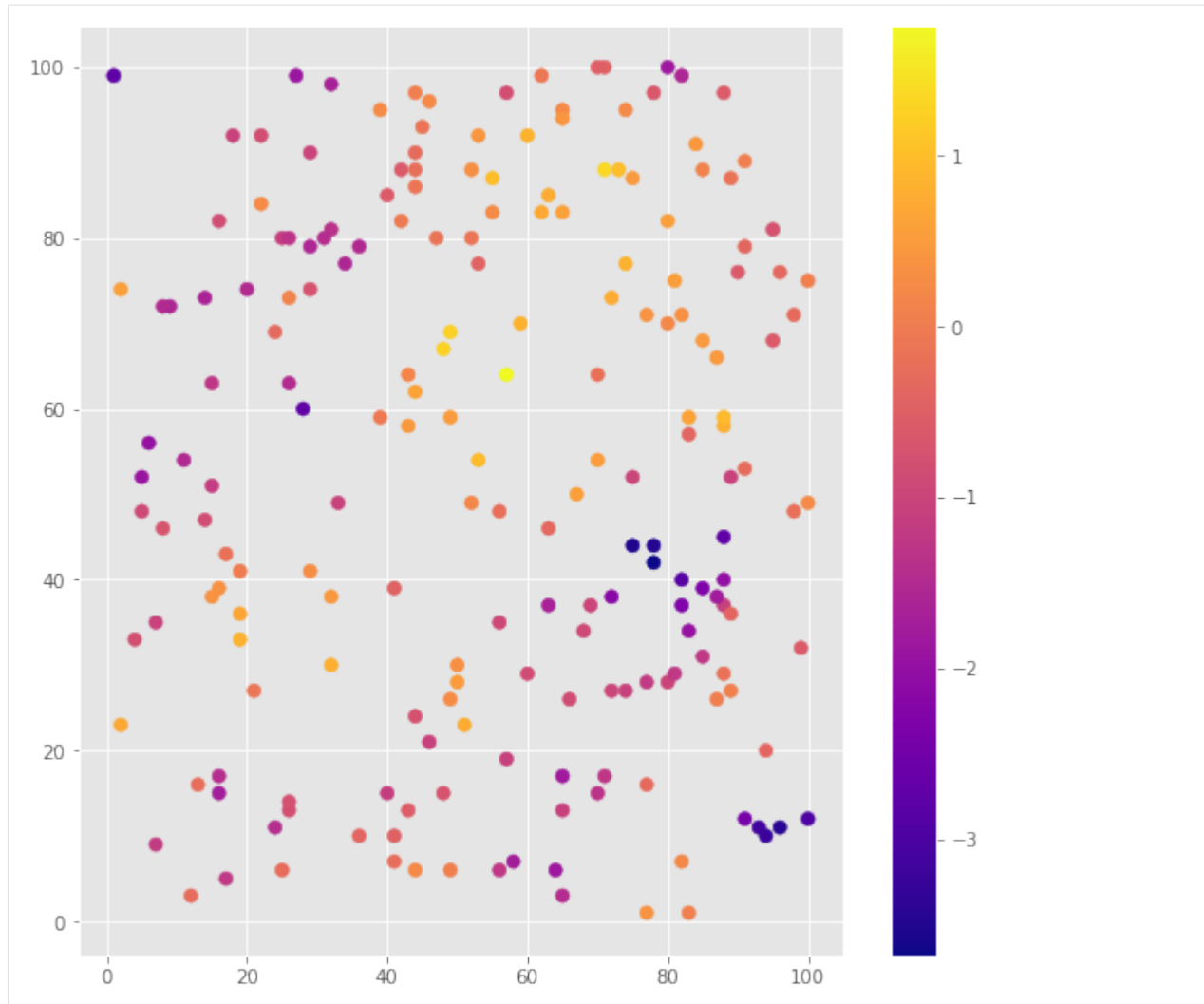
```
[4]: data = pd.read_csv('./data/sample_sr.csv')
print("Loaded %d rows and %d columns" % data.shape)
data.head()
```

```
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

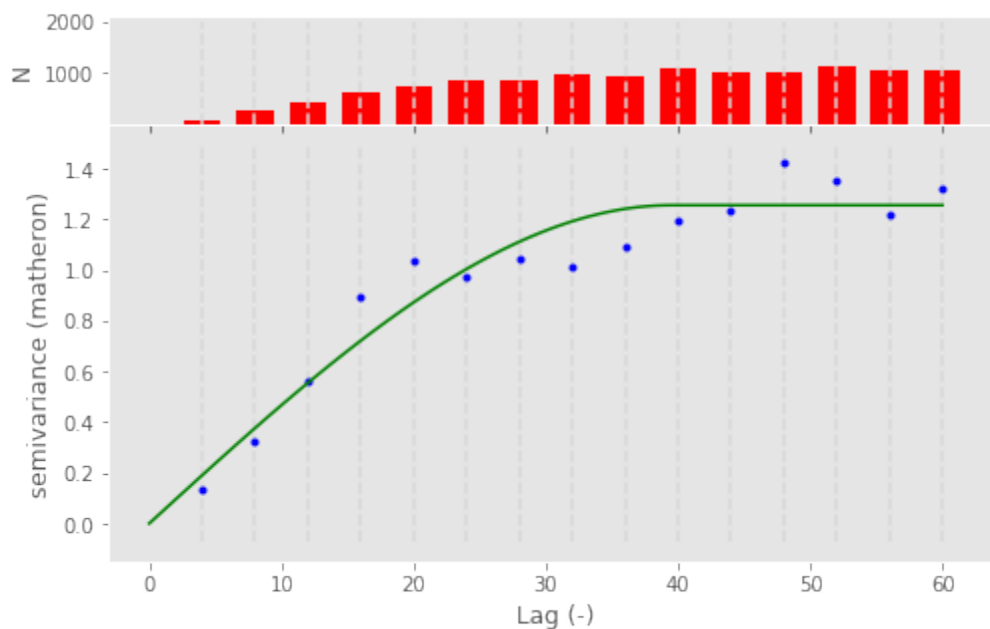
As a quick reminder, the variogram relates pair-wise separating distances of `coordinates` and relates them to the *semi-variance* of the corresponding values pairs. The default estimator used is the Matheron estimator:

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

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 object 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')
```

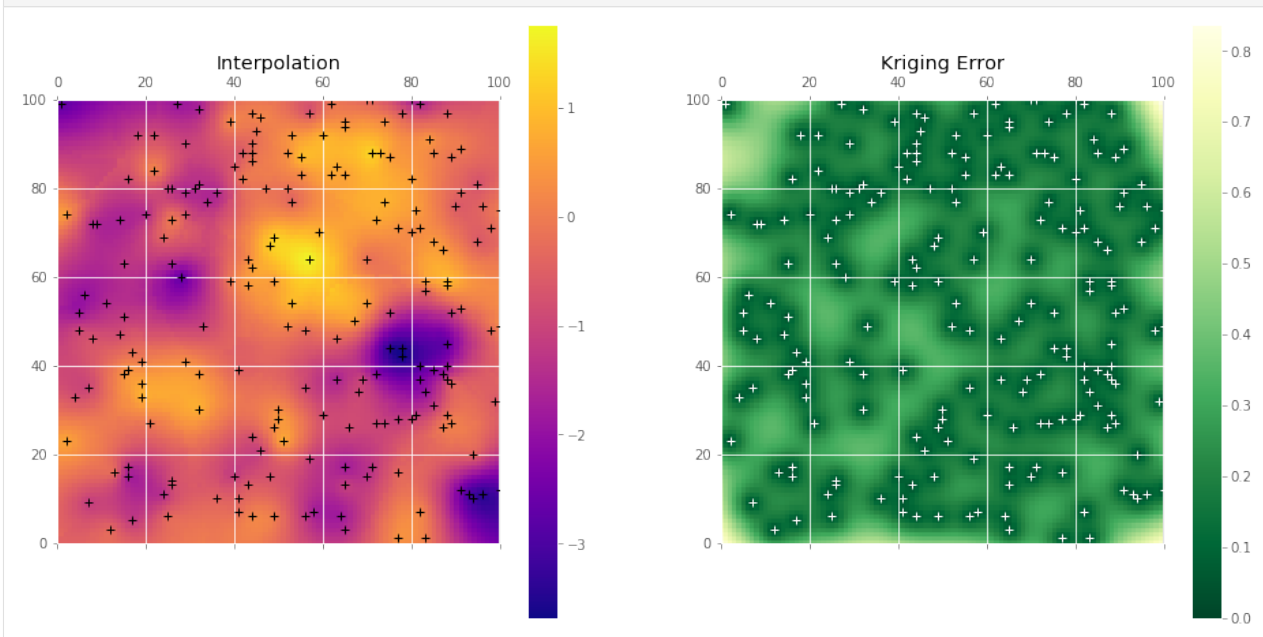
The `transform` method will apply the interpolation for passed arrays of coordinates. It requires each dimension as a single 1D array. We can easily build a meshgrid of 100x100 coordinates and pass them to the interpolator. To receive a 2D result, we can simply reshape the result. The Kriging error will 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 02 - 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 appropriate 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
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

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

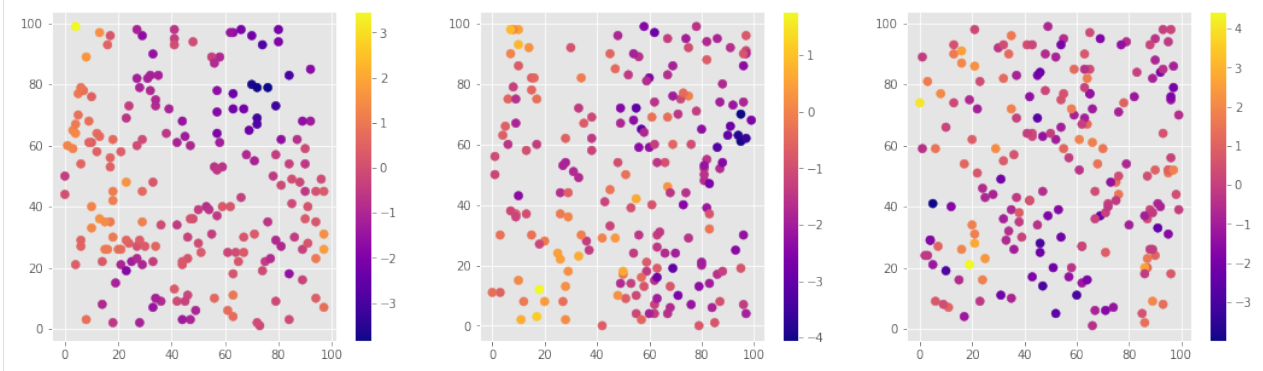
2.1 Load data

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)

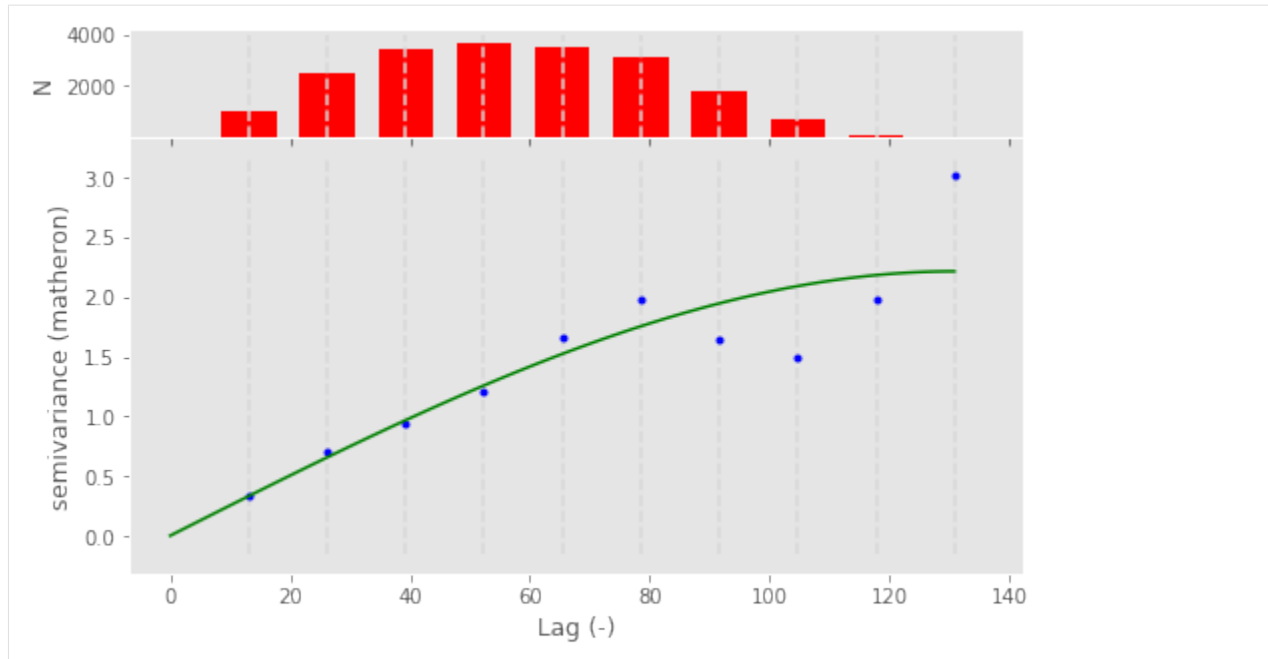
[5]: fig, axes = plt.subplots(1, 3, figsize=(18, 5))
for data, ax in zip((data1, data2, data3), axes.flatten()):
    plot_scatter(data, ax)
```



2.2 Comparing theoretical models

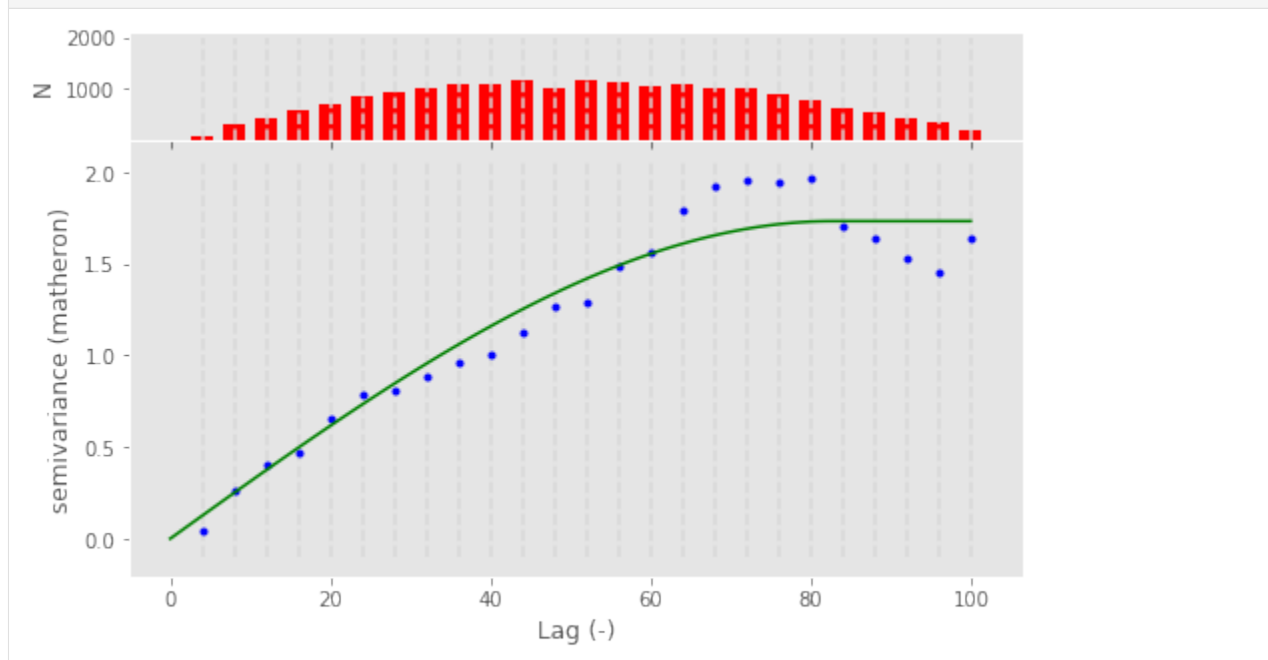
One of the features of `scikit-gstat` is the fact that it is programmed object oriented. That means, we can just instantiate a `Variogram` object and start changing arguments until it models spatial dependency in our observations well.

```
[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);
```

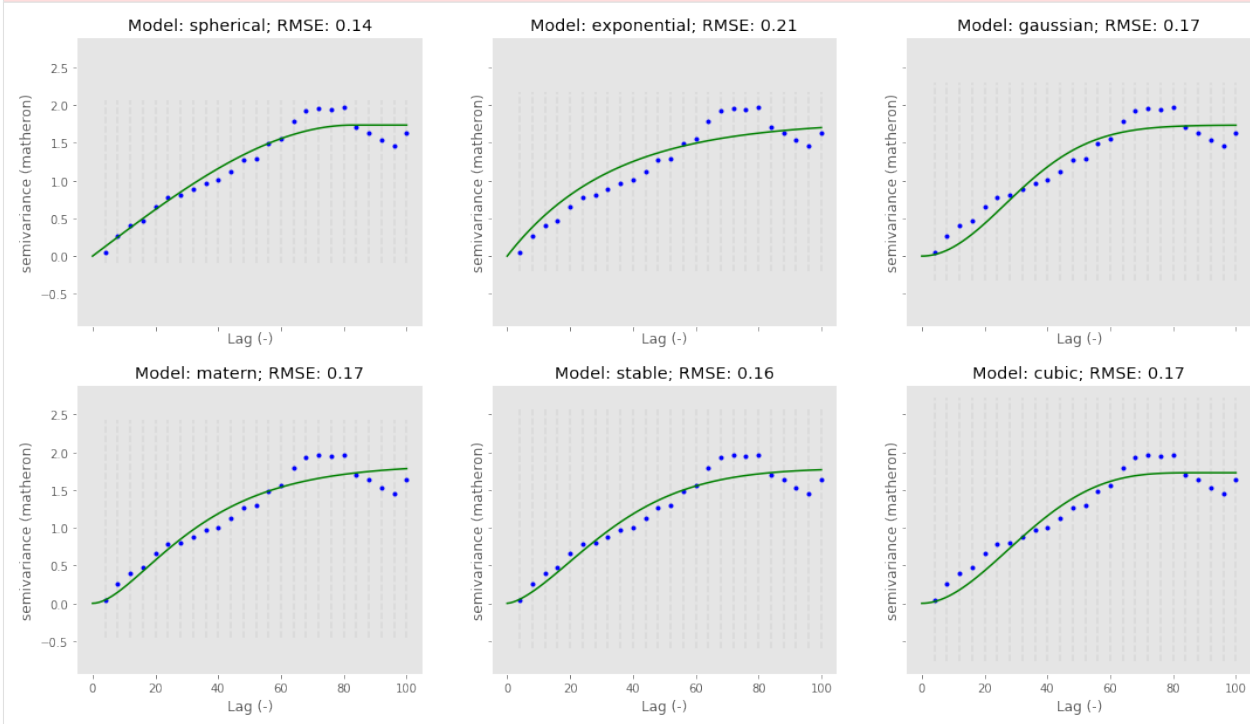


That's not too bad. Now, we can try different theoretical models. It is always a good idea to judge the fit visually, Especially, because we want it to fit to close bins more accurately than to distant bins, as they will ultimately determine the Kriging weights. Nevertheless, `Variogram` has a `rmse` 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)
```

(continues on next page)

(continued from previous page)

```

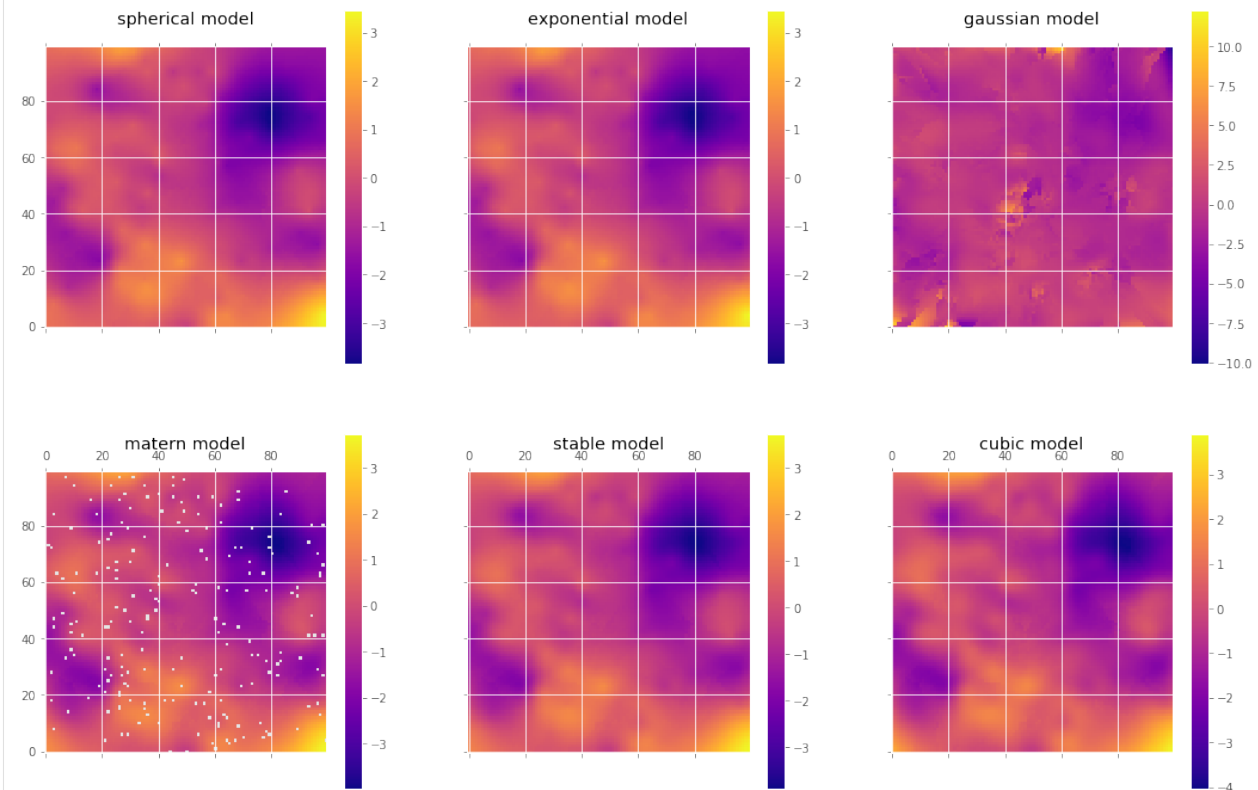
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
    ↪ 'cubic')):
    V1.model = model
    fields.append(interpolate(V1, axes[i]))

```

```

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
count  10000.000000
mean    -0.412559
std       1.165185

```

(continues on next page)

(continued from previous page)

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

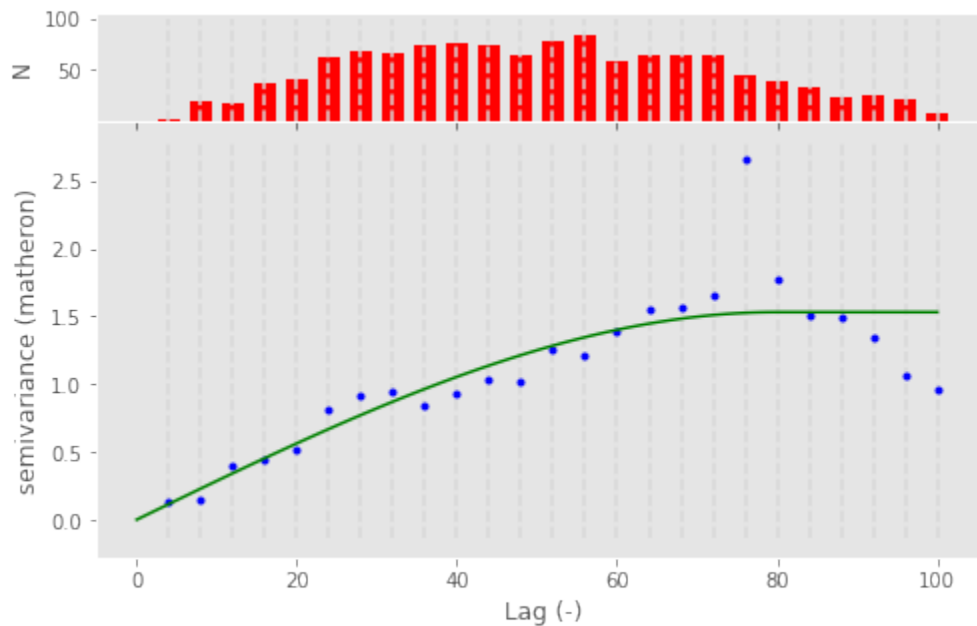
You have to remind that we had quite some observations. How does that look like, when the number of observations is decreased?

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);
```



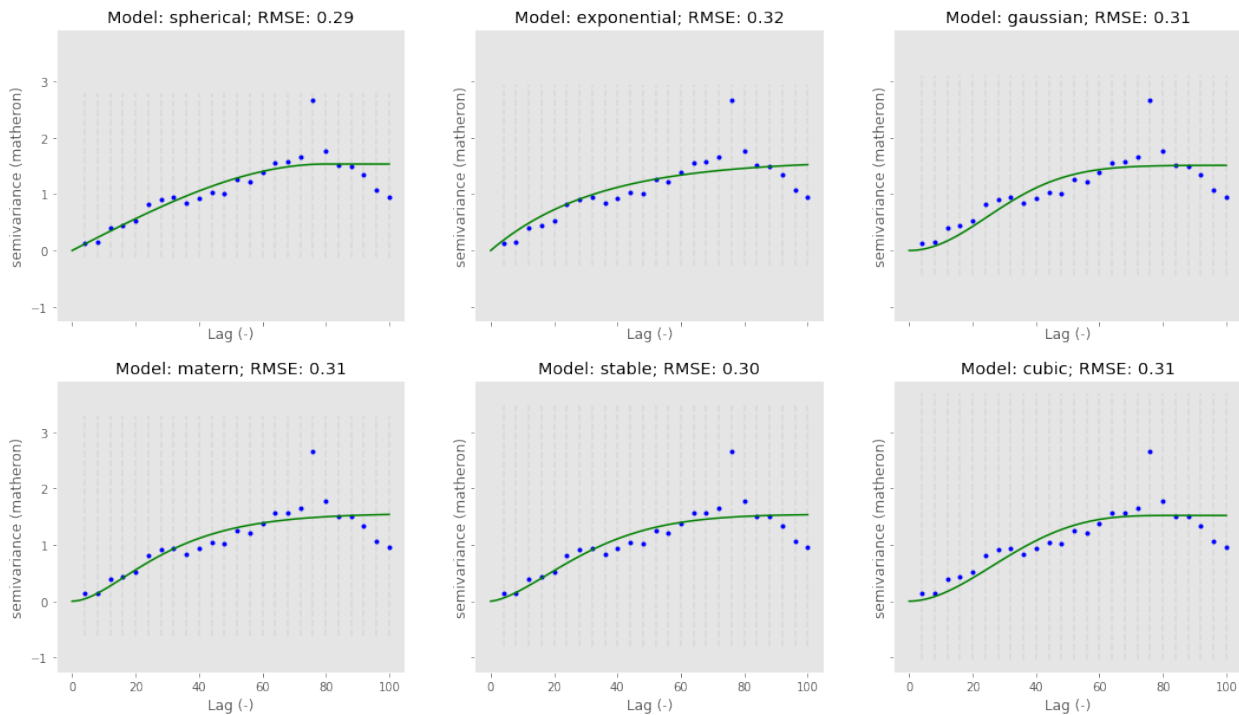
```
[13]: fig, _a = plt.subplots(2,3, figsize=(18, 10), sharex=True, sharey=True)
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
↪ 'cubic')):
```

(continues on next page)

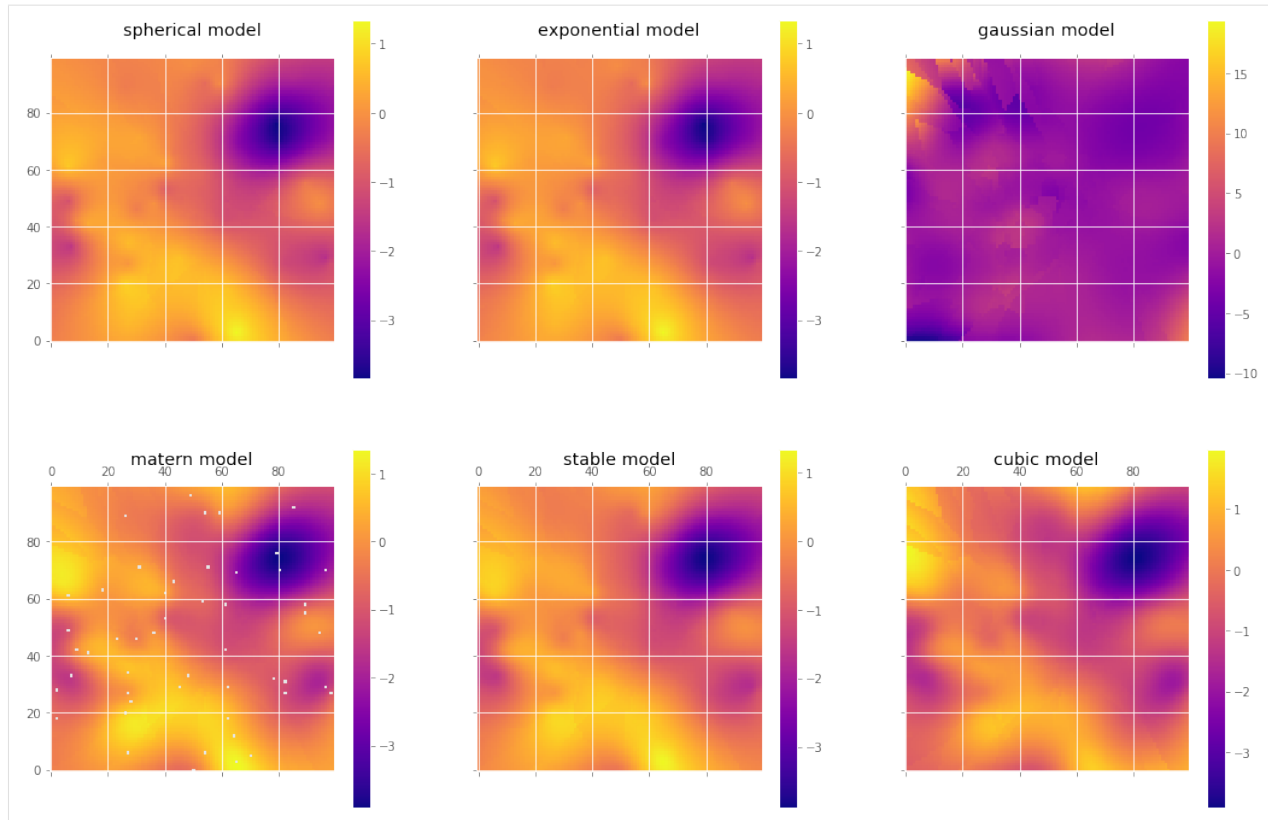
(continued from previous page)

```
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 \
count	10000.000000	10000.000000	10000.000000	9950.000000	10000.000000
mean	-0.492584	-0.498393	-0.229285	-0.474786	-0.482607
std	0.933770	0.913543	2.596812	1.022044	0.987613
min	-3.841342	-3.841342	-10.442055	-3.908236	-3.883908
25%	-0.968383	-0.961082	-1.519022	-1.008619	-0.981728
50%	-0.250866	-0.263681	-0.489828	-0.286338	-0.271219
75%	0.179633	0.159811	0.737149	0.265865	0.228785
max	1.331839	1.331839	19.360127	1.346393	1.331839

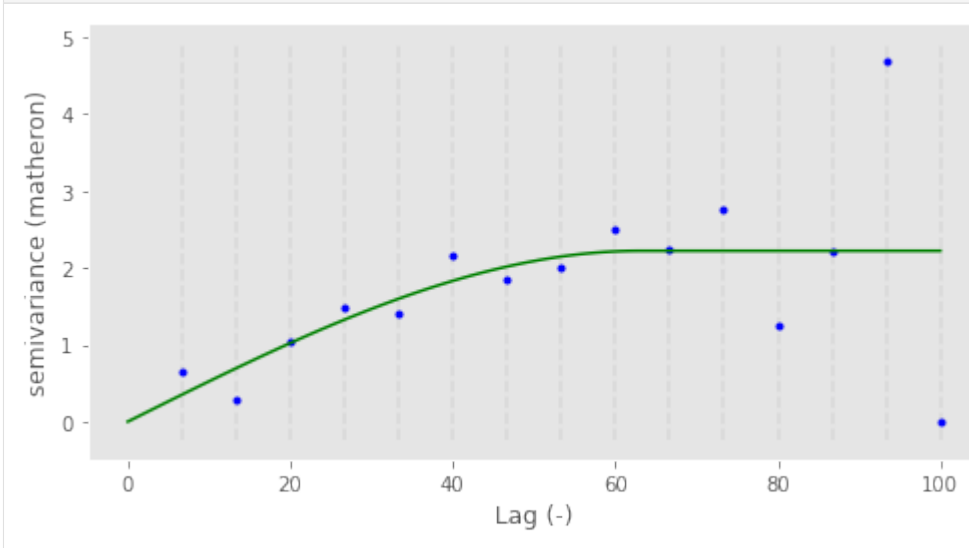
	cubic
count	10000.000000
mean	-0.436488
std	1.093714
min	-3.907538
25%	-1.026079
50%	-0.307099
75%	0.352761
max	1.973759

- 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.

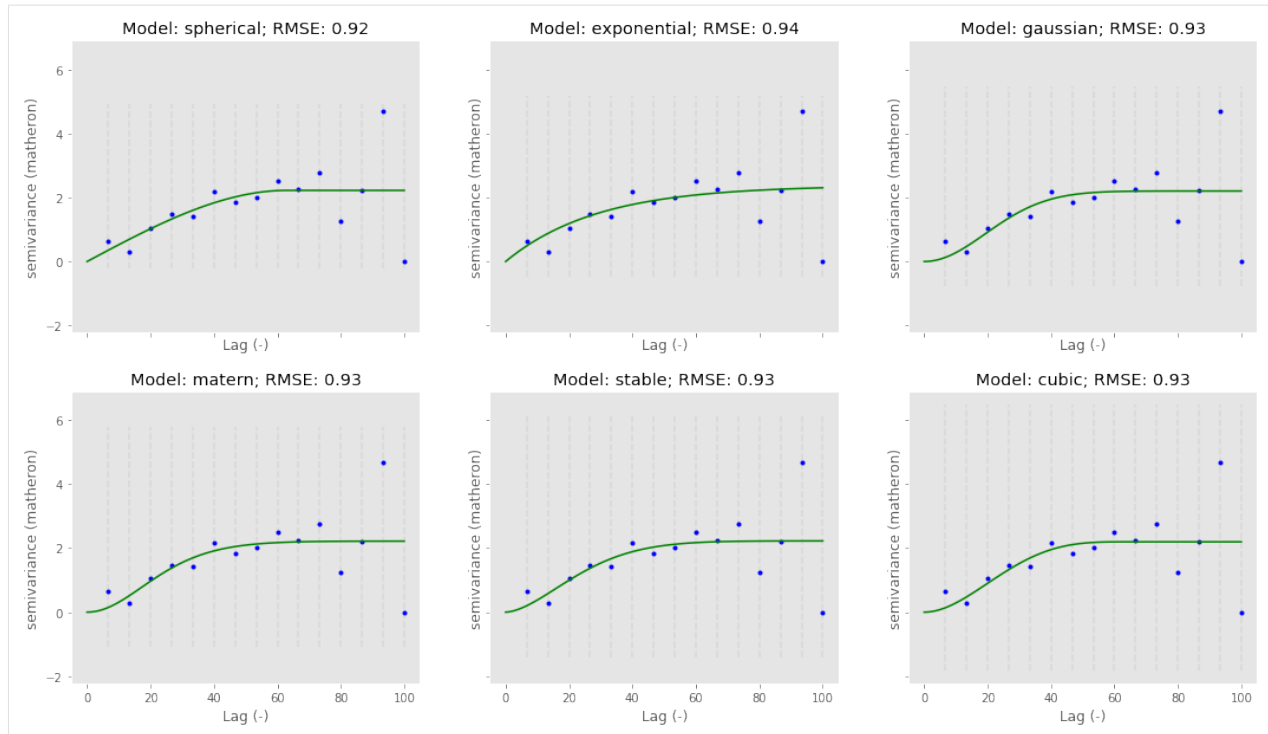
2.3.1 10% of all observations

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);
```



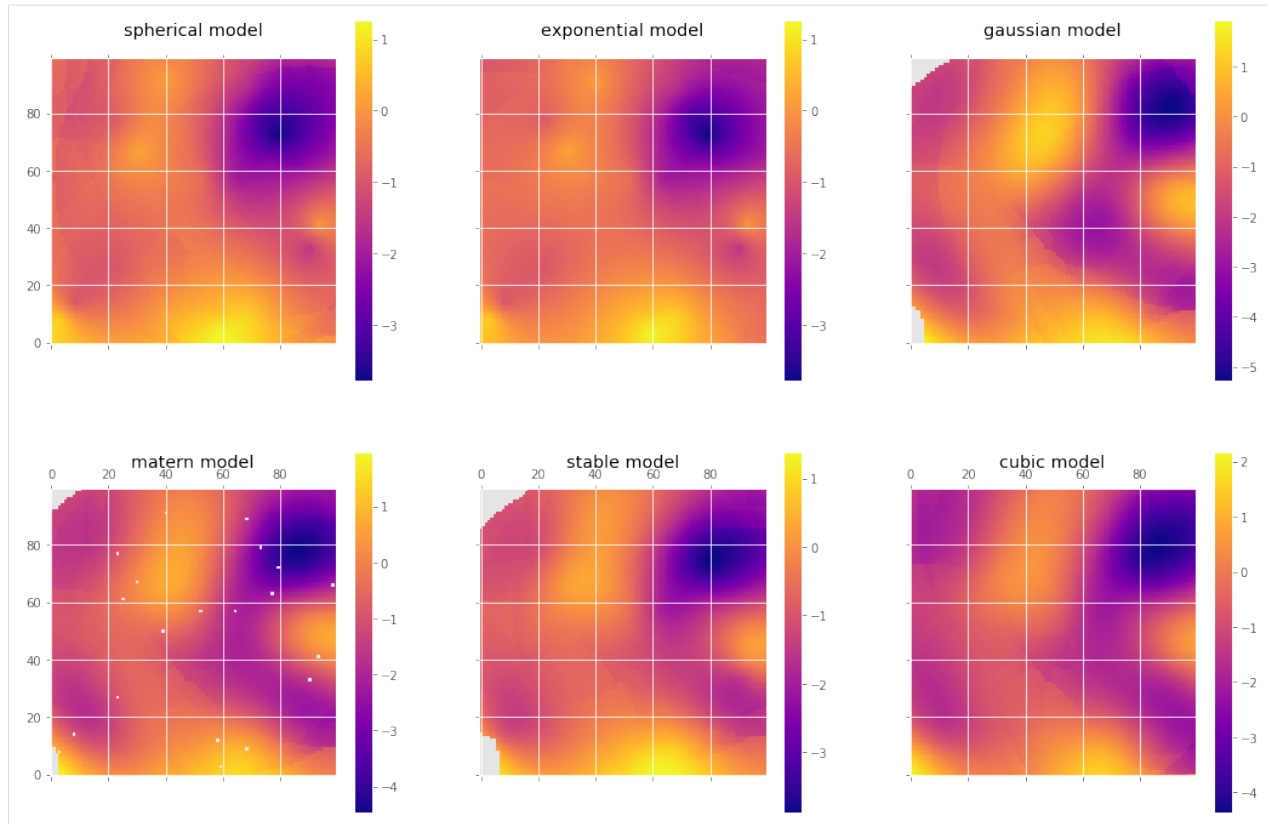
```
[17]: fig, _a = plt.subplots(2,3, figsize=(18, 10), sharex=True, sharey=True)
axes = _a.flatten()
for i, model in enumerate(('spherical', 'exponential', 'gaussian', 'matern', 'stable',
                           ↪ 'cubic')):
    V3.model = model
    V3.plot(axes=axes[i], hist=False, show=False)
    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 \
count	10000.000000	10000.000000	9856.000000	9904.000000	9786.000000
mean	-0.808890	-0.793323	-1.094349	-0.994542	-0.899081
std	0.887960	0.820827	1.366513	1.127764	0.948683
min	-3.785324	-3.785324	-5.277685	-4.458332	-3.880787
25%	-1.063349	-1.033391	-1.904575	-1.629145	-1.314486
50%	-0.671072	-0.621459	-0.963145	-0.899756	-0.801062
75%	-0.288246	-0.334686	-0.125231	-0.207966	-0.277770
max	1.259305	1.259305	1.914923	1.960789	1.380699


```

      cubic
count  10000.000000
mean   -1.058428
std     1.124708
min     -4.364176
25%    -1.698163
50%    -1.004205
75%    -0.275732
max     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 observations 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 03 - 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
```

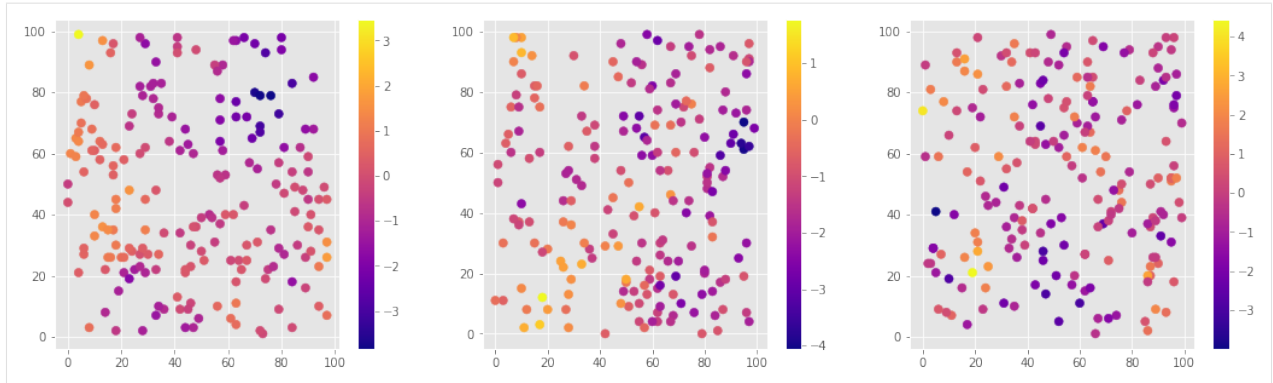
3.1 Load data

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 third 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

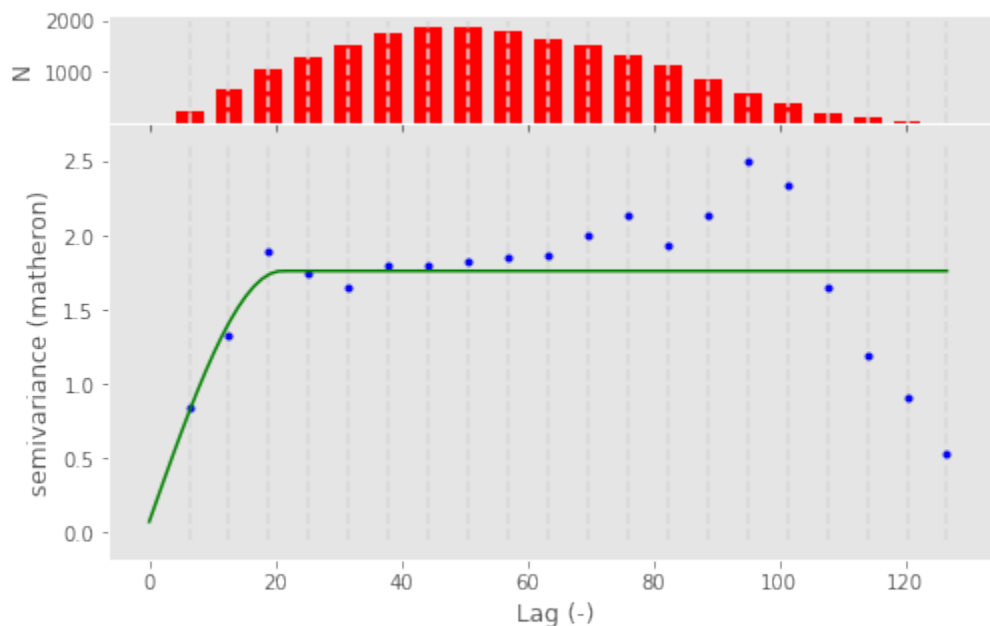
The default estimator configured in `Variogram` is the Math  ron estimator (Math  ron, 1963). It is defined like:

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

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

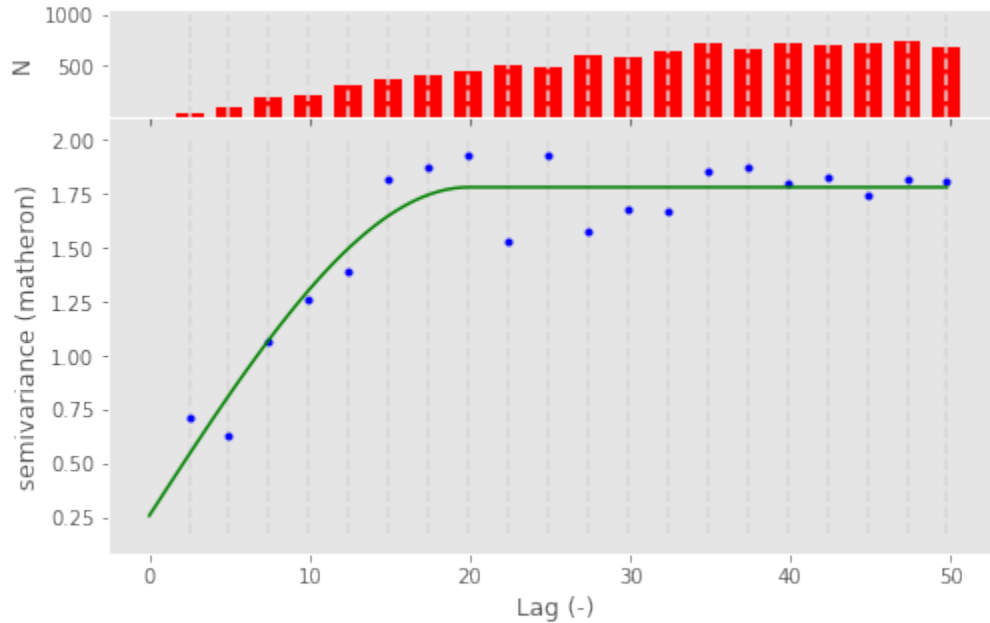
```
[6]: V1 = Variogram(data3[['x', 'y']].values, data3.z.values, normalize=False, n_lags=20,
    ↪ use_nugget=True)
    V1.plot(show=False);
```



Following the histogram, we should set a `maxlag`. This property accepts a number $0 < \text{maxlag} < 1$ 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:

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

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

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

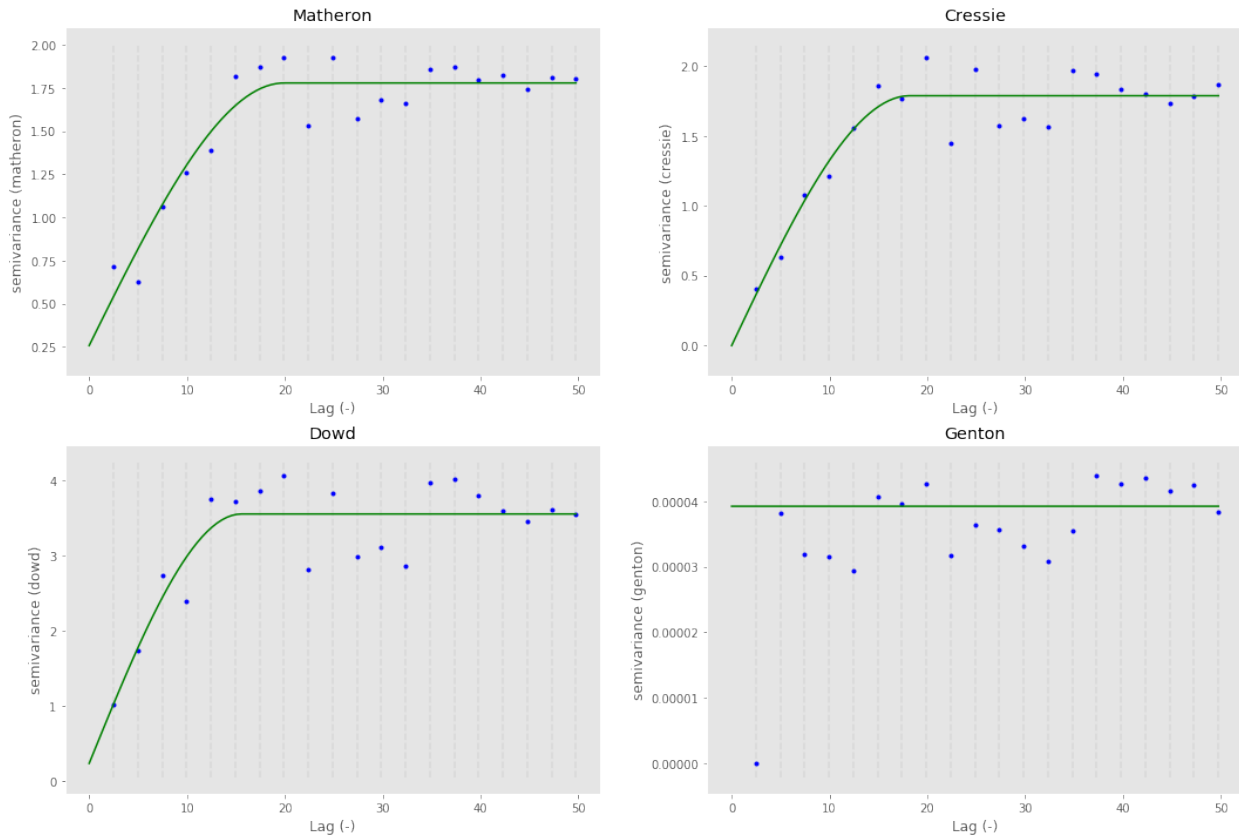
with:

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

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.estimate = est
         V1.plot(axes=ax, hist=False, show=False)
         ax.set_title(est.capitalize())
```

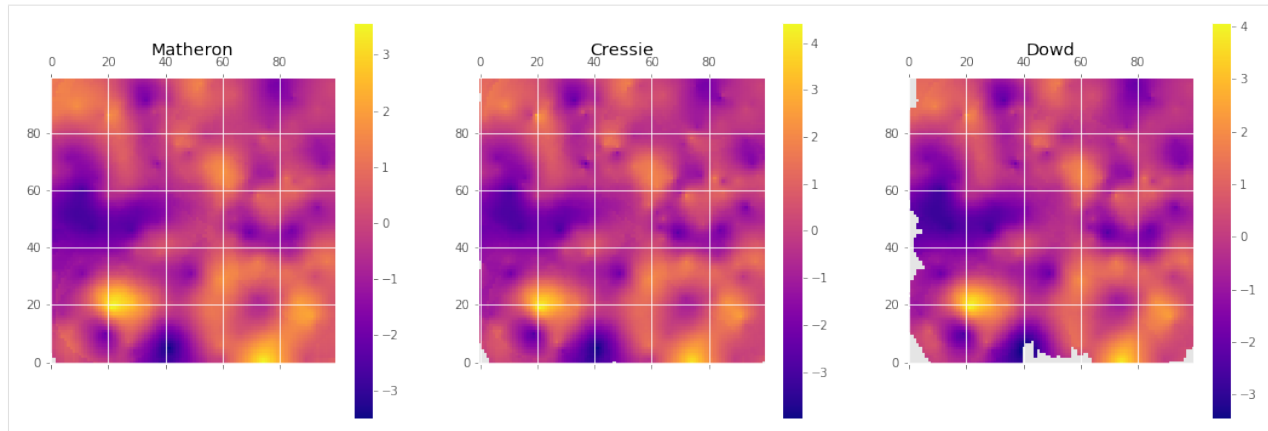


For Kriging, the difference on the first few lag classes is important, as no points will be used for estimation, that lies outside the range.

```
[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.estimate = 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())
```

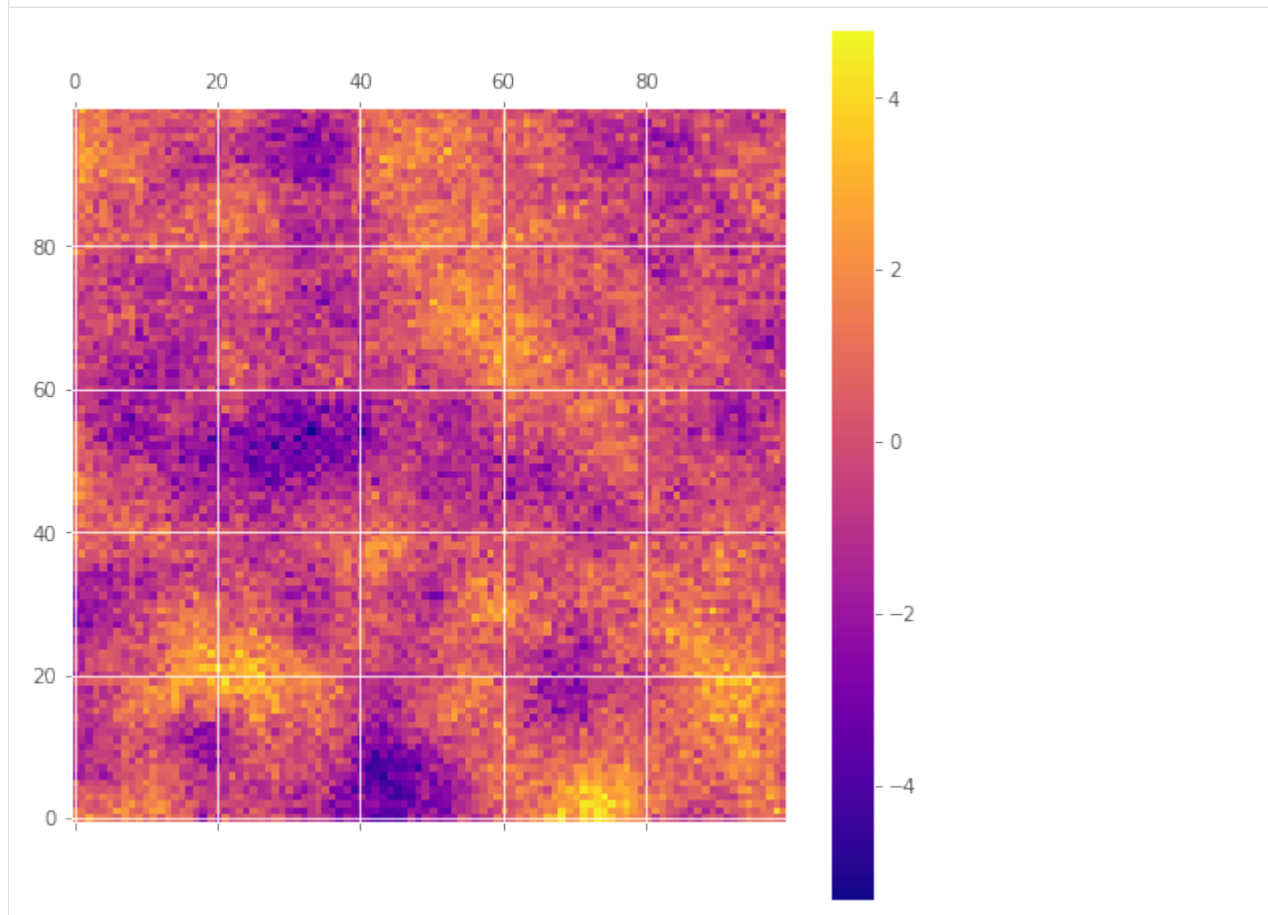
Warning: for 6 locations, not enough neighbors were found within the range.
Warning: for 27 locations, not enough neighbors were found within the range.
Warning: for 286 locations, not enough neighbors were found within the range.



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.

2.4.4 3.X 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>

2.5 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.

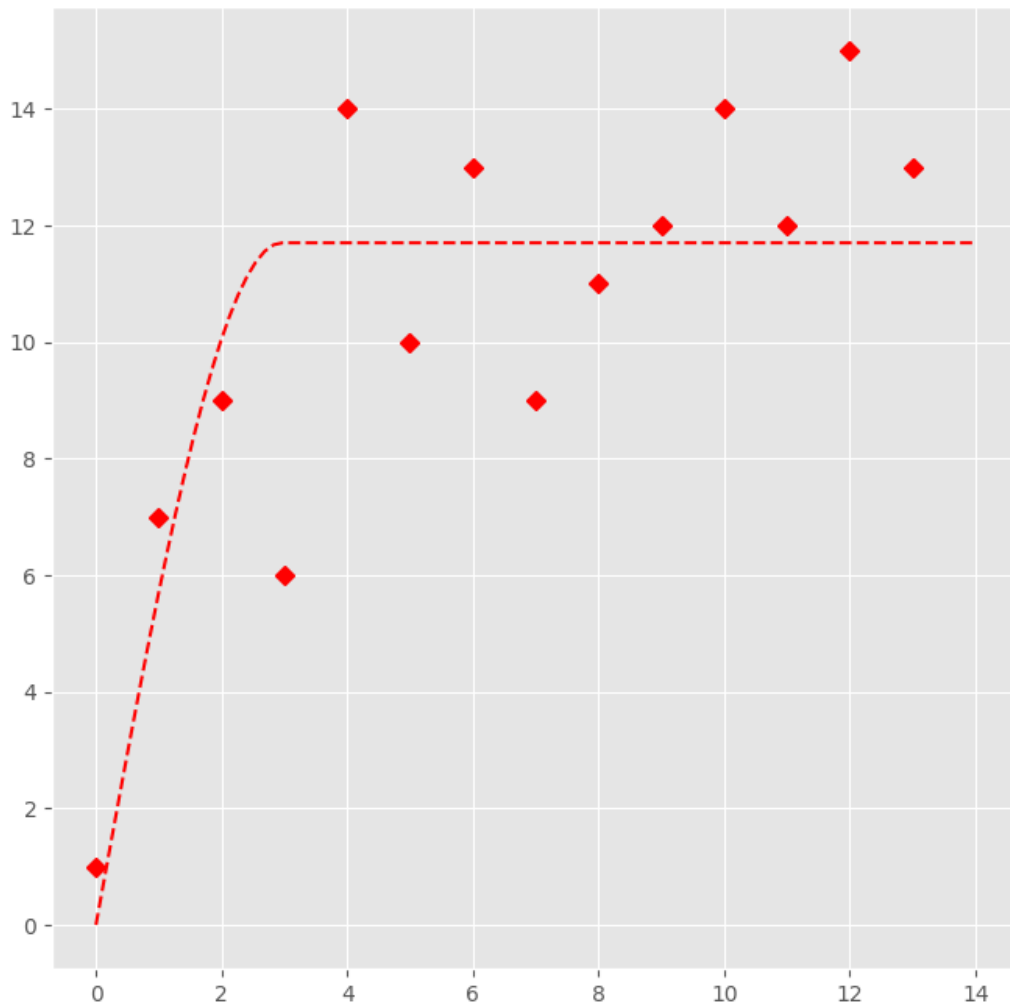
```
In [1]: from scipy.optimize import curve_fit
In [2]: import matplotlib.pyplot as plt
In [3]: plt.style.use('ggplot')
In [4]: import numpy as np
In [5]: from skgstat.models import spherical
```

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 0x7fa917b0fb00>]  
  
In [13]: plt.plot(xi, yi, '--r')  
Out[13]: [<matplotlib.lines.Line2D at 0x7fa917b0ff98>]
```



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='trf'* 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 0x7fa915982860>]

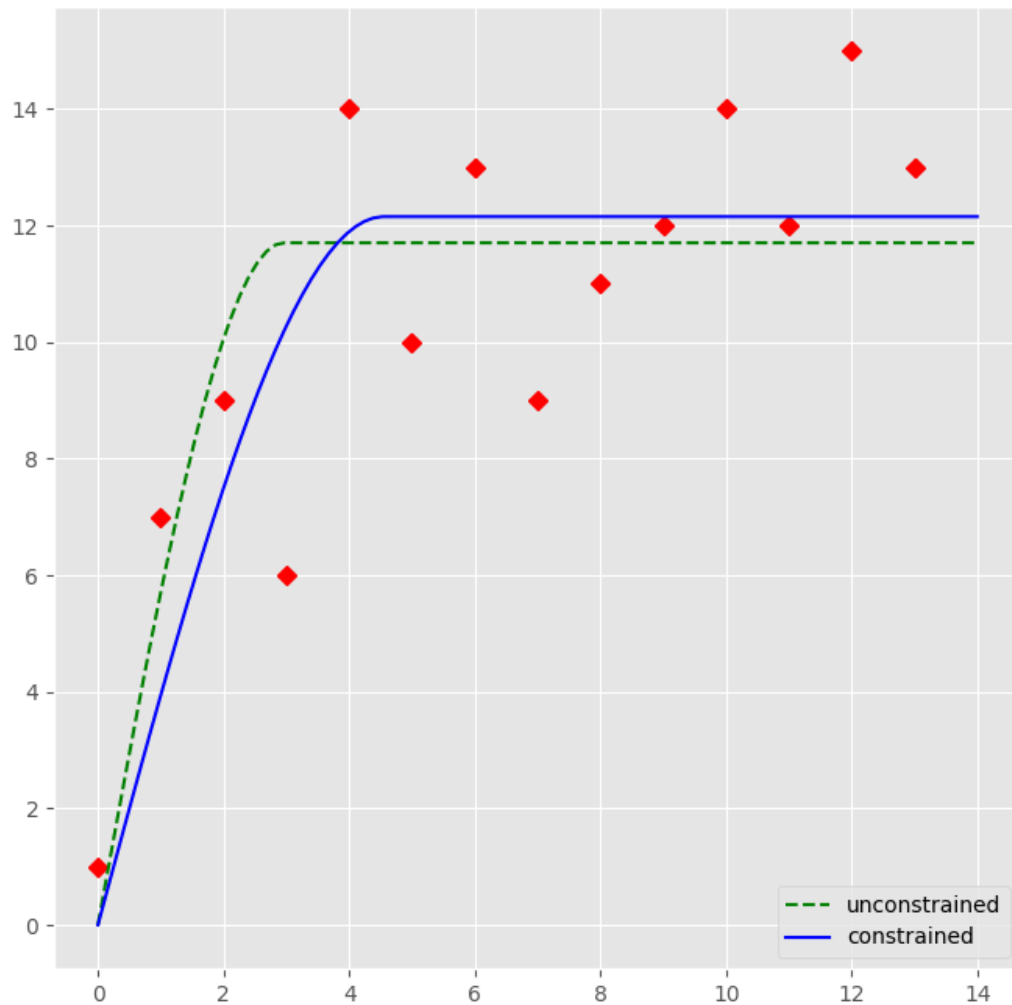
In [15]: plt.plot(xi, yi, '--g', label='unconstrained')
Out[15]: [<matplotlib.lines.Line2D at 0x7fa915982c50>]

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 0x7fa915992048>]

In [19]: plt.legend(loc='lower right')
Out[19]: <matplotlib.legend.Legend at 0x7fa9159600f0>
```



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 $\sigma = [0.1, 0.2, \dots 0.9]$ 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')

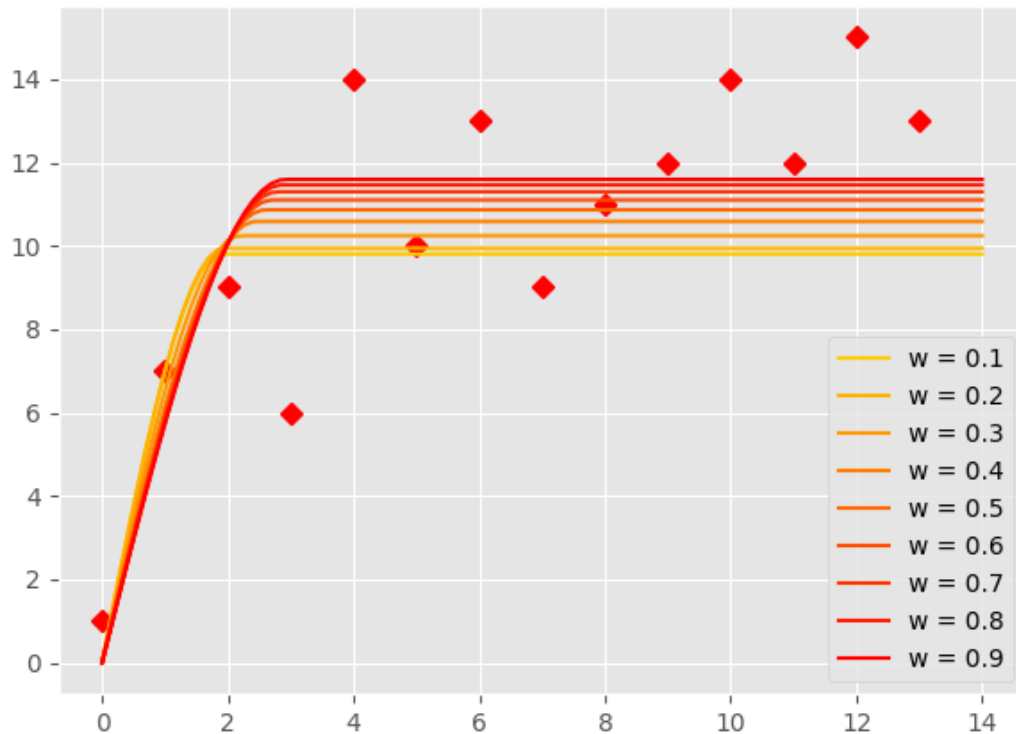
In [25]: sigma = np.ones(len(x))

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

In [27]: ax.plot(x, y, 'rD')
Out[27]: [<matplotlib.lines.Line2D at 0x7fa9187106d8>]

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 0x7fa917e58198>
```



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 avoid zeros
In [31]: X = np.asarray([(x + 1) for x 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 0x7fa915a76630>]
```

```
In [39]: for i in range(4):
```

```
.....:     cof, cov = curve_fit(f, x, y, sigma=s[i], p0=(6.,14.), bounds=(0,(14,
```

```
.....:     ↪14)))
```

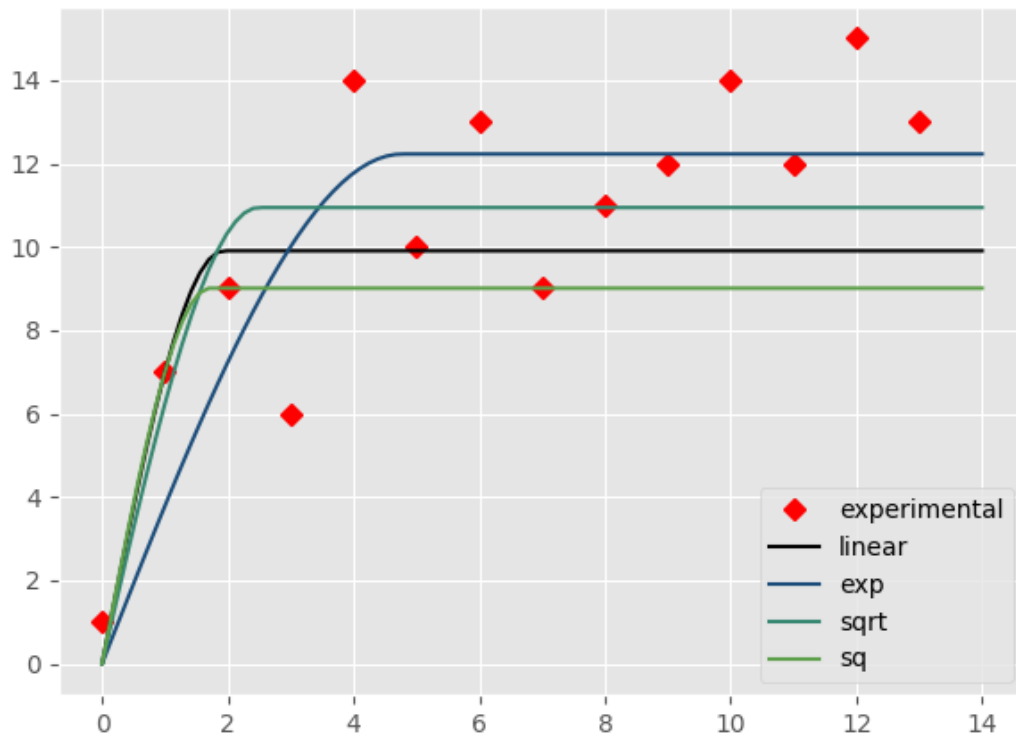
```
.....:     yi = list(map(lambda x: spherical(x, *cof), xi))
```

```
.....:     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 0x7fa917b57588>
```



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 current search area. As the search area is specific to the current poi, it has to be defined as the index of the coordinate to be used. The method is called *search_area*. Using random coordinates, the search area shapes are presented 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)
```

(continues on next page)

(continued from previous page)

```

In [6]: coords = np.random.gamma(15, 6, (40, 2))

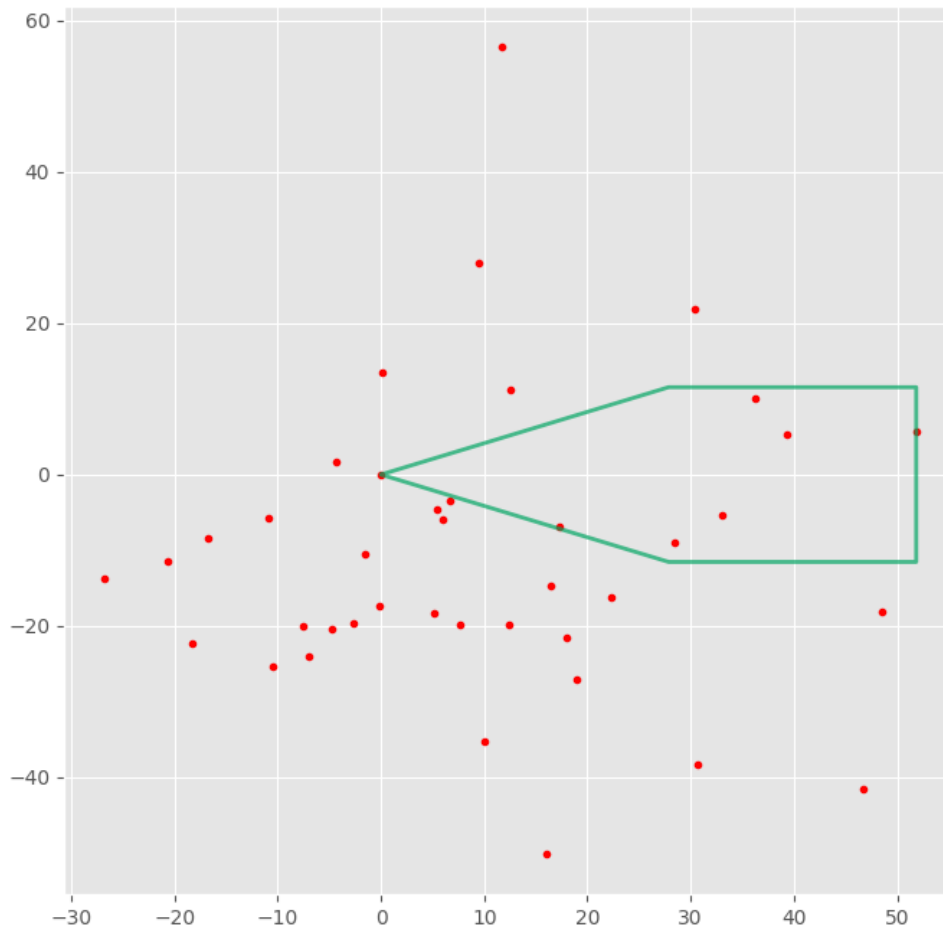
In [7]: np.random.seed(42)

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.search_area(poi=3)
Out[10]: <Figure size 800x800 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))

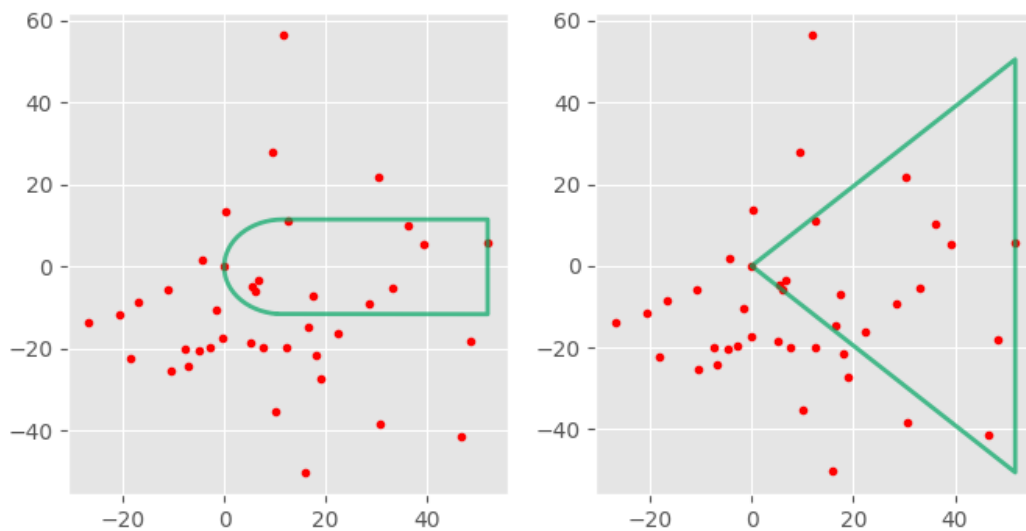
In [12]: DV.set_directional_model('circle')

In [13]: DV.search_area(poi=3, ax=axes[0])
Out[13]: <Figure size 800x400 with 2 Axes>

In [14]: DV.set_directional_model('compass')

In [15]: DV.search_area(poi=3, ax=axes[1])
Out[15]: <Figure size 800x400 with 2 Axes>

In [16]: fig.show()
```



Local Reference System

In order to apply different search area shapes and rotate them considering the given azimuth, a few preprocessing steps are necessary. This can lead to some calculation overhead, as in the case of a *compass* model. The selection of point pairs being directional is implemented by transforming the coordinates into a local reference system iteratively for each coordinate. For multidimensional coordinates, only the first two dimensions are used. They are shifted to make the current point of interest the origin of the local reference system. Then all other points are rotated until the azimuth overlays the local x-axis. This makes the definition of different shapes way easier. In this local system, the bandwidth can easily be applied to the transformed y-axis. The *set_directional_model* can also set a custom function as search area shape, that accepts the current local reference system and returns the search area for the given poi. The search area has to be returned as a shapely Polygon. Unfortunately, the *tolerance* and *bandwidth* parameter are not passed yet.

Note: For the next release, it is planned to pass all necessary parameters to the directional model function. This should greatly improve the definition of custom shapes. Until the implementation, the parameters have to be injected directly.

The following example will illustrate the rotation of the local reference system.

```

In [17]: from matplotlib.patches import FancyArrowPatch as arrow

In [18]: np.random.seed(42)

In [19]: c = np.random.gamma(10, 6, (9, 2))

In [20]: mid = np.array([[np.mean(c[:,0]), np.mean(c[:,1])]])

In [21]: coords = np.append(mid, c, axis=0) - mid

In [22]: DV = DirectionalVariogram(coords, vals[:10],
....:     azimuth=45, tolerance=45)
....:

In [23]: loc = DV.local_reference_system(poi=0)

In [24]: fig, ax = plt.subplots(1, 1, figsize=(6, 6))

In [25]: ax.scatter(coords[:,0], coords[:,1], 15, c='b')
Out[25]: <matplotlib.collections.PathCollection at 0x7fa917ba0d30>

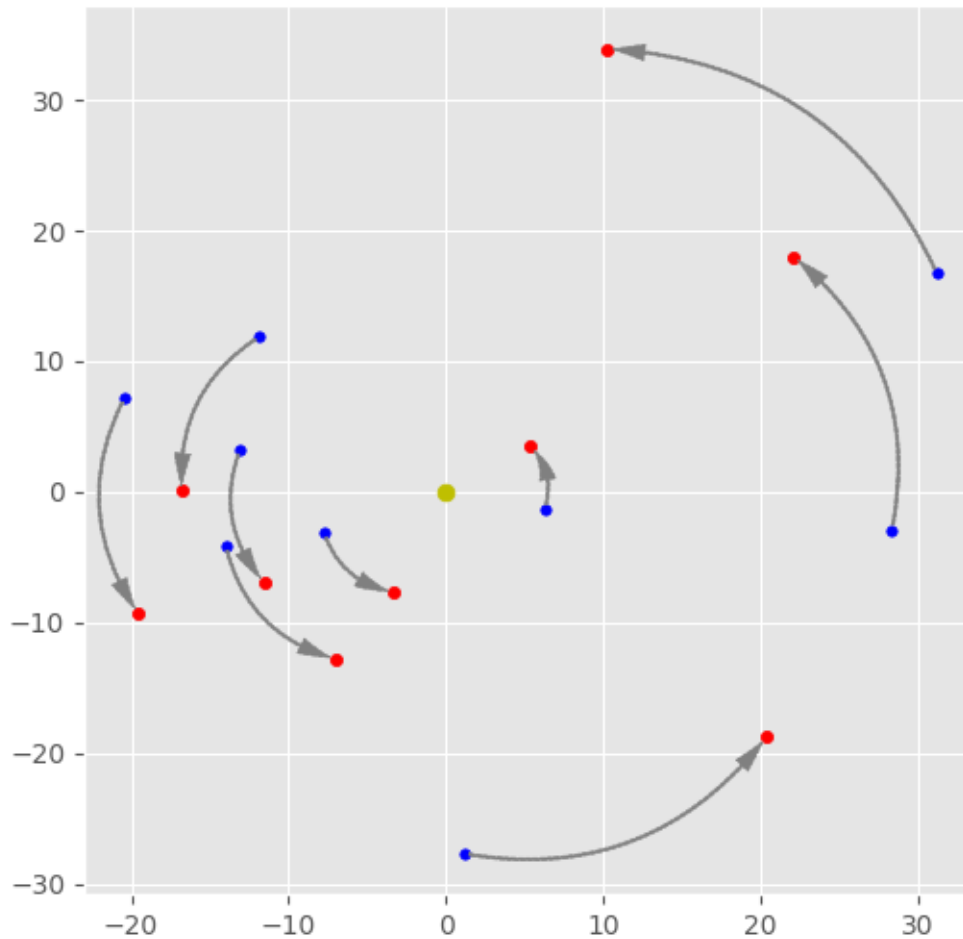
In [26]: ax.scatter(loc[:,0], loc[:,1], 20, c='r')
Out[26]: <matplotlib.collections.PathCollection at 0x7fa917ba0f98>

In [27]: ax.scatter([0], [0], 40, c='y')
Out[27]: <matplotlib.collections.PathCollection at 0x7fa917ba02e8>

In [28]: for u,v in zip(coords[1:], loc[1:]):
....:     arrowstyle="Simple,head_width=6,head_length=12,tail_width=1"
....:     a = arrow(u, v, color='grey', linestyle='--',
....:         connectionstyle="arc3, rad=.3", arrowstyle=arrowstyle)
....:     ax.add_patch(a)
....:

In [29]: fig.show()

```



After moving and shifting, any type of Geometry could be generated and passed as the search area.

```
In [30]: from shapely.geometry import Polygon
```

```
In [31]: def M(loc):
...:     xmax = np.max(loc[:,0])
...:     ymax = np.max(loc[:,1])
...:     return Polygon([
...:         (0, 0),
...:         (0, ymax * 0.6),
...:         (0.05*xmax, ymax * 0.6),
...:         (xmax * 0.3, ymax * 0.3),
...:         (0.55 * xmax, 0.6 * ymax),
...:         (0.6 * xmax, 0.6 * ymax),
...:         (0.6 * xmax, 0),
...:         (0.55 * xmax, 0),
...:         (0.55 * xmax, 0.55 * ymax),
...:         (xmax * 0.325, ymax * 0.275),
```

(continues on next page)

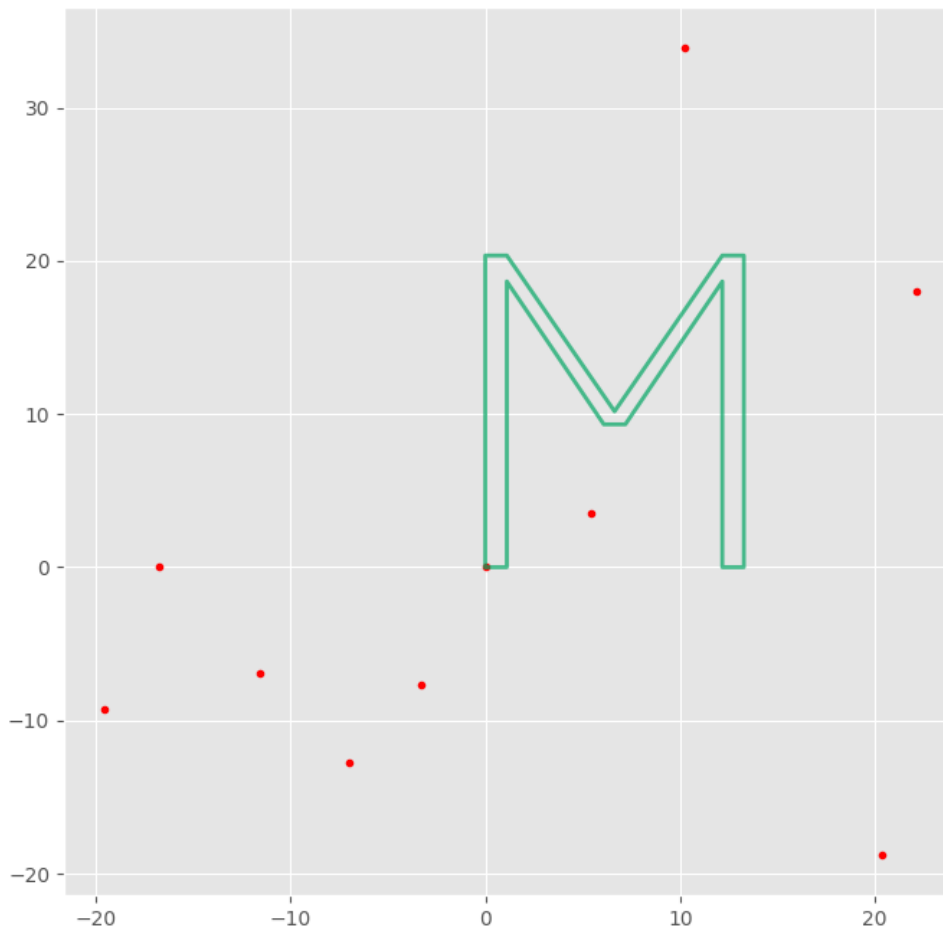
(continued from previous page)

```
.....:      (xmax * 0.275, ymax * 0.275),  
.....:      (0.05 * xmax, 0.55 * ymax),  
.....:      (0.05 * xmax, 0),  
.....:      (0, 0)  
.....:  ])  
.....:
```

In [32]: DV.set_directional_model(M)

In [33]: DV.search_area(poi=0)

Out[33]: <Figure size 800x800 with 1 Axes>



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))

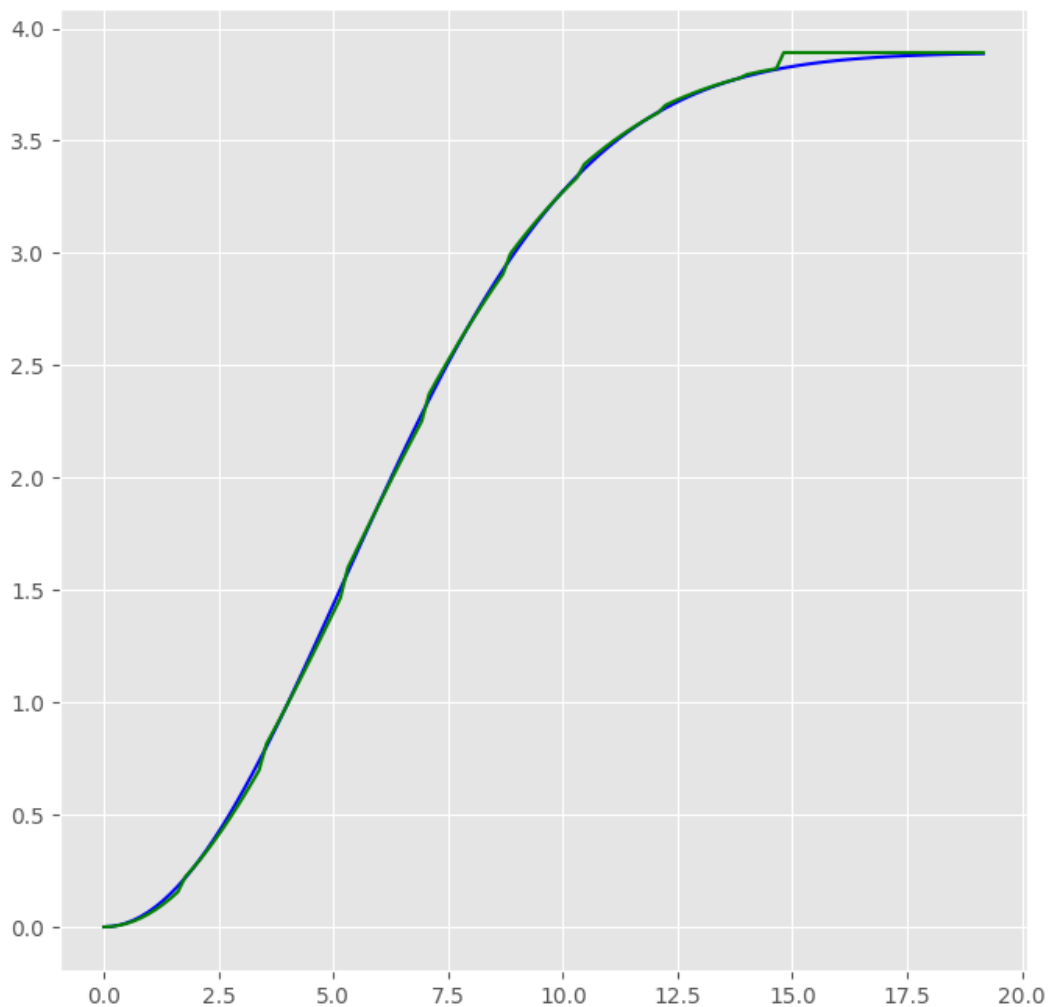
# estimation
In [12]: ok.mode = 'estimate'

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 0x7fa915a0b0b8>]

In [15]: plt.plot(x, y_e, '-g', label='estimated variogram')
Out[15]: [<matplotlib.lines.Line2D at 0x7fa9159e11d0>]

In [16]: plt.legend(loc='lower right')
Out[16]: [<matplotlib.legend.Legend at 0x7fa918714080>]
```



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)

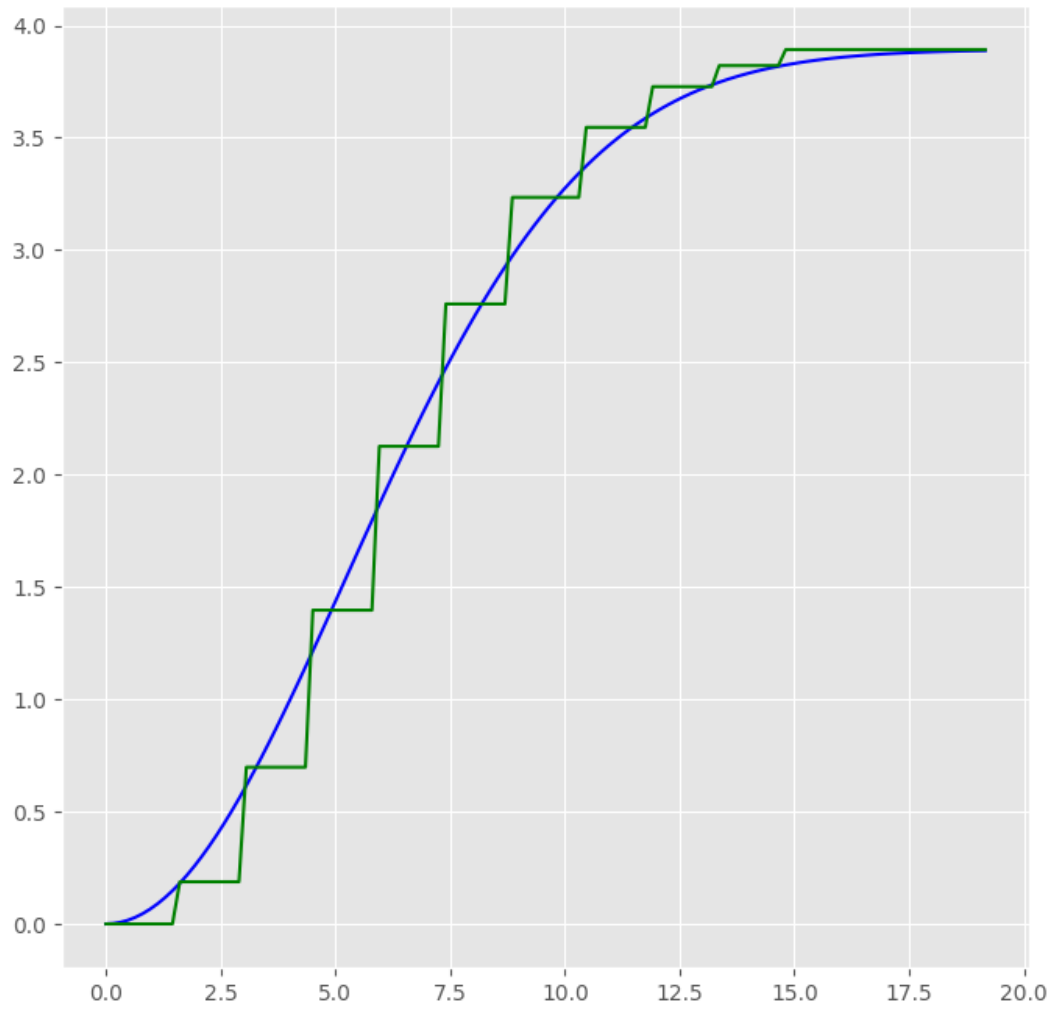
In [19]: plt.plot(x, y_c, '-b')
Out[19]: [<matplotlib.lines.Line2D at 0x7fa918787f28>]

In [20]: plt.plot(x, y_e2, '-g')
```

(continues on next page)

(continued from previous page)

```
Out [20]: [<matplotlib.lines.Line2D at 0x7fa918787da0>]
```



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', nor-  
                        malize=False, fit_method='trf', fit_sigma=None, use_nugget=False,  
                        maxlag=None, n_lags=10, verbose=False)
```

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, n_lags=10, verbose=False)
```

Variogram Class

Note: The directional variogram estimation is not re-implemented yet. Therefore the parameters is-directional, azimuth and tolerance will be ignored at the moment and can be subject to changes.

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 absolute 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*) – String identifying the binning function used to find lag class edges. At the moment there are two possible values: ‘even’ (default) or ‘uniform’. Even will find `n_lags` bins of same width in the interval `[0,maxlag[`. ‘uniform’ will identify `n_lags` bins on the same interval, but with varying edges so that all bins count the same amount of observations.
- **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 unconstrained.
 - ‘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 influence 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 effect 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.

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***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`**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()**

Variogram parameters

Return a dictionary of the variogram parameters.

Returns**Return type** `dict`

distance_difference_plot (*ax=None, plot_bins=True, show=True*)

Raw distance plot

Plots all absolute value differences of all point pair combinations over their separating distance, without sorting them into a lag.

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.estimate` 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.

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', 'trf']`:
 - `lm`: Levenberg-Marquardt algorithms implemented in `scipy.optimize.leastsq` function.
 - `trf`: Trust Region Reflective algorithm implemented in `scipy.optimize.least_squares(method='trf')`
- **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(), scipy.optimize.curve_fit(), scipy.optimize.leastsq(),  
scipy.optimize.least_squares()
```

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']. The parameters described below refer to the setter of this property.

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: $w = w_n^2$

Returns**Return type** void**Notes**

The cost function is defined as:

$$chisq = \sum \frac{r^2}{\sigma}$$

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**lag_classes()**

Iterate over the lag classes

Generates an iterator over all lag classes. Can be zipped with Variogram.bins to identify the lag.

Returns

Return type iterable

lag_groups()

Lag class groups

Returns 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*)

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.

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 dimensionality of the coordinates array. In case 3D coordinates are used, three subplots have to be given.

Returns

Return type `matplotlib.Figure`

property mean_residual

Mean Model residuals

Calculates the mean, absolute deviations between the experimental variogram and theoretical model values.

Returns

Return type `float`

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 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 difference 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 `self.describe` return

Returns

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.

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:

$$RMSE = \sqrt{\frac{\sum_{i=0}^{i=N(x)} (x - y)^2}{N(x)}}$$

set_bin_func (*bin_func*)

Set binning function

Sets a new binning function to be used. The new binning method is set by a string identifying the new function to be used. Can be one of: ['even', 'uniform'].

Parameters **bin_func** (*str*) – Can be one of:

- **'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.

Returns

Return type `void`

See also:

`Variogram.bin_func()`, `skgstat.binning.uniform_count_lags()`, `skgstat.binning.even_width_lags()`

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*)

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.

Parameters **values** (*numpy.ndarray*) –

Returns

Return type `void`

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 calculation, refer to the data function 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()`

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 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.DirectionVariogram(coordinates=None, values=None, estima-  
                                tor='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, ver-  
                                bose=False)
```

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 directional version of a Variogram will only form pairs 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)
```

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 absolute 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*) – String identifying the binning function used to find lag class edges. At the moment there are two possible values: ‘even’ (default) or ‘uniform’. Even will find `n_lags` bins of same width in the interval `[0,maxlag[`. ‘uniform’ will identify `n_lags` bins on the same interval, but with varying edges so that all bins count the same amount of observations.
- **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 unconstrained.
 - ‘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 influence 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, Polygon*) – 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 current local coordinate system and returning a Polygon representing the search area itself can be passed. In this case, the tolerance and bandwidth has to be incorporated by hand into the model. The azimuth is handled by the class. 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.

- **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 effect 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 $0 < \text{maxlag} < 1$, then maxlag is relative and $\text{maxlag} * \max(\text{Variogram.distance})$ 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 $\text{maxlag}=0.5$ 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.

_triangle (*local_ref*)

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 **local_ref** (*numpy.array*) – Array of all coordinates transformed into a local representation with the current point of interest being the origin and the azimuth angle aligned onto the x-axis.

Returns **search_area** – Search Area of triangular shape bounded by the current bandwidth.

Return type Polygon

Notes



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()*

_circle (*local_ref*)

Circular Search Area

Construct a half-circled 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. The radius of the half-circle is set to half the bandwidth.

Parameters **local_ref** (*numpy.array*) – Array of all coordinates transformed into a local representation with the current point of interest being the origin and the azimuth angle aligned onto the x-axis.

Returns **search_area** – Search Area of half-circular shape bounded by the current bandwidth.

Return type Polygon

:raises ValueError : In case the DirectionalVariogram.bandwidth is None or 0.:

See also:

DirectionalVariogram._triangle(), *DirectionalVariogram._compass()*

_compass (*local_ref*)

Compass direction Search Area

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 **local_ref** (*numpy.array*) – Array of all coordinates transformed into a local representation with the current point of interest being the origin and the azimuth angle aligned onto the x-axis.

Returns **search_area** – Search Area of the given compass direction.

Return type Polygon

Notes

The necessary figure is build by searching for the intersection of a half-tolerance angled line with a vertical line at the maximum x-value. Using polar coordinates, these points (positive and negative half-tolerance angle) are the edges of the search area in the local coordinate system. The radius of a polar coordinate can be calculated as:

$$r = \frac{x}{\cos(\alpha/2)}$$

The two bounding points P1 nad P2 (in local coordinates) are then (xmax, y) and (xmax, -y), with xmax being the maximum local x-coordinate representation and y:

$$y = r * \sin\left(\frac{\alpha}{2}\right)$$

See also:

DirectionalVariogram._triangle(), *DirectionalVariogram._circle()*

`_direction_mask()`

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.

TODO: This array is not cached. it is used twice, for binning and grouping.

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:

local_reference_system (*poi*)

Calculate local coordinate system

The coordinates will be transformed into a local reference system that will simplify the directional dependence selection. The point of interest (poi) of the current iteration will be used as origin of the local reference system and the x-axis will be rotated onto the azimuth.

Parameters **poi** (*tuple*) – First two coordinate dimensions of the point of interest. will be used as the new origin

Returns **local_ref** – Array of dimension (m, 2) where m is the length of the coordinates array. Transformed coordinates in the same order as the original coordinates.

Return type numpy.array

search_area (*poi=0, ax=None*)

Plot Search Area

Parameters

- **poi** (*integer*) – Point of interest. Index of the coordinate that shall be used to visualize the search area.
- **ax** (*None, matplotlib.AxesSubplot*) – If not None, the Search Area will be plotted into the given Subplot object. If None, a new matplotlib Figure will be created and returned

Returns

Return type plot

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 function can be passed that accepts the current local coordinate system and returns a Polygon representing the search area. In this case, the tolerance and bandwidth has to be incorporated by hand into the model. The azimuth is handled by the class. 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 the current local coordinate system and returns a Polygon of the search area.

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 $0 \leq \text{angle} \leq 360$.

:raises ValueError : in case $\text{angle} < 0$ or $\text{angle} > 360$:

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', estima-  
                                tor='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 calculated 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 expensive in calculation time.
- **levels** (*int*) – Number of levels to be formed for finding contour lines. More levels result in more detailed plots, but are expensive 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 increase the node density. This is fast, but cannot interpolate *behind* any NaN occurrence. '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 calculated 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 expensive in calculation time.
- **levels** (*int*) – Number of levels to be formed for finding contour lines. More levels result in more detailed plots, but are expensive 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 increase the node density. This is fast, but cannot interpolate *behind* any NaN occurrence. ‘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 many 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*) – 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 has to 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='g', 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*) – Specifies 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.spatial.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 $n \leq 1$ 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.spatial.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, azimuth=220, color='g', 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 $0 \leq \alpha \leq 1$, 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')

# 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 ‘coordinantes’ 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 Estimator Functions

Scikit-GStat implements various semi-variance estimators. These functions 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()`

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 publication¹ and the notes on their application². Following the 1962 publication¹, 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 Appliquée*, Tome 1. Mémoires de Bureau de Recherches Géologiques et Minières, Paris.

² Matheron, G. (1965): *Les variables régionalisées et leur estimation*. Editions Masson et Cie, 212 S., Paris.

References

Cressie

`skgstat.estimated.cressie()`
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 x_i and x_{i+h} fall into the h separating distance class, x should contain $\text{abs}(x_i - x_{i+h})$ as an element.

Returns

Return type `numpy.float64`

Notes

This implementation is done after the publication by Cressie and Hawkins from 1980³:

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

with:

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

where x is exactly the input array x .

References

Dowd

`skgstat.estimated.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 x_i and x_{i+h} fall into the h separating distance class, x should contain $\text{abs}(x_i - x_{i+h})$ 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 extreme values at the cost of variability. This implementation follows Dowd's publication⁴:

$$2\gamma(h) = 2.198 * \text{median}(x)^2$$

with:

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

where x is exactly the input array x.

References

Genton

`skgstat.estimators.genton()`

Genton robust semi-variance estimator

Return the Genton semi-variance of the given sample x. Genton is a highly robust variogram 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 x_i and x_{i+h} fall into the h separating distance class, x should contain $\text{abs}(x_i - x_{i+h})$ as an element.

Returns

Return type `numpy.float64`

Notes

The Genton estimator is described in great detail in the original publication⁵ 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 $N \geq 500$, (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.estimated.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 x_i and x_{i+h} fall into the h separating distance class, x should contain $\text{abs}(x_i - x_{i+h})$ 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.estimated.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 x_i and x_{i+h} fall into the h separating distance class, x should contain $\text{abs}(x_i - x_{i+h})$ 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.estimated.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 x_i and x_{i+h} fall into the h separating distance class, x should contain $\text{abs}(x_i - x_{i+h})$ as an element.
- **p** (`int`) – Desired percentile. Should be given as whole numbers $0 < p < 100$.

Returns

Return type `np.float64`

2.6.5 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 optional 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 follows⁶:

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

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 $C_0 + b$.
- **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 following⁷ and⁸ 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.

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 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

This implementation follows⁹:

$$\gamma = b + c_0 * \left(1 - e^{-\frac{h^2}{a^2}}\right)$$

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.

⁹ Chiles, J.P., Delfiner, P. (1999). Geostatistics. Modeling Spatial Uncertainty. Wiley Interscience.

- **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 like:

$$\gamma = b + c_0 * \left[7 * \left(\frac{h^2}{a^2} \right) - \frac{35}{4} * \left(\frac{h^3}{a^3} \right) + \frac{7}{2} * \left(\frac{h^5}{a^5} \right) - \frac{3}{4} * \left(\frac{h^7}{a^7} \right) \right]$$

a is the range parameter. For the cubic function, the effective range and range parameter are the same.

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.

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 $a = \frac{r}{3^{\frac{1}{s}}}$. 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 $s \leq 2$ 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:

$$\gamma = b + C_0 * \left(1 - e^{-\frac{h}{a}s}\right)$$

a is the range parameter and is calculated from the effective range r as:

$$a = \frac{r}{3^{\frac{1}{s}}}$$

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.

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 $a = \frac{r}{2}$. 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 $C_0 + b$.
- **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 $+\infty$ 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 formula and references will follow.

2.6.6 Kriging Class

```
class skgstat.OrdinaryKriging (variogram, min_points=5, max_points=15, mode='exact', precision=100, solver='inv', n_jobs=1, perf=False)
```

```
__init__ (variogram, min_points=5, max_points=15, mode='exact', precision=100, solver='inv', n_jobs=1, perf=False)
```

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
- **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.

```
transform (*x)
```

Kriging

returns an estimation of the observable for the given unobserved locations. Each coordinate dimension should be a 1D array.

Parameters **x** (*numpy.array*) – 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*

2.7 Changelog

2.7.1 Version 0.3

- [DOCS] Tutorial added and Dockerfile finalized
- [Variogram] normalize default value changed to 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`.

2.7.2 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

2.7.3 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.

2.7.4 Version 0.2.5

- added `OrdinaryKriging` for using a `Variogram` to perform an interpolation.

2.7.5 Version 0.2.4

- added `SpaceTimeVariogram` for calculating dispersion functions depending on a space and a time lag.

2.7.6 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.

2.7.7 Version 0.2.2

- added `DirectionalVariogram` class for direction-dependent variograms
- [Variogram] changed default values for *estimator* and *model* from function to string

2.7.8 Version 0.2.1

- added various unittests

2.7.9 Version 0.2.0

- completely rewritten `Variogram` class compared to v0.1.8