GeoStatTools Documentation

Release 1.1.1

Lennart Schueler, Sebastian Mueller

May 20, 2021

CONTENTS

CHAPTER 1

GSTOOLS QUICKSTART

GeoStatTools provides geostatistical tools for random field generation and variogram estimation based on many readily provided and even user-defined covariance models.

1.1 Installation

The package can be installed via pip. On Windows you can install WinPython to get Python and pip running. Also conda provides pip support. Install GSTools by typing the following into the command prompt:

```
pip install gstools
```

To get the latest development version you can install it directly from GitHub:

```
pip install https://github.com/GeoStat-Framework/GSTools/archive/develop.zip
```

To enable the OpenMP support, you have to provide a C compiler, Cython and OpenMP. To get all other dependencies, it is recommended to first install gstools once in the standard way just decribed. Then use the following command:

```
pip install --global-option="--openmp" gstools
```

Or for the development version:

```
pip install --global-option="--openmp" https://github.com/GeoStat-Framework/
↪GSTools/archive/develop.zip
```

If something went wrong during installation, try the `-I` flag from pip.

1.2 Citation

At the moment you can cite the Zenodo code publication of GSTools:

Sebastian Müller, & Lennart Schüler. (2019, October 1). GeoStat-Framework/GSTools: Reverberating Red (Version v1.1.0). Zenodo. http://doi.org/10.5281/zenodo.3468230

A publication for the GeoStat-Framework is in preperation.

1.3 Tutorials and Examples

The documentation also includes some tutorials, showing the most important use cases of GSTools, which are

- Random Field Generation
- The Covariance Model
- Variogram Estimation
- Random Vector Field Generation
- Kriging
- Conditioned random field generation
- Field transformations

Some more examples are provided in the examples folder.

1.4 Spatial Random Field Generation

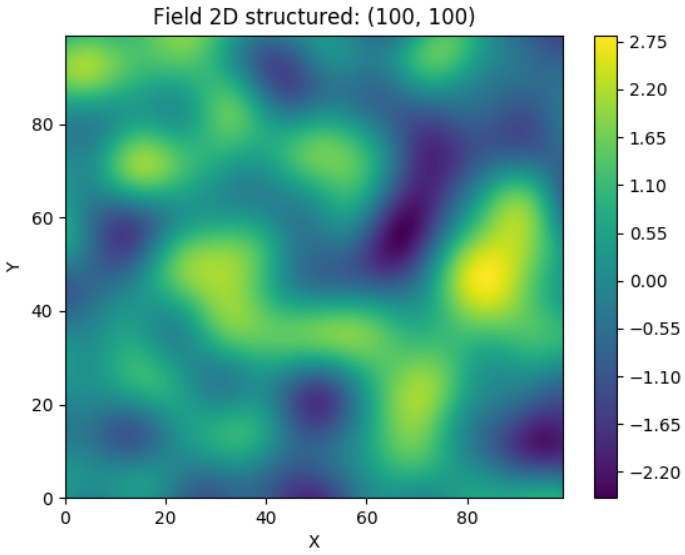
The core of this library is the generation of spatial random fields. These fields are generated using the randomisation method, described by Heße et al. 2014.

Examples

Gaussian Covariance Model

This is an example of how to generate a 2 dimensional spatial random field (*SRF*) with a *Gaussian* covariance model.

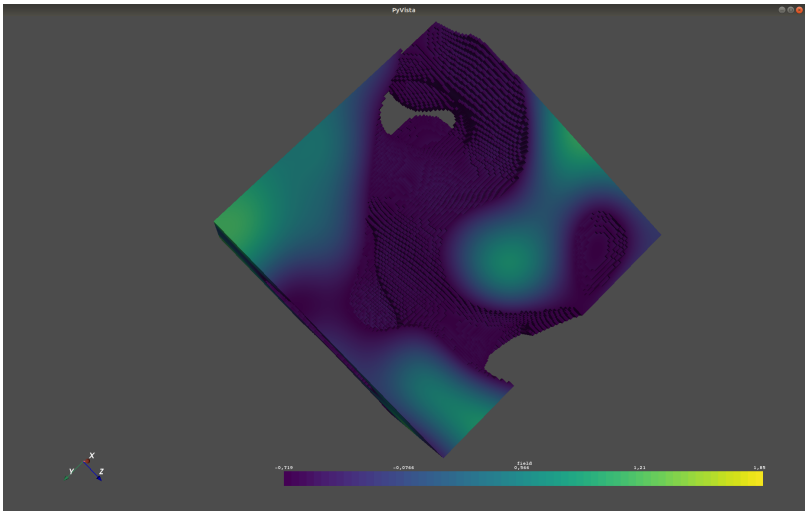
```
from gstools import SRF, Gaussian
import matplotlib.pyplot as plt
# structured field with a size 100x100 and a grid-size of 1x1
x = y = range(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model)
srf((x, y), mesh_type='structured')
srf.plot()
```



A similar example but for a three dimensional field is exported to a VTK file, which can be visualized with ParaView or PyVista in Python:

```
from gstools import SRF, Gaussian
import matplotlib.pyplot as pt
# structured field with a size 100x100x100 and a grid-size of 1x1x1
x = y = z = range(100)
model = Gaussian(dim=3, var=0.6, len_scale=20)
srf = SRF(model)
srf((x, y, z), mesh_type='structured')
srf.vtk_export('3d_field') # Save to a VTK file for ParaView

mesh = srf.to_pyvista() # Create a PyVista mesh for plotting in Python
mesh.threshold_percent(0.5).plot()
```



Truncated Power Law Model

GSTools also implements truncated power law variograms, which can be represented as a superposition of scale dependant modes in form of standard variograms, which are truncated by a lower- ℓ_{low} and an upper length-scale ℓ_{up}.

This example shows the truncated power law (*TPLStable*) based on the *Stable* covariance model and is given

by

$$\gamma_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}}(r) = \int_{\ell_{\mathrm{low}}}^{\ell_{\mathrm{up}}} \gamma(r,\lambda)\frac{\mathrm{d}\lambda}{\lambda}$$

with *Stable* modes on each scale:

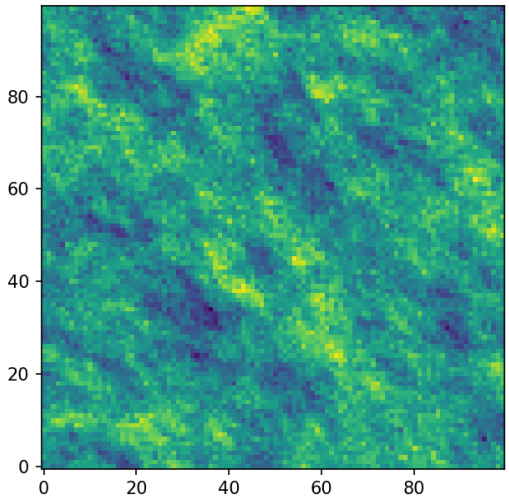
$$\gamma(r,\lambda) = \sigma^2(\lambda)\cdot\left(1-\exp\left[-\left(\frac{r}{\lambda}\right)^{\alpha}\right]\right)$$
$$\sigma^2(\lambda) = C\cdot\lambda^{2H}$$

which gives Gaussian modes for `alpha=2` or Exponential modes for `alpha=1`.

For $\ell_{\mathrm{low}} = 0$ this results in:

$$\gamma_{\ell_{\mathrm{up}}}(r) = \sigma^2_{\ell_{\mathrm{up}}}\cdot\left(1-\frac{2H}{\alpha}\cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell_{\mathrm{up}}}\right)^{\alpha}\right]\right)$$
$$\sigma^2_{\ell_{\mathrm{up}}} = C\cdot\frac{\ell_{\mathrm{up}}^{2H}}{2H}$$

```
import numpy as np
import matplotlib.pyplot as plt
from gstools import SRF, TPLStable
x = y = np.linspace(0, 100, 100)
model = TPLStable(
    dim=2,            # spatial dimension
    var=1,            # variance (C calculated internally, so that `var` is 1)
    len_low=0,        # lower truncation of the power law
    len_scale=10,     # length scale (a.k.a. range), len_up = len_low + len_scale
    nugget=0.1,       # nugget
    anis=0.5,         # anisotropy between main direction and transversal ones
    angles=np.pi/4,   # rotation angles
    alpha=1.5,        # shape parameter from the stable model
    hurst=0.7,        # hurst coefficient from the power law
)
srf = SRF(model, mean=1, mode_no=1000, seed=19970221, verbose=True)
srf((x, y), mesh_type='structured')
srf.plot()
```



1.5 Estimating and fitting variograms

The spatial structure of a field can be analyzed with the variogram, which contains the same information as the covariance function.

All covariance models can be used to fit given variogram data by a simple interface.

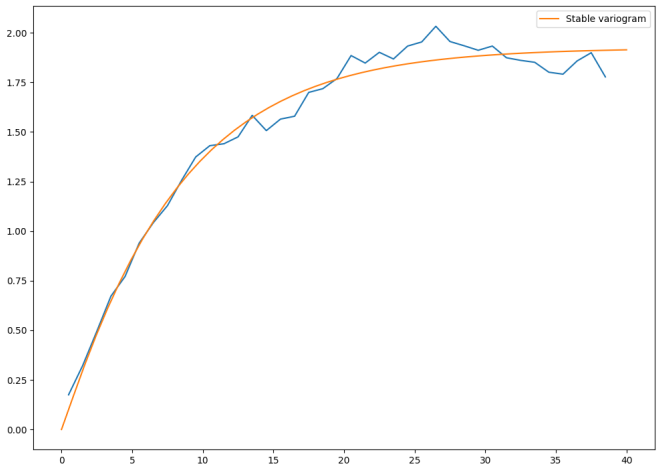
Examples

This is an example of how to estimate the variogram of a 2 dimensional unstructured field and estimate the parameters of the covariance model again.

```
import numpy as np
from gstools import SRF, Exponential, Stable, vario_estimate_unstructured
# generate a synthetic field with an exponential model
x = np.random.RandomState(19970221).rand(1000) * 100.
y = np.random.RandomState(20011012).rand(1000) * 100.
model = Exponential(dim=2, var=2, len_scale=8)
srf = SRF(model, mean=0, seed=19970221)
field = srf((x, y))
# estimate the variogram of the field with 40 bins
bins = np.arange(40)
bin_center, gamma = vario_estimate_unstructured((x, y), field, bins)
# fit the variogram with a stable model. (no nugget fitted)
fit_model = Stable(dim=2)
fit_model.fit_variogram(bin_center, gamma, nugget=False)
# output
ax = fit_model.plot(x_max=40)
ax.plot(bin_center, gamma)
print(fit_model)
```

Which gives:

```
Stable(dim=2, var=1.92, len_scale=8.15, nugget=0.0, anis=[1.], angles=[0.],␣
↪alpha=1.05)
```



1.6 Kriging and Conditioned Random Fields

An important part of geostatistics is Kriging and conditioning spatial random fields to measurements. With conditioned random fields, an ensemble of field realizations with their variability depending on the proximity of the measurements can be generated.

Example

For better visualization, we will condition a 1d field to a few "measurements", generate 100 realizations and plot them:

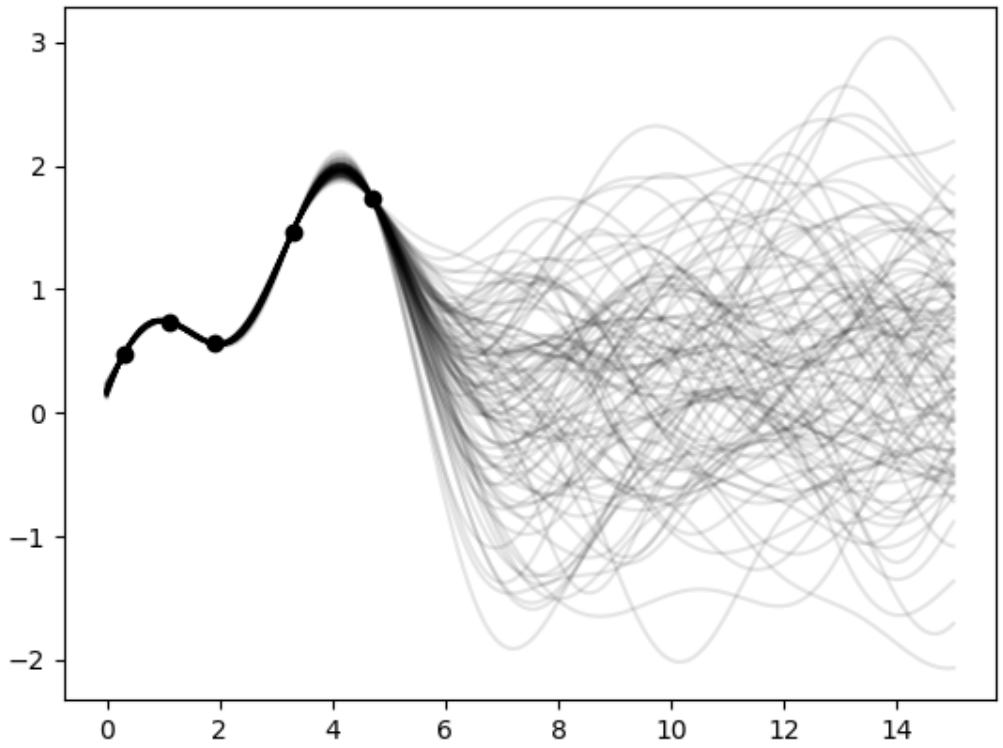
```
import numpy as np
from gstools import Gaussian, SRF
import matplotlib.pyplot as plt

# conditions
cond_pos = [0.3, 1.9, 1.1, 3.3, 4.7]
cond_val = [0.47, 0.56, 0.74, 1.47, 1.74]

gridx = np.linspace(0.0, 15.0, 151)

# spatial random field class
model = Gaussian(dim=1, var=0.5, len_scale=2)
srf = SRF(model)
srf.set_condition(cond_pos, cond_val, "ordinary")

# generate the ensemble of field realizations
fields = []
for i in range(100):
    fields.append(srf(gridx, seed=i))
    plt.plot(gridx, fields[i], color="k", alpha=0.1)
plt.scatter(cond_pos, cond_val, color="k")
plt.show()
```



1.7 User defined covariance models

One of the core-features of GSTools is the powerfull *CovModel* class, which allows to easy define covariance models by the user.

Example

Here we re-implement the Gaussian covariance model by defining just the correlation function, which takes a non-dimensional distance `h = r/l`

```
from gstools import CovModel
import numpy as np
# use CovModel as the base-class
class Gau(CovModel):
    def cor(self, h):
        return np.exp(-h**2)
```

And that's it! With `Gau` you now have a fully working covariance model, which you could use for field generation or variogram fitting as shown above.

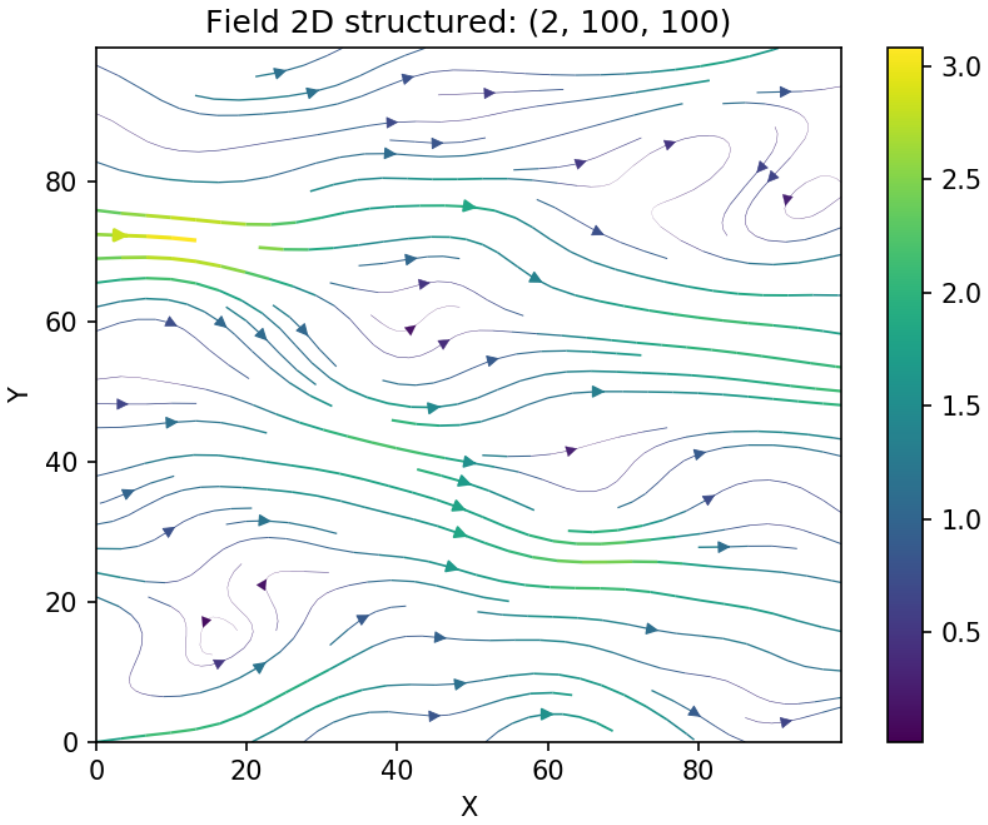
1.8 Incompressible Vector Field Generation

Using the original Kraichnan method, incompressible random spatial vector fields can be generated.

Example

```
import numpy as np
import matplotlib.pyplot as plt
from gstools import SRF, Gaussian
x = np.arange(100)
y = np.arange(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model, generator='VectorField')
srf((x, y), mesh_type='structured', seed=19841203)
srf.plot()
```

yielding



1.9 VTK/PyVista Export

After you have created a field, you may want to save it to file, so we provide a handy VTK export routine using the `vtk_export()` or you could create a VTK/PyVista dataset for use in Python with to `to_pyvista()` method:

```
from gstools import SRF, Gaussian
x = y = range(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model)
srf((x, y), mesh_type='structured')
srf.vtk_export("field") # Saves to a VTK file
mesh = srf.to_pyvista() # Create a VTK/PyVista dataset in memory
mesh.plot()
```

Which gives a RectilinearGrid VTK file `field.vtr` or creates a PyVista mesh in memory for immediate 3D plotting in Python.

1.10 Requirements

- Numpy >= 1.14.5
- SciPy >= 1.1.0
- hankel >= 0.3.6
- emcee >= 3.0.0
- pyevtk
- six

Optional

- matplotlib
- pyvista

1.11 License

CHAPTER 2

GSTOOLS TUTORIALS

In the following you will find several Tutorials on how to use GSTools to explore its whole beauty and power.

2.1 Tutorial 1: Random Field Generation

The main feature of GSTools is the spatial random field generator *SRF*, which can generate random fields following a given covariance model. The generator provides a lot of nice features, which will be explained in the following

Theoretical Background

GSTools generates spatial random fields with a given covariance model or semi-variogram. This is done by using the so-called randomization method. The spatial random field is represented by a stochastic Fourier integral and its discretised modes are evaluated at random frequencies.

GSTools supports arbitrary and non-isotropic covariance models.

A very Simple Example

We are going to start with a very simple example of a spatial random field with an isotropic Gaussian covariance model and following parameters:

- variance $\sigma^2 = 1$
- correlation length $\lambda = 10$

First, we set things up and create the axes for the field. We are going to need the *SRF* class for the actual generation of the spatial random field. But *SRF* also needs a covariance model and we will simply take the *Gaussian* model.

```
from gstools import SRF, Gaussian

x = y = range(100)
```

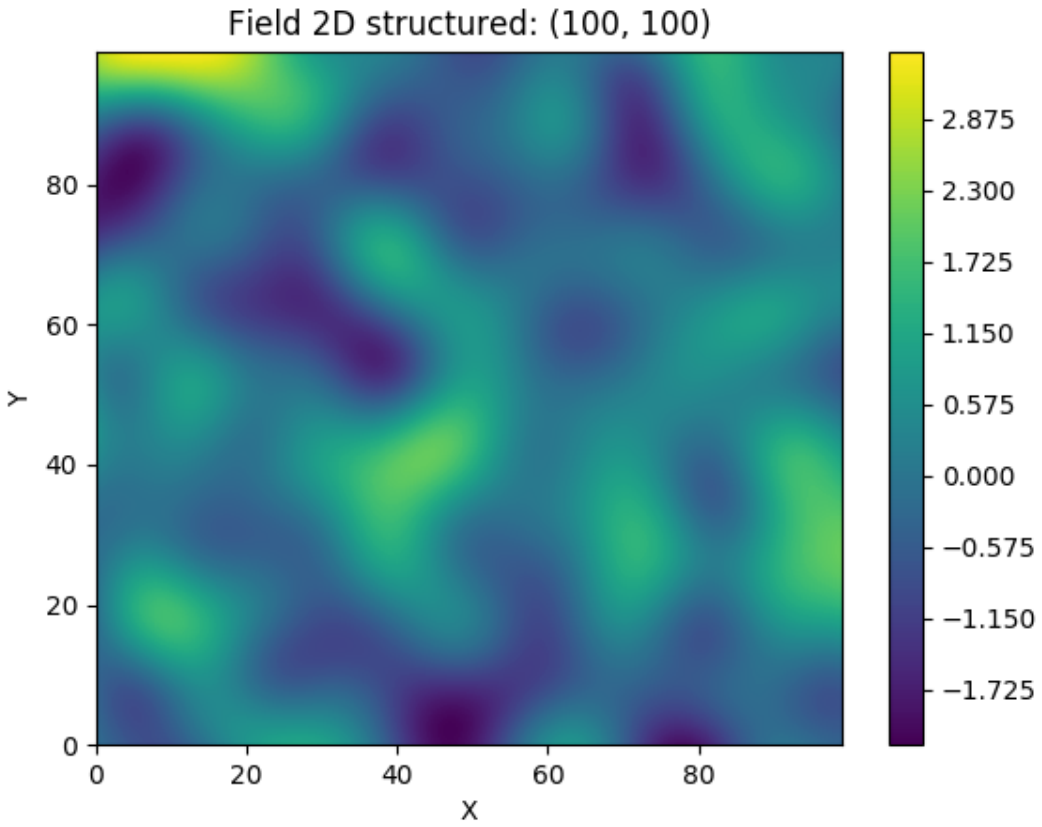
Now we create the covariance model with the parameters σ^2 and λ and hand it over to *SRF*. By specifying a seed, we make sure to create reproducible results:

```
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model, seed=20170519)
```

With these simple steps, everything is ready to create our first random field. We will create the field on a structured grid (as you might have guessed from the *x* and *y*), which makes it easier to plot.

```
field = srf.structured([x, y])
srf.plot()
```

Yielding



Wow, that was pretty easy!

The script can be found in `gstools/examples/00_gaussian.py`

Creating an Ensemble of Fields

Creating an ensemble of random fields would also be a great idea. Let's reuse most of the previous code.

```
import numpy as np
import matplotlib.pyplot as pt
from gstools import SRF, Gaussian

x = y = np.arange(100)

model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model)
```

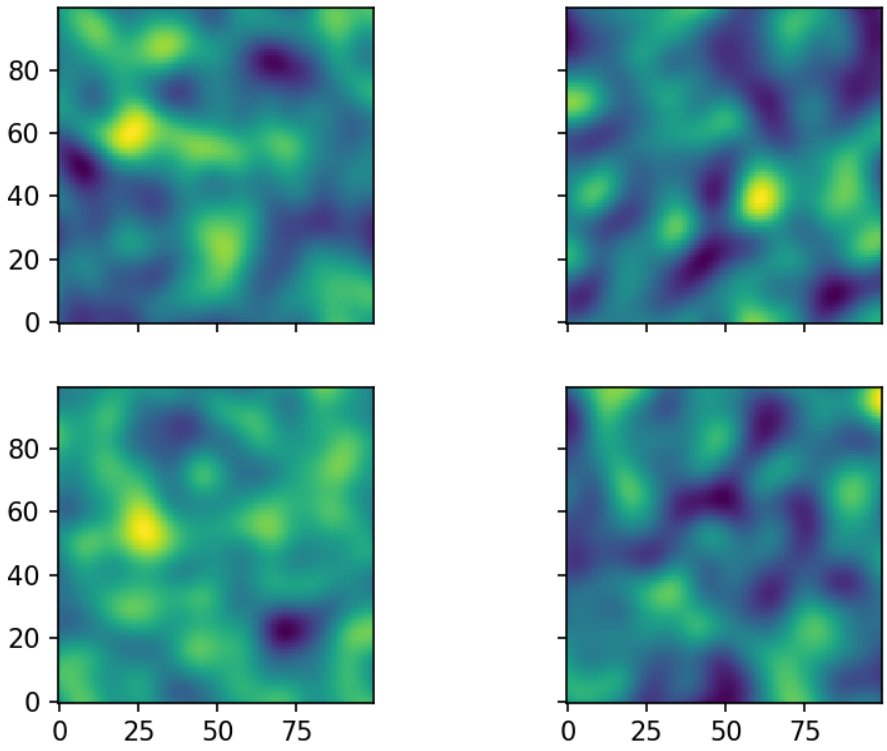
This time, we did not provide a seed to *SRF*, as the seeds will used during the actual computation of the fields. We will create four ensemble members, for better visualisation and save them in a list and in a first step, we will be using the loop counter as the seeds.

```
ens_no = 4
field = []
for i in range(ens_no):
    field.append(srf.structured([x, y], seed=i))
```

Now let's have a look at the results:

```
fig, ax = pt.subplots(2, 2, sharex=True, sharey=True)
ax = ax.flatten()
for i in range(ens_no):
    ax[i].imshow(field[i].T, origin='lower')
pt.show()
```

Yielding



The script can be found in `gstools/examples/05_srf_ensemble.py`

Using better Seeds

It is not always a good idea to use incrementing seeds. Therefore GSTools provides a seed generator *MasterRNG*. The loop, in which the fields are generated would then look like

```
from gstools.random import MasterRNG
seed = MasterRNG(20170519)
for i in range(ens_no):
    field.append(srf.structured([x, y], seed=seed()))
```

Creating Fancier Fields

Only using Gaussian covariance fields gets boring. Now we are going to create much rougher random fields by using an exponential covariance model and we are going to make them anisotropic.

The code is very similar to the previous examples, but with a different covariance model class *Exponential*. As model parameters we a using following

- variance $\sigma^2 = 1$
- correlation length $\lambda = (12, 3)^T$
- rotation angle $\theta = \pi/8$

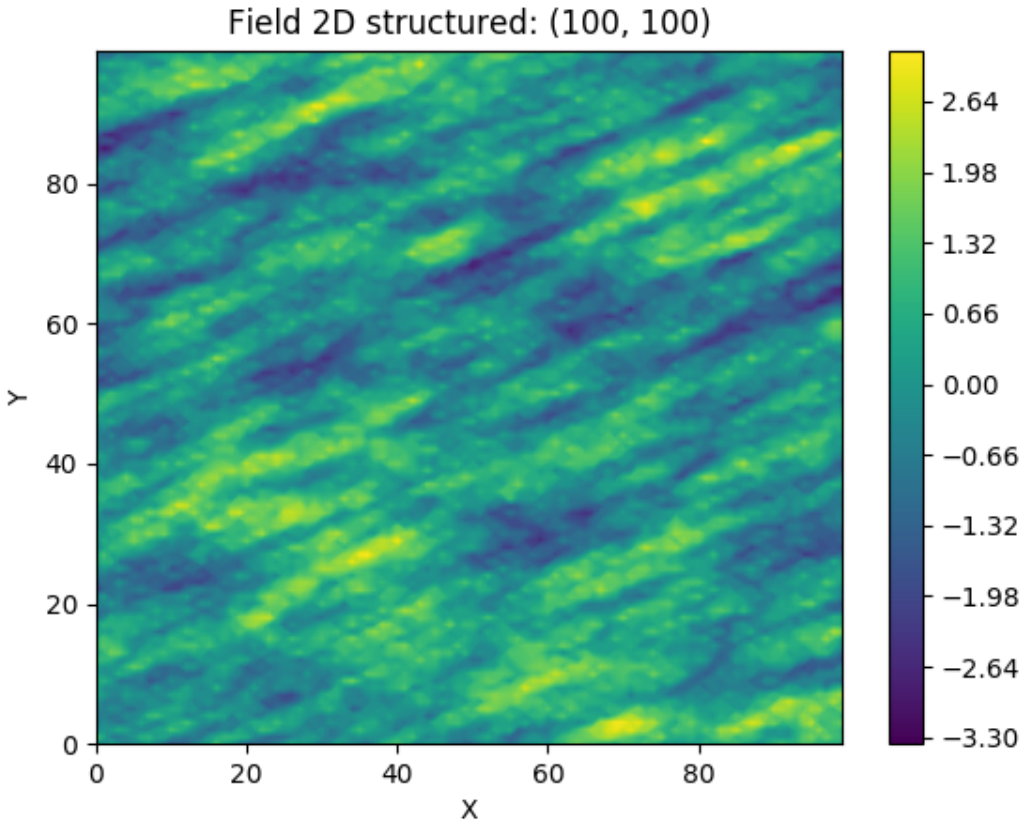
```
import numpy as np
from gstools import SRF, Exponential

x = y = np.arange(100)

model = Exponential(dim=2, var=1, len_scale=[12., 3.], angles=np.pi/8.)
srf = SRF(model, seed=20170519)

srf.structured([x, y])
srf.plot()
```

Yielding



The anisotropy ratio could also have been set with

```
model = Exponential(dim=2, var=1, len_scale=12., anis=3./12., angles=np.pi/8.)
```

Using an Unstructured Grid

For many applications, the random fields are needed on an unstructured grid. Normally, such a grid would be read in, but we can simply generate one and then create a random field at those coordinates.

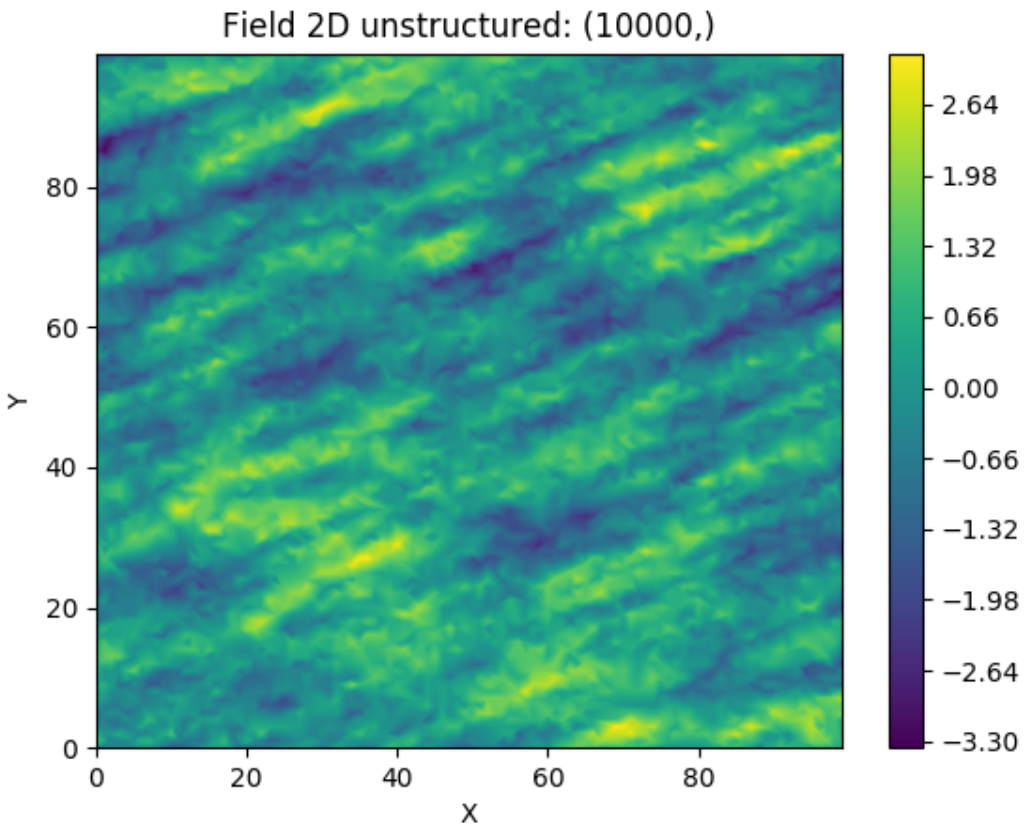
```
import numpy as np
from gstools import SRF, Exponential
from gstools.random import MasterRNG

seed = MasterRNG(19970221)
rng = np.random.RandomState(seed())
x = rng.randint(0, 100, size=10000)
y = rng.randint(0, 100, size=10000)

model = Exponential(dim=2, var=1, len_scale=[12., 3.], angles=np.pi/8.)

srf = SRF(model, seed=20170519)
srf([x, y])
srf.plot()
```

Yielding



Comparing this image to the previous one, you can see that be using the same seed, the same field can be computed on different grids.

The script can be found in `gstools/examples/06_unstr_srf_export.py`

Exporting a Field

Using the field from *previous example*, it can simply be exported to the file `field.vtu` and viewed by e.g. paraview with following lines of code

```
srf.vtk_export("field")
```

Or it could visualized immediately in Python using PyVista:

```
mesh = srf.to_pyvista("field")
mesh.plot()
```

The script can be found in `gstools/examples/04_export.py` and in `gstools/examples/06_unstr_srf_export.py`

Merging two Fields

We can even generate the same field realisation on different grids. Let's try to merge two unstructured rectangular fields. The first field will be generated exactly like in example *Using an Unstructured Grid*:

```
import numpy as np
import matplotlib.pyplot as pt
from gstools import SRF, Exponential
from gstools.random import MasterRNG

seed = MasterRNG(19970221)
rng = np.random.RandomState(seed())
x = rng.randint(0, 100, size=10000)
y = rng.randint(0, 100, size=10000)

model = Exponential(dim=2, var=1, len_scale=[12., 3.], angles=np.pi/8.)

srf = SRF(model, seed=20170519)

field = srf([x, y])
```

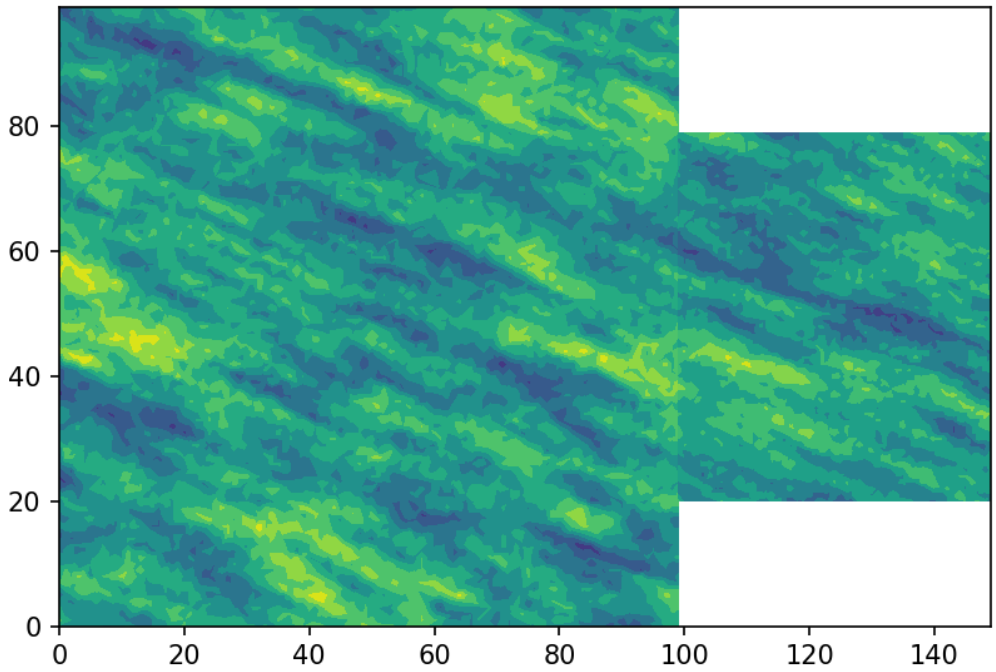
But now we extend the field on the right hand side by creating a new unstructured grid and calculating a field with the same parameters and the same seed on it:

```
# new grid
seed = MasterRNG(20011012)
rng = np.random.RandomState(seed())
x2 = rng.randint(99, 150, size=10000)
y2 = rng.randint(20, 80, size=10000)

field2 = srf((x2, y2))

pt.tricontourf(x, y, field.T)
pt.tricontourf(x2, y2, field2.T)
pt.axes().set_aspect('equal')
pt.show()
```

Yielding



The slight mismatch where the two fields were merged is merely due to interpolation problems of the plotting routine. You can convince yourself be increasing the resolution of the grids by a factor of 10.

Of course, this merging could also have been done by appending the grid point `(x2, y2)` to the original grid `(x, y)` before generating the field. But one application scenario would be to generate hugh fields, which would not fit into memory anymore.

The script can be found in `gstools/examples/07_srf_merge.py`

2.2 Tutorial 2: The Covariance Model

One of the core-features of GSTools is the powerful *CovModel* class, which allows you to easily define arbitrary covariance models by yourself. The resulting models provide a bunch of nice features to explore the covariance models.

Theoretical Backgound

A covariance model is used to characterize the semi-variogram, denoted by γ, of a spatial random field. In GSTools, we use the following form for an isotropic and stationary field:

$$\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$$

Where:

- $\mathrm{cor}(r)$ is the so called correlation function depending on the distance r
- σ^2 is the variance
- n is the nugget (subscale variance)

Note: We are not limited to isotropic models. We support anisotropy ratios for length scales in orthogonal transversal directions like:

- x (main direction)
- y (1. transversal direction)
- z (2. transversal direction)

These main directions can also be rotated, but we will come to that later.

Example

Let us start with a short example of a self defined model (Of course, we provide a lot of predefined models [See: *gstools.covmodel*], but they all work the same way). Therefore we reimplement the Gaussian covariance model by defining just the correlation function:

```
from gstools import CovModel
import numpy as np
# use CovModel as the base-class
class Gau(CovModel):
    def correlation(self, r):
        return np.exp(-(r/self.len_scale)**2)
```

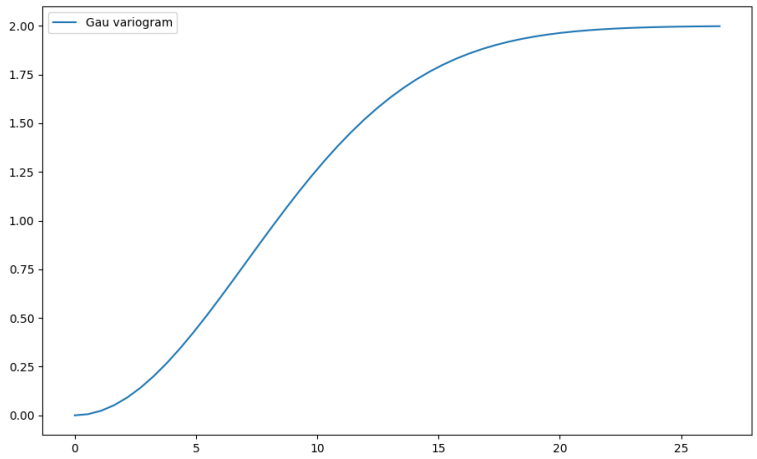
Now we can instantiate this model:

```
model = Gau(dim=2, var=2., len_scale=10)
```

To have a look at the variogram, let's plot it:

```
from gstools.covmodel.plot import plot_variogram
plot_variogram(model)
```

Which gives:



Parameters

We already used some parameters, which every covariance models has. The basic ones are:

- **dim** : dimension of the model
- **var** : variance of the model (on top of the subscale variance)
- **len_scale** : length scale of the model
- **nugget** : nugget (subscale variance) of the model

These are the common parameters used to characterize a covariance model and are therefore used by every model in GSTools. You can also access and reset them:

```
print(model.dim, model.var, model.len_scale, model.nugget, model.sill)
model.dim = 3
model.var = 1
model.len_scale = 15
model.nugget = 0.1
print(model.dim, model.var, model.len_scale, model.nugget, model.sill)
```

Which gives:

```
2 2.0 10 0.0 2.0
3 1.0 15 0.1 1.1
```

Note:

- The sill of the variogram is calculated by `sill = variance + nugget` So we treat the variance as everything **above** the nugget, which is sometimes called **partial sill**.
- A covariance model can also have additional parameters.

Anisotropy

The internally used (semi-) variogram represents the isotropic case for the model. Nevertheless, you can provide anisotropy ratios by:

```
model = Gau(dim=3, var=2., len_scale=10, anis=0.5)
print(model.anis)
print(model.len_scale_vec)
```

Which gives:

```
[0.5 1. ]
[10.  5. 10.]
```

As you can see, we defined just one anisotropy-ratio and the second transversal direction was filled up with `1.` and you can get the length-scales in each direction by the attribute *`len_scale_vec`*. For full control you can set a list of anistropy ratios: `anis=[0.5, 0.4]`.

Alternatively you can provide a list of length-scales:

```
model = Gau(dim=3, var=2., len_scale=[10, 5, 4])
print(model.anis)
print(model.len_scale)
print(model.len_scale_vec)
```

Which gives:

```
[0.5 0.4]
10
[10.  5.  4.]
```

Rotation Angles

The main directions of the field don't have to coincide with the spatial directions x, y and z. Therefore you can provide rotation angles for the model:

```
model = Gau(dim=3, var=2., len_scale=10, angles=2.5)
print(model.angles)
```

Which gives:

```
[2.5 0.  0. ]
```

Again, the angles were filled up with `0.` to match the dimension and you could also provide a list of angles. The number of angles depends on the given dimension:

- in 1D: no rotation performable
- in 2D: given as rotation around z-axis
- in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

Methods

The covariance model class *`CovModel`* of GSTools provides a set of handy methods.

Basics

One of the following functions defines the main characterization of the variogram:

- `variogram` : The variogram of the model given by

$$\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$$

- `covariance` : The (auto-)covariance of the model given by

$$C(r) = \sigma^2 \cdot \mathrm{cor}(r)$$

- `correlation` : The (auto-)correlation (or normalized covariance) of the model given by

$$\mathrm{cor}\,(r)$$

As you can see, it is the easiest way to define a covariance model by giving a correlation function as demonstrated by the above model `Gau`. If one of the above functions is given, the others will be determined:

```
model = Gau(dim=3, var=2., len_scale=10, nugget=0.5)
print(model.variogram(10.))
print(model.covariance(10.))
print(model.correlation(10.))
```

Which gives:

```
1.7642411176571153
0.6321205588285577
0.7357588823428847
0.36787944117144233
```

Spectral methods

The spectrum of a covariance model is given by:

$$S(\mathbf{k}) = \left(\frac{1}{2\pi}\right)^n \int C(\|\mathbf{r}\|) e^{ib\mathbf{k}\cdot\mathbf{r}} d^n\mathbf{r}$$

Since the covariance function $C(r)$ is radially symmetric, we can calculate this by the hankel-transformation:

$$S(k) = \left(\frac{1}{2\pi}\right)^n \cdot \frac{(2\pi)^{n/2}}{(bk)^{n/2-1}} \int_0^\infty r^{n/2-1} C(r) J_{n/2-1}(bkr) r dr$$

Where $k = \|\mathbf{k}\|$.

Depending on the spectrum, the spectral-density is defined by:

$$\tilde{S}(k) = \frac{S(k)}{\sigma^2}$$

You can access these methods by:

```
model = Gau(dim=3, var=2., len_scale=10)
print(model.spectrum(0.1))
print(model.spectral_density(0.1))
```

Which gives:

```
34.96564773852395
17.482823869261974
```

Note: The spectral-density is given by the radius of the input phase. But it is **not** a probability density function for the radius of the phase. To obtain the pdf for the phase-radius, you can use the methods *`spectral_rad_pdf`* or *`ln_spectral_rad_pdf`* for the logarithm.

The user can also provide a cdf (cumulative distribution function) by defining a method called `spectral_rad_cdf` and/or a ppf (percent-point function) by `spectral_rad_ppf`.

The attributes *`has_cdf`* and *`has_ppf`* will check for that.

Different scales

Besides the length-scale, there are many other ways of characterizing a certain scale of a covariance model. We provide two common scales with the covariance model.

Integral scale

The integral scale of a covariance model is calculated by:

$$I = \int_0^\infty \mathrm{cor}(r) dr$$

You can access it by:

```
model = Gau(dim=3, var=2., len_scale=10)
print(model.integral_scale)
print(model.integral_scale_vec)
```

Which gives:

```
8.862269254527579
[8.86226925 8.86226925 8.86226925]
```

You can also specify integral length scales like the ordinary length scale, and len_scale/anis will be recalculated:

```
model = Gau(dim=3, var=2., integral_scale=[10, 4, 2])
print(model.anis)
print(model.len_scale)
print(model.len_scale_vec)
print(model.integral_scale)
print(model.integral_scale_vec)
```

Which gives:

```
[0.4 0.2]
11.283791670955127
[11.28379167  4.51351667  2.25675833]
10.000000000000002
[10.  4.  2.]
```

Percentile scale

Another scale characterizing the covariance model, is the percentile scale. It is the distance, where the normalized variogram reaches a certain percentage of its sill.

```
model = Gau(dim=3, var=2., len_scale=10)
print(model.percentile_scale(0.9))
```

Which gives:

```
15.174271293851463
```

Note: The nugget is neglected by this percentile_scale.

Additional Parameters

Let's pimp our self-defined model `Gau` by setting the exponent as an additional parameter:

$$\mathrm{cor}(r) := \exp\left(-\left(\frac{r}{\ell}\right)^{\alpha}\right)$$

This leads to the so called **stable** covariance model and we can define it by

```
class Stab(CovModel):
    def default_opt_arg(self):
        return {"alpha": 1.5}
    def correlation(self, r):
        return np.exp(-(r/self.len_scale)**self.alpha)
```

As you can see, we override the method *CovModel.default_opt_arg* to provide a standard value for the optional argument `alpha` and we can access it in the correlation function by `self.alpha`

Now we can instantiate this model:

```
model1 = Stab(dim=2, var=2., len_scale=10)
model2 = Stab(dim=2, var=2., len_scale=10, alpha=0.5)
print(model1)
print(model2)
```

Which gives:

```
Stab(dim=2, var=2.0, len_scale=10, nugget=0.0, anis=[1.], angles=[0.], alpha=1.5)
Stab(dim=2, var=2.0, len_scale=10, nugget=0.0, anis=[1.], angles=[0.], alpha=0.5)
```

Note: You don't have to overrid the *CovModel.default_opt_arg*, but you will get a ValueError if you don't set it on creation.

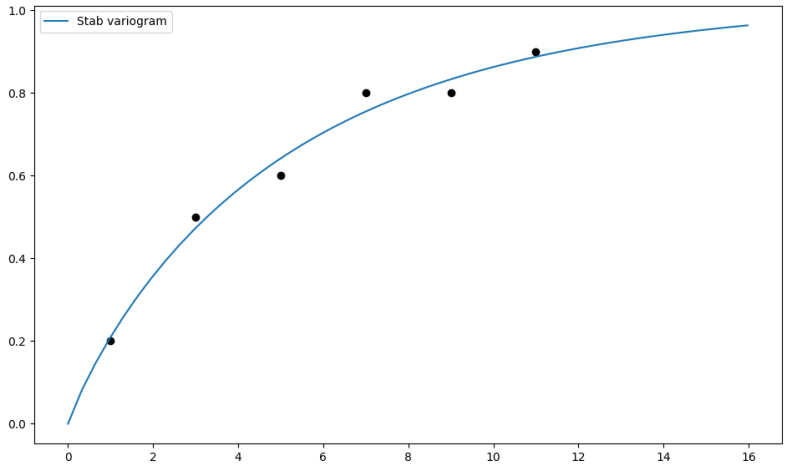
Fitting variogram data

The model class comes with a routine to fit the model-parameters to given variogram data. Have a look at the following:

```
# data
x = [1.0, 3.0, 5.0, 7.0, 9.0, 11.0]
y = [0.2, 0.5, 0.6, 0.8, 0.8, 0.9]
# fitting model
model = Stab(dim=2)
# we have to provide boundaries for the parameters
model.set_arg_bounds(alpha=[0, 3])
# fit the model to given data, deselect nugget
results, pcov = model.fit_variogram(x, y, nugget=False)
print(results)
# show the fitting
from matplotlib import pyplot as plt
from gstools.covmodel.plot import plot_variogram
plt.scatter(x, y, color="k")
plot_variogram(model)
plt.show()
```

Which gives:

```
{'var': 1.024575782651677,
 'len_scale': 5.081620691462197,
 'nugget': 0.0,
 'alpha': 0.906705123369987}
```



As you can see, we have to provide boundaries for the parameters. As a default, the following bounds are set:

- additional parameters: `[-np.inf, np.inf]`
- variance: `[0.0, np.inf]`
- len_scale: `[0.0, np.inf]`
- nugget: `[0.0, np.inf]`

Also, you can deselect parameters from fitting, so their predefined values will be kept. In our case, we fixed a `nugget` of `0.0`, which was set by default. You can deselect any standard or optional argument of the covariance model. The second return value `pcov` is the estimated covariance of `popt` from the used scipy routine `scipy.optimize.curve_fit`.

You can use the following methods to manipulate the used bounds:

CovModel.default_opt_arg_bounds()	Provide default boundaries for optional arguments.
CovModel.default_arg_bounds()	Provide default boundaries for arguments.
CovModel.set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
CovModel.check_arg_bounds()	Check arguments to be within the given bounds.

You can override the *CovModel.default_opt_arg_bounds* to provide standard bounds for your additional parameters.

To access the bounds you can use:

CovModel.var_bounds	Bounds for the variance.
CovModel.len_scale_bounds	Bounds for the lenght scale.
CovModel.nugget_bounds	Bounds for the nugget.
CovModel.opt_arg_bounds	Bounds for the optional arguments.
CovModel.arg_bounds	Bounds for all parameters.

Provided Covariance Models

The following standard covariance models are provided by GSTools

Gaussian([dim, var, len_scale, nugget, ...])	The Gaussian covariance model.
Exponential([dim, var, len_scale, nugget, ...])	The Exponential covariance model.
Matern([dim, var, len_scale, nugget, anis, ...])	The Matérn covariance model.
Stable([dim, var, len_scale, nugget, anis, ...])	The stable covariance model.
Rational([dim, var, len_scale, nugget, ...])	The rational quadratic covariance model.

Continued on next page

Table 3 – continued from previous page

Linear([dim, var, len_scale, nugget, anis, ...])	The bounded linear covariance model.
Circular([dim, var, len_scale, nugget, ...])	The circular covariance model.
Spherical([dim, var, len_scale, nugget, ...])	The Spherical covariance model.
Intersection([dim, var, len_scale, nugget, ...])	The Intersection covariance model.

As a special feature, we also provide truncated power law (TPL) covariance models

TPLGaussian([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Gaussian modes.
TPLExponential([dim, var, len_scale, ...])	Truncated-Power-Law with Exponential modes.
TPLStable([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Stable modes.

2.3 Tutorial 3: Variogram Estimation

Estimating the spatial correlations is an important part of geostatistics. These spatial correlations can be expressed by the variogram, which can be estimated with the subpackage *gstools.variogram*. The variograms can be estimated on structured and unstructured grids.

Theoretical Background

The same (semi-)variogram as *the Covariance Model* is being used by this subpackage.

An Example with Actual Data

This example is going to be a bit more extensive and we are going to do some basic data preprocessing for the actual variogram estimation. But this example will be self-contained and all data gathering and processing will be done in this example script.

The complete script can be found in `gstools/examples/08_variogram_estimation.py`

This example will only work with Python 3.

The Data

We are going to analyse the Herten aquifer, which is situated in Southern Germany. Multiple outcrop faces where surveyed and interpolated to a 3D dataset. In these publications, you can find more information about the data:

Bayer, Peter; Comunian, Alessandro; Höyng, Dominik; Mariethoz, Gregoire (2015): Physicochemical properties and 3D geostatistical simulations of the Herten and the Descalvado aquifer analogs. PANGAEA, https://doi.org/10.1594/PANGAEA.844167,
Supplement to: Bayer, P et al. (2015): Three-dimensional multi-facies realizations of sedimentary reservoir and aquifer analogs. Scientific Data, 2, 150033, https://doi.org/10.1038/sdata.2015.33

Retrieving the Data

To begin with, we need to download and extract the data. Therefore, we are going to use some built-in Python libraries. For simplicity, many values and strings will be hardcoded.

```
import os
import urllib.request
import zipfile
import numpy as np
import matplotlib.pyplot as pt

def download_herten():
    # download the data, warning: its about 250MB
    print('Downloading Herten data')
    data_filename = 'data.zip'
    data_url = 'http://store.pangaea.de/Publications/Bayer_et_al_2015/Herten-
↪analog.zip'
    urllib.request.urlretrieve(data_url, 'data.zip')

    # extract the data
    with zipfile.ZipFile(data_filename, 'r') as zf:
        zf.extract(os.path.join('Herten-analog', 'sim-big_1000x1000x140',
                                'sim.vtk'))
```

That was that. But we also need a script to convert the data into a format we can use. This script is also kindly provided by the authors. We can download this script in a very similar manner as the data:

```
def download_scripts():
    # download a script for file conversion
    print('Downloading scripts')
    tools_filename = 'scripts.zip'
    tool_url = 'http://store.pangaea.de/Publications/Bayer_et_al_2015/tools.zip'
    urllib.request.urlretrieve(tool_url, tools_filename)

    # only extract the script we need
    with zipfile.ZipFile(tools_filename, 'r') as zf:
        zf.extract(os.path.join('tools', 'vtk2gslib.py'))
```

These two functions can now be called:

```
download_herten()
download_scripts()
```

Preprocessing the Data

First of all, we have to convert the data with the script we just downloaded

```
# import the downloaded conversion script
from tools.vtk2gslib import vtk2numpy

# load the Herten aquifer with the downloaded vtk2numpy routine
print('Loading data')
herten, grid = vtk2numpy(os.path.join('Herten-analog', 'sim-big_1000x1000x140',
↪'sim.vtk'))
```

The data only contains facies, but from the supplementary data, we know the hydraulic conductivity values of each facies, which we will simply paste here and assign them to the correct facies

```
# conductivity values per fazies from the supplementary data
cond = np.array([2.50E-04, 2.30E-04, 6.10E-05, 2.60E-02, 1.30E-01,
                 9.50E-02, 4.30E-05, 6.00E-07, 2.30E-03, 1.40E-04,])

# asign the conductivities to the facies
herten_cond = cond[herten]
```

Next, we are going to calculate the transmissivity, by integrating over the vertical axis

```
# integrate over the vertical axis, calculate transmissivity
herten_log_trans = np.log(np.sum(herten_cond, axis=2) * grid['dz'])
```

The Herten data provides information about the grid, which was already used in the previous code block. From this information, we can create our own grid on which we can estimate the variogram. As a first step, we are going to estimate an isotropic variogram, meaning that we will take point pairs from all directions into account. An unstructured grid is a natural choice for this. Therefore, we are going to create an unstructured grid from the given, structured one. For this, we are going to write another small function

```
def create_unstructured_grid(x_s, y_s):
    x_u, y_u = np.meshgrid(x_s, y_s)
    len_unstruct = len(x_s) * len(y_s)
    x_u = np.reshape(x_u, len_unstruct)
    y_u = np.reshape(y_u, len_unstruct)
    return x_u, y_u

# create a structured grid on which the data is defined
```

(continues on next page)

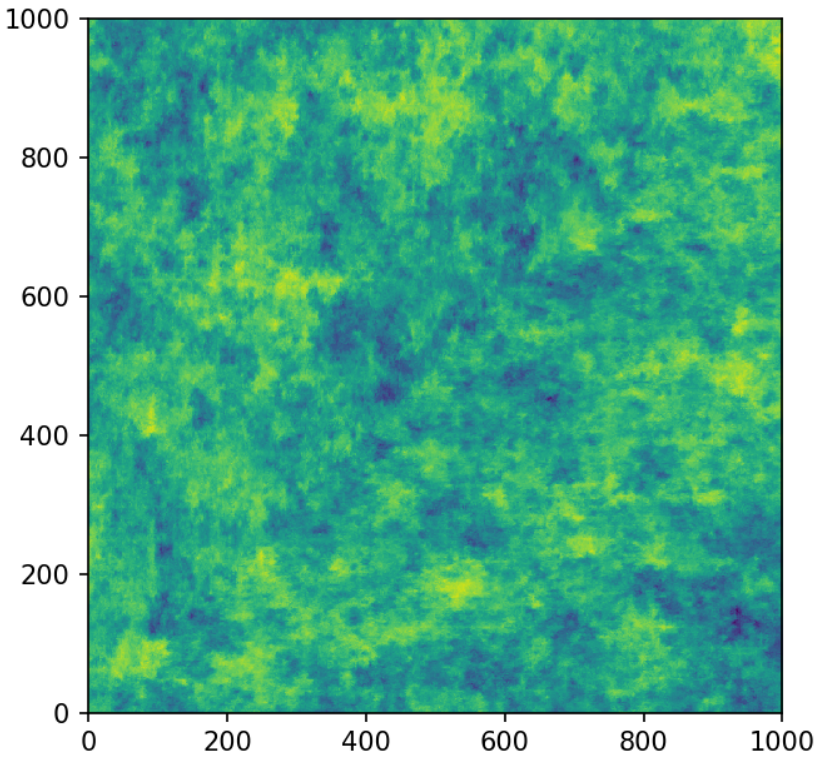
(continued from previous page)

```
x_s = np.arange(grid['ox'], grid['nx']*grid['dx'], grid['dx'])
y_s = np.arange(grid['oy'], grid['ny']*grid['dy'], grid['dy'])

# create an unstructured grid for the variogram estimation
x_u, y_u = create_unstructured_grid(x_s, y_s)
```

Let's have a look at the transmissivity field of the Herten aquifer

```
pt.imshow(herten_log_trans.T, origin='lower', aspect='equal')
pt.show()
```



Estimating the Variogram

Finally, everything is ready for the variogram estimation. For the unstructured method, we have to define the bins on which the variogram will be estimated. Through expert knowledge (i.e. fiddling around), we assume that the main features of the variogram will be below 10 metres distance. And because the data has a high spatial resolution, the resolution of the bins can also be high. The transmissivity data is still defined on a structured grid, but we can simply flatten it with `numpy.ndarray.flatten`, in order to bring it into the right shape. It might be more memory efficient to use `herten_log_trans.reshape(-1)`, but for better readability, we will stick to `numpy.ndarray.flatten`. Taking all data points into account would take a very long time (expert knowledge *wink*), thus we will only take 2000 datapoints into account, which are sampled randomly. In order to make the exact results reproducible, we can also set a seed.

```
from gstools import vario_estimate_unstructured

bins = np.linspace(0, 10, 50)
print('Estimating unstructured variogram')
```

(continues on next page)

(continued from previous page)

```
bin_center, gamma = vario_estimate_unstructured(
    (x_u, y_u),
    herten_log_trans.flatten(),
    bins,
    sampling_size=2000,
    sampling_seed=19920516,
    )
```

The estimated variogram is calculated on the centre of the given bins, therefore, the `bin_center` array is also returned.

Fitting the Variogram

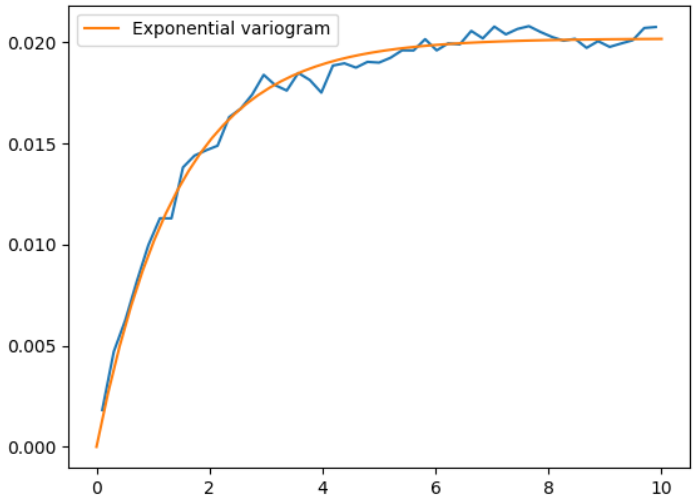
Now, we can see, if the estimated variogram can be modelled by a common variogram model. Let's try the *Exponential* model.

```
from gstools import Exponential

# fit an exponential model
fit_model = Exponential(dim=2)
fit_model.fit_variogram(bin_center, gamma, nugget=False)
```

Finally, we can visualise some results. For quickly plotting a covariance model, GSTools provides some helper functions.

```
from gstools.covmodel.plot import plot_variogram
pt.plot(bin_center, gamma)
plot_variogram(fit_model, x_max=bins[-1])
pt.show()
```



That looks like a pretty good fit! By printing the model, we can directly see the fitted parameters

```
print(fit_model)
```

which gives

```
Exponential(dim=2, var=0.020193095802479327, len_scale=1.4480057557321007,␣
↪nugget=0.0, anis=[1.], angles=[0.])
```

With this data, we could start generating new ensembles of the Herten aquifer with the *SRF* class.

Estimating the Variogram in Specific Directions

Estimating a variogram on a structured grid gives us the possibility to only consider values in a specific direction. This could be a first test, to see if the data is anisotropic. In order to speed up the calculations, we are going to only use every 10th datapoint and for a comparison with the isotropic variogram calculated earlier, we only need the first 21 array items.

```
x_s = x_s[::10][:21]
y_s = y_s[::10][:21]
herten_trans_log = herten_log_trans[::10,::10]
```

With this much smaller data set, we can immediately estimate the variogram in the x- and y-axis

```
from gstools import vario_estimate_structured
print('Estimating structured variograms')
gamma_x = vario_estimate_structured(herten_trans_log, direction='x')[:21]
gamma_y = vario_estimate_structured(herten_trans_log, direction='y')[:21]
```

With these two estimated variograms, we can start fitting *Exponential* covariance models

```
fit_model_x = Exponential(dim=2)
fit_model_x.fit_variogram(x_s, gamma_x, nugget=False)
fit_model_y = Exponential(dim=2)
fit_model_y.fit_variogram(y_s, gamma_y, nugget=False)
```

Now, the isotropic variogram and the two variograms in x- and y-direction can be plotted together with their respective models, which will be plotted with dashed lines.

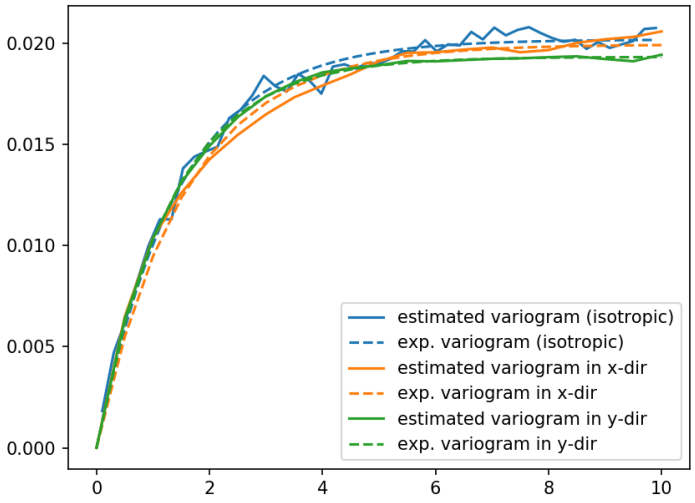
```
line, = pt.plot(bin_center, gamma, label='estimated variogram (isotropic)')
pt.plot(bin_center, fit_model.variogram(bin_center), color=line.get_color(),
        linestyle='--', label='exp. variogram (isotropic)')

line, = pt.plot(x_s, gamma_x, label='estimated variogram in x-dir')
pt.plot(x_s, fit_model_x.variogram(x_s), color=line.get_color(),
        linestyle='--', label='exp. variogram in x-dir')

line, = pt.plot(y_s, gamma_y, label='estimated variogram in y-dir')
pt.plot(y_s, fit_model_y.variogram(y_s),
    color=line.get_color(), linestyle='--', label='exp. variogram in y-dir')

pt.legend()
pt.show()
```

Giving



The plot might be a bit cluttered, but at least it is pretty obvious that the Herten aquifer has no apparent anisotropies in its spatial structure.

Creating a Spatial Random Field from the Herten Parameters

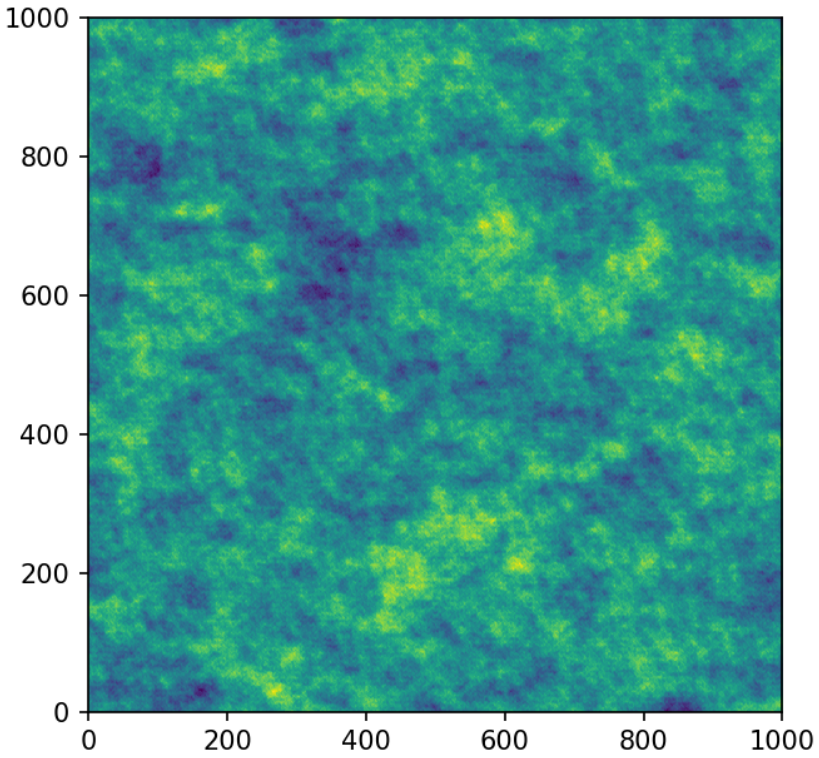
With all the hard work done, it's straight forward now, to generate new *Herten realisations*

```
from gstools import SRF

srf = SRF(fit_model, seed=19770928)
new_herten = srf((x_s, y_s), mesh_type='structured')

pt.imshow(new_herten.T, origin='lower')
pt.show()
```

Yielding



That's pretty neat! Executing the code given on this site, will result in a lower resolution of the field, because we overwrote *x_s* and *y_s* for the directional variogram estimation. In the example script, this is not the case and you will get a high resolution field.

And Now for Some Cleanup

In case you want all the downloaded data and scripts to be deleted, use following commands

```
from shutil import rmtree
os.remove('data.zip')
os.remove('scripts.zip')
rmtree('Herten-analog')
rmtree('tools')
```

And in case you want to play around a little bit more with the data, you can comment out the function calls `download_herten()` and `download_scripts()`, after they where called at least once and also comment out the cleanup. This way, the data will not be downloaded with every script execution.

2.4 Tutorial 4: Random Vector Field Generation

In 1970, Kraichnan was the first to suggest a randomization method. For studying the diffusion of single particles in a random incompressible velocity field, he came up with a randomization method which includes a projector which ensures the incompressibility of the vector field.

Theoretical Background

Without loss of generality we assume that the mean velocity $\bar{U}$ is oriented towards the direction of the first basis vector $\mathbf{e}_1$. Our goal is now to generate random fluctuations with a given covariance model around this mean velocity. And at the same time, making sure that the velocity field remains incompressible or in other words, ensure $\nabla \cdot \mathbf{U} = 0$. This can be done by using the randomization method we already know, but adding a projector to every mode being summed:

$$\mathbf{U}(\mathbf{x}) = \bar{U}\mathbf{e}_1 - \sqrt{\frac{\sigma^2}{N}} \sum_{i=1}^{N} \mathbf{p}(\mathbf{k}_i) \left[Z_{1,i} \cos\left(\langle \mathbf{k}_i, \mathbf{x} \rangle\right) + \sin\left(\langle \mathbf{k}_i, \mathbf{x} \rangle\right) \right]$$

with the projector

$$\mathbf{p}(\mathbf{k}_i) = \mathbf{e}_1 - \frac{\mathbf{k}_i k_1}{k^2} \, .$$

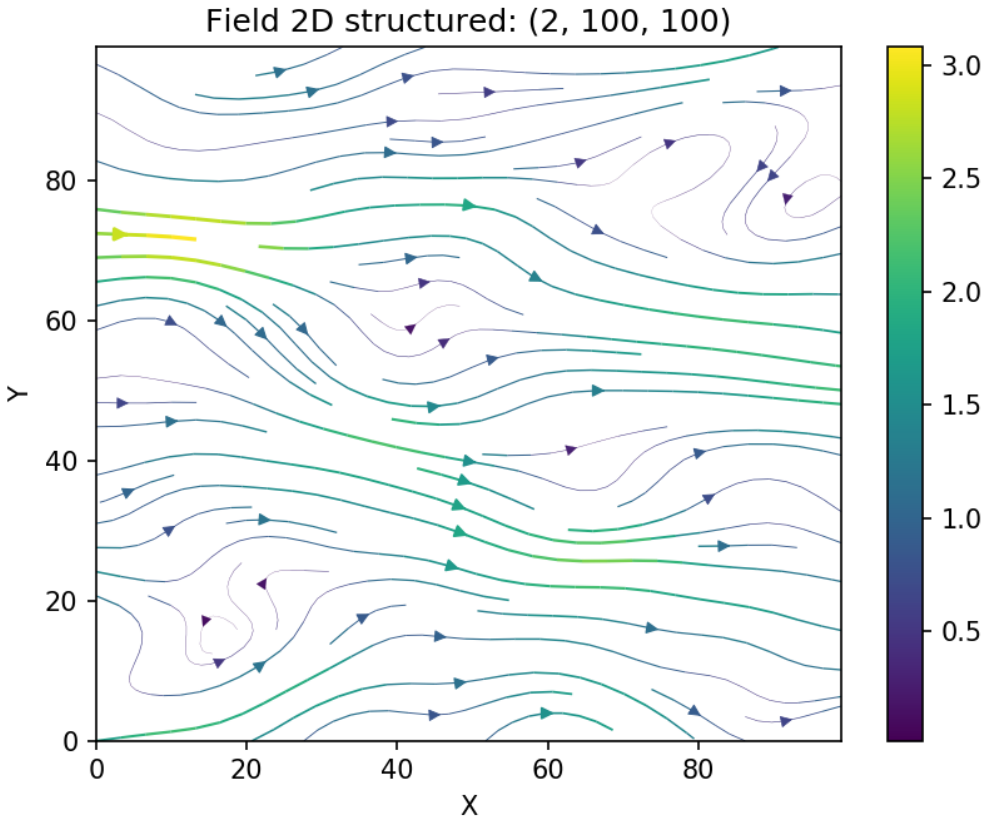
By calculating $\nabla \cdot \mathbf{U} = 0$, it can be verified, that the resulting field is indeed incompressible.

Generating a Random Vector Field

As a first example we are going to generate a vector field with a Gaussian covariance model on a structured grid:

```
import numpy as np
import matplotlib.pyplot as plt
from gstools import SRF, Gaussian
x = np.arange(100)
y = np.arange(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model, generator='VectorField')
srf((x, y), mesh_type='structured', seed=19841203)
srf.plot()
```

And we get a beautiful streamflow plot:

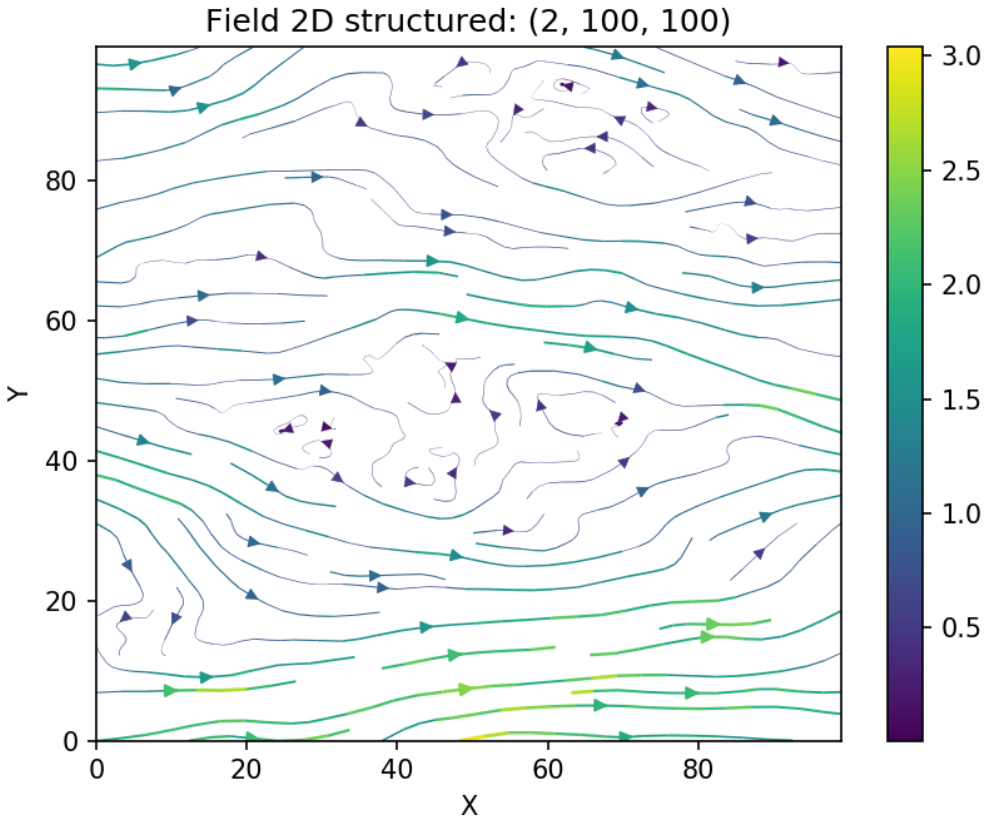


Let us have a look at the influence of the covariance model. Choosing the exponential model and keeping all other parameters the same

```
from gstools import Exponential

model2 = Exponential(dim=2, var=1, len_scale=10)
srf.model = model2
srf((x, y), mesh_type='structured', seed=19841203)
srf.plot()
```

we get following result



and we see, that the wiggles are much "rougher" than the smooth Gaussian ones.

Applications

One great advantage of the Kraichnan method is, that after some initializations, one can compute the velocity field at arbitrary points, online, with hardly any overhead. This means, that for a Lagrangian transport simulation for example, the velocity can be evaluated at each particle position very efficiently and without any interpolation. These field interpolations are a common problem for Lagrangian methods.

2.5 Tutorial 5: Kriging

The subpackage *gstools.krige* provides routines for Gaussian process regression, also known as kriging. Kriging is a method of data interpolation based on predefined covariance models.

We provide two kinds of kriging routines:

- Simple: The data is interpolated with a given mean value for the kriging field.
- Ordinary: The mean of the resulting field is unkown and estimated during interpolation.

Theoretical Background

The aim of kriging is to derive the value of a field at some point x_0, when there are fixed observed values $z(x_1) \dots z(x_n)$ at given points x_i.

The resluting value z_0 at x_0 is calculated as a weighted mean:

$$z_0 = \sum_{i=1}^{n} w_i \cdot z_i$$

The weights $W = (w_1, \dots, w_n)$ depent on the given covariance model and the location of the target point.

The different kriging approaches provide different ways of calculating W.

Implementation

The routines for kriging are almost identical to the routines for spatial random fields. First you define a covariance model, as described in *the SRF tutorial*, then you initialize the kriging class with this model:

```
from gstools import Gaussian, krige
# condtions
cond_pos = ...
cond_val = ...
model = Gaussian(dim=1, var=0.5, len_scale=2)
krig = krige.Simple(model, mean=1, cond_pos=cond_pos, cond_val=cond_val)
```

The resulting field instance `krig` has the same methods as the *SRF* class. You can call it to evaluate the kriged field at different points, you can plot the latest field or you can export the field and so on. Have a look at the documentation of *Simple* and *Ordinary*.

Simple Kriging

Simple kriging assumes a known mean of the data. For simplicity we assume a mean of 0, which can be achieved by subtracting the mean from the observed values and subsequently adding it to the resulting data.

The resulting equation system for W is given by:

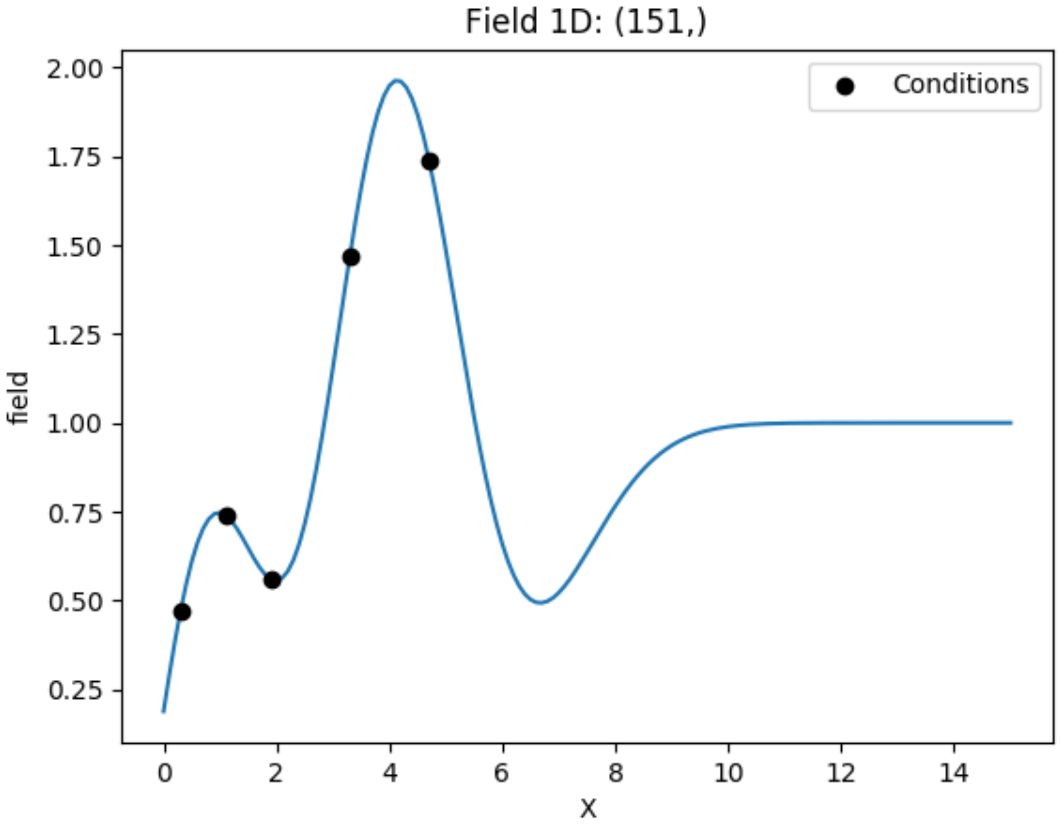
$$W = \begin{pmatrix} c(x_1, x_1) & \cdots & c(x_1, x_n) \\ \vdots & \ddots & \vdots \\ c(x_n, x_1) & \cdots & c(x_n, x_n) \end{pmatrix}^{-1} \begin{pmatrix} c(x_1, x_0) \\ \vdots \\ c(x_n, x_0) \end{pmatrix}$$

Thereby $c(x_i, x_j)$ is the covariance of the given observations.

Example

Here we use simple kriging in 1D (for plotting reasons) with 5 given observations/conditions. The mean of the field has to be given beforehand.

```
import numpy as np
from gstools import Gaussian, krige
# condtions
cond_pos = [0.3, 1.9, 1.1, 3.3, 4.7]
cond_val = [0.47, 0.56, 0.74, 1.47, 1.74]
# resulting grid
gridx = np.linspace(0.0, 15.0, 151)
# spatial random field class
model = Gaussian(dim=1, var=0.5, len_scale=2)
krig = krige.Simple(model, mean=1, cond_pos=cond_pos, cond_val=cond_val)
krig(gridx)
ax = krig.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
ax.legend()
```



Ordinary Kriging

Ordinary kriging will estimate an appropriate mean of the field, based on the given observations/conditions and the covariance model used.

The resulting system of equations for W is given by:

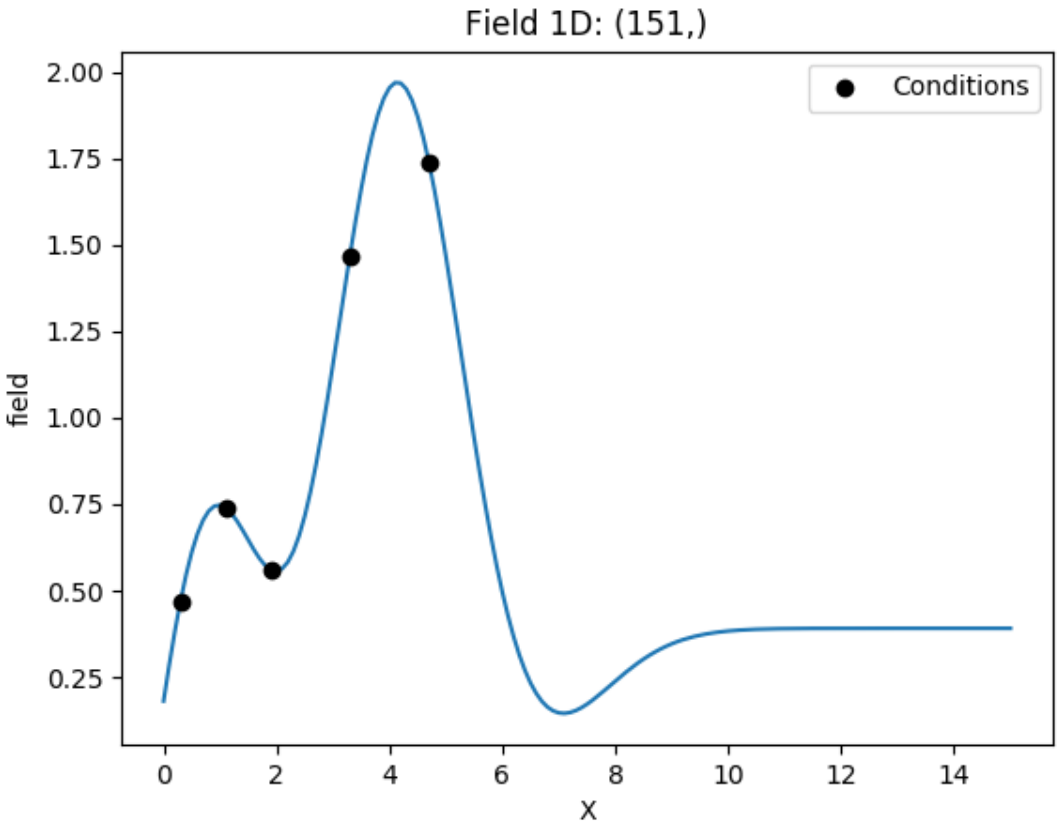
$$\begin{pmatrix} W \\ \mu \end{pmatrix} = \begin{pmatrix} \gamma(x_1,x_1) & \cdots & \gamma(x_1,x_n) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \gamma(x_n,x_1) & \cdots & \gamma(x_n,x_n) & 1 \\ 1 & \cdots & 1 & 0 \end{pmatrix}^{-1} \begin{pmatrix} \gamma(x_1,x_0) \\ \vdots \\ \gamma(x_n,x_0) \\ 1 \end{pmatrix}$$

Thereby $\gamma(x_i,x_j)$ is the semi-variogram of the given observations and μ is a Lagrange multiplier to minimize the kriging error and estimate the mean.

Example

Here we use ordinary kriging in 1D (for plotting reasons) with 5 given observations/conditions. The estimated mean can be accessed by `krig.mean`.

```
import numpy as np
from gstools import Gaussian, krige
# condtions
cond_pos = [0.3, 1.9, 1.1, 3.3, 4.7]
cond_val = [0.47, 0.56, 0.74, 1.47, 1.74]
# resulting grid
gridx = np.linspace(0.0, 15.0, 151)
# spatial random field class
model = Gaussian(dim=1, var=0.5, len_scale=2)
krig = krige.Ordinary(model, cond_pos=cond_pos, cond_val=cond_val)
krig(gridx)
ax = krig.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
ax.legend()
```



Interface to PyKrige

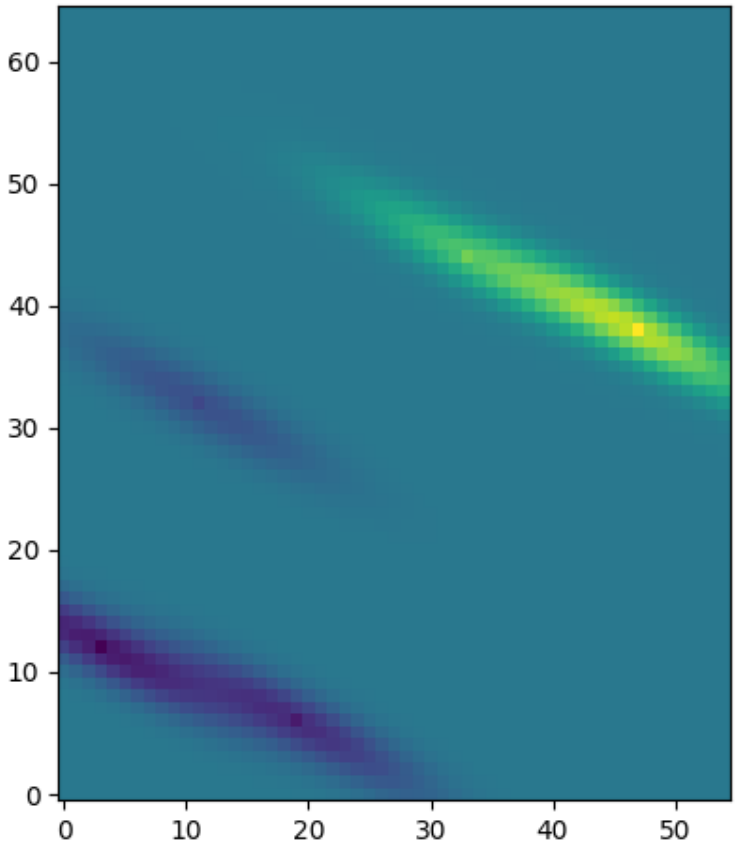
To use fancier methods like universal kriging, we provide an interface to PyKrige.

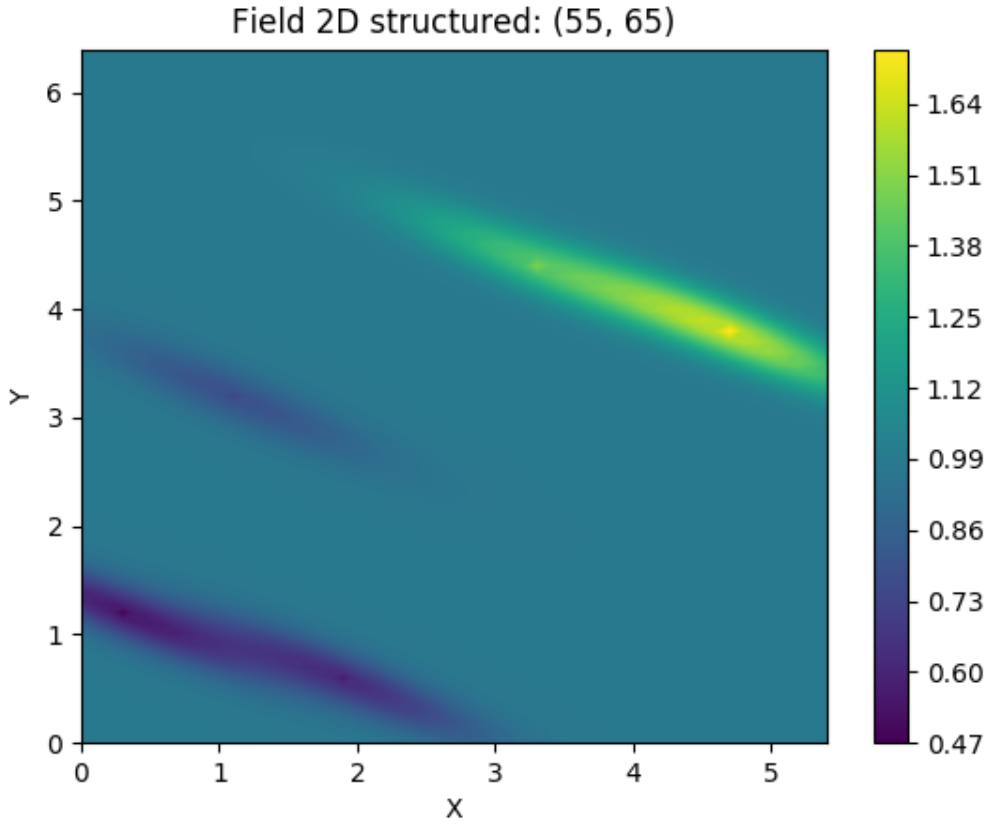
You can pass a GSTools Covariance Model to the PyKrige routines as `variogram_model`.

To demonstrate the general workflow, we compare the ordinary kriging of PyKrige with GSTools in 2D:

```
import numpy as np
from gstools import Gaussian, krige
from pykrige.ok import OrdinaryKriging
from matplotlib import pyplot as plt

# conditioning data
data = np.array([[0.3, 1.2, 0.47],
                 [1.9, 0.6, 0.56],
                 [1.1, 3.2, 0.74],
                 [3.3, 4.4, 1.47],
                 [4.7, 3.8, 1.74]])
# grid definition for output field
gridx = np.arange(0.0, 5.5, 0.1)
gridy = np.arange(0.0, 6.5, 0.1)
# a GSTools based covariance model
cov_model = Gaussian(dim=2, len_scale=1, anis=.2, angles=-.5, var=.5, nugget=.1)
# ordinary kriging with pykrige
OK1 = OrdinaryKriging(data[:, 0], data[:, 1], data[:, 2], cov_model)
z1, ss1 = OK1.execute('grid', gridx, gridy)
plt.imshow(z1, origin="lower")
plt.show()
# ordinary kriging with gstools for comparison
OK2 = krige.Ordinary(cov_model, [data[:, 0], data[:, 1]], data[:, 2])
OK2.structured([gridx, gridy])
OK2.plot()
```


60
50
40
30
20
10
0
0
10
20
30
40
50


Field 2D structured: (55, 65)
Y
X
0
1
2
3
4
5
6
1.64
1.51
1.38
1.25
1.12
0.99
0.86
0.73
0.60
0.47

2.6 Tutorial 6: Conditioned Fields

Kriged fields tend to approach the field mean outside the area of observations. To generate random fields, that coincide with given observations, but are still random according to a given covariance model away from the observations proximity, we provide the generation of conditioned random fields.

Theoretical Background

The idea behind conditioned random fields builds up on kriging. First we generate a field with a kriging method, then we generate a random field, and finally we generate another kriged field to eliminate the error between the random field and the kriged field of the given observations.

To do so, you can choose between ordinary and simple kriging. In case of ordinary kriging, the mean of the SRF will be overwritten by the estimated mean.

The setup of the spatial random field is the same as described in *the SRF tutorial*. You just need to add the conditions as described in *the kriging tutorial*:

```
srf.set_condition(cond_pos, cond_val, "simple")
```

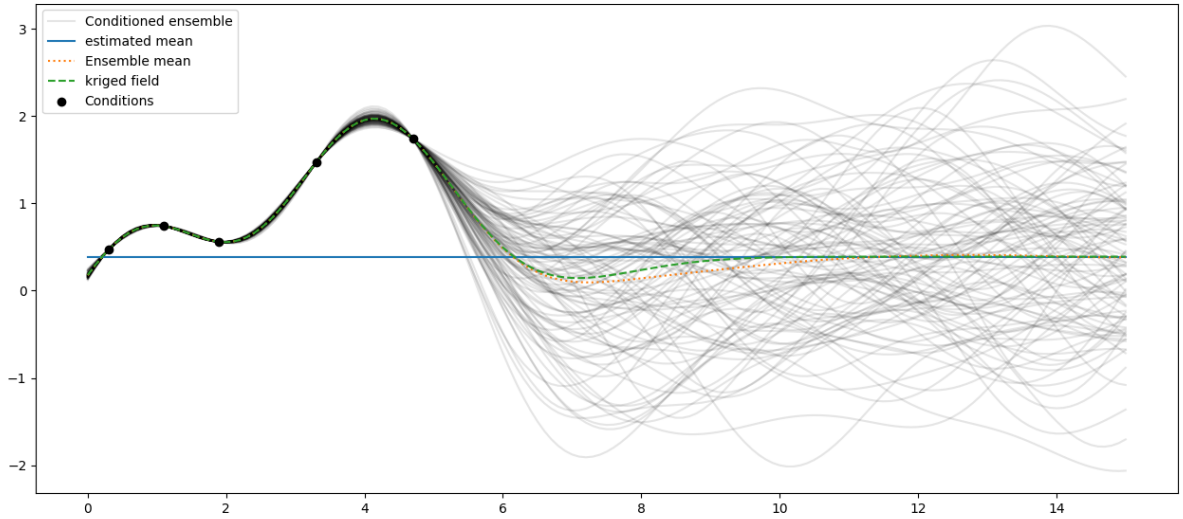
or:

```
srf.set_condition(cond_pos, cond_val, "ordinary")
```

Example: Conditioning with Ordinary Kriging

Here we use ordinary kriging in 1D (for plotting reasons) with 5 given observations/conditions, to generate an ensemble of conditioned random fields. The estimated mean can be accessed by `srf.mean`.

```
import numpy as np
from gstools import Gaussian, SRF
import matplotlib.pyplot as plt
# conditions
cond_pos = [0.3, 1.9, 1.1, 3.3, 4.7]
cond_val = [0.47, 0.56, 0.74, 1.47, 1.74]
gridx = np.linspace(0.0, 15.0, 151)
# spatial random field class
model = Gaussian(dim=1, var=0.5, len_scale=2)
srf = SRF(model)
srf.set_condition(cond_pos, cond_val, "ordinary")
fields = []
for i in range(100):
    if i % 10 == 0: print(i)
    fields.append(srf(gridx, seed=i))
    label = "Conditioned ensemble" if i == 0 else None
    plt.plot(gridx, fields[i], color="k", alpha=0.1, label=label)
plt.plot(gridx, np.full_like(gridx, srf.mean), label="estimated mean")
plt.plot(gridx, np.mean(fields, axis=0), linestyle=':', label="Ensemble mean")
plt.plot(gridx, srf.krige_field, linestyle='dashed', label="kriged field")
plt.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
plt.legend()
plt.show()
```



As you can see, the kriging field coincides with the ensemble mean of the conditioned random fields and the estimated mean is the mean of the far-field.

2.7 Tutorial 7: Field transformations

The generated fields of gstools are ordinary Gaussian random fields. In application there are several transformations to describe real world problems in an appropriate manner.

GStools provides a submodule *gstools.transform* with a range of common transformations:

binary(fld[, divide, upper, lower])	Binary transformation.
boxcox(fld[, lmbda, shift])	Box-Cox transformation.
zinnharvey(fld[, conn])	Zinn and Harvey transformation to connect low or high values.
normal_force_moments(fld)	Force moments of a normal distributed field.
normal_to_lognormal(fld)	Transform normal distribution to log-normal distribution.
normal_to_uniform(fld)	Transform normal distribution to uniform distribution on [0, 1].
normal_to_arcsin(fld[, a, b])	Transform normal distribution to the bimodal arcsin distribution.
normal_to_uquad(fld[, a, b])	Transform normal distribution to U-quadratic distribution.

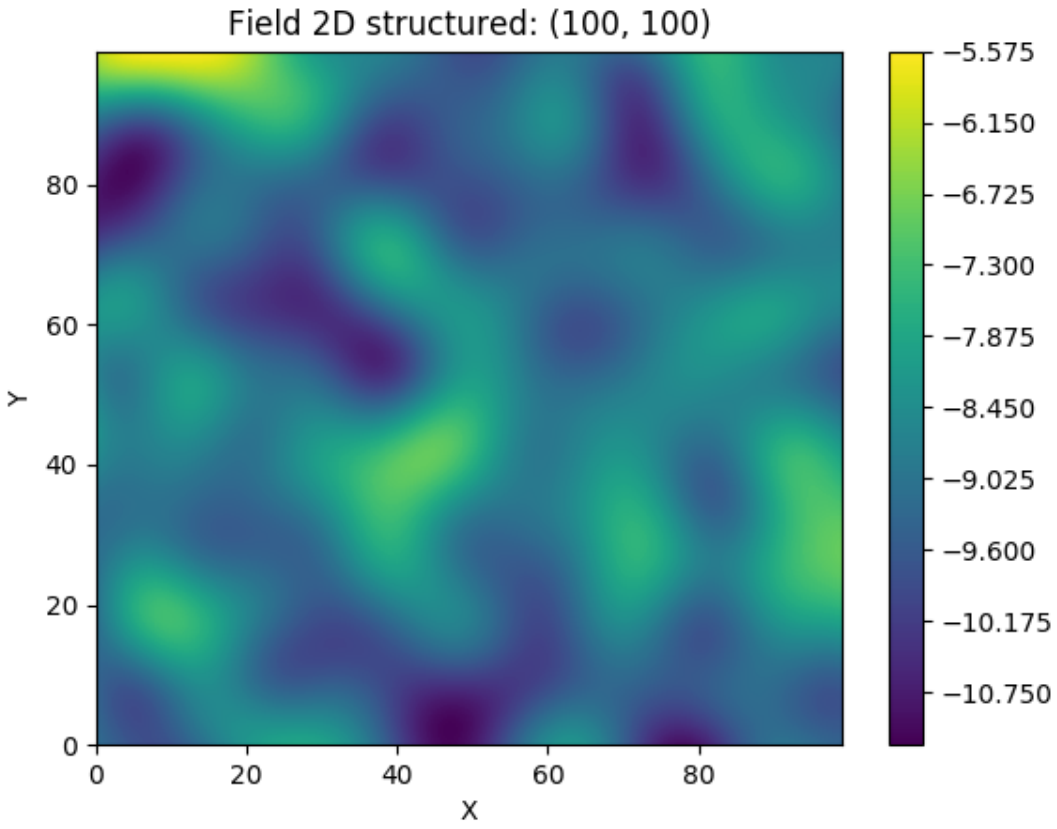
Implementation

All the transformations take a field class, that holds a generated field, as input and will manipulate this field inplace.

Simply import the transform submodule and apply a transformation to the srf class:

```
from gstools import transform as tf
...
tf.normal_to_lognormal(srf)
```

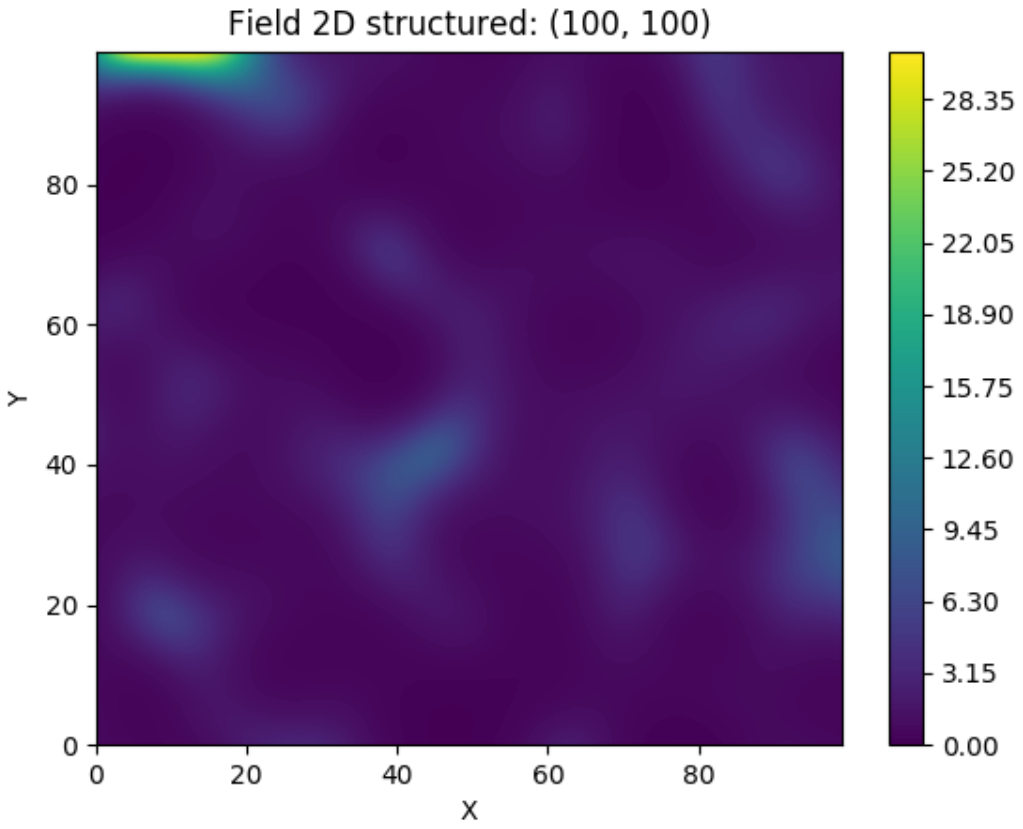
In the following we will start from a simple random field following a Gaussian covariance:



1. Example: log-normal fields

Here we transform a field to a log-normal distribution:

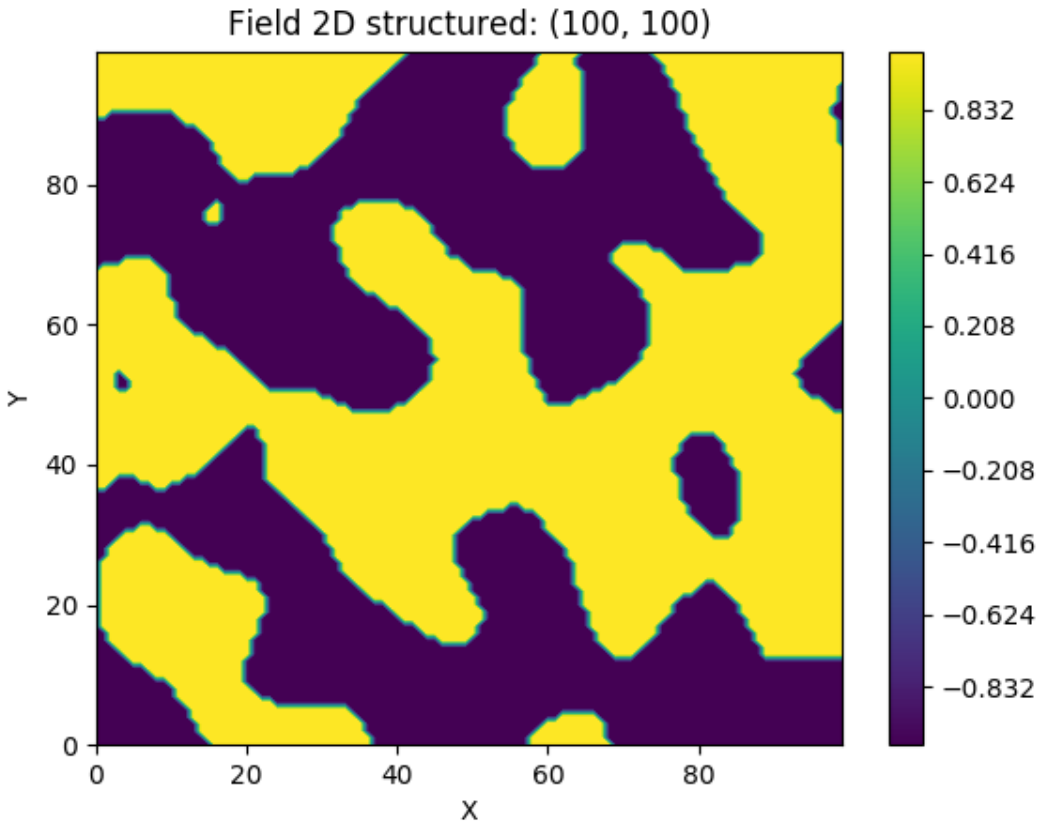
```
from gstools import SRF, Gaussian
from gstools import transform as tf
# structured field with a size of 100x100 and a grid-size of 1x1
x = y = range(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model, seed=20170519)
srf.structured([x, y])
tf.normal_to_lognormal(srf)
srf.plot()
```



2. Example: binary fields

Here we transform a field to a binary field with only two values. The dividing value is the mean by default and the upper and lower values are derived to preserve the variance.

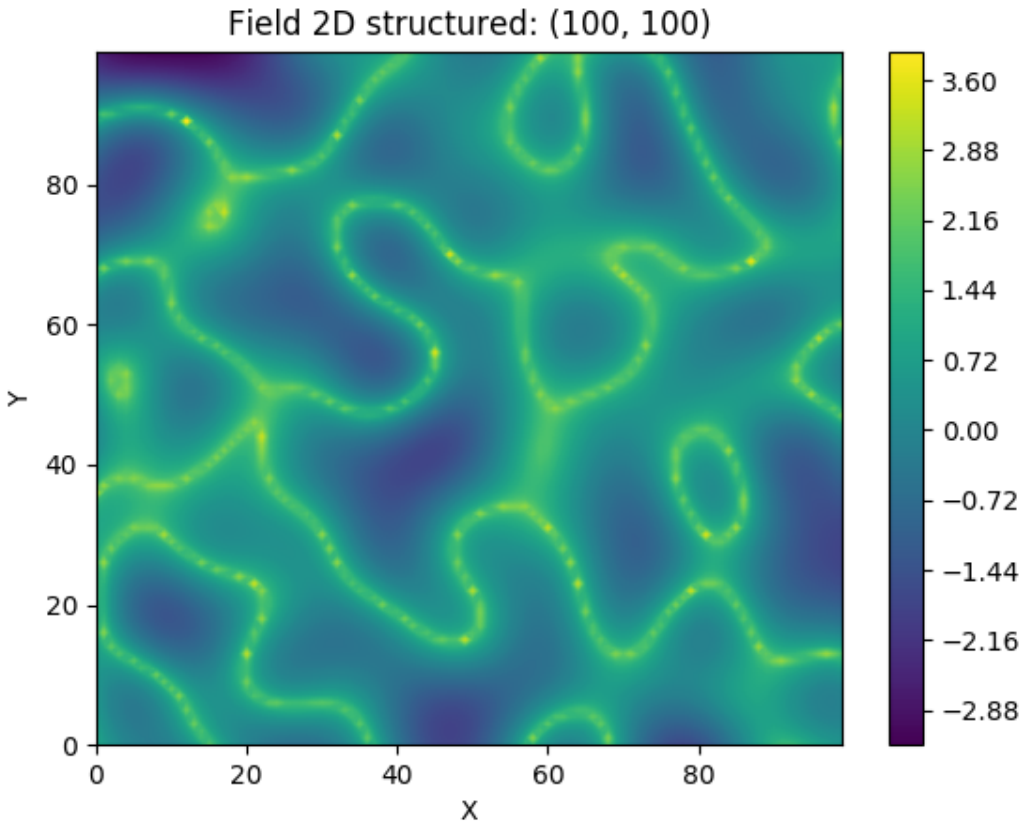
```
from gstools import SRF, Gaussian
from gstools import transform as tf
# structured field with a size of 100x100 and a grid-size of 1x1
x = y = range(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model, seed=20170519)
srf.structured([x, y])
tf.binary(srf)
srf.plot()
```



3. Example: Zinn & Harvey transformation

Here, we transform a field with the so called "Zinn & Harvey" transformation presented in Zinn & Harvey (2003). With this transformation, one could overcome the restriction that in ordinary Gaussian random fields the mean values are the ones being the most connected.

```
from gstools import SRF, Gaussian
from gstools import transform as tf
# structured field with a size of 100x100 and a grid-size of 1x1
x = y = range(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model, seed=20170519)
srf.structured([x, y])
tf.zinnharvey(srf, conn="high")
srf.plot()
```



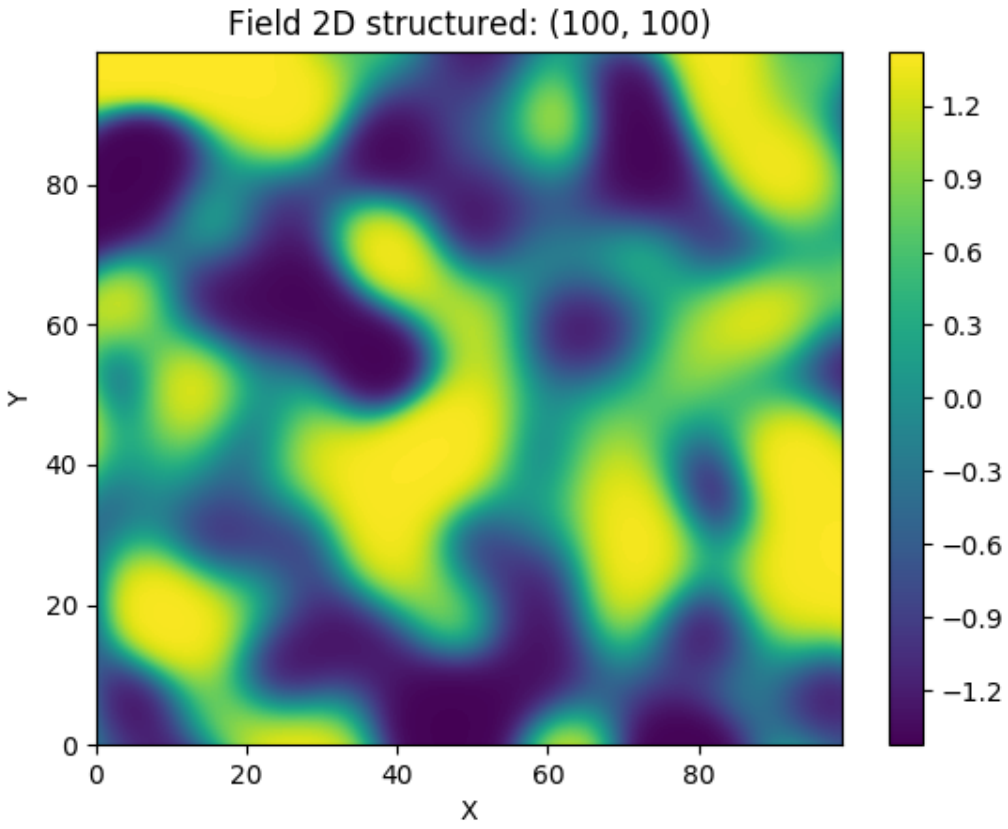
4. Example: bimodal fields

We provide two transformations to obtain bimodal distributions:

- arcsin.
- uquad.

Both transformations will preserve the mean and variance of the given field by default.

```
from gstools import SRF, Gaussian
from gstools import transform as tf
# structured field with a size of 100x100 and a grid-size of 1x1
x = y = range(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model, seed=20170519)
field = srf.structured([x, y])
tf.normal_to_arcsin(srf)
srf.plot()
```

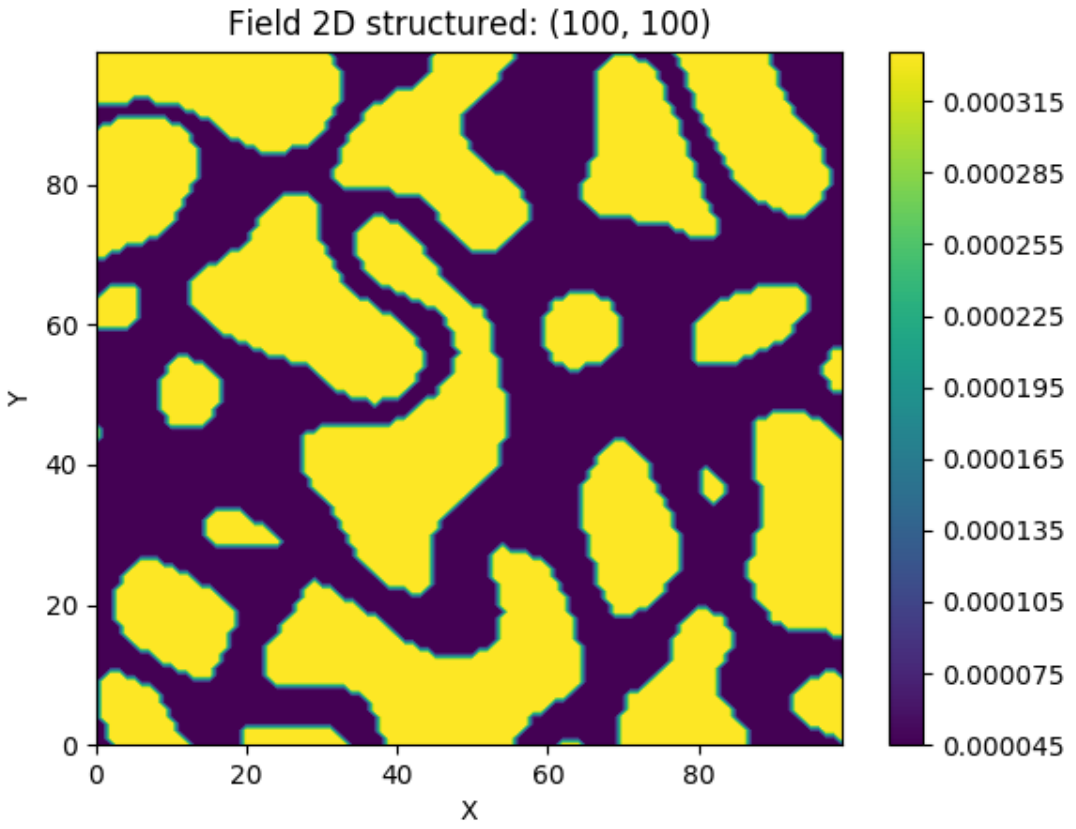


5. Example: Combinations

You can combine different transformations simply by successively applying them.

Here, we first force the single field realization to hold the given moments, namely mean and variance. Then we apply the Zinn & Harvey transformation to connect the low values. Afterwards the field is transformed to a binary field and last but not least, we transform it to log-values.

```
from gstools import SRF, Gaussian
from gstools import transform as tf
# structured field with a size of 100x100 and a grid-size of 1x1
x = y = range(100)
model = Gaussian(dim=2, var=1, len_scale=10)
srf = SRF(model, mean=-9, seed=20170519)
srf.structured([x, y])
tf.normal_force_moments(srf)
tf.zinnharvey(srf, conn="low")
tf.binary(srf)
tf.normal_to_lognormal(srf)
srf.plot()
```



The resulting field could be interpreted as a transmissivity field, where the values of low permeability are the ones being the most connected and only two kinds of soil exist.

CHAPTER 3

GSTOOLS API

3.1 Purpose

GeoStatTools is a library providing geostatistical tools for random field generation, conditioned field generation, kriging and variogram estimation based on a list of provided or even user-defined covariance models.

The following functionalities are directly provided on module-level.

3.2 Subpackages

covmodel	GStools subpackage providing a set of handy covariance models.
field	GStools subpackage providing tools for spatial random fields.
variogram	GStools subpackage providing tools for estimating and fitting variograms.
krige	GStools subpackage providing kriging.
random	GStools subpackage for random number generation.
tools	GStools subpackage providing miscellaneous tools.
transform	GStools subpackage providing transformations.

3.3 Classes

Spatial Random Field

Class for random field generation

SRF(model[, mean, upscaling, generator])	A class to generate spatial random fields (SRF).

Covariance Base-Class

Class to construct user defined covariance models

CovModel([dim, var, len_scale, nugget, ...])	Base class for the GSTools covariance models.

Covariance Models

Standard Covariance Models

Gaussian([dim, var, len_scale, nugget, ...])	The Gaussian covariance model.
Exponential([dim, var, len_scale, nugget, ...])	The Exponential covariance model.
Matern([dim, var, len_scale, nugget, anis, ...])	The Matérn covariance model.
Rational([dim, var, len_scale, nugget, ...])	The rational quadratic covariance model.
Stable([dim, var, len_scale, nugget, anis, ...])	The stable covariance model.
Linear([dim, var, len_scale, nugget, anis, ...])	The bounded linear covariance model.
Circular([dim, var, len_scale, nugget, ...])	The circular covariance model.
Spherical([dim, var, len_scale, nugget, ...])	The Spherical covariance model.
Intersection([dim, var, len_scale, nugget, ...])	The Intersection covariance model.

Truncated Power Law Covariance Models

TPLGaussian([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Gaussian modes.
TPLExponential([dim, var, len_scale, ...])	Truncated-Power-Law with Exponential modes.
TPLStable([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Stable modes.

3.4 Functions

VTK-Export

Routines to export fields to the vtk format

to_vtk	
vtk_export(filename, pos, fields[, mesh_type])	Export a field to vtk.
to_vtk_structured	
vtk_export_structured(filename, pos, fields)	Export a field to vtk structured rectilinear grid file.
to_vtk_unstructured	
vtk_export_unstructured(filename, pos, fields)	Export a field to vtk unstructured grid file.

variogram estimation

Estimate the variogram of a given field

vario_estimate_structured(field[, direction])	Estimates the variogram on a regular grid.
vario_estimate_unstructured(pos, field, ...)	Estimates the variogram on a unstructured grid.

3.5 gstools.covmodel

GStools subpackage providing a set of handy covariance models.

Subpackages

base	GStools subpackage providing the base class for covariance models.
models	GStools subpackage providing different covariance models.
tpl_models	GStools subpackage providing truncated power law covariance models.
plot	GStools subpackage providing plotting routines for the covariance models.

Covariance Base-Class

Class to construct user defined covariance models

CovModel([dim, var, len_scale, nugget, ...])	Base class for the GSTools covariance models.

Covariance Models

Standard Covariance Models

Gaussian([dim, var, len_scale, nugget, ...])	The Gaussian covariance model.
Exponential([dim, var, len_scale, nugget, ...])	The Exponential covariance model.
Matern([dim, var, len_scale, nugget, anis, ...])	The Matérn covariance model.
Rational([dim, var, len_scale, nugget, ...])	The rational quadratic covariance model.
Stable([dim, var, len_scale, nugget, anis, ...])	The stable covariance model.
Linear([dim, var, len_scale, nugget, anis, ...])	The bounded linear covariance model.
Circular([dim, var, len_scale, nugget, ...])	The circular covariance model.
Spherical([dim, var, len_scale, nugget, ...])	The Spherical covariance model.
Intersection([dim, var, len_scale, nugget, ...])	The Intersection covariance model.

Truncated Power Law Covariance Models

TPLGaussian([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Gaussian modes.
TPLExponential([dim, var, len_scale, ...])	Truncated-Power-Law with Exponential modes.
TPLStable([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Stable modes.

gstools.covmodel.base

GStools subpackage providing the base class for covariance models.

The following classes are provided

CovModel([dim, var, len_scale, nugget, ...])	Base class for the GSTools covariance models.

class gstools.covmodel.base.**CovModel**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: `object`

Base class for the GSTools covariance models.

Parameters

- **dim** (`int`, optional) – dimension of the model. Default: `3`
- **var** (`float`, optional) – variance of the model (the nugget is not included in "this" variance) Default: `1.0`
- **len_scale** (`float` or `list`, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: `1.0`
- **nugget** (`float`, optional) – nugget of the model. Default: `0.0`
- **anis** (`float` or `list`, optional) – anisotropy ratios in the transversal directions [y, z]. Default: `1.0`
- **angles** (`float` or `list`, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: `0.0`
- **integral_scale** (`float` or `list` or `None`, optional) – If given, `len_scale` will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: `None`
- **var_raw** (`float` or `None`, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, `var` will be ignored. (This is just for models that override *CovModel.var_factor*) Default: `None`
- **hankel_kw** (`dict` or `None`, optional) – Modify the init-arguments of `hankel.SymmetricFourierTransform` used for the spectrum calculation. Use with caution (Better: Don't!). `None` is equivalent to `{"a": -1, "b": 1, "N": 1000, "h": 0.001}`. Default: `None`

Examples

```
>>> from gstools import CovModel
>>> import numpy as np
>>> class Gau(CovModel):
...     def cor(self, h):
...         return np.exp(-h**2)
...
>>> model = Gau()
>>> model.spectrum(2)
0.00825830126008459
```

Attributes

angles `numpy.ndarray`: Rotation angles (in rad) of the model.

anis `numpy.ndarray`: The anisotropy factors of the model.

arg `list` of `str`: Names of all arguments.

arg_bounds `dict`: Bounds for all parameters.

dim `int`: The dimension of the model.

dist_func `tuple` of `callable`: pdf, cdf and ppf.

do_rotation `bool`: State if a rotation is performed.

hankel_kw `dict`: `hankel.SymmetricFourierTransform` kwargs.

has_cdf `bool`: State if a cdf is defined by the user.

has_ppf `bool`: State if a ppf is defined by the user.

integral_scale `float`: The main integral scale of the model.

integral_scale_vec `numpy.ndarray`: The integral scales in each direction.

len_scale `float`: The main length scale of the model.

len_scale_bounds `list`: Bounds for the lenght scale.

len_scale_vec `numpy.ndarray`: The length scales in each direction.

name `str`: The name of the CovModel class.

nugget `float`: The nugget of the model.

nugget_bounds `list`: Bounds for the nugget.

opt_arg `list` of `str`: Names of the optional arguments.

opt_arg_bounds `dict`: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill `float`: The sill of the variogram.

var `float`: The variance of the model.

var_bounds `list`: Bounds for the variance.

var_raw `float`: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.

Continued on next page

Table 13 – continued from previous page

cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Provide default optional arguments by the user.
default_opt_arg_bounds()	Provide default boundaries for optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.

calc_integral_scale()
Calculate the integral scale of the isotrope model.

check_arg_bounds()
Check arguments to be within the given bounds.

check_opt_arg()
Run checks for the optional arguments.

This is in addition to the bound-checks

Notes

- You can use this to raise a ValueError/warning
- Any return value will be ignored
- This method will only be run once, when the class is initialized

cor_spatial(*pos*)
Spatial correlation respecting anisotropy and rotation.

cov_nugget(*r*)
Covariance of the model respecting the nugget at r=0.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

cov_spatial(*pos*)
Spatial covariance respecting anisotropy and rotation.

default_arg_bounds()
Provide default boundaries for arguments.

Given as a dictionary.

default_opt_arg()
Provide default optional arguments by the user.

Should be given as a dictionary.

default_opt_arg_bounds()
Provide default boundaries for optional arguments.

fit_variogram(*x_data*, *y_data*, *maxfev=1000*, ***para_deselect*)
Fiting the isotropic variogram-model to given data.

Parameters

- **x_data** (numpy.ndarray) – The radii of the meassured variogram.
- **y_data** (numpy.ndarray) – The messured variogram
- **maxfev** (*int, optional*) – The maximum number of calls to the function in scipy curvefit. Default: 1000
- ****para_deselect** – You can deselect the parameters to be fitted, by setting them "False" as keywords. By default, all parameters are fitted.

Returns

- **fit_para** (dict) – Dictonary with the fitted parameter values
- **pcov** (numpy.ndarray) – The estimated covariance of *popt* from scipy.optimize.curve_fit

---

Notes

You can set the bounds for each parameter by accessing *CovModel.set_arg_bounds*.

The fitted parameters will be instantly set in the model.

---

fix_dim()
Set a fix dimension for the model.

ln_spectral_rad_pdf(*r*)
Log radial spectral density of the model.

percentile_scale(*per=0.9*)
Calculate the percentile scale of the isotrope model.

This is the distance, where the given percentile of the variance is reached by the variogram

plot(*func='variogram'*, ***kwargs*)
Plot a function of a the CovModel.

Parameters

- **func** (str, optional) – Function to be plotted. Could be one of:
  - "variogram"
  - "covariance"
  - "correlation"
  - "vario_spatial"
  - "cov_spatial"
  - "cor_spatial"
  - "spectrum"
  - "spectral_density"

– "spectral_rad_pdf"

- ****kwargs** – Keyword arguments forwarded to the plotting function *"plot_" + func* in `gstools.covmodel.plot`.

See also:

`gstools.covmodel.plot()`

pykrige_vario(*args=None*, *r=0*)
Isotropic variogram of the model for pykrige.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

set_arg_bounds(***kwargs*)
Set bounds for the parameters of the model.

Parameters ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values as value:

- `[a, b]` or
- `[a, b, <type>]`

<type> is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

spectral_density(*k*)
Spectral density of the covariance model.

This is given by:

$$\tilde{S}(k) = \frac{S(k)}{\sigma^2}$$

Where $S(k)$ is the spectrum of the covariance model.

Parameters **k** (`float`) – Radius of the phase: $k = \|\mathbf{k}\|$

spectral_rad_pdf(*r*)
Radial spectral density of the model.

spectrum(*k*)
Spectrum of the covariance model.

This is given by:

$$S(k) = \left(\frac{1}{2\pi}\right)^n \int C(r) e^{ib\mathbf{k}\cdot\mathbf{r}} d^n\mathbf{r}$$

Internally, this is calculated by the hankel transformation:

$$S(k) = \left(\frac{1}{2\pi}\right)^n \cdot \frac{(2\pi)^{n/2}}{k^{n/2-1}} \int_0^\infty r^{n/2} C(r) J_{n/2-1}(kr) dr$$

Where $C(r)$ is the covariance function of the model.

Parameters **k** (`float`) – Radius of the phase: $k = \|\mathbf{k}\|$

var_factor()
Factor for the variance.

vario_nugget(*r*)
Isotropic variogram of the model respecting the nugget at r=0.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

vario_spatial(*pos*)
Spatial variogram respecting anisotropy and rotation.

angles
Rotation angles (in rad) of the model.

Type numpy.ndarray

anis
The anisotropy factors of the model.

Type numpy.ndarray

arg
Names of all arguments.

Type list of str

arg_bounds
Bounds for all parameters.

Notes

Keys are the opt-arg names and values are lists of 2 or 3 values:

- [a, b] or
- [a, b, <type>]

<type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type dict

dim
The dimension of the model.

Type int

dist_func
pdf, cdf and ppf.

Spectral distribution info from the model.

Type tuple of callable

do_rotation
State if a rotation is performed.

Type bool

hankel_kw
hankel.SymmetricFourierTransform kwargs.

Type dict

has_cdf
State if a cdf is defined by the user.

Type bool

has_ppf
State if a ppf is defined by the user.

Type bool

integral_scale
The main integral scale of the model.

Raises ValueError – If integral scale is not setable.

Type float

integral_scale_vec
The integral scales in each direction.

Notes

This is calculated by:

- integral_scale_vec[0] = integral_scale
- integral_scale_vec[1] = integral_scale*anis[0]
- integral_scale_vec[2] = integral_scale*anis[1]

Type numpy.ndarray

len_scale
The main length scale of the model.

Type float

len_scale_bounds
Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values:

- [a, b] or
- [a, b, <type>]

<type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type list

len_scale_vec
The length scales in each direction.

Notes

This is calculated by:

- len_scale_vec[0] = len_scale
- len_scale_vec[1] = len_scale*anis[0]
- len_scale_vec[2] = len_scale*anis[1]

Type numpy.ndarray

name
The name of the CovModel class.

Type str

nugget
The nugget of the model.

Type float

nugget_bounds
Bounds for the nugget.

Notes

Is a list of 2 or 3 values:

- `[a, b]` or
- `[a, b, <type>]`

<type> is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `list`

opt_arg
Names of the optional arguments.

Type `list` of `str`

opt_arg_bounds
Bounds for the optional arguments.

Notes

Keys are the opt-arg names and values are lists of 2 or 3 values:

- `[a, b]` or
- `[a, b, <type>]`

<type> is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `dict`

pykrige_angle
2D rotation angle for pykrige.

pykrige_angle_x
3D rotation angle around x for pykrige.

pykrige_angle_y
3D rotation angle around y for pykrige.

pykrige_angle_z
3D rotation angle around z for pykrige.

pykrige_anis
2D anisotropy ratio for pykrige.

pykrige_anis_y
3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z
3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs
Keyword arguments for pykrige routines.

sill
The sill of the variogram.

Notes

This is calculated by:

- `sill = variance + nugget`

Type `float`

var
The variance of the model.

Type `float`

var_bounds
Bounds for the variance.

Notes

Is a list of 2 or 3 values:

- `[a, b]` or
- `[a, b, <type>]`

<type> is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `list`

var_raw
The raw variance of the model without factor.

(See. CovModel.var_factor)

Type `float`

gstools.covmodel.models

GStools subpackage providing different covariance models.

The following classes and functions are provided

Gaussian([dim, var, len_scale, nugget, ...])	The Gaussian covariance model.
Exponential([dim, var, len_scale, nugget, ...])	The Exponential covariance model.
Matern([dim, var, len_scale, nugget, anis, ...])	The Matérn covariance model.
Stable([dim, var, len_scale, nugget, anis, ...])	The stable covariance model.
Rational([dim, var, len_scale, nugget, ...])	The rational quadratic covariance model.
Linear([dim, var, len_scale, nugget, anis, ...])	The bounded linear covariance model.
Circular([dim, var, len_scale, nugget, ...])	The circular covariance model.
Spherical([dim, var, len_scale, nugget, ...])	The Spherical covariance model.
Intersection([dim, var, len_scale, nugget, ...])	The Intersection covariance model.

class gstools.covmodel.models.**Gaussian**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

The Gaussian covariance model.

Notes

This model is given by the following correlation function:

$$\mathrm{cor}(r) = \exp\left(-\frac{\pi}{4}\cdot\left(\frac{r}{\ell}\right)^2\right)$$

.

Parameters

- **dim** (int, optional) – dimension of the model. Default: 3
- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: 1.0
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: 1.0
- **nugget** (float, optional) – nugget of the model. Default: 0.0
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var

will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None

- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to `{"a": -1, "b": 1, "N": 1000, "h": 0.001}`.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill float: The sill of the variogram.

var float: The variance of the model.

var_bounds list: Bounds for the variance.

var_raw float: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Gaussian correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Provide default optional arguments by the user.
default_opt_arg_bounds()	Provide default boundaries for optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_cdf(r)	Radial spectral cdf.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectral_rad_ppf(u)	Radial spectral ppf.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

calc_integral_scale()
Calculate the integral scale of the isotrope model.

correlation(*r*)
Gaussian correlation function.

$$\mathrm{cor}(r) = \exp\left(-\frac{\pi}{4} \cdot \left(\frac{r}{\ell}\right)^2\right)$$

covariance(*r*)
Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

spectral_density(*k*)
Spectral density of the covariance model.

This is given by:

$$\tilde{S}(k) = \frac{S(k)}{\sigma^2}$$

Where $S(k)$ is the spectrum of the covariance model.

Parameters **k** (float) – Radius of the phase: $k = \|\mathbf{k}\|$

spectral_rad_cdf(*r*)
Radial spectral cdf.

spectral_rad_ppf(*u*)
Radial spectral ppf.

Notes

Not defined for 3D.

---

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

class gstools.covmodel.models.**Exponential**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

The Exponential covariance model.

Notes

This model is given by the following correlation function:

$$\mathrm{cor}(r) = \exp\left(-\frac{r}{\ell}\right)$$

.

Parameters

- **dim** (int, optional) – dimension of the model. Default: 3
- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: 1.0
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: 1.0
- **nugget** (float, optional) – nugget of the model. Default: 0.0
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None

- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None
- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill float: The sill of the variogram.

var float: The variance of the model.

var_bounds list: Bounds for the variance.

var_raw float: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Exponential correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Provide default optional arguments by the user.
default_opt_arg_bounds()	Provide default boundaries for optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_cdf(r)	Radial spectral cdf.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectral_rad_ppf(u)	Radial spectral ppf.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

calc_integral_scale()
Calculate the integral scale of the isotrope model.

correlation(*r*)
Exponential correlation function.

$$\mathrm{cor}(r) = \exp\left(-\frac{r}{\ell}\right)$$

covariance(*r*)
Covariance of the model.

Given by: $C\left(r\right) = \sigma^2 \cdot \mathrm{cor}\left(r\right)$

Where $\mathrm{cor}(r)$ is the correlation function.

spectral_density(*k*)
Spectral density of the covariance model.

This is given by:

$$\tilde{S}(k) = \frac{S(k)}{\sigma^2}$$

Where $S(k)$ is the spectrum of the covariance model.

Parameters **k** (float) – Radius of the phase: $k = \|\mathbf{k}\|$

spectral_rad_cdf(*r*)
Radial spectral cdf.

spectral_rad_ppf(*u*)
Radial spectral ppf.

Notes

Not defined for 3D.

---

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

class gstools.covmodel.models.**Matern**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

The Matérn covariance model.

Notes

This model is given by the following correlation function:

$$\mathrm{cor}(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \cdot \left(\sqrt{\nu} \cdot \frac{r}{\ell}\right)^{\nu} \cdot \mathrm{K}_{\nu}\left(\sqrt{\nu} \cdot \frac{r}{\ell}\right)$$

Where Γ is the gamma function and K_{ν} is the modified Bessel function of the second kind.

ν is a shape parameter and should be >= 0.2.

If $\nu > 20$, a gaussian model is used, since it is the limit case:

$$\mathrm{cor}(r) = \exp\left(-\frac{1}{4} \cdot \left(\frac{r}{\ell}\right)^2\right)$$

Other Parameters

- ****opt_arg** – The following parameters are covered by these keyword arguments
- **nu** (float, optional) – Shape parameter. Standard range: `[0.2, 30]` Default: `1.0`
- **.**

Parameters

- **dim** (int, optional) – dimension of the model. Default: `3`
- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: `1.0`
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: `1.0`
- **nugget** (float, optional) – nugget of the model. Default: `0.0`
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: `1.0`

- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None
- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill `float`: The sill of the variogram.

var `float`: The variance of the model.

var_bounds `list`: Bounds for the variance.

var_raw `float`: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Matérn correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Defaults for the optional arguments.
default_opt_arg_bounds()	Defaults for boundaries of the optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

calc_integral_scale()
Calculate the integral scale of the isotrope model.

correlation(*r*)
Matérn correlation function.

$$\mathrm{cor}(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \cdot \left(\sqrt{\nu} \cdot \frac{r}{\ell}\right)^{\nu} \cdot \mathrm{K}_{\nu}\left(\sqrt{\nu} \cdot \frac{r}{\ell}\right)$$

covariance(*r*)
Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

default_opt_arg()
Defaults for the optional arguments.

- `{"nu": 1.0}`

Returns Defaults for optional arguments

Return type `dict`

default_opt_arg_bounds()
Defaults for boundaries of the optional arguments.

- `{"nu": [0.5, 30.0, "cc"]}`

Returns Boundaries for optional arguments

Return type `dict`

spectral_density(k)
Spectral density of the covariance model.

This is given by:

$$\tilde{S}(k) = \frac{S(k)}{\sigma^2}$$

Where $S(k)$ is the spectrum of the covariance model.

Parameters **k** (`float`) – Radius of the phase: $k = \|\mathbf{k}\|$

variogram(r)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

class gstools.covmodel.models.**Stable**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)
Bases: *gstools.covmodel.base.CovModel*

The stable covariance model.

Notes

This model is given by the following correlation function:

$$\mathrm{cor}(r) = \exp\left(-\left(\frac{r}{\ell}\right)^{\alpha}\right)$$

α is a shape parameter with $\alpha \in (0, 2]$

Other Parameters

- ****opt_arg** – The following parameters are covered by these keyword arguments
- **alpha** (`float`, optional) – Shape parameter. Standard range: `(0, 2]` Default: `1.5`
- **.**

Parameters

- **dim** (`int`, optional) – dimension of the model. Default: `3`

- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: 1.0
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: 1.0
- **nugget** (float, optional) – nugget of the model. Default: 0.0
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None
- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill float: The sill of the variogram.

var float: The variance of the model.

var_bounds list: Bounds for the variance.

var_raw float: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Check the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Stable correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Defaults for the optional arguments.
default_opt_arg_bounds()	Defaults for boundaries of the optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

check_opt_arg()
Check the optional arguments.

Warns **alpha** – If alpha is < 0.3, the model tends to a nugget model and gets numerically unstable.

correlation(*r*)
Stable correlation function.

$$\mathrm{cor}(r) = \exp\left(-\left(\frac{r}{\ell}\right)^{\alpha}\right)$$

covariance(*r*)
Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

default_opt_arg()
Defaults for the optional arguments.

- `{"alpha": 1.5}`

Returns Defaults for optional arguments

Return type `dict`

default_opt_arg_bounds()
Defaults for boundaries of the optional arguments.

- `{"alpha": [0, 2, "oc"]}`

Returns Boundaries for optional arguments

Return type `dict`

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

class gstools.covmodel.models.**Rational**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)
Bases: *gstools.covmodel.base.CovModel*

The rational quadratic covariance model.

Notes

This model is given by the following correlation function:

$$\mathrm{cor}(r) = \left(1 + \frac{1}{2\alpha} \cdot \left(\frac{r}{\ell}\right)^2\right)^{-\alpha}$$

α is a shape parameter and should be > 0.5.

Other Parameters

- ****opt_arg** – The following parameters are covered by these keyword arguments
- **alpha** (`float`, optional) – Shape parameter. Standard range: `(0, inf)` Default: `1.0`

- .

Parameters

- **dim** (int, optional) – dimension of the model. Default: 3
- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: 1.0
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: 1.0
- **nugget** (float, optional) – nugget of the model. Default: 0.0
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None
- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

name `str`: The name of the CovModel class.

nugget `float`: The nugget of the model.

nugget_bounds `list`: Bounds for the nugget.

opt_arg `list` of `str`: Names of the optional arguments.

opt_arg_bounds `dict`: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill `float`: The sill of the variogram.

var `float`: The variance of the model.

var_bounds `list`: Bounds for the variance.

var_raw `float`: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Rational correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Defaults for the optional arguments.
default_opt_arg_bounds()	Defaults for boundaries of the optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.

Continued on next page

Table 19 – continued from previous page

vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

correlation(*r*)
Rational correlation function.

$$\mathrm{cor}(r) = \left(1 + \frac{1}{2\alpha} \cdot \left(\frac{r}{\ell}\right)^2\right)^{-\alpha}$$

covariance(*r*)
Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

default_opt_arg()
Defaults for the optional arguments.

- `{"alpha": 1.0}`

Returns Defaults for optional arguments

Return type dict

default_opt_arg_bounds()
Defaults for boundaries of the optional arguments.

- `{"alpha": [0.5, inf]}`

Returns Boundaries for optional arguments

Return type dict

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

class gstools.covmodel.models.**Linear**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)
Bases: *gstools.covmodel.base.CovModel*

The bounded linear covariance model.

This model is derived from the relative intersection area of two lines in 1D, where the middle points have a distance of r and the line lengths are ℓ.

Notes

This model is given by the following correlation function:

$$\mathrm{cor}(r) = \begin{cases} 1 - \frac{r}{\ell} & r < \ell \\ 0 & r \geq \ell \end{cases}$$

.

Parameters

- **dim** (int, optional) – dimension of the model. Default: 3
- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: 1.0
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: 1.0
- **nugget** (float, optional) – nugget of the model. Default: 0.0
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None
- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

name str: The name of the CovModel class.

nugget `float`: The nugget of the model.

nugget_bounds `list`: Bounds for the nugget.

opt_arg `list` of `str`: Names of the optional arguments.

opt_arg_bounds `dict`: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill `float`: The sill of the variogram.

var `float`: The variance of the model.

var_bounds `list`: Bounds for the variance.

var_raw `float`: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Linear correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Provide default optional arguments by the user.
default_opt_arg_bounds()	Provide default boundaries for optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.

Continued on next page

Table 20 – continued from previous page

vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

correlation(*r*)
Linear correlation function.

$$\mathrm{cor}(r) = \begin{cases} 1 - \frac{r}{\ell} & r < \ell \\ 0 & r \geq \ell \end{cases}$$

covariance(*r*)
Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

class gstools.covmodel.models.**Circular**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

The circular covariance model.

This model is derived as the relative intersection area of two discs in 2D, where the middle points have a distance of r and the diameters are given by ℓ.

Notes

This model is given by the following correlation function:

$$\mathrm{cor}(r) = \begin{cases} \frac{2}{\pi} \cdot \left(\cos^{-1}\left(\frac{r}{\ell}\right) - \frac{r}{\ell} \cdot \sqrt{1 - \left(\frac{r}{\ell}\right)^2} \right) & r < \ell \\ 0 & r \geq \ell \end{cases}$$

.

Parameters

- **dim** (int, optional) – dimension of the model. Default: 3
- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: 1.0
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: 1.0
- **nugget** (float, optional) – nugget of the model. Default: 0.0
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis

  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: `0.0`

- **integral_scale** (`float` or `list` or `None`, optional) – If given, `len_scale` will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: `None`
- **var_raw** (`float` or `None`, optional) – raw variance of the model which will be multiplied with *`CovModel.var_factor`* to result in the actual variance. If given, `var` will be ignored. (This is just for models that override *`CovModel.var_factor`*) Default: `None`
- **hankel_kw** (`dict` or `None`, optional) – Modify the init-arguments of `hankel.SymmetricFourierTransform` used for the spectrum calculation. Use with caution (Better: Don't!). `None` is equivalent to `{"a": -1, "b": 1, "N": 1000, "h": 0.001}`.

Attributes

angles `numpy.ndarray`: Rotation angles (in rad) of the model.

anis `numpy.ndarray`: The anisotropy factors of the model.

arg `list` of `str`: Names of all arguments.

arg_bounds `dict`: Bounds for all parameters.

dim `int`: The dimension of the model.

dist_func `tuple` of `callable`: pdf, cdf and ppf.

do_rotation `bool`: State if a rotation is performed.

hankel_kw `dict`: `hankel.SymmetricFourierTransform` kwargs.

has_cdf `bool`: State if a cdf is defined by the user.

has_ppf `bool`: State if a ppf is defined by the user.

integral_scale `float`: The main integral scale of the model.

integral_scale_vec `numpy.ndarray`: The integral scales in each direction.

len_scale `float`: The main length scale of the model.

len_scale_bounds `list`: Bounds for the lenght scale.

len_scale_vec `numpy.ndarray`: The length scales in each direction.

name `str`: The name of the CovModel class.

nugget `float`: The nugget of the model.

nugget_bounds `list`: Bounds for the nugget.

opt_arg `list` of `str`: Names of the optional arguments.

opt_arg_bounds `dict`: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill float: The sill of the variogram.

var float: The variance of the model.

var_bounds list: Bounds for the variance.

var_raw float: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Circular correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Provide default optional arguments by the user.
default_opt_arg_bounds()	Provide default boundaries for optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

correlation(*r*)
Circular correlation function.

$$\mathrm{cor}(r) = \begin{cases} \frac{2}{\pi} \cdot \left(\cos^{-1}\left(\frac{r}{\ell}\right) - \frac{r}{\ell} \cdot \sqrt{1 - \left(\frac{r}{\ell}\right)^2} \right) & r < \ell \\ 0 & r \geq \ell \end{cases}$$

covariance(*r*)
Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

class gstools.covmodel.models.**Spherical**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

The Spherical covariance model.

This model is derived from the relative intersection area of two spheres in 3D, where the middle points have a distance of r and the diameters are given by ℓ.

Notes

This model is given by the following correlation function:

$$\mathrm{cor}(r) = \begin{cases} 1 - \frac{3}{2} \cdot \frac{r}{\ell} + \frac{1}{2} \cdot \left(\frac{r}{\ell}\right)^3 & r < \ell \\ 0 & r \geq \ell \end{cases}$$

.

Parameters

- **dim** (int, optional) – dimension of the model. Default: 3
- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: 1.0
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: 1.0
- **nugget** (float, optional) – nugget of the model. Default: 0.0
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None
- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill float: The sill of the variogram.

var float: The variance of the model.

var_bounds list: Bounds for the variance.

var_raw float: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Spherical correlation function.

Continued on next page

Table 22 – continued from previous page

cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Provide default optional arguments by the user.
default_opt_arg_bounds()	Provide default boundaries for optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

correlation(*r*)
Spherical correlation function.

$$\mathrm{cor}(r) = \begin{cases} 1 - \frac{3}{2}\cdot\frac{r}{\ell} + \frac{1}{2}\cdot\left(\frac{r}{\ell}\right)^3 & r < \ell \\ 0 & r \geq \ell \end{cases}$$

covariance(*r*)
Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

class gstools.covmodel.models.**Intersection**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

The Intersection covariance model.

This model is derived from the relative intersection area of two d-dimensional spheres, where the middle points have a distance of r and the diameters are given by ℓ.

In 1D this is the Linear model, in 2D this is the Circular model and in 3D this is the Spherical model.

Notes

This model is given by the following correlation functions.

In 1D:

$$\mathrm{cor}(r) = \begin{cases} 1 - \frac{r}{\ell} & r < \ell \\ 0 & r \geq \ell \end{cases}$$

In 2D:

$$\mathrm{cor}(r) = \begin{cases} \frac{2}{\pi} \cdot \left(\cos^{-1}\left(\frac{r}{\ell}\right) - \frac{r}{\ell} \cdot \sqrt{1 - \left(\frac{r}{\ell}\right)^2} \right) & r < \ell \\ 0 & r \geq \ell \end{cases}$$

In 3D:

$$\mathrm{cor}(r) = \begin{cases} 1 - \frac{3}{2} \cdot \frac{r}{\ell} + \frac{1}{2} \cdot \left(\frac{r}{\ell}\right)^3 & r < \ell \\ 0 & r \geq \ell \end{cases}$$

.

Parameters

- **dim** (int, optional) – dimension of the model. Default: 3
- **var** (float, optional) – variance of the model (the nugget is not included in "this" variance) Default: 1.0
- **len_scale** (float or list, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: 1.0
- **nugget** (float, optional) – nugget of the model. Default: 0.0
- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None
- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill float: The sill of the variogram.

var float: The variance of the model.

var_bounds list: Bounds for the variance.

var_raw float: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.

Continued on next page

Table 23 – continued from previous page

default_opt_arg()	Provide default optional arguments by the user.
default_opt_arg_bounds()	Provide default boundaries for optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for the variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

correlation

correlation(*r*)

covariance(*r*)

Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

spectral_density(*k*)

Spectral density of the covariance model.

This is given by:

$$\tilde{S}(k) = \frac{S(k)}{\sigma^2}$$

Where $S(k)$ is the spectrum of the covariance model.

Parameters **k** (float) – Radius of the phase: $k = \|\mathbf{k}\|$

variogram(*r*)

Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

gstools.covmodel.tpl_models

GStools subpackage providing truncated power law covariance models.

The following classes and functions are provided

TPLGaussian([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Gaussian modes.
TPLExponential([dim, var, len_scale, ...])	Truncated-Power-Law with Exponential modes.
TPLStable([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Stable modes.

class gstools.covmodel.tpl_models.**TPLGaussian**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

Truncated-Power-Law with Gaussian modes.

Notes

The truncated power law is given by a superposition of scale-dependent variograms:

$$\gamma_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}}(r) = \int_{\ell_{\mathrm{low}}}^{\ell_{\mathrm{up}}} \gamma(r,\lambda)\frac{\mathrm{d}\lambda}{\lambda}$$

with *Gaussian* modes on each scale:

$$\gamma(r,\lambda) = \sigma^2(\lambda)\cdot\left(1-\exp\left[-\left(\frac{r}{\lambda}\right)^2\right]\right)$$
$$\sigma^2(\lambda) = C\cdot\lambda^{2H}$$

This results in:

$$\gamma_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}}(r) = \sigma^2_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}}\cdot\left(1-H\cdot\frac{\ell_{\mathrm{up}}^{2H}\cdot E_{1+H}\left[\left(\frac{r}{\ell_{\mathrm{up}}}\right)^2\right]-\ell_{\mathrm{low}}^{2H}\cdot E_{1+H}\left[\left(\frac{r}{\ell_{\mathrm{low}}}\right)^2\right]}{\ell_{\mathrm{up}}^{2H}-\ell_{\mathrm{low}}^{2H}}\right)$$
$$\sigma^2_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}} = \frac{C\cdot\left(\ell_{\mathrm{up}}^{2H}-\ell_{\mathrm{low}}^{2H}\right)}{2H}$$

The "length scale" of this model is equivalent by the integration range:

$$\ell = \ell_{\mathrm{up}} - \ell_{\mathrm{low}}$$

If you want to define an upper scale truncation, you should set `len_low` and `len_scale` accordingly.

The following Parameters occure:

- $C > 0$: The scaling factor from the Power-Law. This parameter will be calculated internally by the given variance. You can access C directly by `model.var_raw`
- $0 < H < 1$: The hurst coefficient (`model.hurst`)
- $\ell_{\mathrm{low}} \geq 0$: The lower length scale truncation of the model (`model.len_low`)
- $\ell_{\mathrm{up}} \geq 0$: The upper length scale truncation of the model (`model.len_up`)

  This will be calculated internally by:

  - `len_up = len_low + len_scale`

That means, that the `len_scale` in this model actually represents the integration range for the truncated power law.

- $E_s(x)$ is the exponential integral.

Other Parameters

- ****opt_arg** – The following parameters are covered by these keyword arguments
- **hurst** (`float`, optional) – Hurst coefficient of the power law. Standard range: `(0, 1)`. Default: `0.5`
- **len_low** (`float`, optional) – The lower length scale truncation of the model. Standard range: `[0, 1000]`. Default: `0.0`
- **.**

Parameters

- **dim** (`int`, optional) – dimension of the model. Default: `3`
- **var** (`float`, optional) – variance of the model (the nugget is not included in "this" variance) Default: `1.0`
- **len_scale** (`float` or `list`, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: `1.0`
- **nugget** (`float`, optional) – nugget of the model. Default: `0.0`
- **anis** (`float` or `list`, optional) – anisotropy ratios in the transversal directions [y, z]. Default: `1.0`
- **angles** (`float` or `list`, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: `0.0`
- **integral_scale** (`float` or `list` or `None`, optional) – If given, `len_scale` will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: `None`
- **var_raw** (`float` or `None`, optional) – raw variance of the model which will be multiplied with *`CovModel.var_factor`* to result in the actual variance. If given, `var` will be ignored. (This is just for models that override *`CovModel.var_factor`*) Default: `None`
- **hankel_kw** (`dict` or `None`, optional) – Modify the init-arguments of `hankel.SymmetricFourierTransform` used for the spectrum calculation. Use with caution (Better: Don't!). `None` is equivalent to `{"a": -1, "b": 1, "N": 1000, "h": 0.001}`.

Attributes

angles `numpy.ndarray`: Rotation angles (in rad) of the model.

anis `numpy.ndarray`: The anisotropy factors of the model.

arg `list` of `str`: Names of all arguments.

arg_bounds `dict`: Bounds for all parameters.

dim `int`: The dimension of the model.

dist_func `tuple` of `callable`: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

len_up float: Upper length scale truncation of the model.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill float: The sill of the variogram.

var float: The variance of the model.

var_bounds list: Bounds for the variance.

var_raw float: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Truncated-Power-Law with Gaussian modes - correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.

Continued on next page

Table 25 – continued from previous page

default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Defaults for the optional arguments.
default_opt_arg_bounds()	Defaults for boundaries of the optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for C (Power-Law factor) to result in variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

correlation(*r*)

Truncated-Power-Law with Gaussian modes - correlation function.

If `len_low=0` we have a simple representation:

$$\mathrm{cor}(r) = H \cdot E_{1+H}\left[\left(\frac{r}{\ell}\right)^2\right]$$

The general case:

$$\mathrm{cor}(r) = H \cdot \frac{\ell_{\mathrm{up}}^{2H} \cdot E_{1+H}\left[\left(\frac{r}{\ell_{\mathrm{up}}}\right)^2\right] - \ell_{\mathrm{low}}^{2H} \cdot E_{1+H}\left[\left(\frac{r}{\ell_{\mathrm{low}}}\right)^2\right]}{\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}}$$

covariance(*r*)

Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

default_arg_bounds()

Provide default boundaries for arguments.

Given as a dictionary.

default_opt_arg()

Defaults for the optional arguments.

- `{"hurst": 0.5, "len_low": 0.0}`

Returns Defaults for optional arguments

Return type `dict`

default_opt_arg_bounds()

Defaults for boundaries of the optional arguments.

- `{"hurst": [0, 1, "oo"], "len_low": [0, 1000, "cc"]}`

Returns Boundaries for optional arguments

Return type dict

var_factor()
Factor for C (Power-Law factor) to result in variance.

This is used to result in the right variance, which is depending on the hurst coefficient and the length-scale extents

$$\frac{\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}}{2H}$$

Returns factor

Return type float

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

len_up
Upper length scale truncation of the model.

- `len_up = len_low + len_scale`

Type float

class gstools.covmodel.tpl_models.**TPLExponential**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

Truncated-Power-Law with Exponential modes.

Notes

The truncated power law is given by a superposition of scale-dependent variograms:

$$\gamma_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}}(r) = \int_{\ell_{\mathrm{low}}}^{\ell_{\mathrm{up}}} \gamma(r,\lambda)\frac{\mathrm{d}\lambda}{\lambda}$$

with *Exponential* modes on each scale:

$$\gamma(r,\lambda) = \sigma^2(\lambda) \cdot \left(1 - \exp\left[-\frac{r}{\lambda}\right]\right)$$
$$\sigma^2(\lambda) = C \cdot \lambda^{2H}$$

This results in:

$$\gamma_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}}(r) = \sigma^2_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}} \cdot \left(1 - 2H \cdot \frac{\ell_{\mathrm{up}}^{2H} \cdot E_{1+2H}\left[\frac{r}{\ell_{\mathrm{up}}}\right] - \ell_{\mathrm{low}}^{2H} \cdot E_{1+2H}\left[\frac{r}{\ell_{\mathrm{low}}}\right]}{\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}}\right)$$
$$\sigma^2_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}} = \frac{C \cdot \left(\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}\right)}{2H}$$

The "length scale" of this model is equivalent by the integration range:

$$\ell = \ell_{\mathrm{up}} - \ell_{\mathrm{low}}$$

If you want to define an upper scale truncation, you should set `len_low` and `len_scale` accordingly.

The following Parameters occure:

- $C > 0$: The scaling factor from the Power-Law. This parameter will be calculated internally by the given variance. You can access C directly by `model.var_raw`
- $0 < H < \frac{1}{2}$: The hurst coefficient (`model.hurst`)
- $\ell_{\text{low}} \geq 0$: The lower length scale truncation of the model (`model.len_low`)
- $\ell_{\text{up}} \geq 0$: The upper length scale truncation of the model (`model.len_up`)

  This will be calculated internally by:

  - `len_up = len_low + len_scale`

  That means, that the `len_scale` in this model actually represents the integration range for the truncated power law.
- $E_s(x)$ is the exponential integral.

Other Parameters

- ****opt_arg** – The following parameters are covered by these keyword arguments
- **hurst** (`float`, optional) – Hurst coefficient of the power law. Standard range: `(0, 1)`. Default: `0.5`
- **len_low** (`float`, optional) – The lower length scale truncation of the model. Standard range: `[0, 1000]`. Default: `0.0`
- **.**

Parameters

- **`dim`** (`int`, optional) – dimension of the model. Default: `3`
- **`var`** (`float`, optional) – variance of the model (the nugget is not included in "this" variance) Default: `1.0`
- **`len_scale`** (`float` or `list`, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: `1.0`
- **`nugget`** (`float`, optional) – nugget of the model. Default: `0.0`
- **`anis`** (`float` or `list`, optional) – anisotropy ratios in the transversal directions [y, z]. Default: `1.0`
- **`angles`** (`float` or `list`, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: `0.0`
- **`integral_scale`** (`float` or `list` or `None`, optional) – If given, `len_scale` will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: `None`
- **`var_raw`** (`float` or `None`, optional) – raw variance of the model which will be multiplied with *`CovModel.var_factor`* to result in the actual variance. If given, `var` will be ignored. (This is just for models that override *`CovModel.var_factor`*) Default: `None`

- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

len_up float: Upper length scale truncation of the model.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill float: The sill of the variogram.

var float: The variance of the model.

var_bounds list: Bounds for the variance.

var_raw float: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Run checks for the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Truncated-Power-Law with Exponential modes - correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Defaults for the optional arguments.
default_opt_arg_bounds()	Defaults for boundaries of the optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for C (Power-Law factor) to result in variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

correlation(*r*)

Truncated-Power-Law with Exponential modes - correlation function.

If `len_low=0` we have a simple representation:

$$\mathrm{cor}(r) = H \cdot E_{1+H}\left[\frac{r}{\ell}\right]$$

The general case:

$$\mathrm{cor}(r) = 2H \cdot \frac{\ell_{\mathrm{up}}^{2H} \cdot E_{1+2H}\left[\frac{r}{\ell_{\mathrm{up}}}\right] - \ell_{\mathrm{low}}^{2H} \cdot E_{1+2H}\left[\frac{r}{\ell_{\mathrm{low}}}\right]}{\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}}$$

covariance(*r*)

Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

default_arg_bounds()

Provide default boundaries for arguments.

Given as a dictionary.

default_opt_arg()
Defaults for the optional arguments.

- `{"hurst": 0.25, "len_low": 0.0}`

Returns Defaults for optional arguments

Return type `dict`

default_opt_arg_bounds()
Defaults for boundaries of the optional arguments.

- `{"hurst": [0, 1, "oo"], "len_low": [0, 1000, "cc"]}`

Returns Boundaries for optional arguments

Return type `dict`

var_factor()
Factor for C (Power-Law factor) to result in variance.

This is used to result in the right variance, which is depending on the hurst coefficient and the length-scale extents

$$\frac{\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}}{2H}$$

Returns factor

Return type `float`

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

len_up
Upper length scale truncation of the model.

- `len_up = len_low + len_scale`

Type `float`

class gstools.covmodel.tpl_models.**TPLStable**(*dim=3*, *var=1.0*, *len_scale=1.0*, *nugget=0.0*, *anis=1.0*, *angles=0.0*, *integral_scale=None*, *var_raw=None*, *hankel_kw=None*, ***opt_arg*)

Bases: *gstools.covmodel.base.CovModel*

Truncated-Power-Law with Stable modes.

Notes

The truncated power law is given by a superposition of scale-dependent variograms:

$$\gamma_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}}(r) = \int_{\ell_{\mathrm{low}}}^{\ell_{\mathrm{up}}} \gamma(r,\lambda)\frac{\mathrm{d}\lambda}{\lambda}$$

with *Stable* modes on each scale:

$$\gamma(r,\lambda) = \sigma^2(\lambda) \cdot \left(1 - \exp\left[-\left(\frac{r}{\lambda}\right)^{\alpha}\right]\right)$$
$$\sigma^2(\lambda) = C \cdot \lambda^{2H}$$

This results in:

$$\gamma_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}}(r) = \sigma^2_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}} \cdot \left(1 - \frac{2H}{\alpha} \cdot \frac{\ell_{\mathrm{up}}^{2H} \cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell_{\mathrm{up}}}\right)^{\alpha}\right] - \ell_{\mathrm{low}}^{2H} \cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell_{\mathrm{low}}}\right)^{\alpha}\right]}{\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}}\right)$$

$$\sigma^2_{\ell_{\mathrm{low}},\ell_{\mathrm{up}}} = \frac{C \cdot \left(\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}\right)}{2H}$$

The "length scale" of this model is equivalent by the integration range:

$$\ell = \ell_{\mathrm{up}} - \ell_{\mathrm{low}}$$

If you want to define an upper scale truncation, you should set `len_low` and `len_scale` accordingly.

The following Parameters occure:

- $0 < \alpha \leq 2$: The shape parameter of the Stable model.
  - $\alpha = 1$: Exponential modes
  - $\alpha = 2$: Gaussian modes
- $C > 0$: The scaling factor from the Power-Law. This parameter will be calculated internally by the given variance. You can access C directly by `model.var_raw`
- $0 < H < \frac{\alpha}{2}$: The hurst coefficient (`model.hurst`)
- $\ell_{\mathrm{low}} \geq 0$: The lower length scale truncation of the model (`model.len_low`)
- $\ell_{\mathrm{up}} \geq 0$: The upper length scale truncation of the model (`model.len_up`)

  This will be calculated internally by:

  - `len_up = len_low + len_scale`

  That means, that the `len_scale` in this model actually represents the integration range for the truncated power law.
- $E_s(x)$ is the exponential integral.

Other Parameters

- ****opt_arg** – The following parameters are covered by these keyword arguments
- **hurst** (`float`, optional) – Hurst coefficient of the power law. Standard range: `(0, 1)`. Default: `0.5`
- **alpha** (`float`, optional) – Shape parameter of the stable model. Standard range: `(0, 2]`. Default: `1.5`
- **len_low** (`float`, optional) – The lower length scale truncation of the model. Standard range: `[0, 1000]`. Default: `0.0`
- **.**

Parameters

- **dim** (`int`, optional) – dimension of the model. Default: `3`
- **var** (`float`, optional) – variance of the model (the nugget is not included in "this" variance) Default: `1.0`
- **len_scale** (`float` or `list`, optional) – length scale of the model. If a single value is given, the same length-scale will be used for every direction. If multiple values (for main and transversal directions) are given, *anis* will be recalculated accordingly. Default: `1.0`
- **nugget** (`float`, optional) – nugget of the model. Default: `0.0`

- **anis** (float or list, optional) – anisotropy ratios in the transversal directions [y, z]. Default: 1.0
- **angles** (float or list, optional) – angles of rotation:
  - in 2D: given as rotation around z-axis
  - in 3D: given by yaw, pitch, and roll (known as Tait–Bryan angles)

  Default: 0.0
- **integral_scale** (float or list or None, optional) – If given, len_scale will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with *CovModel.var_factor* to result in the actual variance. If given, var will be ignored. (This is just for models that override *CovModel.var_factor*) Default: None
- **hankel_kw** (dict or None, optional) – Modify the init-arguments of hankel.SymmetricFourierTransform used for the spectrum calculation. Use with caution (Better: Don't!). None is equivalent to {"a": -1, "b": 1, "N": 1000, "h": 0.001}.

Attributes

angles numpy.ndarray: Rotation angles (in rad) of the model.

anis numpy.ndarray: The anisotropy factors of the model.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

dim int: The dimension of the model.

dist_func tuple of callable: pdf, cdf and ppf.

do_rotation bool: State if a rotation is performed.

hankel_kw dict: hankel.SymmetricFourierTransform kwargs.

has_cdf bool: State if a cdf is defined by the user.

has_ppf bool: State if a ppf is defined by the user.

integral_scale float: The main integral scale of the model.

integral_scale_vec numpy.ndarray: The integral scales in each direction.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the lenght scale.

len_scale_vec numpy.ndarray: The length scales in each direction.

len_up float: Upper length scale truncation of the model.

name str: The name of the CovModel class.

nugget float: The nugget of the model.

nugget_bounds list: Bounds for the nugget.

opt_arg list of str: Names of the optional arguments.

opt_arg_bounds dict: Bounds for the optional arguments.

pykrige_angle 2D rotation angle for pykrige.

pykrige_angle_x 3D rotation angle around x for pykrige.

pykrige_angle_y 3D rotation angle around y for pykrige.

pykrige_angle_z 3D rotation angle around z for pykrige.

pykrige_anis 2D anisotropy ratio for pykrige.

pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.

pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.

pykrige_kwargs Keyword arguments for pykrige routines.

sill `float`: The sill of the variogram.

var `float`: The variance of the model.

var_bounds `list`: Bounds for the variance.

var_raw `float`: The raw variance of the model without factor.

Methods

calc_integral_scale()	Calculate the integral scale of the isotrope model.
check_arg_bounds()	Check arguments to be within the given bounds.
check_opt_arg()	Check the optional arguments.
cor_spatial(pos)	Spatial correlation respecting anisotropy and rotation.
correlation(r)	Truncated-Power-Law with Stable modes - correlation function.
cov_nugget(r)	Covariance of the model respecting the nugget at r=0.
cov_spatial(pos)	Spatial covariance respecting anisotropy and rotation.
covariance(r)	Covariance of the model.
default_arg_bounds()	Provide default boundaries for arguments.
default_opt_arg()	Defaults for the optional arguments.
default_opt_arg_bounds()	Defaults for boundaries of the optional arguments.
fit_variogram(x_data, y_data[, maxfev])	Fiting the isotropic variogram-model to given data.
fix_dim()	Set a fix dimension for the model.
ln_spectral_rad_pdf(r)	Log radial spectral density of the model.
percentile_scale([per])	Calculate the percentile scale of the isotrope model.
plot([func])	Plot a function of a the CovModel.
pykrige_vario([args, r])	Isotropic variogram of the model for pykrige.
set_arg_bounds(**kwargs)	Set bounds for the parameters of the model.
spectral_density(k)	Spectral density of the covariance model.
spectral_rad_pdf(r)	Radial spectral density of the model.
spectrum(k)	Spectrum of the covariance model.
var_factor()	Factor for C (Power-Law factor) to result in variance.
vario_nugget(r)	Isotropic variogram of the model respecting the nugget at r=0.
vario_spatial(pos)	Spatial variogram respecting anisotropy and rotation.
variogram(r)	Isotropic variogram of the model.

check_opt_arg()
Check the optional arguments.

Warns **alpha** – If alpha is < 0.3, the model tends to a nugget model and gets numerically unstable.

correlation(*r*)

Truncated-Power-Law with Stable modes - correlation function.

If `len_low=0` we have a simple representation:

$$\mathrm{cor}(r) = \frac{2H}{\alpha} \cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell}\right)^{\alpha}\right]$$

The general case:

$$\mathrm{cor}(r) = \frac{2H}{\alpha} \cdot \frac{\ell_{\mathrm{up}}^{2H} \cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell_{\mathrm{up}}}\right)^{\alpha}\right] - \ell_{\mathrm{low}}^{2H} \cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell_{\mathrm{low}}}\right)^{\alpha}\right]}{\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}}$$

covariance(*r*)
Covariance of the model.

Given by: $C(r) = \sigma^2 \cdot \mathrm{cor}(r)$

Where $\mathrm{cor}(r)$ is the correlation function.

default_arg_bounds()
Provide default boundaries for arguments.

Given as a dictionary.

default_opt_arg()
Defaults for the optional arguments.

- `{"hurst": 0.5, "alpha": 1.5, "len_low": 0.0}`

Returns Defaults for optional arguments

Return type `dict`

default_opt_arg_bounds()
Defaults for boundaries of the optional arguments.

- `{"hurst": [0, 1, "oo"], "alpha": [0, 2, "oc"], "len_low": [0, 1000, "cc"]}`

Returns Boundaries for optional arguments

Return type `dict`

var_factor()
Factor for C (Power-Law factor) to result in variance.

This is used to result in the right variance, which is depending on the hurst coefficient and the length-scale extents

$$\frac{\ell_{\mathrm{up}}^{2H} - \ell_{\mathrm{low}}^{2H}}{2H}$$

Returns factor

Return type `float`

variogram(*r*)
Isotropic variogram of the model.

Given by: $\gamma(r) = \sigma^2 \cdot (1 - \mathrm{cor}(r)) + n$

Where $\mathrm{cor}(r)$ is the correlation function.

len_up
Upper length scale truncation of the model.

- `len_up = len_low + len_scale`

Type `float`

gstools.covmodel.plot

GStools subpackage providing plotting routines for the covariance models.

The following classes and functions are provided

plot_variogram(model[, x_min, x_max, fig, ax])	Plot variogram of a given CovModel.
plot_covariance(model[, x_min, x_max, fig, ax])	Plot covariance of a given CovModel.
plot_correlation(model[, x_min, x_max, fig, ax])	Plot correlation function of a given CovModel.
plot_vario_spatial(model[, x_min, x_max, ...])	Plot spatial variogram of a given CovModel.
plot_cov_spatial(model[, x_min, x_max, fig, ax])	Plot spatial covariance of a given CovModel.
plot_cor_spatial(model[, x_min, x_max, fig, ax])	Plot spatial correlation of a given CovModel.
plot_spectrum(model[, x_min, x_max, fig, ax])	Plot specturm of a given CovModel.
plot_spectral_density(model[, x_min, x_max, ...])	Plot spectral density of a given CovModel.
plot_spectral_rad_pdf(model[, x_min, x_max, ...])	Plot radial spectral pdf of a given CovModel.

gstools.covmodel.plot.**plot_variogram**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot variogram of a given CovModel.

gstools.covmodel.plot.**plot_covariance**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot covariance of a given CovModel.

gstools.covmodel.plot.**plot_correlation**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot correlation function of a given CovModel.

gstools.covmodel.plot.**plot_vario_spatial**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot spatial variogram of a given CovModel.

gstools.covmodel.plot.**plot_cov_spatial**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot spatial covariance of a given CovModel.

gstools.covmodel.plot.**plot_cor_spatial**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot spatial correlation of a given CovModel.

gstools.covmodel.plot.**plot_spectrum**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot specturm of a given CovModel.

gstools.covmodel.plot.**plot_spectral_density**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot spectral density of a given CovModel.

gstools.covmodel.plot.**plot_spectral_rad_pdf**(*model*, *x_min=0.0*, *x_max=None*, *fig=None*, *ax=None*)

Plot radial spectral pdf of a given CovModel.

3.6 gstools.field

GStools subpackage providing tools for spatial random fields.

Subpackages

generator	GStools subpackage providing generators for spatial random fields.
upscaling	GStools subpackage providing upscaling routines for the spatial random field.
base	GStools subpackage providing a base class for spatial fields.

Spatial Random Field

SRF(model[, mean, upscaling, generator])	A class to generate spatial random fields (SRF).

class gstools.field.**SRF**(*model*, *mean=0.0*, *upscaling='no_scaling'*, *generator='RandMeth'*, ***generator_kwargs*)

Bases: *gstools.field.base.Field*

A class to generate spatial random fields (SRF).

Parameters

- **model** (*CovModel*) – Covariance Model of the spatial random field.
- **mean** (float, optional) – mean value of the SRF
- **upscaling** (str, optional) – Method to be used for upscaling the variance at each point depending on the related element volume. See the point_volumes keyword in the *SRF.__call__* routine. At the moment, the following upscaling methods are provided:
  - "no_scaling" : No upscaling is applied to the variance. See: *var_no_scaling*
  - "coarse_graining" : A volume depended variance is calculated by the upscaling technique coarse graining. See: *var_coarse_graining*

  Default: "no_scaling"
- **generator** (str, optional) – Name of the field generator to be used. At the moment, the following generators are provided:
  - "RandMeth" : The Randomization Method. See: *RandMeth*
  - "IncomprRandMeth" : The incompressible Randomization Method. This is the original algorithm proposed by Kraichnan 1970 See: *IncomprRandMeth*
  - "VectorField" : an alias for "IncomprRandMeth"
  - "VelocityField" : an alias for "IncomprRandMeth"

  Default: "RandMeth"
- ****generator_kwargs** – Keyword arguments that are forwarded to the generator in use. Have a look at the provided generators for further information.

Attributes

cond_pos list: The position tuple of the conditions.

cond_val `list`: The values of the conditions.

condition `bool`: State if conditions ar given.

generator `callable`: The generator of the field.

mean `float`: The mean of the field.

model *CovModel*: The covariance model of the field.

upscaling `str`: Name of the upscaling method.

value_type `str`: Type of the field values (scalar, vector).

Methods

__call__(pos[, seed, point_volumes, mesh_type])	Generate the spatial random field.
cond_func(*args, **kwargs)	Conditioning method applied to the field.
del_condition()	Delete Conditions.
mesh(mesh[, points, direction, name])	Generate a field on a given meshio or ogs5py mesh.
plot([field, fig, ax])	Plot the spatial random field.
set_condition([cond_pos, cond_val, krige_type])	Condition a given spatial random field with measurements.
set_generator(generator, **generator_kwargs)	Set the generator for the field.
structured(*args, **kwargs)	Generate a field on a structured mesh.
to_pyvista([field_select, fieldname])	Create a VTK/PyVista grid of the stored field.
unstructured(*args, **kwargs)	Generate a field on an unstructured mesh.
upscaling_func(*args, **kwargs)	Upscaling method applied to the field variance.
vtk_export(filename[, field_select, fieldname])	Export the stored field to vtk.

__call__(*pos*, *seed=nan*, *point_volumes=0.0*, *mesh_type='unstructured'*)
Generate the spatial random field.

The field is saved as *self.field* and is also returned.

Parameters

- **pos** (`list`) – the position tuple, containing main direction and transversal directions
- **seed** (`int`, optional) – seed for RNG for reseting. Default: keep seed from generator
- **point_volumes** (`float` or `numpy.ndarray`) – If your evaluation points for the field are coming from a mesh, they are probably representing a certain element volume. This volume can be passed by *point_volumes* to apply the given variance upscaling. If *point_volumes* is `0` nothing is changed. Default: `0`
- **mesh_type** (`str`) – 'structured' / 'unstructured'

Returns **field** – the SRF

Return type `numpy.ndarray`

cond_func(**args*, ***kwargs*)
Conditioning method applied to the field.

del_condition()
Delete Conditions.

mesh(*mesh*, *points='centroids'*, *direction='xyz'*, *name='field'*, ***kwargs*)
Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (*meshio.Mesh or ogs5py.MSH*) – The given meshio or ogs5py mesh
- **points** (str, optional) – The points to evaluate the field at. Either the "centroids" of the mesh cells (calculated as mean of the cell vertices) or the "points" of the given mesh. Default: "centroids"
- **direction** (str, optional) – Here you can state which direction should be choosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass "xz" Default: "xyz"
- **name** (str, optional) – Name to store the field in the given mesh as point_data or cell_data. Default: "field"
- ****kwargs** – Keyword arguments forwareded to *Field.__call__*.

Notes

This will store the field in the given mesh under the given name, if a meshio mesh was given.

See: https://github.com/nschloe/meshio

See: *Field.__call__*

plot(*field='field'*, *fig=None*, *ax=None*)
Plot the spatial random field.

Parameters

- **field** (str, optional) – Field that should be plotted. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fig** (Figure or None) – Figure to plot the axes on. If *None*, a new one will be created. Default: *None*
- **ax** (Axes or None) – Axes to plot on. If *None*, a new one will be added to the figure. Default: *None*

set_condition(*cond_pos=None*, *cond_val=None*, *krige_type='ordinary'*)
Condition a given spatial random field with measurements.

Parameters

- **cond_pos** (list) – the position tuple of the conditions
- **cond_val** (numpy.ndarray) – the values of the conditions
- **krige_type** (str, optional) – Used kriging type for conditioning. Either 'ordinary' or 'simple'. Default: 'ordinary'

Notes

When using "ordinary" as krige_type, the mean attribute of the spatial random field will be overwritten with the estimated mean.

set_generator(*generator*, ***generator_kwargs*)
Set the generator for the field.

Parameters

- **generator** (str, optional) – Name of the generator to use for field generation. Default: "RandMeth"
- ****generator_kwargs** – keyword arguments that are forwarded to the generator in use.

structured(**args*, ***kwargs*)
Generate a field on a structured mesh.

See *Field.__call__*

to_pyvista(*field_select='field'*, *fieldname='field'*)
Create a VTK/PyVista grid of the stored field.

Parameters

- **field_select** (str, optional) – Field that should be stored. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fieldname** (str, optional) – Name of the field in the VTK file. Default: "field"

unstructured(**args*, ***kwargs*)
Generate a field on an unstructured mesh.

See *Field.__call__*

upscaling_func(**args*, ***kwargs*)
Upscaling method applied to the field variance.

vtk_export(*filename*, *field_select='field'*, *fieldname='field'*)
Export the stored field to vtk.

Parameters

- **filename** (str) – Filename of the file to be saved, including the path. Note that an ending (.vtr or .vtu) will be added to the name.
- **field_select** (str, optional) – Field that should be stored. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fieldname** (str, optional) – Name of the field in the VTK file. Default: "field"

cond_pos
The position tuple of the conditions.

Type list

cond_val
The values of the conditions.

Type list

condition
State if conditions ar given.

Type bool

generator
The generator of the field.

Default: *RandMeth*

Type callable

mean
The mean of the field.

Type float

model
The covariance model of the field.

Type *CovModel*

upscaling
Name of the upscaling method.

See the point_volumes keyword in the *SRF.__call__* routine. Default: "no_scaling"

Type str

value_type
Type of the field values (scalar, vector).

Type str

gstools.field.generator

GStools subpackage providing generators for spatial random fields.

The following classes are provided

RandMeth(model[, mode_no, seed, verbose])	Randomization method for calculating isotropic spatial random fields.
IncomprRandMeth(model[, mean_velocity, ...])	RandMeth for incompressible random vector fields.

class gstools.field.generator.**RandMeth**(*model*, *mode_no=1000*, *seed=None*, *verbose=False*, ***kwargs*)

Bases: object

Randomization method for calculating isotropic spatial random fields.

Parameters

- **model** (*CovModel*) – Covariance model
- **mode_no** (int, optional) – Number of Fourier modes. Default: 1000
- **seed** (int or None, optional) – The seed of the random number generator. If "None", a random seed is used. Default: None
- **verbose** (bool, optional) – Be chatty during the generation. Default: False
- ****kwargs** – Placeholder for keyword-args

Notes

The Randomization method is used to generate isotropic spatial random fields characterized by a given covariance model. The calculation looks like:

$$u(x) = \sqrt{\frac{\sigma^2}{N}} \cdot \sum_{i=1}^{N} \left(Z_{1,i} \cdot \cos\left(\langle k_i, x \rangle\right) + Z_{2,i} \cdot \sin\left(\langle k_i, x \rangle\right) \right)$$

where:

- N : fourier mode number
- $Z_{j,i}$: random samples from a normal distribution
- k_i : samples from the spectral density distribution of the covariance model

Attributes

mode_no int: Number of modes in the randomization method.

model *CovModel*: Covariance model of the spatial random field.

name str: Name of the generator.

seed int: Seed of the master RNG.

value_type str: Type of the field values (scalar, vector).

verbose bool: Verbosity of the generator.

Methods

__call__(x[, y, z, mesh_type])	Calculate the random modes for the randomization method.
reset_seed([seed])	Recalculate the random amplitudes and wave numbers with the given seed.
update([model, seed])	Update the model and the seed.

__call__(*x*, *y=None*, *z=None*, *mesh_type='unstructured'*)
Calculate the random modes for the randomization method.

This method calls the *summate_** Cython methods, which are the heart of the randomization method.

Parameters

- **x** (float, numpy.ndarray) – The x components of the pos. tuple.
- **y** (float, numpy.ndarray, optional) – The y components of the pos. tuple.
- **z** (float, numpy.ndarray, optional) – The z components of the pos. tuple.
- **mesh_type** (str, optional) – 'structured' / 'unstructured'

Returns the random modes

Return type numpy.ndarray

reset_seed(*seed=nan*)
Recalculate the random amplitudes and wave numbers with the given seed.

Parameters **seed** (int or None or numpy.nan, optional) – the seed of the random number generator. If None, a random seed is used. If numpy.nan, the actual seed will be kept. Default: numpy.nan

Notes

Even if the given seed is the present one, modes will be recalculated.

update(*model=None*, *seed=nan*)
Update the model and the seed.

If model and seed are not different, nothing will be done.

Parameters

- **model** (*CovModel* or None, optional) – covariance model. Default: None
- **seed** (int or None or numpy.nan, optional) – the seed of the random number generator. If None, a random seed is used. If numpy.nan, the actual seed will be kept. Default: numpy.nan

mode_no
Number of modes in the randomization method.

Type int

model
Covariance model of the spatial random field.

Type *CovModel*

name
Name of the generator.

Type str

seed
Seed of the master RNG.

Notes

If a new seed is given, the setter property not only saves the new seed, but also creates new random modes with the new seed.

Type int

value_type
Type of the field values (scalar, vector).

Type str

verbose
Verbosity of the generator.

Type bool

class gstools.field.generator.**IncomprRandMeth**(*model*, *mean_velocity=1.0*, *mode_no=1000*, *seed=None*, *verbose=False*, ***kwargs*)

Bases: *gstools.field.generator.RandMeth*

RandMeth for incompressible random vector fields.

Parameters

- **model** (*CovModel*) – covariance model
- **mean_velocity** (float, optional) – the mean velocity in x-direction
- **mode_no** (int, optional) – number of Fourier modes. Default: 1000
- **seed** (int or None, optional) – the seed of the random number generator. If "None", a random seed is used. Default: None
- **verbose** (bool, optional) – State if there should be output during the generation. Default: False
- ****kwargs** – Placeholder for keyword-args

Notes

The Randomization method is used to generate isotropic spatial incompressible random vector fields characterized by a given covariance model. The equation is:

$$u_i(x) = \bar{u_i}\delta_{i1} + \bar{u_i}\sqrt{\frac{\sigma^2}{N}} \cdot \sum_{j=1}^{N} p_i(k_j)\left(Z_{1,j} \cdot \cos\left(\langle k_j, x\rangle\right) + Z_{2,j} \cdot \sin\left(\langle k_j, x\rangle\right)\right)$$

where:

- $\bar{u}$: mean velocity in e_1 direction
- N : fourier mode number
- $Z_{k,j}$: random samples from a normal distribution
- k_j : samples from the spectral density distribution of the covariance model
- $p_i(k_j) = e_1 - \frac{k_i k_1}{k^2}$: the projector ensuring the incompressibility

Attributes

mode_no int: Number of modes in the randomization method.

model *CovModel*: Covariance model of the spatial random field.

name `str`: Name of the generator.

seed `int`: Seed of the master RNG.

value_type `str`: Type of the field values (scalar, vector).

verbose `bool`: Verbosity of the generator.

Methods

__call__(x[, y, z, mesh_type])	Calculate the random modes for the randomization method.
reset_seed([seed])	Recalculate the random amplitudes and wave numbers with the given seed.
update([model, seed])	Update the model and the seed.

__call__(*x*, *y=None*, *z=None*, *mesh_type='unstructured'*)

Calculate the random modes for the randomization method.

This method calls the *summate_incompr_** Cython methods, which are the heart of the randomization method. In this class the method contains a projector to ensure the incompressibility of the vector field.

Parameters

- **x** (`float`, `numpy.ndarray`) – the x components of the position tuple, the shape has to be (len(x), 1, 1) for 3d and accordingly shorter for lower dimensions
- **y** (`float`, `numpy.ndarray`, optional) – the y components of the pos. tuples. Default: `None`
- **z** (`float`, `numpy.ndarray`, optional) – the z components of the pos. tuple. Default: `None`
- **mesh_type** (`str`, optional) – 'structured' / 'unstructured'

Returns the random modes

Return type `numpy.ndarray`

reset_seed(*seed=nan*)

Recalculate the random amplitudes and wave numbers with the given seed.

Parameters **seed** (`int` or `None` or `numpy.nan`, optional) – the seed of the random number generator. If `None`, a random seed is used. If `numpy.nan`, the actual seed will be kept. Default: `numpy.nan`

Notes

Even if the given seed is the present one, modes will be recalculated.

---

update(*model=None*, *seed=nan*)

Update the model and the seed.

If model and seed are not different, nothing will be done.

Parameters

- **model** (*CovModel* or `None`, optional) – covariance model. Default: `None`
- **seed** (`int` or `None` or `numpy.nan`, optional) – the seed of the random number generator. If `None`, a random seed is used. If `numpy.nan`, the actual seed will be kept. Default: `numpy.nan`

mode_no
Number of modes in the randomization method.

Type int

model
Covariance model of the spatial random field.

Type *CovModel*

name
Name of the generator.

Type str

seed
Seed of the master RNG.

Notes

If a new seed is given, the setter property not only saves the new seed, but also creates new random modes with the new seed.

Type int

value_type
Type of the field values (scalar, vector).

Type str

verbose
Verbosity of the generator.

Type bool

gstools.field.upscaling

GStools subpackage providing upscaling routines for the spatial random field.

The following functions are provided

var_coarse_graining(model[, point_volumes])	Coarse Graning procedure to upscale the variance for uniform flow.
var_no_scaling(model, *args, **kwargs)	Dummy function to bypass scaling.

gstools.field.upscaling.**var_coarse_graining**(*model*, *point_volumes=0.0*)

Coarse Graning procedure to upscale the variance for uniform flow.

Parameters

- **model** (*CovModel*) – Covariance Model used for the field.
- **point_volumes** (float or numpy.ndarray) – Volumes of the elements at the given points. Default: 0

Returns **scaled_var** – The upscaled variance

Return type float or numpy.ndarray

Notes

This procedure was presented in [Attinger03]. It applies the upscaling procedure 'Coarse Graining' to the Groundwater flow equation under uniform flow on a lognormal distributed conductivity field following a gaussian covariance function. A filter over a cube with a given edge-length λ is applied and an upscaled conductivity field is obtained. The upscaled field is again following a gaussian covariance function with scale dependent variance and length-scale:

$$\lambda = V^{\frac{1}{d}}$$

$$\sigma^2(\lambda) = \sigma^2 \cdot \left(\frac{\ell^2}{\ell^2 + \left(\frac{\lambda}{2}\right)^2} \right)^{\frac{d}{2}}$$

$$\ell(\lambda) = \left(\ell^2 + \left(\frac{\lambda}{2} \right)^2 \right)^{\frac{1}{2}}$$

Therby λ will be calculated from the given point_volumes V by assuming a cube with the given volume.

The upscaled length scale will be ignored by this routine.

References

gstools.field.upscaling.**var_no_scaling**(*model*, **args*, ***kwargs*)

Dummy function to bypass scaling.

Parameters **model** (*CovModel*) – Covariance Model used for the field.

Returns **var** – The model variance.

Return type float

gstools.field.base

GStools subpackage providing a base class for spatial fields.

The following classes are provided

Field(model[, mean])	A field base class for random and kriging fields ect.

class gstools.field.base.**Field**(*model*, *mean=0.0*)
A field base class for random and kriging fields ect.

Parameters

- **model** (*CovModel*) – Covariance Model related to the field.
- **mean** (float, optional) – Mean value of the field.

Attributes

mean float: The mean of the field.

model *CovModel*: The covariance model of the field.

value_type str: Type of the field values (scalar, vector).

Methods

__call__(**kwargs)	Generate the field.
mesh(mesh[, points, direction, name])	Generate a field on a given meshio or ogs5py mesh.
plot([field, fig, ax])	Plot the spatial random field.
structured(*args, **kwargs)	Generate a field on a structured mesh.
to_pyvista([field_select, fieldname])	Create a VTK/PyVista grid of the stored field.
unstructured(*args, **kwargs)	Generate a field on an unstructured mesh.
vtk_export(filename[, field_select, fieldname])	Export the stored field to vtk.

__call__(***kwargs*)
Generate the field.

mesh(*mesh*, *points='centroids'*, *direction='xyz'*, *name='field'*, ***kwargs*)
Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (*meshio.Mesh or ogs5py.MSH*) – The given meshio or ogs5py mesh
- **points** (str, optional) – The points to evaluate the field at. Either the "centroids" of the mesh cells (calculated as mean of the cell vertices) or the "points" of the given mesh. Default: "centroids"
- **direction** (str, optional) – Here you can state which direction should be choosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass "xz" Default: "xyz"
- **name** (str, optional) – Name to store the field in the given mesh as point_data or cell_data. Default: "field"
- ****kwargs** – Keyword arguments forwareded to *Field.__call__*.

Notes

This will store the field in the given mesh under the given name, if a meshio mesh was given.

See: https://github.com/nschloe/meshio

See: *Field.__call__*

---

plot (*field='field'*, *fig=None*, *ax=None*)
Plot the spatial random field.

Parameters

- **field** (str, optional) – Field that should be plotted. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fig** (Figure or None) – Figure to plot the axes on. If *None*, a new one will be created. Default: *None*
- **ax** (Axes or None) – Axes to plot on. If *None*, a new one will be added to the figure. Default: *None*

structured (**args*, ***kwargs*)
Generate a field on a structured mesh.

See *Field.__call__*

to_pyvista (*field_select='field'*, *fieldname='field'*)
Create a VTK/PyVista grid of the stored field.

Parameters

- **field_select** (str, optional) – Field that should be stored. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fieldname** (str, optional) – Name of the field in the VTK file. Default: "field"

unstructured (**args*, ***kwargs*)
Generate a field on an unstructured mesh.

See *Field.__call__*

vtk_export (*filename*, *field_select='field'*, *fieldname='field'*)
Export the stored field to vtk.

Parameters

- **filename** (str) – Filename of the file to be saved, including the path. Note that an ending (.vtr or .vtu) will be added to the name.
- **field_select** (str, optional) – Field that should be stored. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fieldname** (str, optional) – Name of the field in the VTK file. Default: "field"

mean
The mean of the field.

Type float

model
The covariance model of the field.

Type *CovModel*

value_type
Type of the field values (scalar, vector).

Type str

3.7 gstools.variogram

GStools subpackage providing tools for estimating and fitting variograms.

Variogram estimation

vario_estimate_unstructured(pos, field, ...)	Estimates the variogram on a unstructured grid.
vario_estimate_structured(field[, direction])	Estimates the variogram on a regular grid.

gstools.variogram.**vario_estimate_unstructured**(*pos*, *field*, *bin_edges*, *sampling_size=None*, *sampling_seed=None*)

Estimates the variogram on a unstructured grid.

The algorithm calculates following equation:

$$\gamma(r_k) = \frac{1}{2N}\sum_{i=1}^{N}(z(\mathbf{x}_i) - z(\mathbf{x}_i'))^2, \text{ with}$$
$$r_k \leq \|\mathbf{x}_i - \mathbf{x}_i'\| < r_{k+1}$$

Notes

Internally uses double precision and also returns doubles.

Parameters

- **pos** (list) – the position tuple, containing main direction and transversal directions
- **field** (numpy.ndarray) – the spatially distributed data
- **bin_edges** (numpy.ndarray) – the bins on which the variogram will be calculated
- **sampling_size** (int or None, optional) – for large input data, this method can take a long time to compute the variogram, therefore this argument specifies the number of data points to sample randomly Default: None
- **sampling_seed** (int or None, optional) – seed for samples if sampling_size is given. Default: None

Returns the estimated variogram and the bin centers

Return type tuple of numpy.ndarray

gstools.variogram.**vario_estimate_structured**(*field*, *direction='x'*)

Estimates the variogram on a regular grid.

The indices of the given direction are used for the bins. The algorithm calculates following equation:

$$\gamma(r_k) = \frac{1}{2N}\sum_{i=1}^{N}(z(\mathbf{x}_i) - z(\mathbf{x}_i'))^2, \text{ with}$$
$$r_k \leq \|\mathbf{x}_i - \mathbf{x}_i'\| < r_{k+1}$$

Warning: It is assumed that the field is defined on an equidistant Cartesian grid.

Notes

Internally uses double precision and also returns doubles.

Parameters

- **field** (numpy.ndarray) – the spatially distributed data
- **direction** (str) – the axis over which the variogram will be estimated (x, y, z)

Returns the estimated variogram along the given direction.

Return type numpy.ndarray

3.8 gstools.krige

GStools subpackage providing kriging.

Kriging Classes

Simple(model, mean, cond_pos, cond_val)	A class for simple kriging.
Ordinary(model, cond_pos, cond_val)	A class for ordinary kriging.

class gstools.krige.**Simple**(*model*, *mean*, *cond_pos*, *cond_val*)

Bases: *gstools.field.base.Field*

A class for simple kriging.

Parameters

- **model** (*CovModel*) – Covariance Model used for kriging.
- **mean** (float, optional) – mean value of the kriging field
- **cond_pos** (list) – tuple, containing the given condition positions (x, [y, z])
- **cond_val** (numpy.ndarray) – the values of the conditions

Attributes

cond_pos list: The position tuple of the conditions.

cond_val list: The values of the conditions.

mean float: The mean of the field.

model *CovModel*: The covariance model of the field.

value_type str: Type of the field values (scalar, vector).

Methods

__call__(pos[, mesh_type])	Generate the simple kriging field.
mesh(mesh[, points, direction, name])	Generate a field on a given meshio or ogs5py mesh.
plot([field, fig, ax])	Plot the spatial random field.
set_condition(cond_pos, cond_val)	Set the conditions for kriging.
structured(*args, **kwargs)	Generate a field on a structured mesh.
to_pyvista([field_select, fieldname])	Create a VTK/PyVista grid of the stored field.
unstructured(*args, **kwargs)	Generate a field on an unstructured mesh.
vtk_export(filename[, field_select, fieldname])	Export the stored field to vtk.

__call__(*pos*, *mesh_type='unstructured'*)

Generate the simple kriging field.

The field is saved as *self.field* and is also returned.

Parameters

- **pos** (list) – the position tuple, containing main direction and transversal directions (x, [y, z])
- **mesh_type** (str) – 'structured' / 'unstructured'

Returns

- **field** (numpy.ndarray) – the kriged field
- **krige_var** (numpy.ndarray) – the kriging error variance

mesh(*mesh*, *points='centroids'*, *direction='xyz'*, *name='field'*, ***kwargs*)
Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (*meshio.Mesh or ogs5py.MSH*) – The given meshio or ogs5py mesh
- **points** (str, optional) – The points to evaluate the field at. Either the "centroids" of the mesh cells (calculated as mean of the cell vertices) or the "points" of the given mesh. Default: "centroids"
- **direction** (str, optional) – Here you can state which direction should be choosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass "xz" Default: "xyz"
- **name** (str, optional) – Name to store the field in the given mesh as point_data or cell_data. Default: "field"
- ****kwargs** – Keyword arguments forwareded to *Field.__call__*.

Notes

This will store the field in the given mesh under the given name, if a meshio mesh was given.

See: https://github.com/nschloe/meshio

See: *Field.__call__*

---

plot(*field='field'*, *fig=None*, *ax=None*)
Plot the spatial random field.

Parameters

- **field** (str, optional) – Field that should be plotted. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fig** (Figure or None) – Figure to plot the axes on. If *None*, a new one will be created. Default: *None*
- **ax** (Axes or None) – Axes to plot on. If *None*, a new one will be added to the figure. Default: *None*

set_condition(*cond_pos*, *cond_val*)
Set the conditions for kriging.

Parameters

- **cond_pos** (list) – the position tuple of the conditions (x, [y, z])
- **cond_val** (numpy.ndarray) – the values of the conditions

structured(**args*, ***kwargs*)
Generate a field on a structured mesh.

See *Field.__call__*

to_pyvista(*field_select='field'*, *fieldname='field'*)
Create a VTK/PyVista grid of the stored field.

Parameters

- **field_select** (str, optional) – Field that should be stored. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"

- **fieldname** (str, optional) – Name of the field in the VTK file. Default: "field"

unstructured(**args*, ***kwargs*)
Generate a field on an unstructured mesh.

See *Field.__call__*

vtk_export(*filename*, *field_select='field'*, *fieldname='field'*)
Export the stored field to vtk.

Parameters

- **filename** (str) – Filename of the file to be saved, including the path. Note that an ending (.vtr or .vtu) will be added to the name.
- **field_select** (str, optional) – Field that should be stored. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fieldname** (str, optional) – Name of the field in the VTK file. Default: "field"

cond_pos
The position tuple of the conditions.

Type list

cond_val
The values of the conditions.

Type list

mean
The mean of the field.

Type float

model
The covariance model of the field.

Type *CovModel*

value_type
Type of the field values (scalar, vector).

Type str

class gstools.krige.**Ordinary**(*model*, *cond_pos*, *cond_val*)
Bases: *gstools.field.base.Field*

A class for ordinary kriging.

Parameters

- **model** (*CovModel*) – Covariance Model used for kriging.
- **cond_pos** (list) – tuple, containing the given condition positions (x, [y, z])
- **cond_val** (numpy.ndarray) – the values of the conditions

Attributes

cond_pos list: The position tuple of the conditions.

cond_val list: The values of the conditions.

mean float: The mean of the field.

model *CovModel*: The covariance model of the field.

value_type str: Type of the field values (scalar, vector).

Methods

__call__(pos[, mesh_type])	Generate the ordinary kriging field.
mesh(mesh[, points, direction, name])	Generate a field on a given meshio or ogs5py mesh.
plot([field, fig, ax])	Plot the spatial random field.
set_condition(cond_pos, cond_val)	Set the conditions for kriging.
structured(*args, **kwargs)	Generate a field on a structured mesh.
to_pyvista([field_select, fieldname])	Create a VTK/PyVista grid of the stored field.
unstructured(*args, **kwargs)	Generate a field on an unstructured mesh.
vtk_export(filename[, field_select, fieldname])	Export the stored field to vtk.

__call__ (*pos*, *mesh_type='unstructured'*)
Generate the ordinary kriging field.

The field is saved as *self.field* and is also returned.

Parameters

- **pos** (list) – the position tuple, containing main direction and transversal directions (x, [y, z])
- **mesh_type** (str) – 'structured' / 'unstructured'

Returns

- **field** (numpy.ndarray) – the kriged field
- **krige_var** (numpy.ndarray) – the kriging error variance

mesh (*mesh*, *points='centroids'*, *direction='xyz'*, *name='field'*, ***kwargs*)
Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (*meshio.Mesh or ogs5py.MSH*) – The given meshio or ogs5py mesh
- **points** (str, optional) – The points to evaluate the field at. Either the "centroids" of the mesh cells (calculated as mean of the cell vertices) or the "points" of the given mesh. Default: "centroids"
- **direction** (str, optional) – Here you can state which direction should be choosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass "xz" Default: "xyz"
- **name** (str, optional) – Name to store the field in the given mesh as point_data or cell_data. Default: "field"
- ****kwargs** – Keyword arguments forwareded to *Field.__call__*.

Notes

This will store the field in the given mesh under the given name, if a meshio mesh was given.

See: https://github.com/nschloe/meshio

See: *Field.__call__*

---

plot (*field='field'*, *fig=None*, *ax=None*)
Plot the spatial random field.

Parameters

- **field** (str, optional) – Field that should be plotted. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"

- **fig** (Figure or None) – Figure to plot the axes on. If *None*, a new one will be created. Default: *None*
- **ax** (Axes or None) – Axes to plot on. If *None*, a new one will be added to the figure. Default: *None*

set_condition(*cond_pos*, *cond_val*)
Set the conditions for kriging.

Parameters

- **cond_pos** (list) – the position tuple of the conditions (x, [y, z])
- **cond_val** (numpy.ndarray) – the values of the conditions

structured(**args*, ***kwargs*)
Generate a field on a structured mesh.

See *Field.__call__*

to_pyvista(*field_select='field'*, *fieldname='field'*)
Create a VTK/PyVista grid of the stored field.

Parameters

- **field_select** (str, optional) – Field that should be stored. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fieldname** (str, optional) – Name of the field in the VTK file. Default: "field"

unstructured(**args*, ***kwargs*)
Generate a field on an unstructured mesh.

See *Field.__call__*

vtk_export(*filename*, *field_select='field'*, *fieldname='field'*)
Export the stored field to vtk.

Parameters

- **filename** (str) – Filename of the file to be saved, including the path. Note that an ending (.vtr or .vtu) will be added to the name.
- **field_select** (str, optional) – Field that should be stored. Can be: "field", "raw_field", "krige_field", "err_field" or "krige_var". Default: "field"
- **fieldname** (str, optional) – Name of the field in the VTK file. Default: "field"

cond_pos
The position tuple of the conditions.

Type list

cond_val
The values of the conditions.

Type list

mean
The mean of the field.

Type float

model
The covariance model of the field.

Type *CovModel*

value_type
Type of the field values (scalar, vector).

Type str

3.9 gstools.random

GStools subpackage for random number generation.

Random Number Generator

RNG([seed])	A random number generator for different distributions and multiple streams.

Seed Generator

MasterRNG(seed)	Master random number generator for generating seeds.

Distribution factory

dist_gen([pdf_in, cdf_in, ppf_in])	Distribution Factory.

class gstools.random.**RNG**(*seed=None*)

A random number generator for different distributions and multiple streams.

Parameters **seed** (int or None, optional) – The seed of the master RNG, if None, a random seed is used. Default: None

Attributes

random numpy.random.mtrand.RandomState: Randomstate.

seed int: Seed of the master RNG.

Methods

sample_dist([pdf, cdf, ppf, size])	Sample from a distribution given by pdf, cdf and/or ppf.
sample_ln_pdf(ln_pdf[, size, sample_around, ...])	Sample from a distribution given by ln(pdf).
sample_sphere(dim[, size])	Uniform sampling on a d-dimensional sphere.

sample_dist(*pdf=None*, *cdf=None*, *ppf=None*, *size=None*, ***kwargs*)

Sample from a distribution given by pdf, cdf and/or ppf.

Parameters

- **pdf** (callable or None, optional) – Probability density function of the given distribution, that takes a single argument Default: None
- **cdf** (callable or None, optional) – Cumulative distribution function of the given distribution, that takes a single argument Default: None
- **ppf** (callable or None, optional) – Percent point function of the given distribution, that takes a single argument Default: None
- **size** (int or None, optional) – sample size. Default: None

- ****kwargs** – Keyword-arguments that are forwarded to `scipy.stats.rv_continuous`.

Returns **samples** – the samples from the given distribution

Return type `float` or `numpy.ndarray`

Notes

At least pdf or cdf needs to be given.

---

sample_ln_pdf(*ln_pdf*, *size=None*, *sample_around=1.0*, *nwalkers=50*, *burn_in=20*, *oversampling_factor=10*)

Sample from a distribution given by ln(pdf).

This algorithm uses the `emcee.EnsembleSampler`

Parameters

- **ln_pdf** (`callable`) – The logarithm of the Probability density function of the given distribution, that takes a single argument
- **size** (`int` or `None`, optional) – sample size. Default: None
- **sample_around** (`float`, optional) – Starting point for initial guess Default: 1.
- **nwalkers** (`int`, optional) – The number of walkers in the mcmc sampler. Used for the emcee.EnsembleSampler class. Default: 100
- **burn_in** (`int`, optional) – Number of burn-in runs in the mcmc algorithm. Default: 100
- **oversampling_factor** (`int`, optional) – To guess the sample number needed for proper results, we use a factor for oversampling. The intern used sample-size is calculated by

  ```
  sample_size = max(burn_in, (size/nwalkers)*oversampling_factor)
  ```

  So at least, as much as the burn-in runs. Default: 10

sample_sphere(*dim*, *size=None*)

Uniform sampling on a d-dimensional sphere.

Parameters

- **dim** (`int`) – Dimension of the sphere. Just 1, 2, and 3 supported.
- **size** (`int`, optional) – sample size

Returns **coord** – x[, y[, z]] coordinates on the sphere with shape (dim, size)

Return type `numpy.ndarray`

random

Randomstate.

Get a stream to the numpy Random number generator. You can use this, to call any provided distribution from `numpy.random.mtrand.RandomState`.

Type `numpy.random.mtrand.RandomState`

seed

Seed of the master RNG.

The setter property not only saves the new seed, but also creates a new master RNG function with the new seed.

Type `int`

class gstools.random.**MasterRNG**(*seed*)
Master random number generator for generating seeds.

Parameters **seed** (int or None, optional) – The seed of the master RNG, if None, a random seed is used. Default: None

Attributes

seed int: Seed of the master RNG.

Methods

__call__()	Return a random seed.

__call__()
Return a random seed.

seed
Seed of the master RNG.

The setter property not only saves the new seed, but also creates a new master RNG function with the new seed.

Type int

gstools.random.**dist_gen**(*pdf_in=None*, *cdf_in=None*, *ppf_in=None*, ***kwargs*)
Distribution Factory.

Parameters

- **pdf_in** (callable or None, optional) – Proprobability distribution function of the given distribution, that takes a single argument Default: None
- **cdf_in** (callable or None, optional) – Cumulative distribution function of the given distribution, that takes a single argument Default: None
- **ppf_in** (callable or None, optional) – Percent point function of the given distribution, that takes a single argument Default: None
- ****kwargs** – Keyword-arguments forwarded to scipy.stats.rv_continuous.

Returns **dist** – The constructed distribution.

Return type scipy.stats.rv_continuous

Notes

At least pdf or cdf needs to be given.

3.10 gstools.tools

GStools subpackage providing miscellaneous tools.

Export

to_vtk_structured	
vtk_export_structured(filename, pos, fields)	Export a field to vtk structured rectilinear grid file.
to_vtk_unstructured	
vtk_export_unstructured(filename, pos, fields)	Export a field to vtk unstructured grid file.
to_vtk	
vtk_export(filename, pos, fields[, mesh_type])	Export a field to vtk.

Special functions

inc_gamma(s, x)	The (upper) incomplete gamma function.
exp_int(s, x)	The exponential integral $E_s(x)$.
inc_beta(a, b, x)	The incomplete Beta function.
tplstable_cor(r, len_scale, hurst, alpha)	The correlation function of the TPLStable model.

Geometric

xyz2pos(x[, y, z, dtype, max_dim])	Convert x, y, z to postional arguments.
pos2xyz(pos[, dtype, calc_dim, max_dim])	Convert postional arguments to x, y, z.
r3d_x(theta)	Rotation matrix about x axis.
r3d_y(theta)	Rotation matrix about y axis.
r3d_z(theta)	Rotation matrix about z axis.

gstools.tools.**vtk_export_structured**(*filename*, *pos*, *fields*)

Export a field to vtk structured rectilinear grid file.

Parameters

- **filename** (str) – Filename of the file to be saved, including the path. Note that an ending (.vtr) will be added to the name.
- **pos** (list) – the position tuple, containing main direction and transversal directions
- **fields** (dict or numpy.ndarray) – Structured fields to be saved. Either a single numpy array as returned by SRF, or a dictionary of fields with theirs names as keys.

gstools.tools.**vtk_export_unstructured**(*filename*, *pos*, *fields*)

Export a field to vtk unstructured grid file.

Parameters

- **filename** (str) – Filename of the file to be saved, including the path. Note that an ending (.vtu) will be added to the name.
- **pos** (list) – the position tuple, containing main direction and transversal directions
- **fields** (dict or numpy.ndarray) – Unstructured fields to be saved. Either a single numpy array as returned by SRF, or a dictionary of fields with theirs names as keys.

gstools.tools.**vtk_export**(*filename*, *pos*, *fields*, *mesh_type='unstructured'*)
Export a field to vtk.

Parameters

- **filename** (str) – Filename of the file to be saved, including the path. Note that an ending (.vtr or .vtu) will be added to the name.
- **pos** (list) – the position tuple, containing main direction and transversal directions
- **fields** (dict or numpy.ndarray) – [Un]structured fields to be saved. Either a single numpy array as returned by SRF, or a dictionary of fields with theirs names as keys.
- **mesh_type** (str, optional) – 'structured' / 'unstructured'. Default: structured

gstools.tools.**inc_gamma**(*s*, *x*)
The (upper) incomplete gamma function.

Given by: $\Gamma(s,x) = \int_x^\infty t^{s-1}\, e^{-t}\, \mathrm{d}t$

Parameters

- **s** (float) – exponent in the integral
- **x** (numpy.ndarray) – input values

gstools.tools.**exp_int**(*s*, *x*)
The exponential integral $E_s(x)$.

Given by: $E_s(x) = \int_1^\infty \frac{e^{-xt}}{t^s}\, \mathrm{d}t$

Parameters

- **s** (float) – exponent in the integral (should be > -100)
- **x** (numpy.ndarray) – input values

gstools.tools.**inc_beta**(*a*, *b*, *x*)
The incomplete Beta function.

Given by: $B(a,b;\, x) = \int_0^x t^{a-1}\, (1-t)^{b-1}\, dt$

Parameters

- **a** (float) – first exponent in the integral
- **b** (float) – second exponent in the integral
- **x** (numpy.ndarray) – input values

gstools.tools.**tplstable_cor**(*r*, *len_scale*, *hurst*, *alpha*)
The correlation function of the TPLStable model.

Given by

$$\mathrm{cor}(r) = \frac{2H}{\alpha} \cdot E_{1+\frac{2H}{\alpha}}\left(\left(\frac{r}{\ell}\right)^\alpha\right)$$

Parameters

- **r** (numpy.ndarray) – input values
- **len_scale** (float) – length-scale of the model.
- **hurst** (float) – Hurst coefficient of the power law.
- **alpha** (float, optional) – Shape parameter of the stable model.

gstools.tools.**xyz2pos**(*x*, *y=None*, *z=None*, *dtype=None*, *max_dim=3*)
Convert x, y, z to postional arguments.

Parameters

- **x** (numpy.ndarray) – grid axis in x-direction if structured, or first components of position vectors if unstructured
- **y** (numpy.ndarray, optional) – analog to x
- **z** (numpy.ndarray, optional) – analog to x
- **dtype** (*data-type, optional*) – The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. Default: None
- **max_dim** (int, optional) – Cut of information above the given dimension. Default: 3

Returns **pos** – the position tuple

Return type numpy.ndarray

gstools.tools.**pos2xyz**(*pos*, *dtype=None*, *calc_dim=False*, *max_dim=3*)

Convert postional arguments to x, y, z.

Parameters

- **pos** (iterable) – the position tuple, containing main direction and transversal directions
- **dtype** (*data-type, optional*) – The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence. Default: None
- **calc_dim** (bool, optional) – State if the dimension should be returned. Default: False
- **max_dim** (int, optional) – Cut of information above the given dimension. Default: 3

Returns

- **x** (numpy.ndarray) – first components of position vectors
- **y** (numpy.ndarray or None) – analog to x
- **z** (numpy.ndarray or None) – analog to x
- **dim** (int, optional) – dimension (only if calc_dim is True)

Notes

If len(pos) > 3, everything after pos[2] will be ignored.

---

gstools.tools.**r3d_x**(*theta*)

Rotation matrix about x axis.

Parameters **theta** (float) – Rotation angle

Returns Rotation matrix.

Return type numpy.ndarray

gstools.tools.**r3d_y**(*theta*)

Rotation matrix about y axis.

Parameters **theta** (float) – Rotation angle

Returns Rotation matrix.

Return type numpy.ndarray

gstools.tools.**r3d_z**(*theta*)

Rotation matrix about z axis.

Parameters **theta** (float) – Rotation angle

Returns Rotation matrix.

Return type numpy.ndarray

3.11 gstools.transform

GStools subpackage providing transformations.

Field-Transformations

binary(fld[, divide, upper, lower])	Binary transformation.
boxcox(fld[, lmbda, shift])	Box-Cox transformation.
zinnharvey(fld[, conn])	Zinn and Harvey transformation to connect low or high values.
normal_force_moments(fld)	Force moments of a normal distributed field.
normal_to_lognormal(fld)	Transform normal distribution to log-normal distribution.
normal_to_uniform(fld)	Transform normal distribution to uniform distribution on [0, 1].
normal_to_arcsin(fld[, a, b])	Transform normal distribution to the bimodal arcsin distribution.
normal_to_uquad(fld[, a, b])	Transform normal distribution to U-quadratic distribution.

gstools.transform.**binary**(*fld*, *divide=None*, *upper=None*, *lower=None*)

Binary transformation.

After this transformation, the field only has two values.

Parameters

- **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.
- **divide** (float, optional) – The dividing value. Default: `fld.mean`
- **upper** (float, optional) – The resulting upper value of the field. Default: `mean + sqrt(fld.model.sill)`
- **lower** (float, optional) – The resulting lower value of the field. Default: `mean - sqrt(fld.model.sill)`

gstools.transform.**boxcox**(*fld*, *lmbda=1*, *shift=0*)

Box-Cox transformation.

After this transformation, the again Box-Cox transformed field is normal distributed.

See: https://en.wikipedia.org/wiki/Power_transform#Box%E2%80%93Cox_transformation

Parameters

- **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.
- **lmbda** (float, optional) – The lambda parameter of the Box-Cox transformation. For `lmbda=0` one obtains the log-normal transformation. Default: `1`
- **shift** (float, optional) – The shift parameter from the two-parametric Box-Cox transformation. The field will be shifted by that value before transformation. Default: `0`

gstools.transform.**zinnharvey**(*fld*, *conn='high'*)

Zinn and Harvey transformation to connect low or high values.

After this transformation, the field is still normal distributed.

Parameters

- **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.
- **conn** (str, optional) – Desired connectivity. Either "low" or "high". Default: "high"

gstools.transform.**normal_force_moments**(*fld*)
Force moments of a normal distributed field.

After this transformation, the field is still normal distributed.

Parameters **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.

gstools.transform.**normal_to_lognormal**(*fld*)
Transform normal distribution to log-normal distribution.

After this transformation, the field is log-normal distributed.

Parameters **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.

gstools.transform.**normal_to_uniform**(*fld*)
Transform normal distribution to uniform distribution on [0, 1].

After this transformation, the field is uniformly distributed on [0, 1].

Parameters **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.

gstools.transform.**normal_to_arcsin**(*fld*, *a=None*, *b=None*)
Transform normal distribution to the bimodal arcsin distribution.

See: https://en.wikipedia.org/wiki/Arcsine_distribution

After this transformation, the field is arcsin-distributed on [a, b].

Parameters

- **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.
- **a** (float, optional) – Parameter a of the arcsin distribution (lower bound). Default: keep mean and variance
- **b** (float, optional) – Parameter b of the arcsin distribution (upper bound). Default: keep mean and variance

gstools.transform.**normal_to_uquad**(*fld*, *a=None*, *b=None*)
Transform normal distribution to U-quadratic distribution.

See: https://en.wikipedia.org/wiki/U-quadratic_distribution

After this transformation, the field is U-quadratic-distributed on [a, b].

Parameters

- **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.
- **a** (float, optional) – Parameter a of the U-quadratic distribution (lower bound). Default: keep mean and variance
- **b** (float, optional) – Parameter b of the U-quadratic distribution (upper bound). Default: keep mean and variance

BIBLIOGRAPHY

[Attinger03] Attinger, S. 2003, ''Generalized coarse graining procedures for flow in porous media", Computational Geosciences, 7(4), 253–273.

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

X

Z