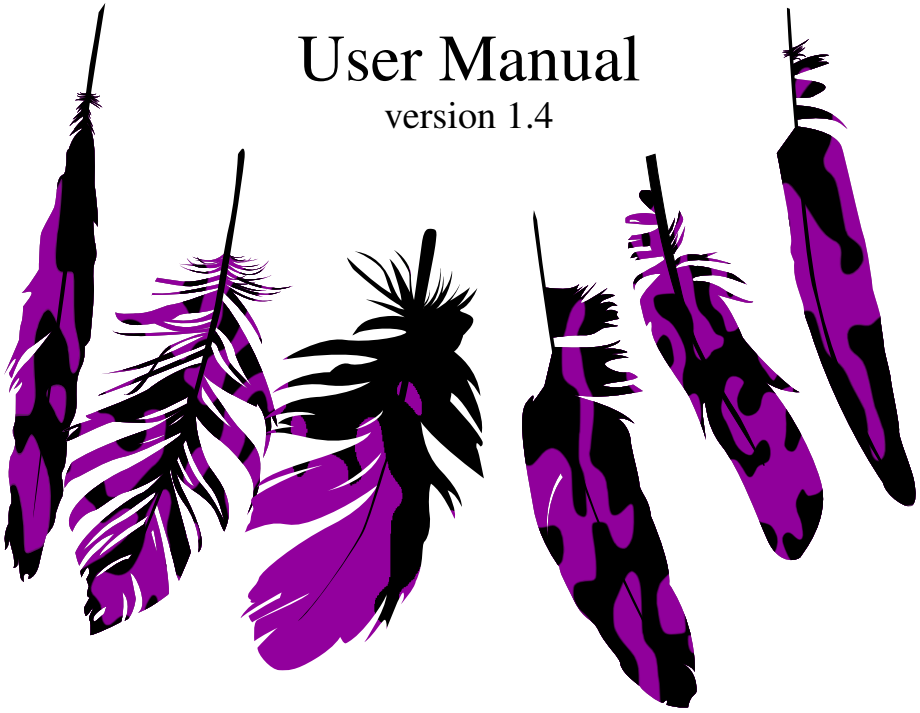
Revarie

User Manual

version 1.4



Dean Price, 2020

Contents

1 Introduction

1.1 What is a Variogram?

A variogram is a tool that can be used to characterize the spatial continuity of a scalar field. For the purposes of this document, a field will refer to a collection of scalar values that each occupy a unique point in a spatial domain. These points can be uniformly, or randomly, spaced. For these applications, an example of a field may be a collection of altitude measurements at different points in a mountain range. In this case, each measurement may have a 2-component coordinate vector and an altitude.

In Figure 1, two different 2D fields are shown, each with a spatially-independent mean of 0 and spatially-independent variance of 1. Clearly, there is a difference in these two fields. In the left field, each point is generated independently of all other points. In the right field, each point is more likely to be similar to points around it. If the relationship between points is strictly a function of proximity, an isotropic variogram can be used to describe their relationship. The variogram used to generate the field shown in the right of Figure 1 is shown in Figure 2. The relationship between a variogram and the generated field will be described in more detail in later sections.

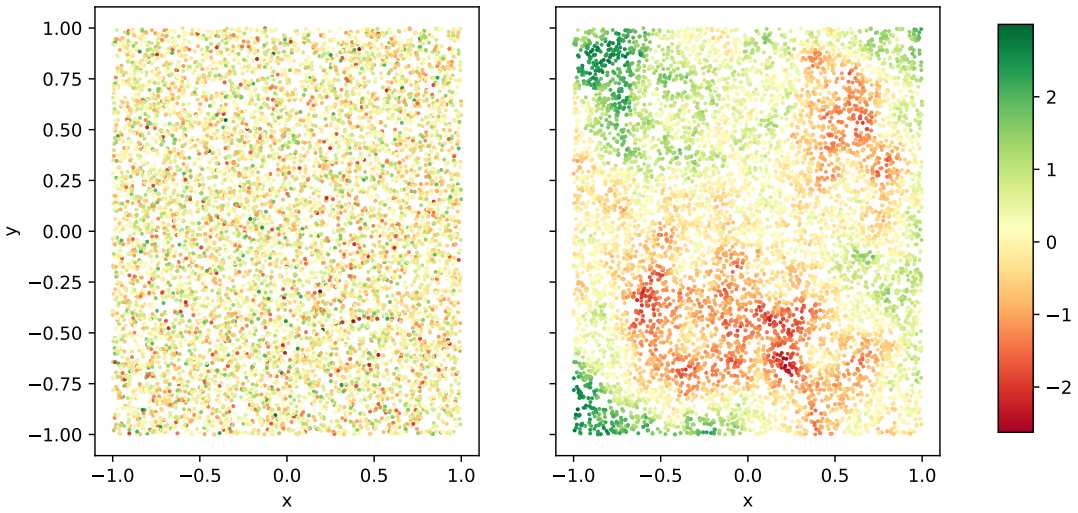


Figure 1: Demonstration of spatially-independent field (left) and field generated using a variogram (right). Both fields have same mean and variance.

1.2 What is Revarie?

Revarie is a spatial statistical analysis toolset written as an easily importable python library which was created with computational physics applications specifically in mind. Its primary focus is on variography of scalar Cartesian fields. Currently, it supports computation and fitting of isotropic variograms along with other tools to support random field generation. It is designed to work with Python's existing numerical libraries to ensure efficient, scalable computation.

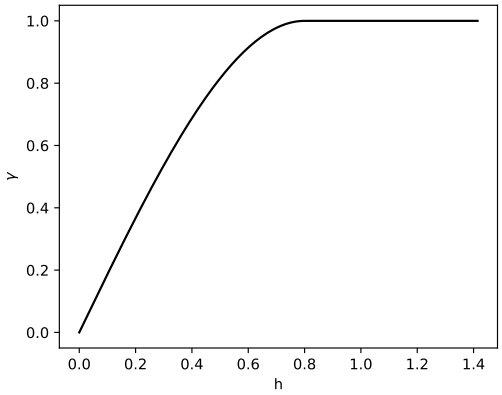


Figure 2: Variogram used to generate field shown in right portion of Figure 1

1.3 Installation

It is recommended to install Revarie using pip:

```
pip install revarie
```

The repository can be cloned via the command below. However, this repository may be unstable so it is recommended for most users to use pip installation.

```
git clone https://github.com/deanrp2/revarie.git
```

1.4 Dependencies

- python 3.5+
- scipy
- numpy
- numpy-indexed

2 Quick Start

The following section should provide most of the information a user would need to use Revarie, examples will be worked through with detailed explanations. For each example, code will be provided in the format given below:

```
foo = "bar"
if foo:
    print(foo)
```

```
>> "bar"
```

Furthermore, it may be helpful to give a bit of code that is helpful for plotting 2D fields in the data format most compatible with Revarie.

```
import matplotlib.pyplot as plt
import numpy as np

r = #numpy array of shape (N,2), each row is coordinates for a point
```

```
f = #numpy array of shape (N,) each element is field value

cm = "RdYlGn" #or any desired colormap from matplotlib

fig, ax = plt.subplots(1,1)
im = ax.scatter(r[:,0], r[:,1], c = f, cmap = cm, s = 2)
fig.subplots_adjust(right = 0.85)
cbar_ax = fig.add_axes([0.87, 0.15, 0.03, 0.7])
fig.colorbar(im, cax = cbar_ax)
plt.show()
```

:

2.1 Calculate Variogram of 2D Field

One of the more basic use cases of Revarie may be calculating the variogram of a given field. In this example the variogram of a 2D field containing 9000 points, provided in Figure 3, will be calculated.

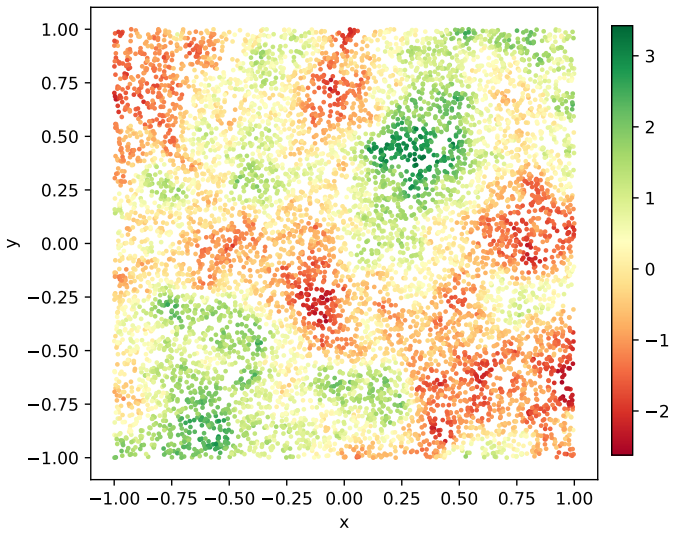


Figure 3: Example field to demonstrate variogram calculation

First, it is necessary to import relevant packages and load the field data into python:

```
import numpy as np
import revarie as rev

r = np.loadtxt("coords.dat")
f = np.loadtxt("vals.dat")
```

:

Numpy is a python library which supports a multidimensional array data structure as well as a number of high-level manipulation and initialization functions for those arrays, it is necessary to hold initial data about the field. In this example case, `r` is stored in a text file called `"coords.dat"`. When imported into python in the manner shown above, it is a numpy array of shape $(2, N)$ where N is the number of points in the field. `f` is stored in a

text file called `"vals.dat"` which contains the field values for every point. This array will be of shape (N,). To demonstrate, some code is given below.

```
print(r.shape)
print(f.shape)
```

```
>> (9000, 2)
>> (9000,)
```

Now, it is time to actually use Revarie to perform a calculation. It is necessary to create a `Variogram` object:

```
v = rev.Variogram(r,f)
```

Upon creation, most of the computation required is performed so it may take time for larger problems. From here, the Matheron method can be called to return the Matheron variogram to the user. By default, the bins will be selected based on the size of the domain of the field using 10 linearly spaced divisions. The user has a few different options in how the variogram is calculated that will be explained in later sections. Also by default, 3 parameters are returned as numpy arrays:

- Lag distance corresponding to the center of each bin
- Number of point pairs corresponding to lag distances contained in the range of a bin.
- Matheron variogram value for that bin

```
centers, n_pairs, matheron_v = v.matheron()
```

From here, the `centers` and `matheron_v` can be plotted using the code below to yield the variogram shown in Figure 4.

```
import matplotlib.pyplot as plt #should be added at top of script

plt.plot(centers, matheron_v, "k")
plt.xlabel("lag distance")
plt.ylabel("variogram")
plt.show()
```

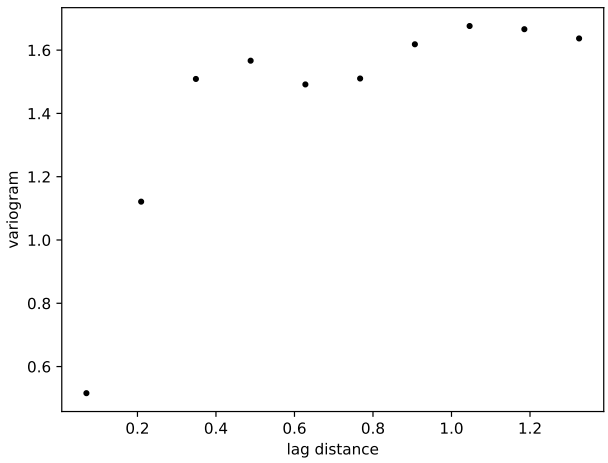


Figure 4: Example Matheron variogram of field given in Figure 3.

2.2 Fit Theoretical Models to Variogram

Often, it is difficult to analyze variograms in the form given in Figure 4, called experimental variogram. Therefore, it may be useful to fit a function to the experimental variogram. Revarie supports a variety of methods of doing this, however, here fitting this experimental variogram to a popular variogram model, called a spherical model, will be shown. More information on the spherical model and other similar theoretical models can be found later in Section 4.3. First, as with the last example, it is important to import the necessary libraries.

```
import numpy as np
import revarie as rev
```

For simplicity, we will use the experimental variogram calculated in the previous example found in Figure 4. Below, the values of this variogram are explicitly input and stored in a numpy array called `ex_vario`. The lag distances corresponding to each variogram value is stored as `ex_lags`. In the previous example, these values were calculated with the `matheron` method and stored as `centers` and `matheron_v`.

```
ex_vario = np.array([0.5159, 1.1211, 1.5089, 1.5666, 1.4916,
                     1.5103, 1.6185, 1.6762, 1.6661, 1.6369])
ex_lags  = np.array([0.0700, 0.2094, 0.3489, 0.4883, 0.6278,
                     0.7672, 0.9067, 1.0462, 1.1856, 1.3251])
```

From here, we use the `fvariogram` function to fit a theoretical variogram to the experimental variogram. Overall, `fvariogram` is a function intended to make it easy for the user to create—or calculate—functional forms of variograms. It supports fitting common variograms, fitting experimental variograms to user-defined models, interpolation and creation of theoretical variograms explicitly given parameters. The user can access its various functionalities using a branch-like process where the function parameters `source` and `methd` guide the user to a specific functionality and the `options` parameter lets the user give inputs for that specific functionality. For this application, a nonlinear least-squares fitting method will be used to fit the experimental variogram to the spherical theoretical variogram model. This can be done with the code given below:

```
th_vario = rev.fvariogram(source = "data",
                          methd = "bmodel",
                          options = [ex_lags, ex_vario, "sph"])
```

There are a few different things to note about the `fvariogram` function, first, the arguments. The `source` argument is selected to be `"data"` this is one of two options that the user can choose from that may lead the user to a desired functionality, the next example will show an application where the `source` argument will be set to `"func"`. Next up, the `methd` argument can be selected from a few different possibilities depending on the application. For this, `"bmodel"` indicates that the provided data is to be fit to a model that is already included in the Revarie package. Finally, the `options` argument, in this application, should be a list containing:

1. The lag values or bin centers to be used for fitting
2. The variogram values to be used for fitting
3. String that points at which built-in model to be used, here it is the spherical model, so `"sph"`

The actual contents of `options` is specific to the selection of the `source` and `methd` parameters, a complete list of functionality for `fvariogram` and how to access it is given in Section 3.2. Moving on to the return data, *the return type will always be a function*. The returned function will take one argument, lag distance, and return the variogram value for that lag distance according to the fitted model. Below, this is demonstrated.

```
print(th_vario(0.3))
```

```
>> 1.579273156321727
```

More practically, the resulting function can take numpy arrays as arguments and return numpy arrays:

```
print(th_vario(ex_lags))
```

```
>> [0.51722172 1.11744264 1.51282021 1.59494506 1.59494506 1.59494506
    1.59494506 1.59494506 1.59494506] 1.59494506]
```

It may also be useful to plot the calculated theoretical variogram on top of the experimental variogram data. This can be done using the code given below, the resulting figure is given in Figure 5:

```
import matplotlib.pyplot as plt #should be added at top of script

h = np.linspace(0, ex_lags.max(), 200) #create lag values where the
                                       #theoretical variogram will be
                                       #plotted
plt.plot(h,th_vario(h), "k", label = "th") #plot theoretical variogram
plt.plot(ex_lags,ex_vario,"k.", label ="ex") #plot experimental variogram
plt.xlabel("lag distance")
plt.ylabel("variogram")
plt.legend()
plt.show()
```

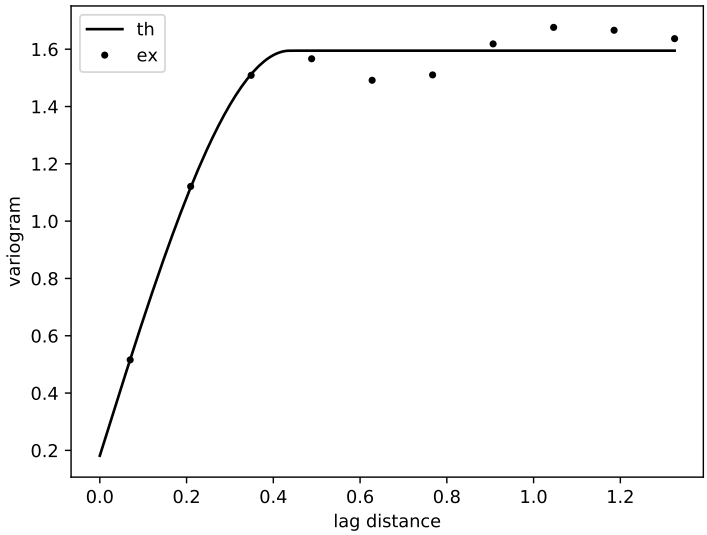


Figure 5: Fitted spherical model theoretical variogram on experimental variogram.

2.3 Returning Fitted Parameters from `fvariogram`

It may not be particularly useful to fit a variogram without information on the variogram parameters that yielded the best fit. Revarie also has an option to return the parameters of the fitted model. As before, imports:

```
import numpy as np
import revarie as rev
```

In order to show another capability of Revarie, returning fitted parameters will be demonstrated on a user-specified model using the `fvariogram` function. A fit with a user-specified function will be used because the procedure is very similar to the previous example. We will use a made-up model that adheres to the convention set by the built-in models (see Section 4.3), that is four parameters are used in the variogram:

1. `h`: a vector of lag distances
2. `nug`: the nugget of the variogram
3. `sill`: the sill of the variogram
4. `rang`: the effective range of the variogram

In this exercise, it will be called the "piecewise linear" model and it will be mathematically defined as:

$$\gamma(h) = \begin{cases} b + \frac{h}{r}(c-b) & h \leq r \\ b + (c-b) & h > r \end{cases} \tag{1}$$

Here, b is the nugget, r is the effective range, c is the sill and h is the lag distance. In this case, the effective range is where the variogram value is equal to the sill. Figure 6 is included to show how this model may look for a few different selections of b, r and c.

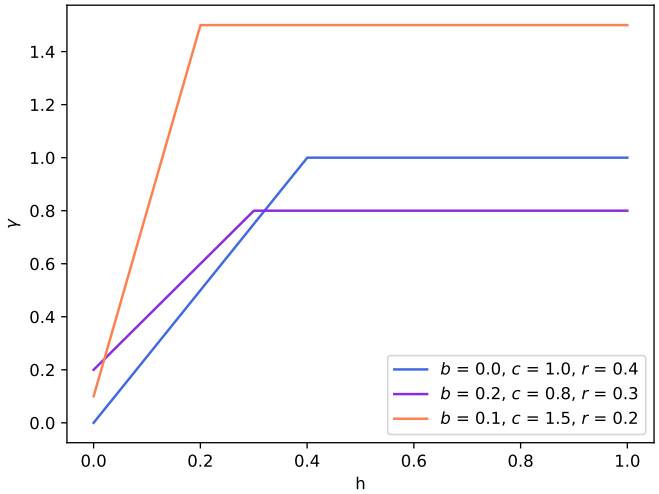


Figure 6: Piecewise linear variogram model used for demonstration purposes.

First, a python function must be created to hold the unfitted model. This initial function should take h, b, r and c as arguments and return a numpy array of the variogram values for

each element in h. This function will never actually be manually called by the user, it will be used as an input to the `fvariogram` function. It is important that the first argument to this function is the lag distances, beyond that, these parameters can be in any order. As a general tip, it is highly recommended to vectorize any user-specified functions that operate with numpy arrays. An example of this is given below where `numpy.piecewise` is used instead of a `for` loop.

```
def piecewise_linear(h, b, r, c):
    """
    Theoretical peicewise linear variogram model with unfitted parameters
    """
    hoverr = np.piecewise(h,
                          [h <= r, h > r],
                          [lambda h: h/r, 1])
    return b + (c - b) * hoverr
```

:

We will use the same experimental variogram used in the last example, so it is useful to define variables to hold this data:

```
ex_vario = np.array([0.5159, 1.1211, 1.5089, 1.5666, 1.4916,
                     1.5103, 1.6185, 1.6762, 1.6661, 1.6369])
ex_lags  = np.array([0.0700, 0.2094, 0.3489, 0.4883, 0.6278,
                     0.7672, 0.9067, 1.0462, 1.1856, 1.3251])
```

:

Now, it is time to actually fit the model:

```
th_vario, opts = rev.fvariogram(source = "data",
                                methd = "umodel",
                options = [ex_lags, ex_vario, piecewise_linear, True])
```

:

There are a few differences between the above code snippet and the code snippet from the previous example which called the `fvariogram` function:

1. The `fvariogram` function is returning two parameters here, `th_vario` and `opts`. `th_vario` is a function which takes lags as an argument and returns variogram values corresponding to each lag based on the fitted model. `opts` is a list containing the fitted parameters from the model, ordered the same as the input parameters for the user specified function.
2. The `methd` argument is set to `"umodel"`. This is simply because in this example, we are fitting a user-specified model and not a built-in model.
3. The third element in the `options` argument contains the name of the function we used to store the user-specified function. For built-in models this element is a string.
4. There is now a fourth element in the `options` parameter, this `True` boolean enables returning the fitted parameters.

Of course, from here it may be useful to know the fitted model parameters. They can be printed out with the code given below:

```
print(opts)
```

```
>> [0.21199756   1.5843875  0.31611229]
```

:

Again, these values are returned to match the order of the input parameters of the user-specified function. Therefore, from this fit, we get a nugget (b) value of 0.212, effective range (r) of 1.584 and sill (c) of 0.316. The fitted theoretical variogram can be plotted on top of the experimental variogram using the code given below, the result is given in Figure 7:

```
import matplotlib.pyplot as plt #should be added at top of script

h = np.linspace(0, ex_lags.max(), 200) #create lag values where the
                                       #theoretical variogram will be
                                       #plotted
plt.plot(h,th_vario(h), "k", label = "th") #plot theoretical variogram
plt.plot(ex_lags,ex_vario,"k.", label ="ex") #plot experimental variogram
plt.xlabel("lag distance")
plt.ylabel("variogram")
plt.legend()
plt.show()
```

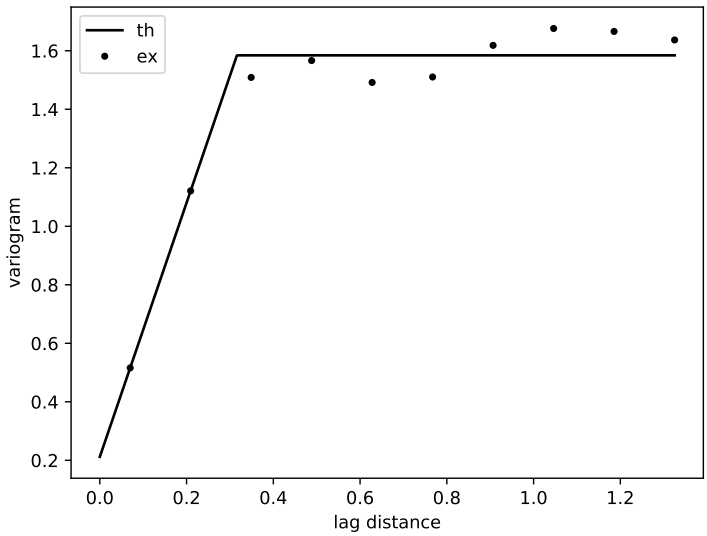


Figure 7: Fitted user-specified theoretical variogram on experimental variogram.

2.4 Generate Variogram from Built-in Models

It may be useful to generate a variogram based on known parameters of a theoretical variogram. Revarie allows the user to store variograms as python functions, which take a numpy array of lag values and return the variogram value at each of those lag distances as a numpy array. In order to demonstrate this functionality in Revarie, we first import necessary modules:

```
import revarie as rev
import numpy as np
```

For this example, we will use an exponential model with a nugget of 0, sill of 0.5 and range of 4. For more information on types of variogram models and the meaning of these parameters, see Section 4. The code below stores the variogram in the variable name `exp_vario` using the `fvariogram` function:

```
nugget = 0.
sill = 0.5
rang = 4 #"e" excluded to avoid "range" function
vario = rev.fvariogram(source = "func",
                       methd = "exp",
                       options = [nugget, sill, rang])
```

From this code snippet, we see another set of selections in the `source`, `methd` and `options` parameters. First, setting `source = "func"` basically means that the user will set all the parameters to define the variogram. There are only two possibilities for this parameter. Next, `methd = "exp"`. This is where the user specifies which built-in model is used, currently the options are `"sph"`, `"exp"` and `"gaus"` for the spherical, exponential and gaussian models, respectively. These models are listed later in Section 4.3 and an alternative way of accessing these models using Revarie is given in Section 2.5.

Furthermore, the value returned from the `fvariogram` function is always a function, more is explained in Section 2.2. This functionality may be particularly useful if it is necessary to apply the same variogram model to multiple different sets of lag distances or it may declutter code in larger projects that may need to make many calls to the same variogram model.

2.5 Accessing Built-in Theoretical Variogram Models

For users who prefer to access the built in models without using the `fvariogram` function, these models can be directly accessed with the base Revarie import. We first import the necessary modules:

```
import revarie as rev
import numpy as np
```

Currently, three models are built into Revarie; they are listed with their equations in Section 4.3. They can be accessed in two ways:

1. directly with a function call
2. using the `mtags` dictionary

For exploratory analysis, the second option may be preferable but for more understandable code, the first option is likely better. For the first option, as with any python library, a function can be called as long as it exists in the global—or a specified—namespace. You simply need to call the function through the revarie namespace:

```
ans = rev.spherical(h = np.array([.1, .4, 4]),
                    nug = 0,
                    sill = 8,
                    rang = 1.4)
print(ans)
```

```
>> [0.85568513 3.33527697 8.        ]
```

As more experienced python users may know, you can also import the function into your global namespace:

```
from revarie import spherical
ans = spherical(h = np.array([.1, .4, 4]),
```

```
                  nug = 0,
                  sill = 8,
                  rang = 1.4)
print(ans)
```

Onto the second option, the `mtags` variable stores a `dict` whose keys are the "model tags" and whose values are the function of each model. The current model tags present in the Revarie library are given in Section 4.3. This was included in the Revarie primarily to provide the developers an easy way to access all models in the Revarie library and provide easy ways to incorporate more models into the package. If the user would like to access the built-in functions using the `mtags` variable, it can be done like below:

```
ans = rev.mtags["sph"](h = np.array([.1, .4, 4]),
                       nug = 0,
                       sill = 8,
                       rang = 1.4)
print(ans)
```

```
>> [0.85568513 3.33527697 8.        ]
```

2.6 Random Field Generation 1D

Revarie also has the capability to generate a field given a particular variogram. In general, this is the most computationally-expensive functionality in the Revarie tool set. First, as with the previous examples, it is necessary to import the relevant modules.

```
import revarie as rev
import numpy as np
import matplotlib.pyplot as plt
```

From here, we also need to create a python function that will be used by the `Revarie` class to generate the field. The `fvariogram` function is specifically created to make it easy to do this (see Section 2.4). In this example, two different variograms are created. First, a spherical variogram with nugget of 0, sill of 1 and effective range of 0.7. Second, a spherical variogram with nugget of 0, sill of 1 and effective range of 0.1. These variograms are stored in `vario_long` and `vario_short`, respectively, in the code below:

```
vario_long = rev.fvariogram(source = "func",
                            methd = "sph",
                            options = [0, 1, 0.7])
vario_short = rev.fvariogram(source = "func",
                             methd = "sph",
                             options = [0, 1, 0.1])
```

These variograms can be plotted with the below code, the resulting plot is shown in Figure 8.

```
h = np.linspace(0,1., 200)

plt.plot(h, vario_long(h), "k", label = "vario_long")
plt.plot(h, vario_short(h), "b", label = "vario_short")
plt.xlabel("lag distance")
plt.ylabel("variogram")
plt.legend()
plt.show()
```

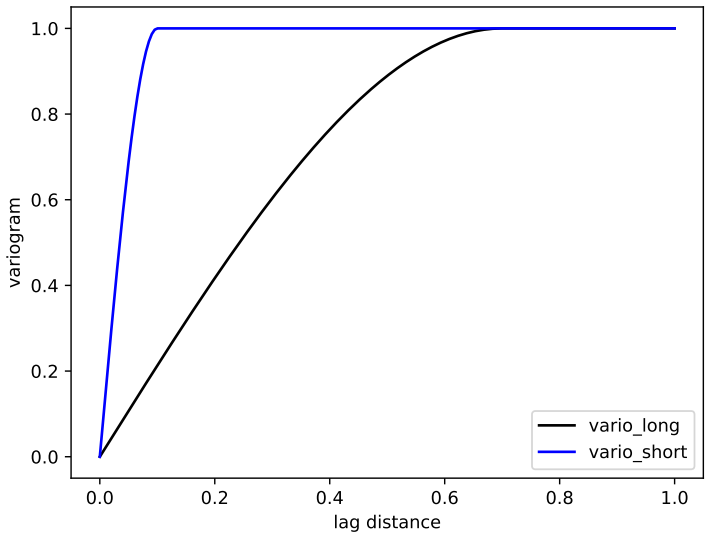


Figure 8: Two variograms used for field generation.

For this example, 1D fields will be generated with uniformly spaced points located between -0.3 and 0.3 with a mean value of 0. Also, the spatially-independent variance will be assumed to be identical to the sill, at 1. The next step is to create a variable (in this case called `x`) that will hold a numpy array that specifies the location of the points in the field. For this, we will use the `numpy.linspace` function

```
x = np.linspace(-0.3, 0.3, 600)
```

From here, we must initialize `Revarie` objects:

```
rlong = rev.Revarie(x = x,
                    mu = 0,
                    sill = 1,
                    model = vario_long)
rshort = rev.Revarie(x = x,
                     mu = 0,
                     sill = 1,
                     model = vario_short)
```

Once these objects are initialized, fields can be generated using the `genf` method. Below, a field is generated and stored in the `flong` variable, the length of the field is the same as `x` because each value in `flong` corresponds to a coordinate in `x`.

```
flong = rlong.genf()
print(flong.shape)
print(x.shape)
```

```
>> (600,)
>> (600,)
```

The field can be plotted with the code given below, the result is shown in Figure 9:

```
plt.plot(x, flong, "k")
plt.xlabel("x")
plt.ylabel("field value")
```

```
plt.show()
```

.

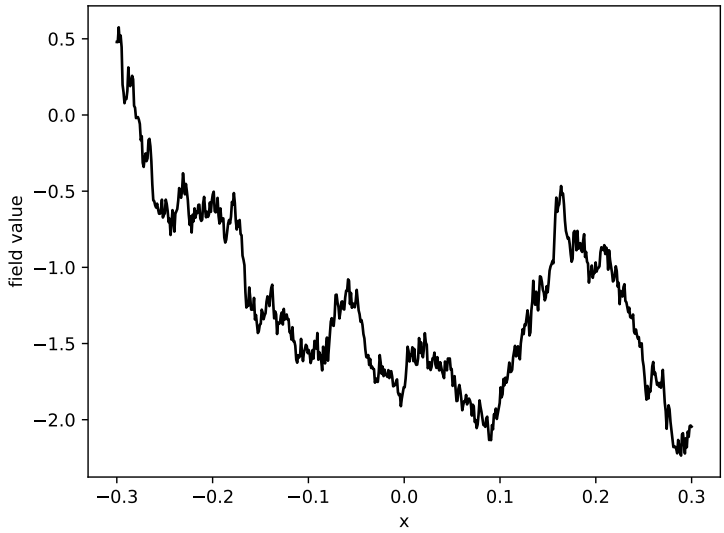


Figure 9: Example 1D field generated with spherical theoretical variogram of range 0.7, sill 1 and nugget 0.

For demonstration purposes, a few different fields resulting from the `rlong` and `rshort` objects are plotted in code given immediately below. The resulting figure is given in Figure 10, the range of x values is likely too small to accurately display the complete behavior from the longer-ranged variogram.

```
fig, ax = plt.subplots(3,3,
                       figsize = (8,8),
                       sharex = True,
                       sharey = True,
                       constrained_layout = True)
ax = ax.flatten()

for i in range(9):
    ax[i].plot(x, rlong.genf(), "k", label = "vario_long")
    ax[i].plot(x, rshort.genf(), "b", label = "vario_short")

    if i % 3 == 0:
        ax[i].set_ylabel("field value")

    if i > 5:
        ax[i].set_xlabel("x")
ax[0].legend()
plt.show()
```

:

2.7 Random Field Generation 2D

Overall, generating random fields in 2D is very similar to generating fields in 1D except `x` should contain the coordinates of the field points in two dimensions. However, there are a few subtleties such as generating `x`, plotting and an increased computational time that should be considered. As usual, import necessary modules:

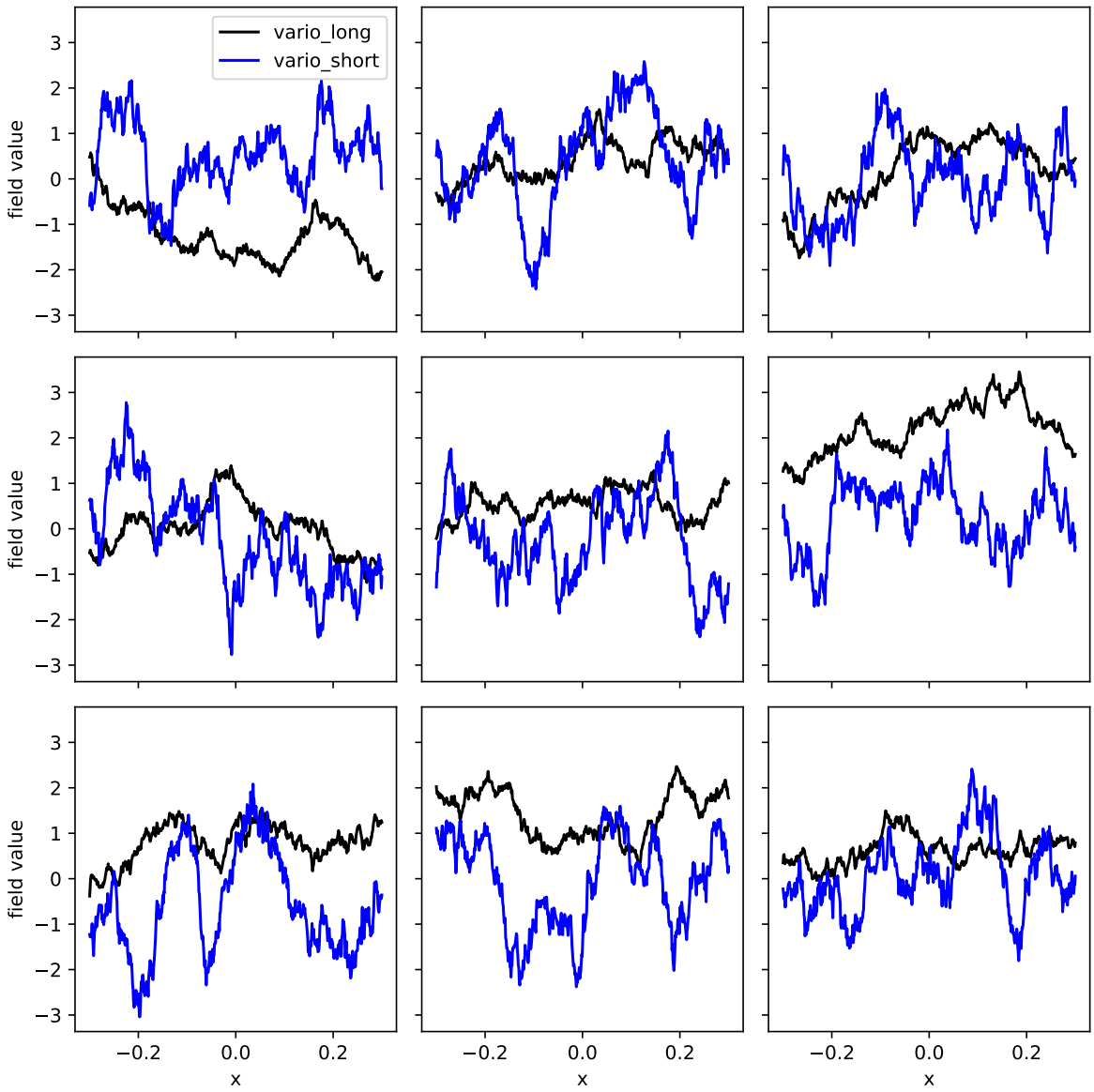


Figure 10: Multiple fields generated with two different effective ranges.

```
import numpy as np
import revarie as rev
import matplotlib.pyplot as plt
```

Now, the variogram must be created. This is done with the `fvariogram` function. We create a spherical variogarm with nugget of 0, sill of 1 and effective range of 0.2:

```
vario = rev.fvariogram(source = "func",
                       methd = "sph",
                       options = [0, 1, 0.3])
```

Now we can generate x, this is not as simple as it was in the 1D case. Although there are many ways to do this, the code given below is included for reference:

```
N = 100 #number of coords in mesh
x = np.zeros((N**2,2)) #initialize x


```

```
g = np.linspace(-0.5, 0.5, N) #generate coords for both dimensions

#create and assign all combinations of coords in both dimensions to
#   each column of x
x[:,0], x[:,1] = [a.flatten() for a in np.meshgrid(g,g)]
```

One should note that field generation run time is highly dependent on N in this case because the number of points in the mesh scales with the square of N. Now, we create a `Revarie` object. We will use a mean of 0 in this case:

```
r = rev.Revarie(x = x, mu = 0, sill = 1, model = vario)
```

From here, we can generate a random field. Note this may take some time:

```
randf = r.genf()
```

Again, there are many ways to plot this field but the code below is given for reference. The result of this snippet is given in Figure 11.

```
cm = "RdYlGn"

fig, ax = plt.subplots(1,1)
im = ax.scatter(x[:,0],x[:,1], #giving locations of points to plot
                c = randf,     #color of each point corresponds with
    value
                               #    in randf
                cmap = cm,     #specifying which colormap to use
                s = 3)         #specify the size of the points

fig.subplots_adjust(right = 0.85) #create space for color bar
cbar_ax = fig.add_axes([0.87,0.15,0.03, 0.7]) #create axis obj for cbar
fig.colorbar(im, cax = cbar_ax) #create a colorbar aside the scatter plot

ax.set_xlabel("x")
ax.set_ylabel("y")
plt.show()
```

3 Documentation

The purpose of this section is to provide documentation on each class or method contained in the Revarie toolbox. Most of the content of this section will be taken directly from the in-code documentation, basic knowledge of object oriented python may be needed to completely understand from this section.

3.1 `Variogram`

Calculates lag and squared difference values for a given field. Various operations can be performed with these quantities within this class.

3.1.1 `Variogram.__init__(self, x, f)`

Create variogram object and calculate lags and squared differences.

Parameters

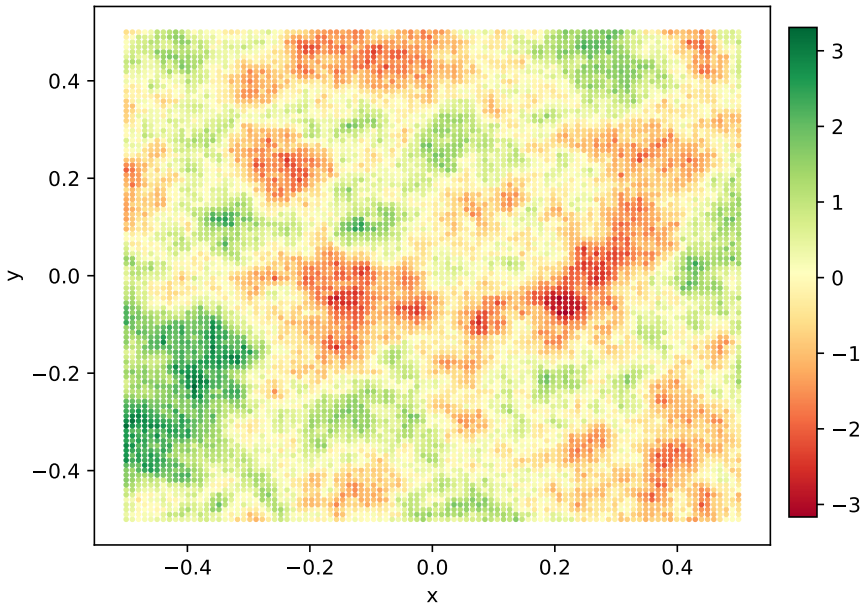


Figure 11: Randomly generated 2D field.

- **x**: numpy.ndarray
  Array of shape (m,n) where n is number of points in an m-dimensional domain.
- **f**: numpy.ndarray
  Array of field values observed at each of the n points.

3.1.2 `Variogram.cloud(self)`

Convenient way to get calculated lags and squared differences from a Variogram instance. Named for variogram cloud plot.

Returns

- **lags**: numpy.ndarray
  Distance between all combinations of points fed into variogram.
- **diffs** : numpy.ndarray
  Squared difference between all combinations of field values fed into variogram.

3.1.3 `Variogram.matheron(self, bin_type = "auto", bins = 10, var = False)`

Calculate Matheron variogram for points and field values previously fed into variogram. A few options for specifying binning exist.

Parameters

- **bin_type** : str
  Descriptor of the format of data passed into the bin parameter. Can be one of:

  - ”auto” : select bounds to be between 0 and the largest lag distance calculated, bin centers will be calculated accordingly based on user given number of bins.
  - ”lin” : bin boundaries will be linearly spaced based on a user given minima, maxima and number of bins. Bin centers will not fall on given maxima and minima.
  - ”bound” : bounds will be completely given by the user as a numpy array

- **bins** : int, list, array-like
  Description of how binning will be performed in Matheron variogram, specific formats given below for each bin type.
  - ”auto” : int giving number of bins to use
  - ”lin” : list containing three entries. First is minima of bin boundaries, second is maxima of bin bounds and third is the number of bins to use as int
  - ”bound” : array-like object specifying boundaries of bins. Number of bins is length of this array - 1.

- **var** : bool
  Set True for bin-wise variance to be calculated and returned

Returns

- **centers** : numpy.ndarray
  Bin centers used for variogram
- **n_bins** : numpy.ndarray
  Number of point relations used to calculate each semivariance
- **v** : numpy.ndarray
  Estimated semivariance values at lags corresponding to bin centers
- **v_var** (optional) : numpy.ndarray
  Variance associated with squared difference values within a bin

3.1.4 `Variogram.reduce(self, typ, bnds, inplace = True)`

Reduce the lag domain of a given Variogram object. Removes data that lay outside the given lag boundaries. Very useful for reducing run times and memory requirements when performing further operations with variogram after initialization such as matheron. Object points and field values are not affected. Resulting Variogram objects are also marked with a self.reduced = True flag to indicate that the self.lags and self.diffs are not reflective of all combinations of self.x and self.f.

Parameters

- **typ** : str
  Descriptor of reduction criteria that is passed to the bnds parameter. Can be one of:
  - ”abs” : eliminate data from self.diffs and self.lags that come from lags greater or less than specified values.
  - ”quant” : eliminate data from self.diffs and self.lags by quantiles.

- **bnds** : list
  2-element list to describe reduction criteria. Specific formats given below for each typ option.
  - ”abs” : List in the format of (min, max) for range of lags to be included after reduction.
  - ”quant” : range of quantiles in format of (min, max) to keep after reduction.
- **inplace** : bool
  Whether or not object is manipulated inplace. If False, will return variogram object with same x and f values but with the reduced lag domain.

Returns

- **new** (optional) : Variogram
  New Variogram instance with same x and f values but reduced lag domain.

3.1.5 `Variogram.rreduce(self, typ, amnt, inplace = False)`

Uniformly downsample stored lags and field value differences for faster calculations and smaller memory size. Object points and field values are not affected. Resulting Variogram objects are also marked with a self.reduced = True flag to indicate that the self.lags and self.diffs are not reflective of all combinations of self.x and self.f.

Parameters

- **typ** : str
  Descriptor of format of reduction amount passed to amnt. Can be one of:
  - ”abs” : Size of remaining lag and field data difference array specified as an absolute size.
  - ”frac” : Size of remaining lag and field data difference array specified as a fraction of original size.
- **amnt** : int, float
  Amount to reduce lag and field difference data vectors. Specific formats given below for each typ option.
  - ”abs” : int giving the size of the remaining lag and field data difference vectors.
  - ”frac” : float between 0 and 1 describing how much of lag and field difference should remain as fraction of original size
- **inplace** : bool
  Whether or not object is manipulated inplace. If False, will return variogram object with same x and f values but with the reduced lag domain.

Returns

- **new** (optional) : Variogram
  New Variogram instance with same x and f values but reduced lag domain.

3.2 `fvariogram`

Figure 12 is included to help users select which arguments for the `source` and `methd` are desired.

3.2.1 `fvariogram(source, methd, options, *args, **kwargs)`

Function intended to make generating python functions to describe variograms centralized. This function can be used to access the built-in models as well as fit around user-defined models. Functionality includes fitting variograms to built-in and user-defined models, interpolation and polynomial fitting. Put simply, can take a wide range of parameters to describe how a function is made that gives variogram as a function of distance. Then, returns that function.

Parameters

- **source** : str
  Descriptor of the general approach used for the variogram model. Can be one of:
  - ”func” : it is desired to return either a user-defined function, or a built-in function with specified parameters. This is not the option to select if any sort of fitting is desired. See below.
  - ”data” : a model will be fit to data that is provided in the ”options” parameter according to a method given in the ”methd” parameter.
- **methd** : str
  Secondary descriptor which depends on the route selected in the source parameter. If ”func” was selected, can be one of:
  - ”ufunc” : user-specified python function object that takes numpy array of lag values and returns variogram values. Should only take a single object.
  - built-in model : pass a string of any tag associated with a built-in model to use that model with parameters specified later in the ”options” parameter.

  If ”data” was selected:
  - ”poly” : uses numpy polynomial fit tool to fit a polynomial to data provided later in the ”options” parameter
  - ”interp” : uses the scipy interpolate tools to interpolate data provided later in the ”options” parameter
  - ”bmodel” : fits the data provided in the ”options” parameter to one of the built-in models using nonlinear least squares
  - ”umodel” : fits the data provided in the ”options” parameter to a user-defined function with an arbitrary number of arguments using nonlinear least squares. If values of the arguments are known, consider using the source = ”func” route because this method will fit function parameters to provided data.
- **options** : list List of data needed for the options selected in the previous parameters. Descriptions of the content of these lists are given below:

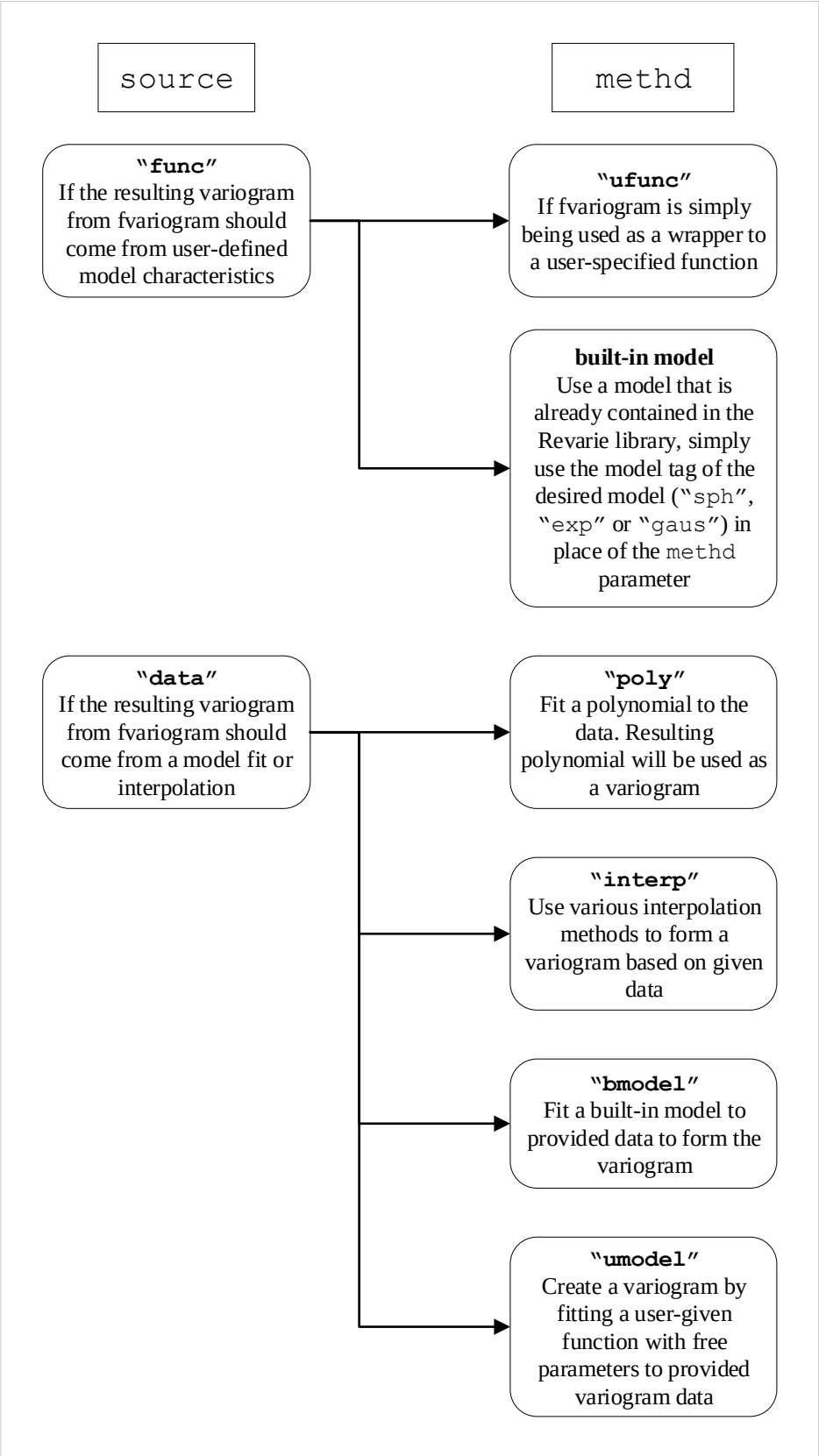


Figure 12: Structure of options in the `fvariogram` function.

  - ”func” → ”ufunc” : [<userfunction>, *args, **kwargs]
    <userfunction> user-defined function to be called with an array of lags as input that returns an array of variogram values. Include extra args to function in *args and **kwargs.
  - ”func” → built-in model : [*args, **kwargs]
    list of parameters to be passed directly to the selected built-in model
  - ”data” → ”poly” : [<h>, <v>, <opt>, <order>]
    <h> array of lag values to fit polynomial to
    <v> array of variogram values to fit polynomial to
    <order> order of polynomial for fitting as int
  - ”data” → ”interp” : [<h>, <v>, <kind>]
    <h> array of lag values to use for interpolation
    <v> array of variogram values to use for interpolation
    <kind> str to specify how interpolation is performed, options include: ”linear”, ”nearest”, ”zero”, ”slinear”, ”quadratic”, ”cubic”, ”previous” or ”next”. See scipy.interpolate.interp1d for more details
  - ”data” → ”bmodel” : [<h>, <v>, <model tag>, <opt>]
    <h> array of lag values to fit model to
    <v> array of variogram values to fit model to
    <model tag> string specifying which of the built in models to fit <opt> Optional if True, also returns optimized parameters
  - ”data” → ”umodel” : [<h>, <v>, <user func>, <opt>]
    <h> array of lag values to fit model to
    <v> array of variogram values to fit model to
    <user func> function object of to use for fitting, first arg must be lag values.
    <opt> Optional, if True returns optimized parameters

- ***args/**kwargs**
  Extra parameters to be passed into external functions for fitting or interpolation
  - ”data” → ”poly”
    Passed into numpy.polyfit function
  - ”data” → ”interp”
    Passed into the scipy.interpolate function
  - ”data” → ”bmodel”
    Passed into scipy.curve_fit
  - ”data” → ”umodel”
    Passed into scipy.curve_fit

Returns

- **fvariogram** : function
  Callable function to be used in later variogram calculations. All returned functions will take one argument, lag distance h as an array. The return type of all returned functions will be an array of variogram values at each of those lag distances.

3.3 Revarie

Class to generate random fields based on a variogram given as a function in the 'model' parameter, mean and variance for a number of points given in the x parameter.

3.3.1 Revarie.__init__(self, x, mu, sill, model)

Parameters

- **x** : numpy.ndarray
  Array of shape (m,n) where n is the number of points in an m-dimensional domain. Each row is a point. This does not need to be the same points used to calculate the original variogram.
- **mu** : float
  Spatially-independent mean of field values
- **sill** : float
  Spatially-independent variance of field values
- **model** : function
  Callable with takes numpy array of lag distances as argument and returns numpy array of variogram values. Should only take a single parameter.

3.3.2 Revarie.genf(self)

Generates random field values according to the data given in the initialization function.

Returns

- **fvs** : numpy array
  Array of field values of length x.shape[0] that correspond to each point in x

3.4 benchmarking

Utility to provide easy runtime benchmarking for users.

3.4.1 benchmarking.bench_variogram(tlimit = 15, path = ".")

Run timing benchmark test for variogram class. Results are formatted and printed to a file named "variogram_timing###.dat".

Parameters

- **tlimit** : float
  Approximate wall time limit for tests to run
- **path** : str, path-like
  Path for results file to be printed to

3.4.2 `benchmarking.bench_revarie(tlimit = 15, path = ".")`

Run timing benchmark test for revarie class. Results are formatted and printed to a file named ”revarie_timing###.dat”.

Parameters

- **tlimit** : float
  Approximate wall time limit for tests to run
- **path** : str, path-like
  Path for results file to be printed to

3.4.3 `benchmarking.suite(tlimit = 30, path = ".")`

Run timing benchmark test for both the revarie and variogram classes. Results are formatted and printed to a file named ”revarie_timing###.dat”. Each benchmark is run with equal time division for each object.

Parameters

- **tlimit** : float
  Approximate wall time limit for tests to run
- **path** : str, path-like
  Path for results file to be printed to

4 Theory

The following section is included in this manual to provide a brief description of the statistical theory used in the methods available in the Revarie library and a brief introduction to the concept of a variogram. It is not intended to act as a comprehensive guide for those interested in the study of variography.

4.1 Brief Introduction to Variogram

At it’s core, variogram calculation is a way of quantifying spatial-autocorrelation. Often, autocorrelation may be used in signal processing to identify periodic trends in a time-varying signal. In variography, it can be used to identify similarity across scales in a space-varying field. As mentioned in previous sections, for the purposes of this document, a field is a set of points, each with a spatial coordinate and value. One example of a field may be a set of altitude measurements taken at different locations in a mountain range, each measurement with an associated location in the mountain range. This subsection will present an example of how a variogram may be calculated by hand with a tangible example with minimal equations. It is included to provide readers with an intuitive approach to variogram calculation that may guide their interpretation of variogram results. First, it is useful to present the field we will use for this exercise, it will consist of 10 field values that are linearly spaced between 1 and 10. The values are plotted on traditional x-y axes in Figure 13. However, for this exercise it may be more useful to look at this field as it is given in Figure 14. This representation will be used moving forward in this subsection.

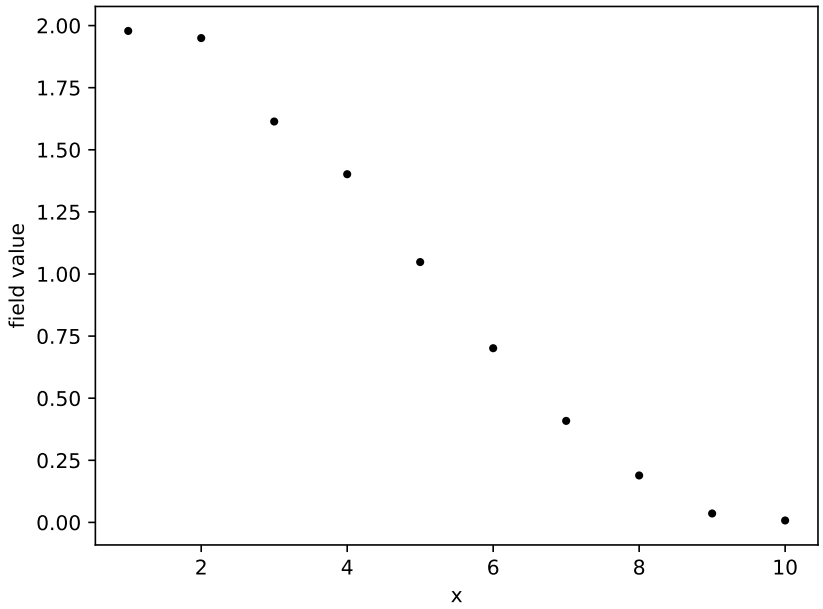


Figure 13: x-y representation of basic field used in example calculation

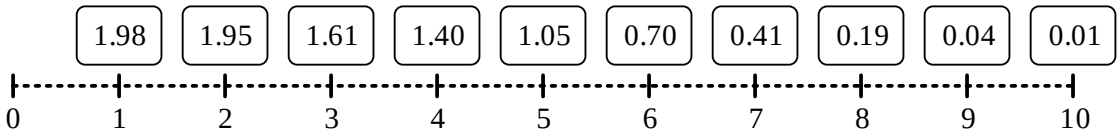
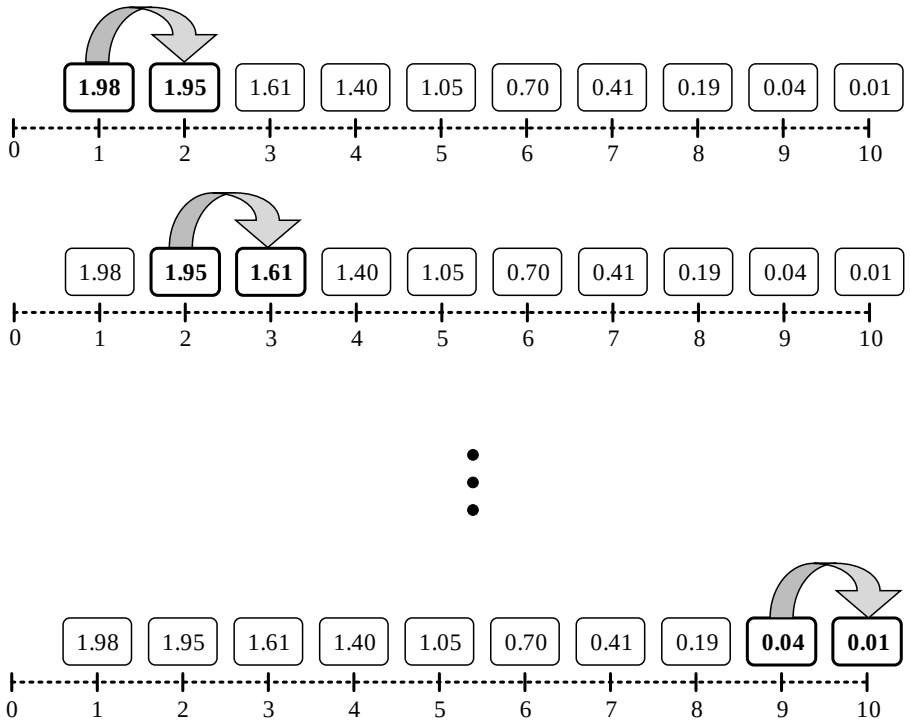


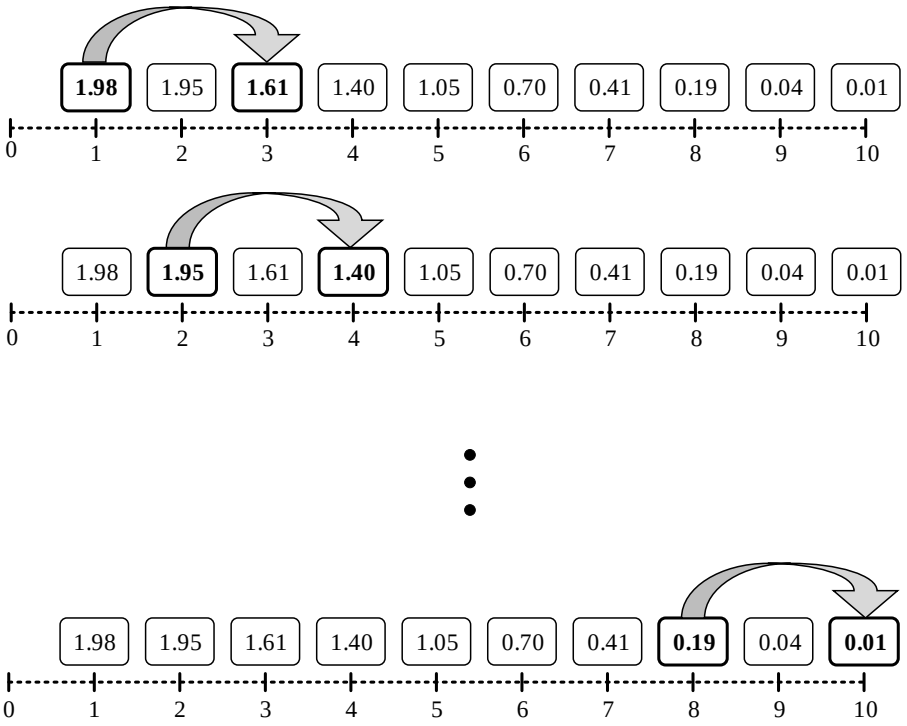
Figure 14: Alternative representation of field. Field values are placed in boxes on top of number line indicating x-coordinate of associated field value.

Now, it is helpful to introduce a tool to aid in our understanding called a ”lag vector”. This vector can start at any point in the field and end at any other point in the field. The point at the beginning of the vector is said to be at the ”tail” of the vector. The point at the termination of the vector is said to be at the ”head” of the vector. Also, the length of the vector is important and will be referred to as a ”lag distance”. Some applications also consider the orientation of the lag vector in higher dimensions but Revarie does not support that sort of variogram calculation so it will be ignored here. Nevertheless, the first useful task is to draw the set of all vectors with a common lag distance and tabulate the values that lie at the head, and tail, of each of these vectors. This is illustrated in the below figure. Here, the representation of the field as shown in Figure 14 is used. Below, the common lag distance depicted is 1 unit. The lag vectors are represented by curved arrows, the tail starting at each field value and the head ending at the field value immediately to the left. This forms a set of adjacent field value pairs that are shown in the table to the right of the figure.



tail	head
1.98	1.95
1.95	1.61
1.61	1.40
1.40	1.05
1.05	0.70
0.70	0.41
0.41	0.19
0.19	0.04
0.04	0.01

Now, it may be useful to depict the set of lag vectors with a common lag distance of 2 units. This is shown below, again with the head/tail values tabulated on the right side of the figure.



tail	head
1.98	1.61
1.95	1.40
1.61	1.05
1.40	0.70
1.05	0.41
0.70	0.19
0.41	0.04
0.19	0.01

By convention, the maximum lag vector length should be half the domain of the field. Therefore, in this application the maximum lag vector length should be 5 units, the head/tail tables can be made for lag distances 3, 4 and 5 just as with the previous examples. Table 1 shows the head/tail tables for the different sizes of lag vectors used in this problem.

1 unit lag		2 unit lag		3 unit lag		4 unit lag		5 unit lag	
tail	head	tail	head	tail	head	tail	head	tail	head
1.98	1.95	1.98	1.61	1.98	1.40	1.98	1.05	1.98	0.70
1.95	1.61	1.95	1.40	1.95	1.05	1.95	0.70	1.95	0.41
1.61	1.40	1.61	1.05	1.61	0.70	1.61	0.41	1.61	0.19
1.40	1.05	1.40	0.70	1.40	0.41	1.40	0.19	1.40	0.04
1.05	0.70	1.05	0.41	1.05	0.19	1.05	0.04	1.05	0.01
0.70	0.41	0.70	0.19	0.70	0.04	0.70	0.01		
0.41	0.19	0.41	0.04	0.41	0.01				
0.19	0.04	0.19	0.01						
0.04	0.01								
R =	0.99		0.98		0.94		0.91		0.89

Table 1: Tabulated head/tail values for assorted lag vector lengths. Pearson correlation coefficient included for each lag vector length.

From here, the correlation between the head/tail values for each lag distance can be plotted, this sort of plot is called a *correlogram*. The correlogram for the current working example is shown n Figure 15. In this form, the analyst can get an idea of the similarity of field values

based on their separation, points separated by small distances are strongly correlated. And oppositely, points separated by large differences are weakly correlated.

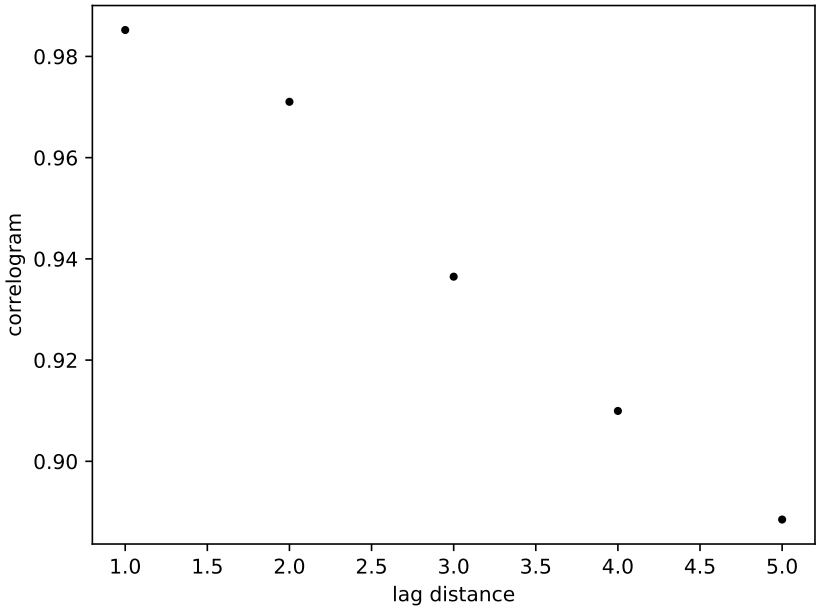


Figure 15: Correlogram of field from example calculation.

Now, separately, we will calculate the variogram. An adaptation to the general Matheron variogram equation for this application is given in Equation 2. Here, γ denotes the variogram value, h denotes the lag distance, N denotes the total number of points in this example problem (10), $D(i,j)$ indicates the distance between two points i and j, f_i denotes the field value at point i and f_j denotes the field value at point j. The expression $\{(i,j) : D(i,j) = h\}$ is the set of point pairs that are h distance apart, in this application, only a finite number of possibilities for h exist.

$$\gamma(h) = \frac{1}{2(N-h)} \sum_{(i,j)\in\{(i,j):D(i,j)=h\}} \left(f_i - f_j\right)^2 \tag{2}$$

Using this equation to calculate the variogram for the example field yields the results shown in Figure 16. We can see here that the variogram increases as the lag distance increases. This suggests that points separated by small distances are more likely to have small differences between their values. And oppositely, points separated by small distances are more likely to have larger differences between their values. *Although not equivalent*, this is a similar observation drawn from the correlogram plot.

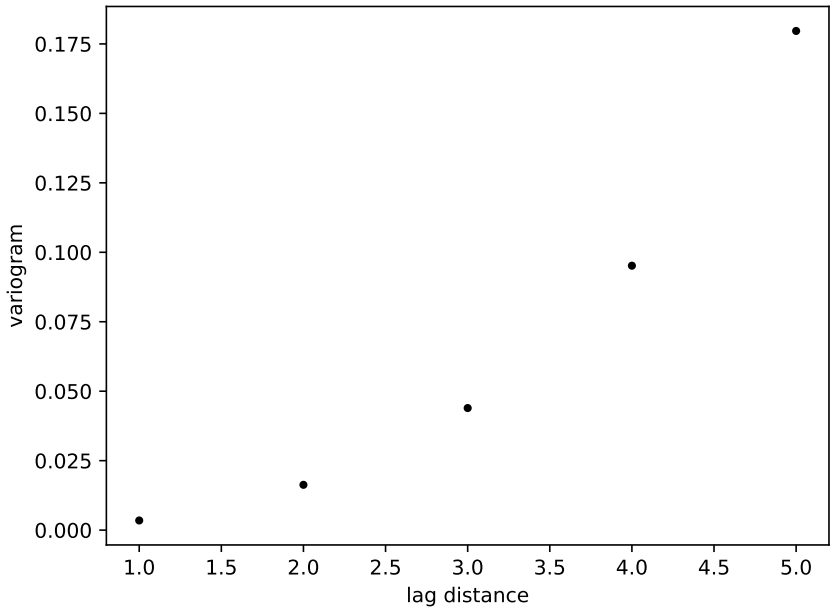


Figure 16: Variogram of field from example calculation.

Overall, if the field was generated as the result of a gaussian process (as is not the case here), the variogram can be related to the correlogram using Equation 3. Here, $c(h)$ denotes the correlogram as a function of lag distance h. $\gamma(h)$ denotes the variogram as a function of lag distance h. σ^2 denotes the spatially-independent variance of the field values.

$$c(h) = 1 - \frac{\gamma(h)}{\sigma^2} \tag{3}$$

4.2 Matheron Variogram

All calculations in the Revarie library rely on the Matheron estimation of the variogram, it is given below in Equation 4 [1]. Here, $\gamma(h)$ denotes the variogram as a function of lag distance, h. $H(h)$ denotes sets of point pairs which are distanced about h apart. $|H(h)|$ denotes the number of point pairs which are distanced about h apart. The set H is dependent on Δ which is the half-bin width of the variogram. Unlike the example given in the previous section, in many applications it may be uncommon to find multiple point pairs that are exactly the same distance apart. Therefore, it is useful to essentially "bin" point pairs to create the set H. This then allows you to approximate a variogram value for h using multiple point pairs which have lag distances equal to $h \pm \Delta$. $D(i, j)$ denotes the exact lag distance between two points i and j. f_i denotes the field value at point i and f_j denotes the field value at point j.

[1] `https://doi.org/10.1016/j.cageo.2007.05.009`

$$\gamma(h) = \frac{1}{|H(h)|} \sum_{(i,j)\in|H(h)|} (f_i - f_j)^2 \tag{4}$$
$$\text{where } H(h) \equiv \{(i,j) : |D(i,j) - h| < \Delta\}$$

4.3 Variogram Models

Currently, there are three different theoretical variogram models built into Revarie, these were selected because they are the most common theoretical variogram models currently used in variography. However before these models are presented, a few notes on general theoretical variograms will be made. Usually, given a variogram model, the variogram can be completely specified with three parameters:

1. **Nugget**: essentially the y-intercept of the variogram, gives some suggestion to the degree of randomness of the field at the smallest scale.
2. **Sill**: value of the variogram where the variogram of the field tends to level off. Typically expected that this value is similar to the spatially-independent variance of the field.
3. **Effective Range**: lag distance where the variogram is equal/similar to its maximum value. Can be used to quantify the range at which similarity between two points due to their proximity is no longer expected. Often, exact definition is different depending on model being used.

Figure 17 is included to show these three parameters on a spherical variogram model.

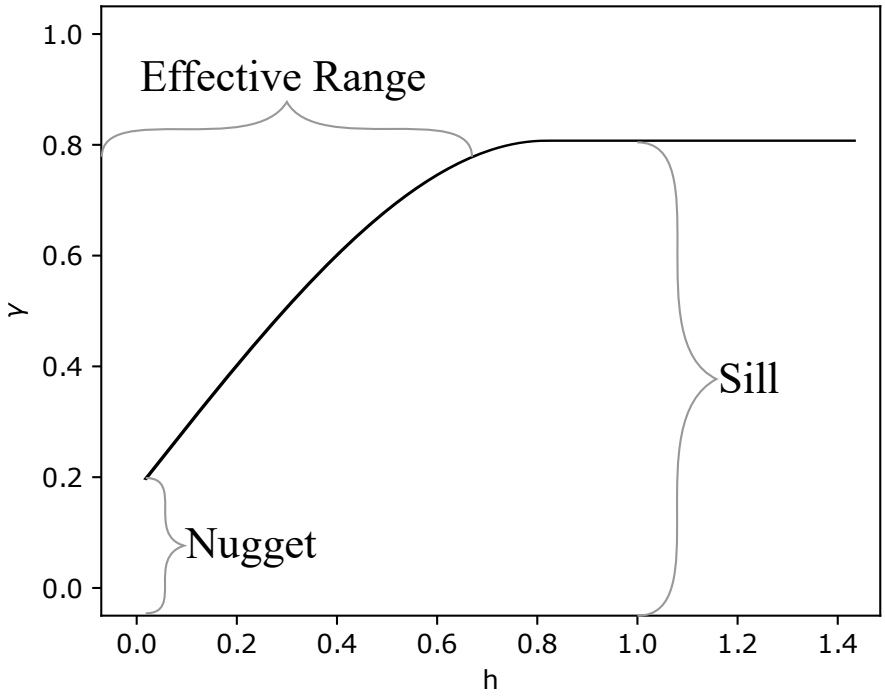


Figure 17: Variogram of field from example calculation.

Currently, there are three models built into the Revarie library. Table 2 lists them and provides their `mtags`. These `mtags` are how these models may be specified within the Revarie library (see Section 2.5 for details).

Name	`mtag`
Spherical	`"sph"`
Exponential	`"exp"`
Gaussian	`"gaus"`

Table 2: Models built into the Revarie library

Moving forward, the equations for each model will be provided. In these equations, the b parameter is the nugget, the r parameter is the effective range and the s parameter is the sill. First, the equation for the spherical model is given in Equation 5. For this model, the effective range is defined as the lag distance where the variogram reaches the sill.

$$\gamma(h) = \begin{cases} b + (c-b)(\frac{3}{2}\frac{h}{r} - \frac{1}{2}\left(\frac{h}{r}\right)^3) & h \leq r \\ b + (c-b) & h > r \end{cases} \tag{5}$$

Second, the equation for the exponential model is given in Equation 6. For this model, the effective range is defined as the lag distance where the variogram reaches about 95% of the sill value (or $100(1-e^{-3})\% \approx 95.0213...\%$ to be exact). It is necessary to make this distinction because for this model the variogram asymptotically approaches the sill, the exact decision is based on convention.

$$\gamma(h) = b + (s-b)(1 - e^{\frac{-3h}{r}}) \tag{6}$$

Finally, the equation for the gaussian model is given in Equation 7. The effective range for this model is defined as the lag distance where the variogram reaches about 98% of the sill value (or $100(1-e^{-4})\% \approx 98.1684...\%$ to be exact).

$$\gamma(h) = b + (s-b)(1 - e^{-\left(\frac{2h}{r}\right)^2}) \tag{7}$$

5 Runtime Benchmarking

Revarie was created for use in computational physics. As such, it is important that this library is capable of large-scale calculations and that the analyst is given the tools they need to evaluate the runtime of the problems they wish to compute. For this, Revarie includes a runtime benchmarking utility easily accessible within the library.

5.1 Benchmark Utility

5.1.1 Import and Run

Due to the numerous configurations for the numpy library that depend on computer characteristics, it may be desirable for a user to do runtime benchmarks for their particular system. In order to make this easy, the `benchmarking` utility is provided. This submodule can be imported with the below code:

```
import revarie.benchmarking
```

This submodule contains three different functions:

- `bench_variogram`: runs a few different variogram calculations for random fields of ascending size until time limit is reached. Records timing results in output file.
- `bench_revarie`: generate a few different random fields of ascending size until time limit is reached. Records timing results in output file.
- `suite`: runs `bench_variogram` and `bench_revarie` with equal time allocations.

An example of how to run one of these would be:

```
revarie.benchmarking.bench_variogram()
```

If there are no previous test output files contained in the current working directory, the output file named "`variogram_timing000.dat`" is created.

5.1.2 Output Format

The output files produced contains 4 sections:

1. **CPU Info:** processor information from system commands
2. **Test Info:** specifications on the test that was run
3. **Results Summary:** total time of all tests run and fitted runtime parameters in form $t = kn^p$ (t is calculation time and n is calculation size).
4. **Timing Data:** columnated calculation size and runtime for each calculation run

Table 3 gives more information on the data contained in the Test Info and Results Summary. A sample output file is given below:

```
% --- CPU info
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  8
On-line CPU(s) list:     0-7
Thread(s) per core:      2
Core(s) per socket:      4
Socket(s):               1
NUMA node(s):            1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   58
Model name:              Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Stepping:                9
CPU MHz:                 3691.324
CPU max MHz:             3900.0000
CPU min MHz:             1600.0000
BogoMIPS:                6784.74
Virtualization:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
```

```
L3 cache:                8192K
NUMA node0 CPU(s):       0-7
Flags:                   fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
    pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
    pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good
    nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64
    monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1
    sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand
    lahf_lm cpuid_fault epb pti ssbd ibrs ibpb stibp tpr_shadow vnmi
    flexpriority ept vpid fsgsbase smep erms xsaveopt dtherm ida arat pln
     pts md_clear flush_l1d

% --- Test Info
Test Date/Time:          2020-08-31 13:56:04.447230
Test Type:               Revarie
Nrange:                  5.00E+01-2.69E+03
Notes:                   1-D

% --- Results Summary
Total Runtime:           20.03746485710144 s
Fitted k:                1.6607E-10
Fitted p:                3.1250E+00
Predicted 1e5 Runtime:   6.9994E+05 s


% --- Timing Data
n              Wall Time [s]
5.00000E+01    5.0044298172E-02
6.00000E+01    2.9106140137E-03
7.20000E+01    3.1726360321E-03
8.60000E+01    2.9566287994E-03
1.03000E+02    8.8076591492E-03
1.23000E+02    6.3507556915E-03
1.47000E+02    7.3390007019E-03
1.76000E+02    1.0611534119E-02
2.11000E+02    1.5202283859E-02
2.53000E+02    1.6932964325E-02
3.03000E+02    2.2617578506E-02
3.63000E+02    3.2533168793E-02
4.35000E+02    4.9937725067E-02
5.22000E+02    7.1791410446E-02
6.26000E+02    1.1121082306E-01
7.51000E+02    1.7229986191E-01
9.01000E+02    2.5178956985E-01
1.08100E+03    4.3161535263E-01
1.29700E+03    8.4677624702E-01
1.55600E+03    1.5259447098E+00
1.86700E+03    2.8681507111E+00
2.24000E+03    4.8895077705E+00
2.68800E+03    8.6389615536E+00
```

:

Result Name	Description
Test Info	
Test Date/Time	date and time test was finished
Test Type	whether the `variogram` or `revarie` functionality was tested
Nrange	range of problem sizes tested
Notes	currently just tells what dimension tests were run in
Results Summary	
Total Runtime	complete runtime for all tests
Fitted k	fitted k parameter for the equation $t = kn^p$
Fitted p	fitted p parameter for the equation $t = kn^p$
Predicted 1e5 Runtime	predicted runtime for a problem with 10^5 points

Table 3: output file data descriptions