



GeoStatTools Documentation

Release 1.3.0

Sebastian Müller, Lennart Schüler

May 20, 2021

1	GSTools Quickstart	1
1.1	Installation	1
	conda	1
	pip	2
1.2	Citation	2
1.3	Tutorials and Examples	2
1.4	Spatial Random Field Generation	3
	Examples	3
1.5	Estimating and fitting variograms	5
	Examples	5
1.6	Kriging and Conditioned Random Fields	6
	Example	6
1.7	User defined covariance models	7
	Example	7
1.8	Incompressible Vector Field Generation	8
	Example	8
1.9	VTK/PyVista Export	9
1.10	Requirements	9
	Optional	10
1.11	License	10
2	GSTools Tutorials	11
2.1	Random Field Generation	11
	Examples	11
2.2	The Covariance Model	24
	Provided Covariance Models	24
	Examples	25
2.3	Variogram Estimation	38
	Examples	38
2.4	Random Vector Field Generation	51
	Examples	52
2.5	Kriging	55
	Provided Kriging Methods	56
	Examples	56
2.6	Conditioned Fields	70
	Examples	70
2.7	Field transformations	73
	Examples	74
2.8	Geographic Coordinates	81
	Examples	82
2.9	Spatio-Temporal Modeling	88

Examples	89
2.10 Normalizing Data	94
Mean, Trend and Normalizers	94
Provided Normalizers	94
Examples	94
2.11 Miscellaneous Tutorials	100
Examples	101
3 GTools API	113
3.1 Purpose	113
3.2 Subpackages	113
3.3 Classes	113
Kriging	113
Spatial Random Field	114
Covariance Base-Class	114
Covariance Models	114
3.4 Functions	114
VTK-Export	114
Geometric	115
Variogram Estimation	115
3.5 Misc	115
3.6 gstools.covmodel	116
Subpackages	116
Covariance Base-Class	116
Covariance Models	128
gstools.covmodel.plot	327
3.7 gstools.field	329
Subpackages	329
Spatial Random Field	329
Field Base Class	338
gstools.field.generator	341
gstools.field.upscaling	347
3.8 gstools.variogram	348
Variogram estimation	348
Binning	351
3.9 gstools.krige	352
Kriging Classes	352
3.10 gstools.random	395
Random Number Generator	395
Seed Generator	395
Distribution factory	395
3.11 gstools.tools	398
Export	398
Special functions	398
Geometric	398
Misc	399
3.12 gstools.transform	405
Field-Transformations	405
3.13 gstools.normalizer	408
Base-Normalizer	408
Field-Normalizer	411
Convenience Routines	429
4 Changelog	431
4.1 1.3.0 - Pure Pink - 2021-04	431
Topics	431
Enhancements	434
Changes	435

	Bugfixes	435
4.2	1.2.1 - Volatile Violet - 2020-04-14	435
	Bugfixes	435
4.3	1.2.0 - Volatile Violet - 2020-03-20	435
	Enhancements	435
	Changes	436
	Bugfixes	436
4.4	1.1.1 - Reverberating Red - 2019-11-08	436
	Enhancements	436
	Changes	436
	Bugfixes	436
4.5	1.1.0 - Reverberating Red - 2019-10-01	437
	Enhancements	437
	Changes	437
	Bugfixes	437
4.6	1.0.1 - Bouncy Blue - 2019-01-18	438
	Bugfixes	438
4.7	1.0.0 - Bouncy Blue - 2019-01-16	438
	Enhancements	438
	Changes	438
	Bugfixes	438
4.8	0.4.0 - Glorious Green - 2018-07-17	439
	Bugfixes	439
4.9	0.3.6 - Original Orange - 2018-07-17	439
Bibliography		441
Python Module Index		443
Index		445



GeoStatTools provides geostatistical tools for various purposes:

- random field generation
- simple, ordinary, universal and external drift kriging
- conditioned field generation
- incompressible random vector field generation
- (automatted) variogram estimation and fitting
- directional variogram estimation and modelling
- data normalization and transformation
- many readily provided and even user-defined covariance models
- metric spatio-temporal modelling
- plotting and exporting routines

1.1 Installation

conda

GSTools can be installed via [conda](#) on Linux, Mac, and Windows. Install the package by typing the following command in a command terminal:

```
conda install gstools
```

In case conda forge is not set up for your system yet, see the easy to follow instructions on [conda forge](#). Using conda, the parallelized version of GSTools should be installed.

pip

GSTools can be installed via [pip](#) on Linux, Mac, and Windows. On Windows you can install [WinPython](#) to get Python and pip running. Install the package by typing the following into command in a command terminal:

```
pip install gstools
```

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

```
pip install git+git://github.com/GeoStat-Framework/GSTools.git@develop
```

If something went wrong during installation, try the [-I flag from pip](#).

To enable the OpenMP support, you have to provide a C compiler and OpenMP. Parallel support is controlled by an environment variable `GSTOOLS_BUILD_PARALLEL`, that can be `0` or `1` (interpreted as `0` if not present). GSTools then needs to be installed from source:

```
export GSTOOLS_BUILD_PARALLEL=1
pip install --no-binary=gstools gstools
```

Note, that the `--no-binary=gstools` option forces pip to not use a wheel for GSTools.

For the development version, you can do almost the same:

```
export GSTOOLS_BUILD_PARALLEL=1
pip install git+git://github.com/GeoStat-Framework/GSTools.git@develop
```

1.2 Citation

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

Sebastian Müller & Lennart Schüller. GeoStat-Framework/GSTools. Zenodo.
<https://doi.org/10.5281/zenodo.1313628>

If you want to cite a specific version, have a look at the Zenodo site.

A publication for the GeoStat-Framework is in preparation.

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
- Geographic Coordinates
- Spatio-Temporal Modelling
- Normalizing Data
- Miscellaneous examples

1.4 Spatial Random Field Generation

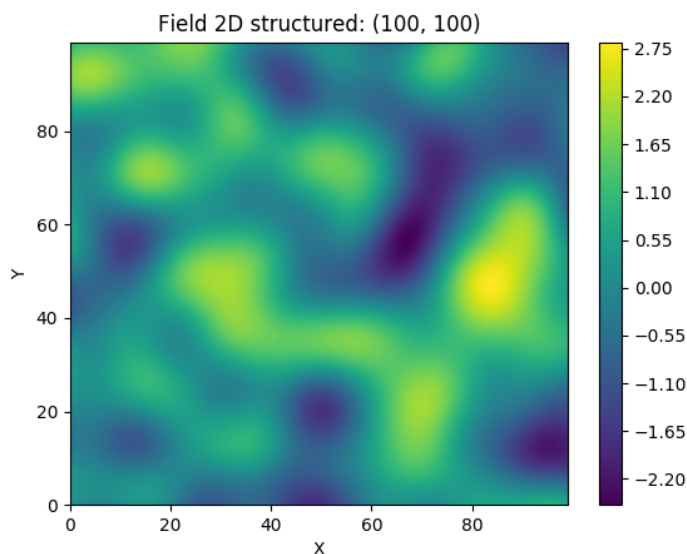
The core of this library is the generation of spatial random fields. These fields are generated using the randomisation method, described by Heße et al. 2014.

Examples

Gaussian Covariance Model

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

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



GSTools also provides support for geographic coordinates. This works perfectly well with *cartopy*.

```
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import gstools as gs
# define a structured field by latitude and longitude
lat = lon = range(-80, 81)
model = gs.Gaussian(latlon=True, len_scale=777, rescale=gs.EARTH_RADIUS)
```

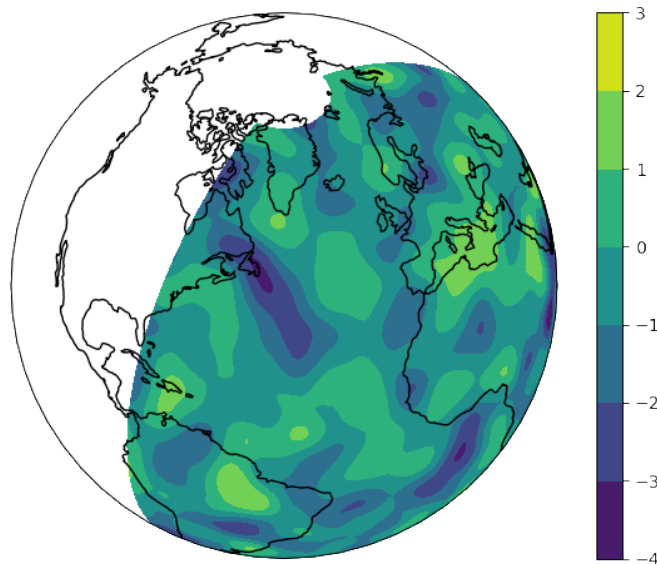
(continues on next page)

(continued from previous page)

```

srf = gs.SRF(model, seed=12345)
field = srf.structured((lat, lon))
# Orthographic plotting with cartopy
ax = plt.subplot(projection=ccrs.Orthographic(-45, 45))
cont = ax.contourf(lon, lat, field, transform=ccrs.PlateCarree())
ax.coastlines()
ax.set_global()
plt.colorbar(cont)

```



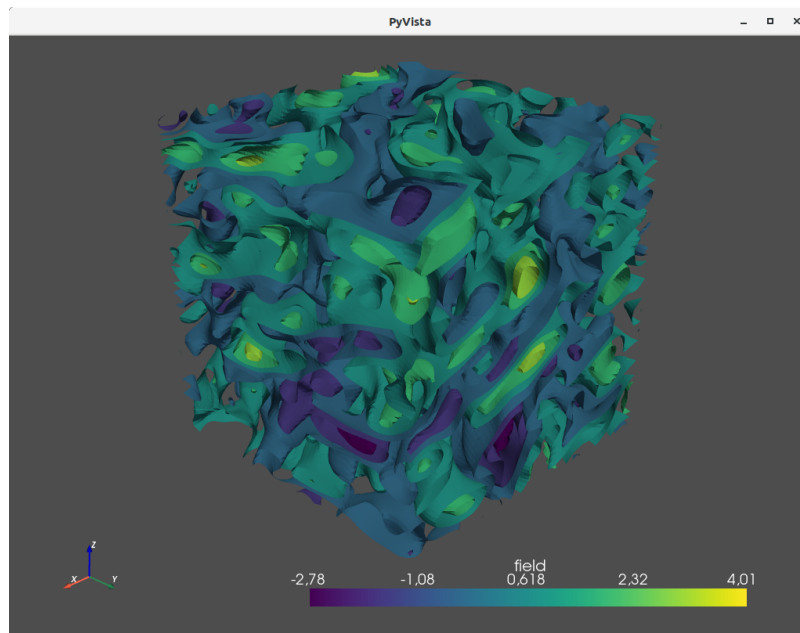
A similar example but for a three dimensional field is exported to a VTK file, which can be visualized with [ParaView](#) or [PyVista](#) in Python:

```

import gstools as gs
# structured field with a size 100x100x100 and a grid-size of 1x1x1
x = y = z = range(100)
model = gs.Gaussian(dim=3, len_scale=[16, 8, 4], angles=(0.8, 0.4, 0.2))
srf = gs.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.contour(isosurfaces=8).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.

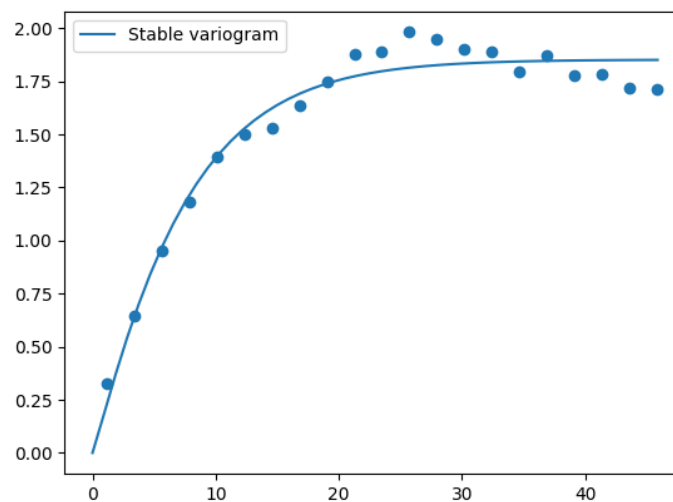
Examples

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
import gstools as gs
# 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 = gs.Exponential(dim=2, var=2, len_scale=8)
srf = gs.SRF(model, mean=0, seed=19970221)
field = srf((x, y))
# estimate the variogram of the field
bin_center, gamma = gs.vario_estimate((x, y), field)
# fit the variogram with a stable model. (no nugget fitted)
fit_model = gs.Stable(dim=2)
fit_model.fit_variogram(bin_center, gamma, nugget=False)
# output
ax = fit_model.plot(x_max=max(bin_center))
ax.scatter(bin_center, gamma)
print(fit_model)
```

Which gives:

```
Stable(dim=2, var=1.85, len_scale=7.42, nugget=0.0, anis=[1.0], angles=[0.0], alpha=1.
↪09)
```



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:

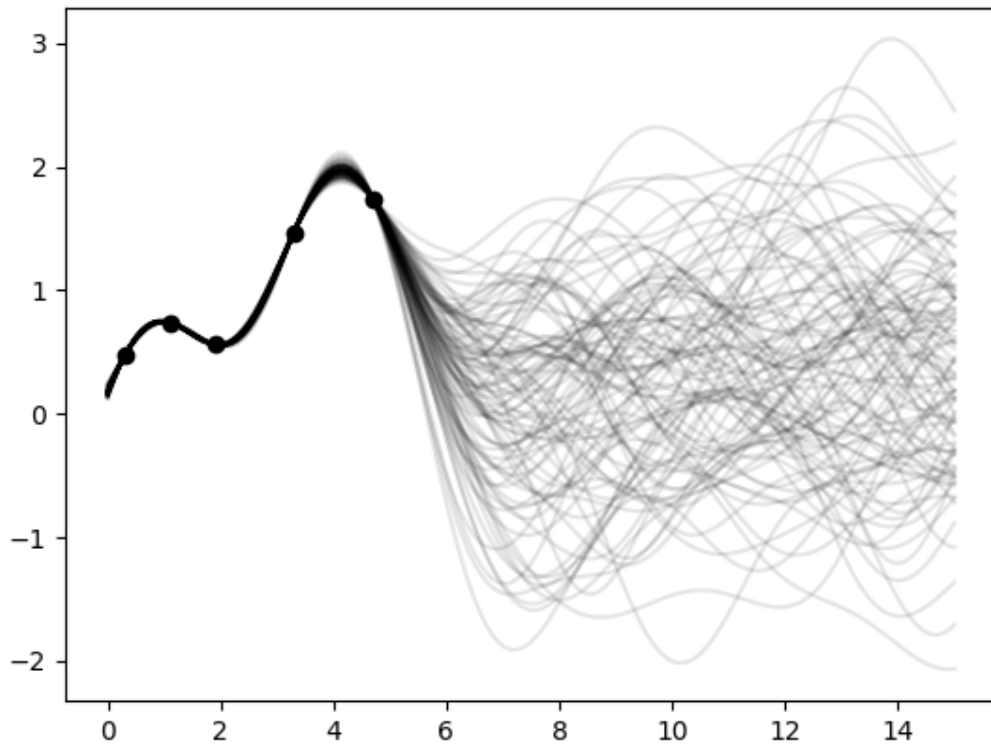
```
import numpy as np
import matplotlib.pyplot as plt
import gstools as gs

# 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)

# conditioned spatial random field class
model = gs.Gaussian(dim=1, var=0.5, len_scale=2)
krige = gs.krige.Ordinary(model, cond_pos, cond_val)
cond_srf = gs.CondSRF(krige)

# generate the ensemble of field realizations
fields = []
for i in range(100):
    fields.append(cond_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 powerful [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 $h = r/l$

```
import numpy as np
import gstools as gs
# use CovModel as the base-class
class Gau(gs.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.

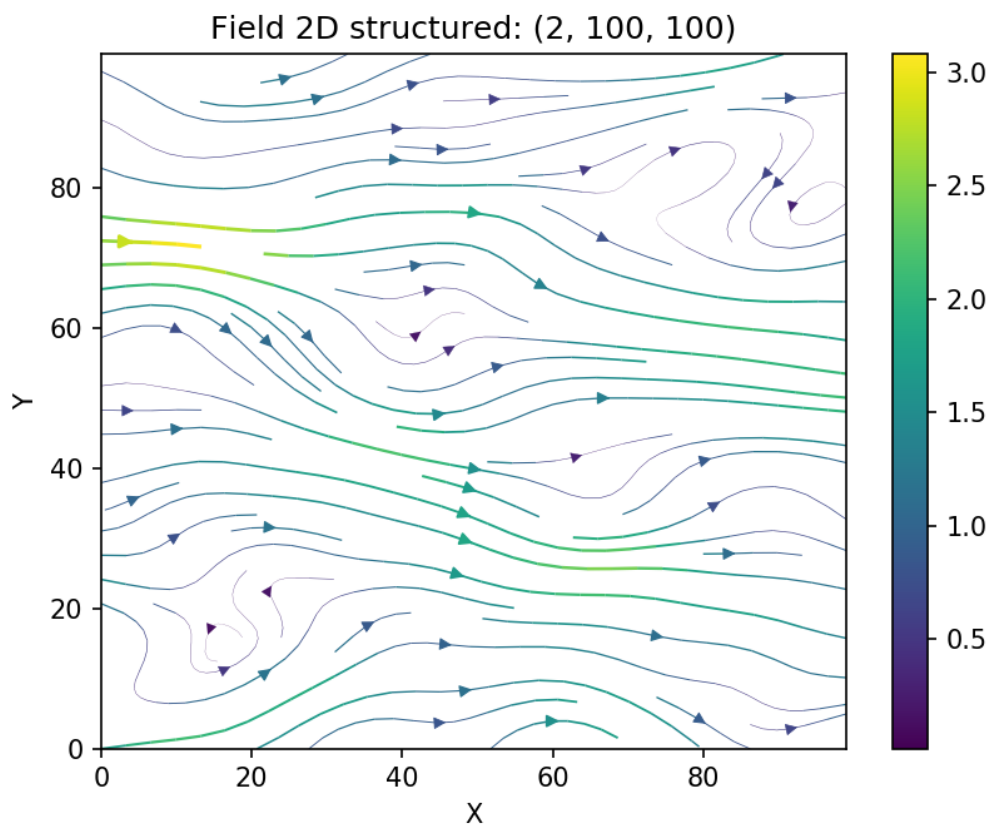
1.8 Incompressible Vector Field Generation

Using the original [Kraichnan method](#), incompressible random spatial vector fields can be generated.

Example

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

yielding

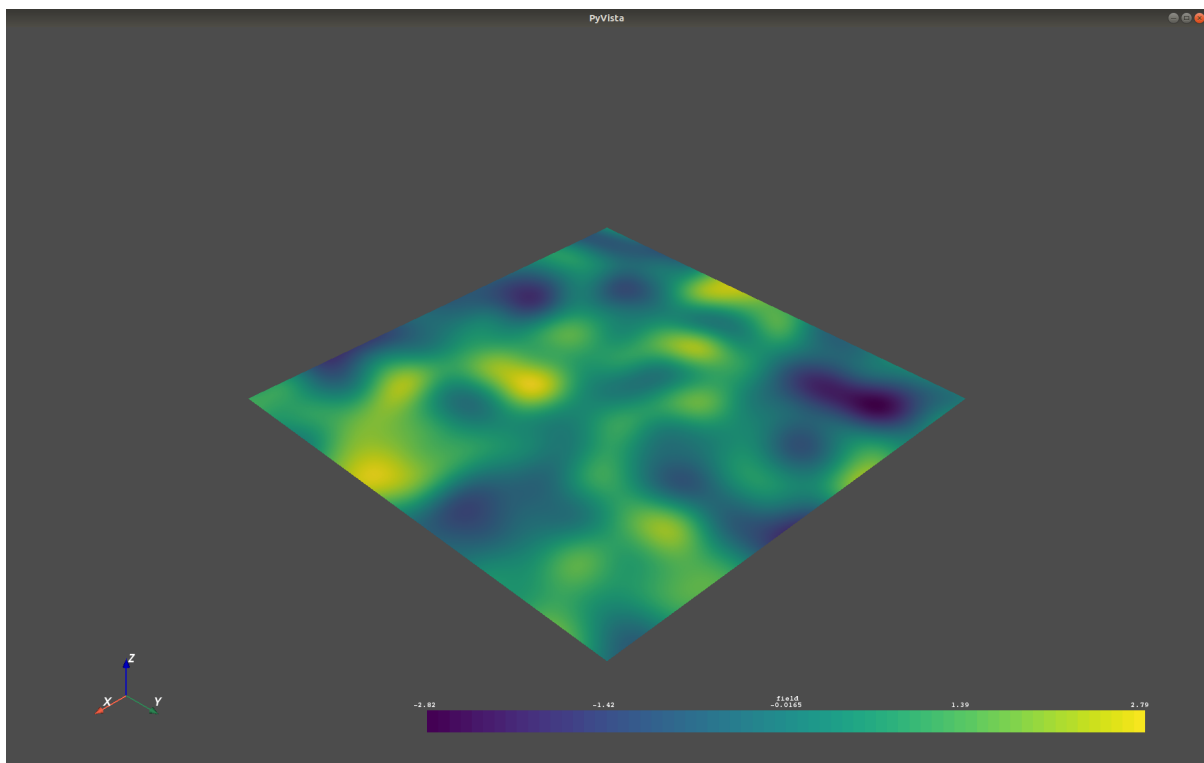


1.9 VTK/PyVista Export

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 the `to_pyvista()` method:

```
import gstools as gs
x = y = range(100)
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.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 >= 1.0.2
- emcee >= 3.0.0
- pyevtk >= 1.1.1
- meshio >= 4.0.3, < 5.0

Optional

- matplotlib
- pyvista

1.11 License

LGPLv3

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

2.1 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

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.

Examples

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.

```
import gstools as gs

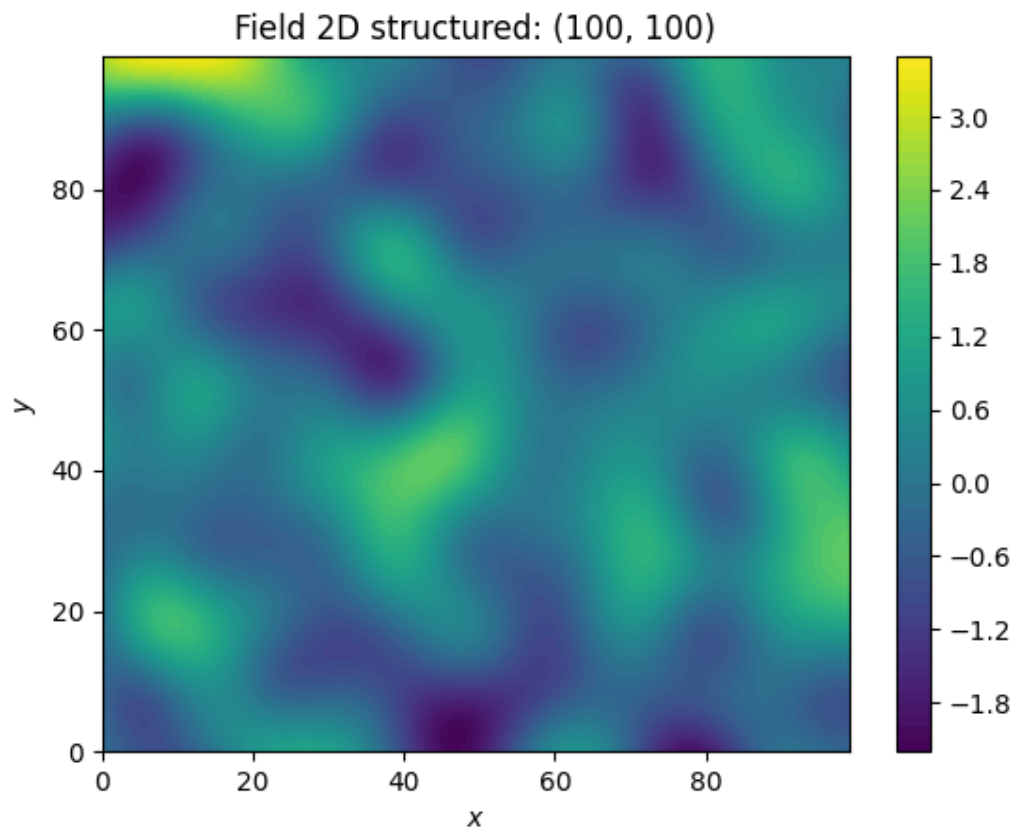
x = y = range(100)
```

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 = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.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()
```



Wow, that was pretty easy!

Total running time of the script: (0 minutes 0.751 seconds)

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 plt
import gstools as gs

x = y = np.arange(100)

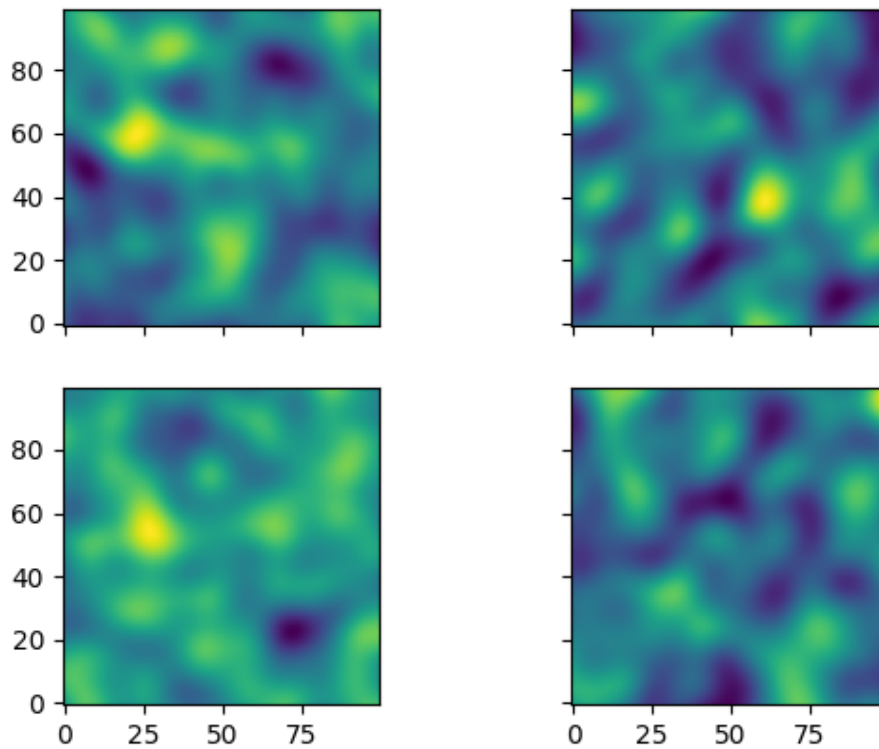
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.SRF(model)
```

This time, we did not provide a seed to *SRF*, as the seeds will be 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 = plt.subplots(2, 2, sharex=True, sharey=True)
ax = ax.flatten()
for i in range(ens_no):
    ax[i].imshow(field[i].T, origin="lower")
plt.show()
```



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()))
```

Total running time of the script: (0 minutes 4.069 seconds)

Creating Fancier Fields

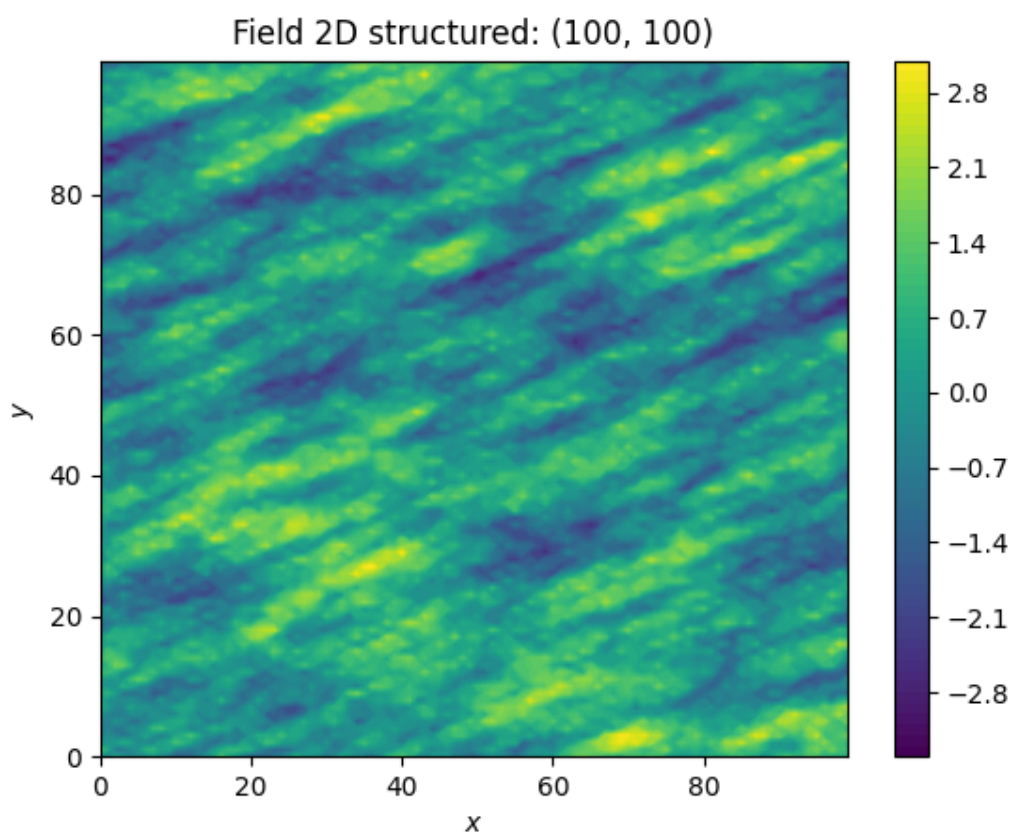
Only using Gaussian covariance fields gets boring. Now we are going to create much rougher random fields by using an exponential covariance model and we are going to make them anisotropic.

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

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

```
import numpy as np
import gstools as gs

x = y = np.arange(100)
model = gs.Exponential(dim=2, var=1, len_scale=[12.0, 3.0], angles=np.pi / 8)
srf = gs.SRF(model, seed=20170519)
srf.structured([x, y])
srf.plot()
```



The anisotropy ratio could also have been set with

```
model = gs.Exponential(dim=2, var=1, len_scale=12, anis=0.25, angles=np.pi / 8)
```

Total running time of the script: (0 minutes 1.050 seconds)

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.

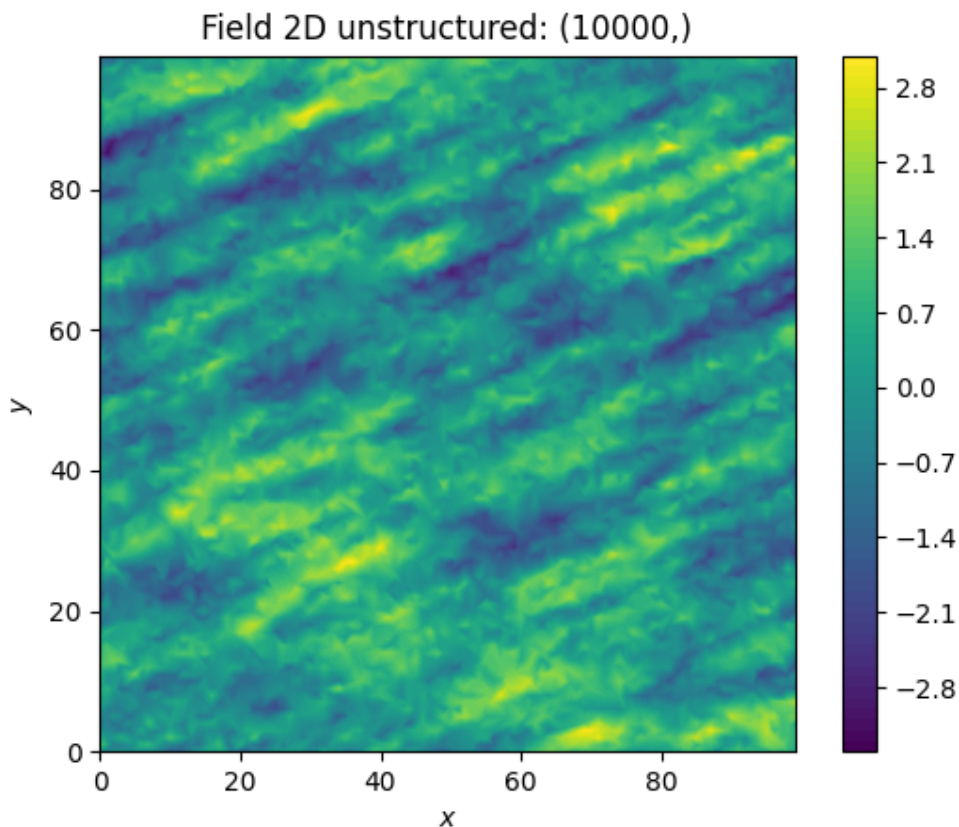
```
import numpy as np
import gstools as gs
```

Creating our own unstructured grid

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

model = gs.Exponential(dim=2, var=1, len_scale=[12, 3], angles=np.pi / 8)
srf = gs.SRF(model, seed=20170519)
field = srf((x, y))
srf.vtk_export("field")
# Or create a PyVista dataset
# mesh = srf.to_pyvista()
```

```
ax = srf.plot()
ax.set_aspect("equal")
```



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

Total running time of the script: (0 minutes 1.109 seconds)

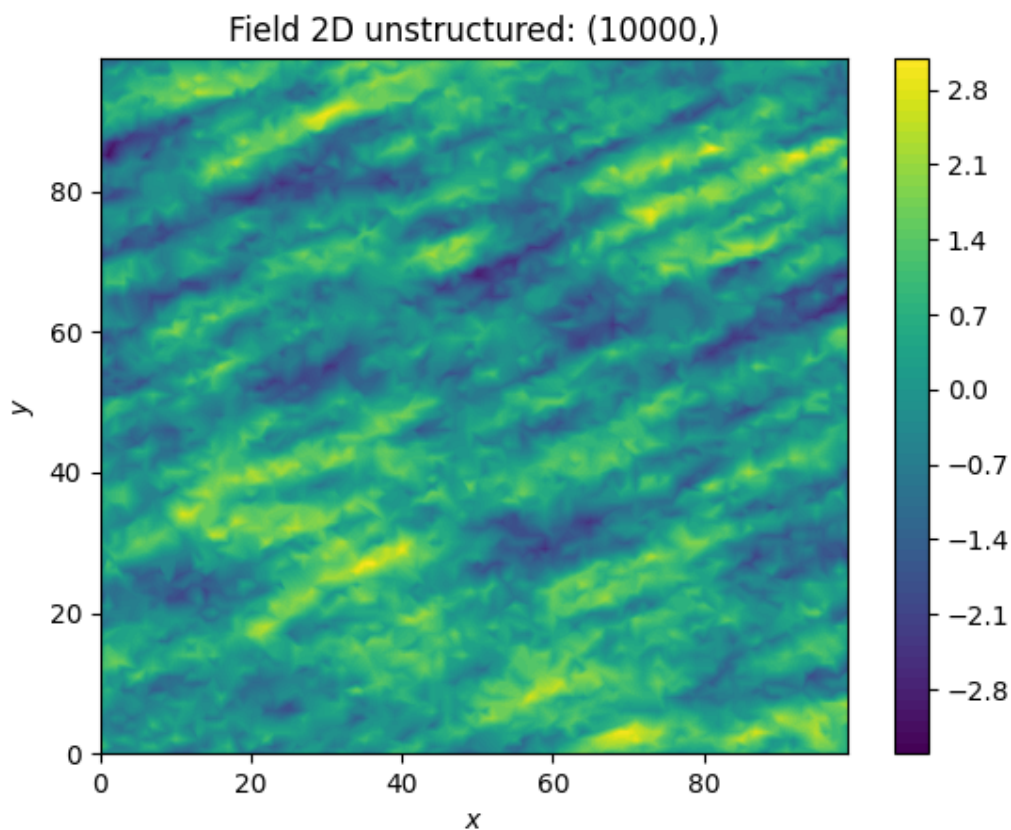
Merging two Fields

We can even generate the same field realisation on different grids. Let's try to merge two unstructured rectangular fields.

```
import numpy as np
import gstools as gs

# creating our own unstructured grid
seed = gs.random.MasterRNG(19970221)
rng = np.random.RandomState(seed())
x = rng.randint(0, 100, size=10000)
y = rng.randint(0, 100, size=10000)

model = gs.Exponential(dim=2, var=1, len_scale=[12, 3], angles=np.pi / 8)
srf = gs.SRF(model, seed=20170519)
field1 = srf((x, y))
srf.plot()
```



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:

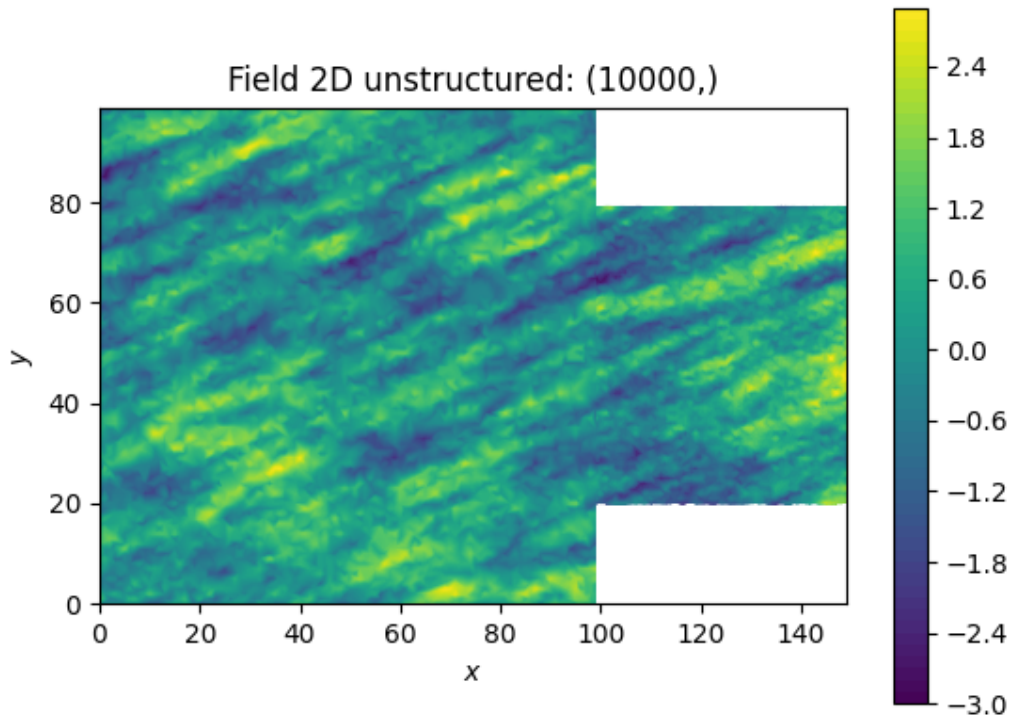
```
# new grid
seed = gs.random.MasterRNG(20011012)
rng = np.random.RandomState(seed())
x2 = rng.randint(99, 150, size=10000)
y2 = rng.randint(20, 80, size=10000)

field2 = srf((x2, y2))
```

(continues on next page)

(continued from previous page)

```
ax = srf.plot()
ax.tricontourf(x, y, field1.T, levels=256)
ax.set_aspect("equal")
```



The slight mismatch where the two fields were merged is merely due to interpolation problems of the plotting routine. You can convince yourself by 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 (x_2 , y_2) to the original grid (x , y) before generating the field. But one application scenario would be to generate huge fields, which would not fit into memory anymore.

Total running time of the script: (0 minutes 2.413 seconds)

Generating Fields on Meshes

GSTools provides an interface for meshes, to support `meshio` and `ogs5py` meshes.

When using `meshio`, the generated fields will be stored immediately in the mesh container.

There are two options to generate a field on a given mesh:

- `points="points"` will generate a field on the mesh points
- `points="centroids"` will generate a field on the cell centroids

In this example, we will generate a simple mesh with the aid of `meshzoo`.

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

(continues on next page)

(continued from previous page)

```
import meshzoo
import meshio
import gstools as gs

# generate a triangulated hexagon with meshzoo
points, cells = meshzoo.ngon(6, 4)
mesh = meshio.Mesh(points, {"triangle": cells})
```

Now we prepare the SRF class as always. We will generate an ensemble of fields on the generated mesh.

```
# number of fields
fields_no = 12
# model setup
model = gs.Gaussian(dim=2, len_scale=0.5)
srf = gs.SRF(model, mean=1)
```

To generate fields on a mesh, we provide a separate method: `SRF.mesh`. First we generate fields on the mesh-centroids controlled by a seed. You can specify the field name by the keyword *name*.

```
for i in range(fields_no):
    srf.mesh(mesh, points="centroids", name="c-field-{}".format(i), seed=i)
```

Now we generate fields on the mesh-points again controlled by a seed.

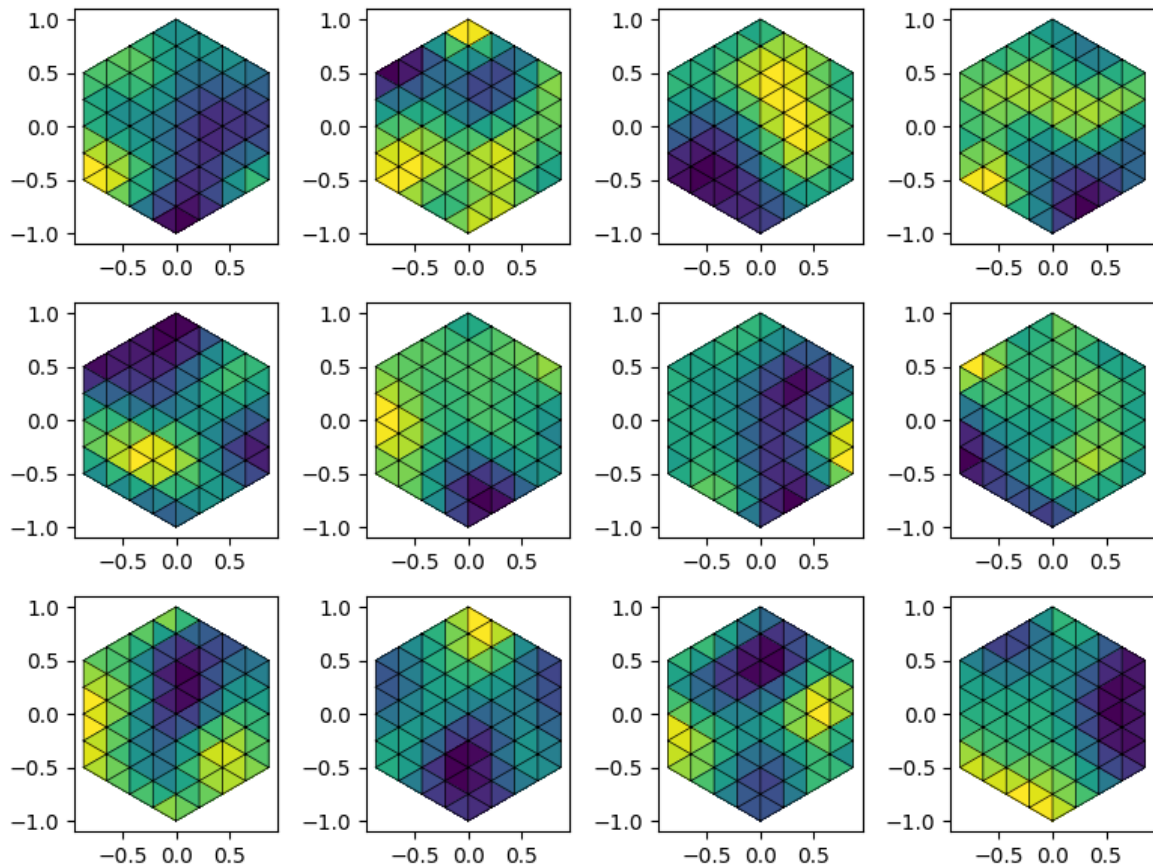
```
for i in range(fields_no):
    srf.mesh(mesh, points="points", name="p-field-{}".format(i), seed=i)
```

To get an impression we now want to plot the generated fields. Luckily, matplotlib supports triangular meshes.

```
triangulation = tri.Triangulation(points[:, 0], points[:, 1], cells)
# figure setup
cols = 4
rows = int(np.ceil(fields_no / cols))
```

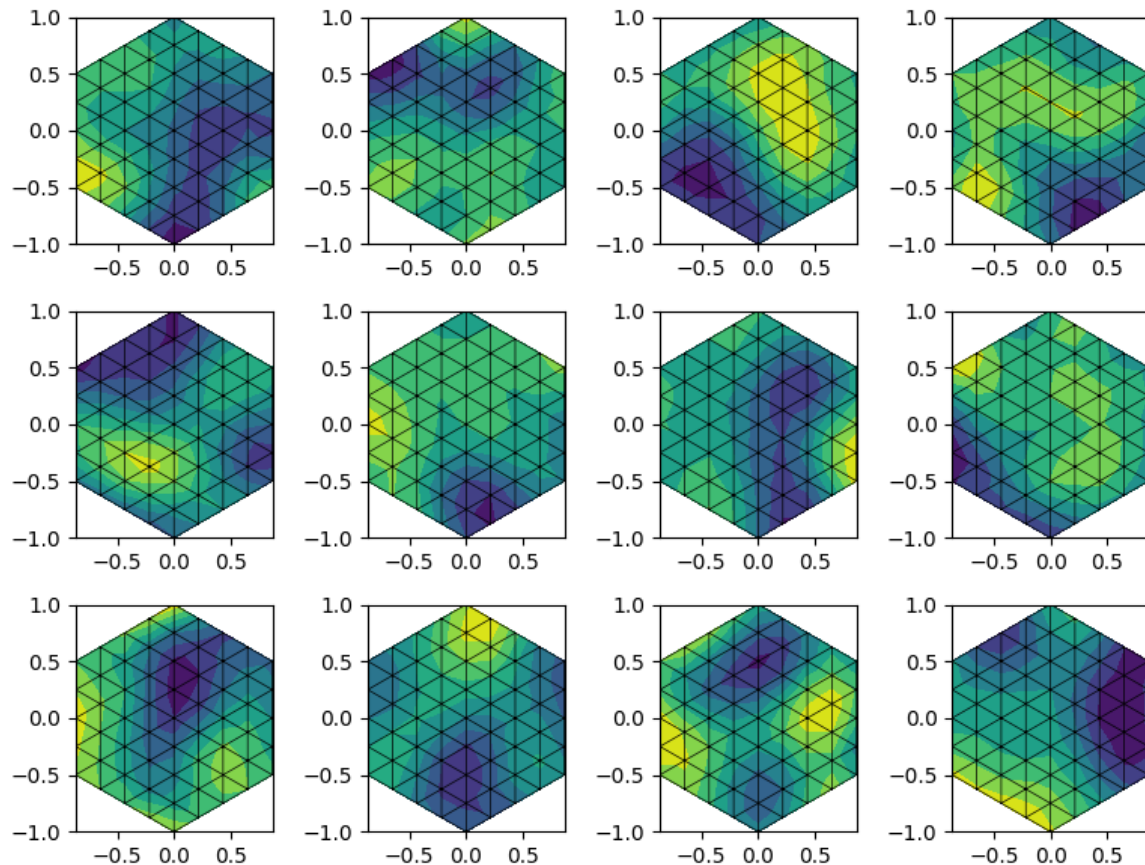
Cell data can be easily visualized with matplotlibs *tripcolor*. To highlight the cell structure, we use *triplot*.

```
fig = plt.figure(figsize=[2 * cols, 2 * rows])
for i, field in enumerate(mesh.cell_data, 1):
    ax = fig.add_subplot(rows, cols, i)
    ax.tripcolor(triangulation, mesh.cell_data[field][0])
    ax.triplot(triangulation, linewidth=0.5, color="k")
    ax.set_aspect("equal")
fig.tight_layout()
```

Point data is plotted via *tricontourf*.

```
fig = plt.figure(figsize=[2 * cols, 2 * rows])
for i, field in enumerate(mesh.point_data, 1):
    ax = fig.add_subplot(rows, cols, i)
    ax.tricontourf(triangulation, mesh.point_data[field])
    ax.triplot(triangulation, linewidth=0.5, color="k")
    ax.set_aspect("equal")
fig.tight_layout()
plt.show()
```



Last but not least, *meshio* can be used for what it does best: Exporting. Tada!

```
mesh.write("mesh_ensemble.vtk")
```

Out:

```
WARNING:root:VTK requires 3D points, but 2D points given. Appending 0 third component.
```

Total running time of the script: (0 minutes 1.741 seconds)

Using PyVista meshes

PyVista is a helper module for the Visualization Toolkit (VTK) that takes a different approach on interfacing with VTK through NumPy and direct array access.

It provides mesh data structures and filtering methods for spatial datasets, makes 3D plotting simple and is built for large/complex data geometries.

The *Field.mesh* method enables easy field creation on PyVista meshes used by the *SRF* or *Krige* class.

```
import pyvista as pv
import gstools as gs
```

We create a structured grid with PyVista containing 50 segments on all three axes each with a length of 2 (whatever unit).

```
dim, spacing = (50, 50, 50), (2, 2, 2)
grid = pv.UniformGrid(dim, spacing)
```

Now we set up the SRF class as always. We'll use an anisotropic model.

```
model = gs.Gaussian(dim=3, len_scale=[16, 8, 4], angles=(0.8, 0.4, 0.2))
srf = gs.SRF(model, seed=19970221)
```

The PyVista mesh can now be directly passed to the `SRF.mesh` method. When dealing with meshes, one can choose if the field should be generated on the mesh-points (“*points*”) or the cell-centroids (“*centroids*”).

In addition we can set a name, under which the resulting field is stored in the mesh.

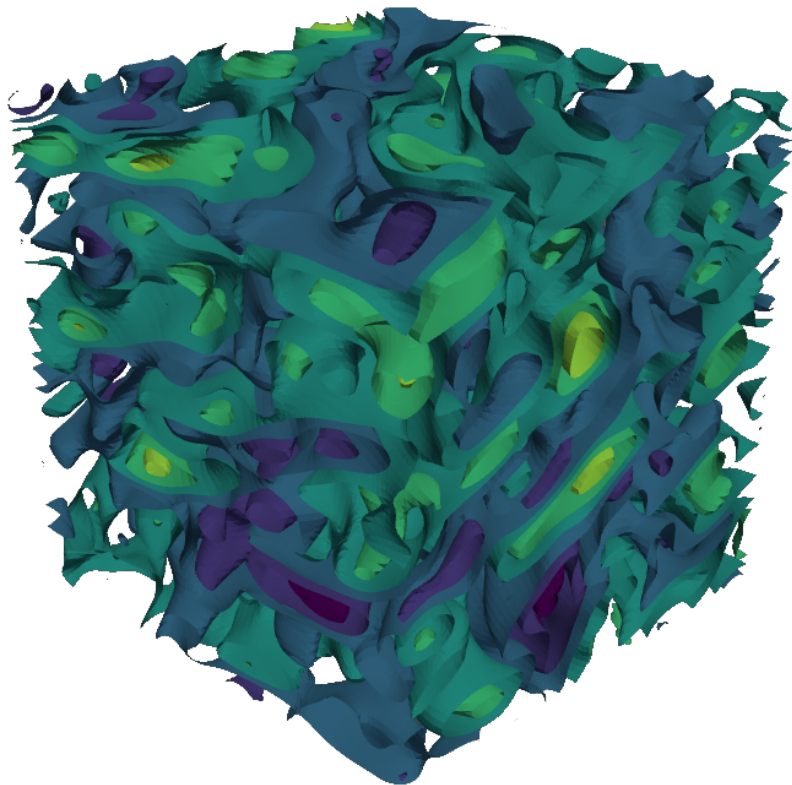
```
srf.mesh(grid, points="points", name="random-field")
```

Now we have access to PyVista’s abundance of methods to explore the field.

Note: PyVista is not working on readthedocs, but you can try it out yourself by uncommenting the following line of code.

```
# grid.contour(isosurfaces=8).plot()
```

The result should look like this:



Total running time of the script: (0 minutes 7.178 seconds)

Higher Dimensions

GSTools provides experimental support for higher dimensions.

Anisotropy is the same as in lower dimensions:

- in n dimensions we need $(n-1)$ anisotropy ratios

Rotation on the other hand is a bit more complex. With increasing dimensions more and more rotation angles are added in order to properly describe the rotated axes of anisotropy.

By design the first rotation angles coincide with the lower ones:

- 2D (rotation in x-y plane) -> 3D: first angle describes xy-plane rotation
- 3D (Tait-Bryan angles) -> 4D: first 3 angles coincide with Tait-Bryan angles

By increasing the dimension from n to $(n+1)$, n angles are added:

- 2D (1 angle) -> 3D: 3 angles (2 added)
- 3D (3 angles) -> 4D: 6 angles (3 added)

the following list of rotation-planes are described by the list of angles in the model:

1. x-y plane
2. x-z plane
3. y-z plane
4. x-v plane
5. y-v plane
6. z-v plane
7. ...

The rotation direction in these planes have alternating signs in order to match Tait-Bryan in 3D.

Let's have a look at a 4D example, where we naively add a 4th dimension.

```
import matplotlib.pyplot as plt
import gstools as gs

dim = 4
size = 20
pos = [range(size)] * dim
model = gs.Exponential(dim=dim, len_scale=5)
srf = gs.SRF(model, seed=20170519)
field = srf.structured(pos)
```

In order to “prove” correctness, we can calculate an empirical variogram of the generated field and fit our model to it.

```
bin_center, vario = gs.vario_estimate(
    pos, field, sampling_size=2000, mesh_type="structured"
)
model.fit_variogram(bin_center, vario)
print(model)
```

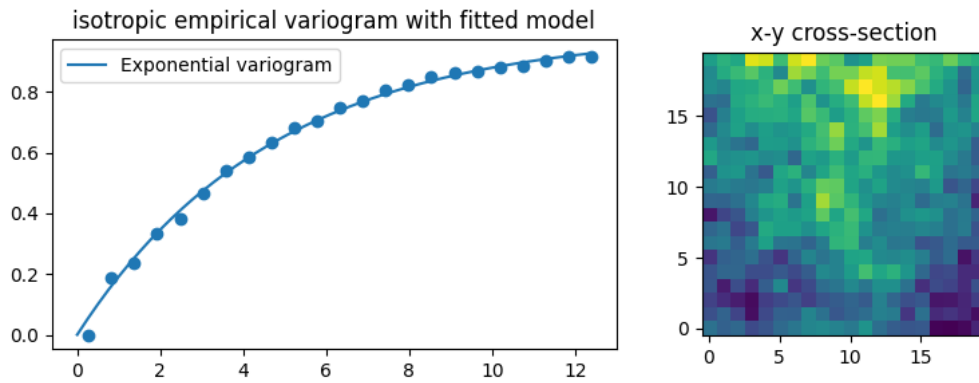
Out:

```
Exponential(dim=4, var=1.0, len_scale=4.7, nugget=1.53e-26)
```

As you can see, the estimated variance and length scale match our input quite well.

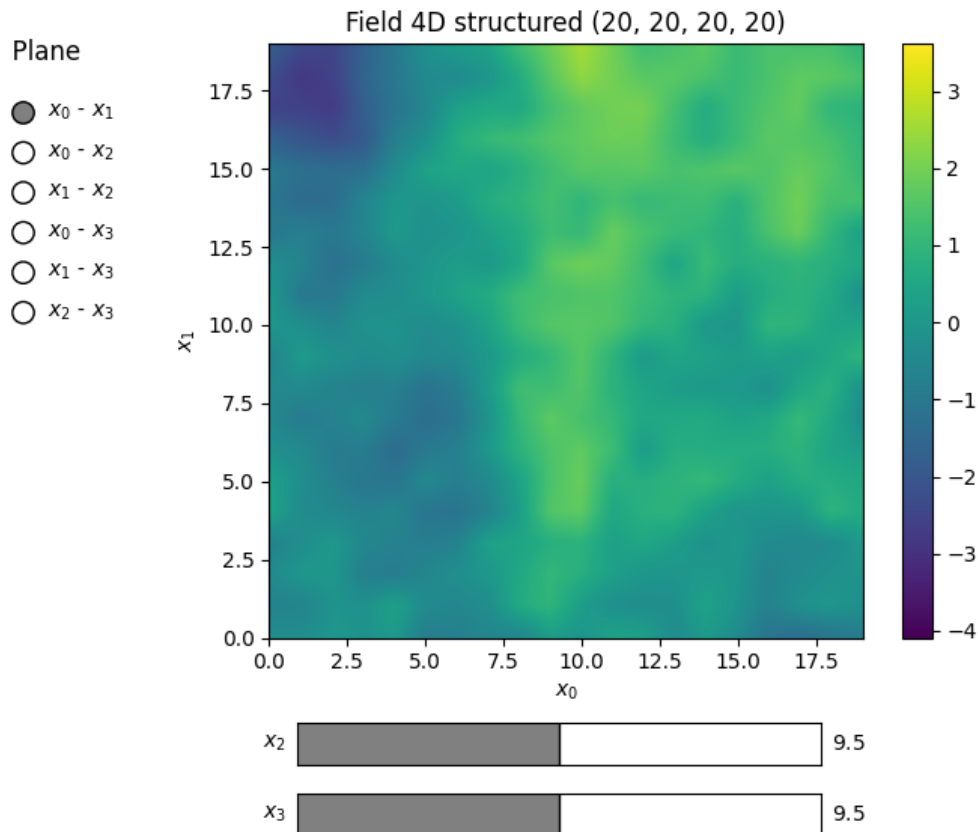
Let's have a look at the fit and a x-y cross-section of the 4D field:

```
f, a = plt.subplots(1, 2, gridspec_kw={"width_ratios": [2, 1]}, figsize=[9, 3])
model.plot(x_max=max(bin_center), ax=a[0])
a[0].scatter(bin_center, vario)
a[1].imshow(field[:, :, 0, 0].T, origin="lower")
a[0].set_title("isotropic empirical variogram with fitted model")
a[1].set_title("x-y cross-section")
f.show()
```



GSTools also provides plotting routines for higher dimensions. Fields are shown by 2D cross-sections, where other dimensions can be controlled via sliders.

```
srf.plot()
```



Total running time of the script: (0 minutes 10.881 seconds)

2.2 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.

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:

$$\gamma(r) = \sigma^2 \cdot \left(1 - \text{cor}\left(s \cdot \frac{r}{\ell}\right)\right) + n$$

Where:

- r is the lag distance
- ℓ is the main correlation length
- s is a scaling factor for unit conversion or normalization
- σ^2 is the variance
- n is the nugget (subscale variance)
- $\text{cor}(h)$ is the normalized correlation function depending on the non-dimensional distance $h = s \cdot \frac{r}{\ell}$

Depending on the normalized correlation function, all covariance models in GSTools are providing the following functions:

- $\rho(r) = \text{cor}\left(s \cdot \frac{r}{\ell}\right)$ is the so called [correlation](#) function
- $C(r) = \sigma^2 \cdot \rho(r)$ is the so called [covariance](#) function, which gives the name for our GSTools class

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

- x_0 (main direction)
- x_1 (1. transversal direction)
- x_2 (2. transversal direction)
- ...

These main directions can also be rotated. Just have a look at the corresponding examples.

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.
Cubic ([dim, var, len_scale, nugget, anis, ...])	The Cubic 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.
HyperSpherical ([dim, var, len_scale, ...])	The Hyper-Spherical covariance model.
SuperSpherical ([dim, var, len_scale, ...])	The Super-Spherical covariance model.
JBessel ([dim, var, len_scale, nugget, anis, ...])	The J-Bessel hole model.
TPLSimple ([dim, var, len_scale, nugget, ...])	The simply truncated power law model.

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

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

These models provide a lower and upper length scale truncation for superpositioned models.

Examples

Introductory 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 “normalized” `correlation` function:

```
import numpy as np
import gstools as gs

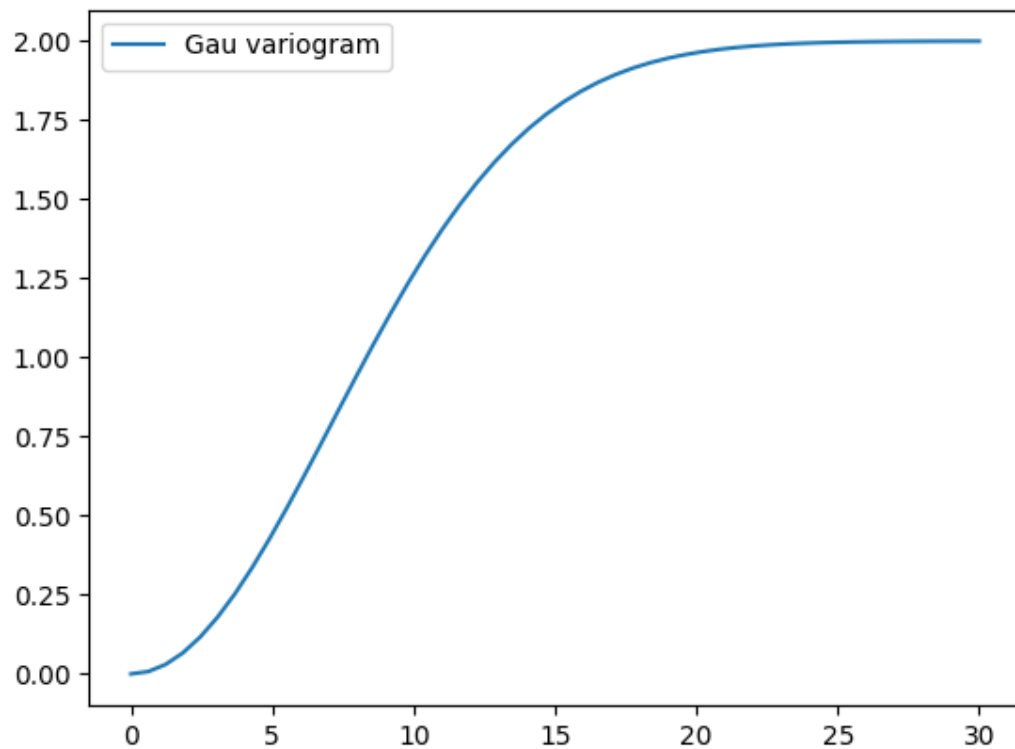
# use CovModel as the base-class
class Gau(gs.CovModel):
    def cor(self, h):
        return np.exp(-(h ** 2))
```

Here the parameter `h` stands for the normalized range $r / \text{len_scale}$. Now we can instantiate this model:

```
model = Gau(dim=2, var=2.0, len_scale=10)
```

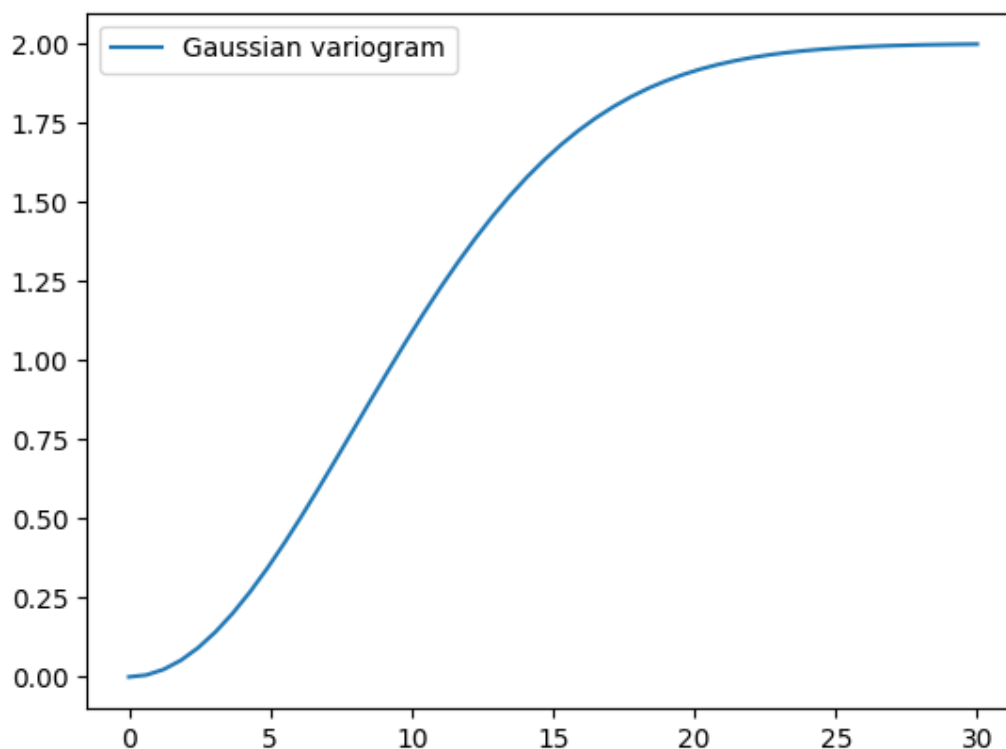
To have a look at the variogram, let's plot it:

```
model.plot()
```



This is almost identical to the already provided *Gaussian* model. There, a scaling factor is implemented so the `len_scale` coincides with the integral scale:

```
gau_model = gs.Gaussian(dim=2, var=2.0, len_scale=10)
gau_model.plot()
```

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("old model:", model)
model.dim = 3
model.var = 1
model.len_scale = 15
model.nugget = 0.1
print("new model:", model)
```

Out:

```
old model: Gau(dim=2, var=2.0, len_scale=10.0, nugget=0.0)
new model: Gau(dim=3, var=1.0, len_scale=15.0, nugget=0.1)
```

Note:

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

Total running time of the script: (0 minutes 0.193 seconds)

Basic Methods

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

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

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

$$\gamma(r) = \sigma^2 \cdot (1 - \rho(r)) + n$$

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

$$C(r) = \sigma^2 \cdot \rho(r)$$

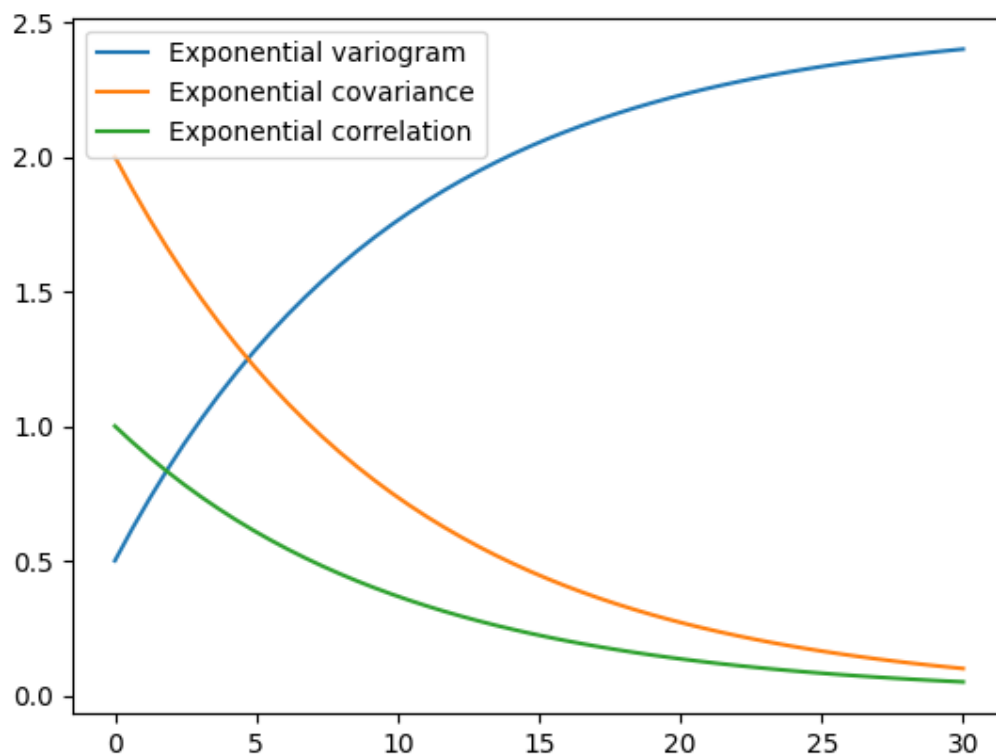
- `CovModel.correlation`: The (auto-)correlation (or normalized covariance) of the model given by

$$\rho(r)$$

- `CovModel.cor`: The normalized correlation taking a normalized range given by:

$$\text{cor}\left(\frac{r}{\ell}\right) = \rho(r)$$

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



```
import gstools as gs

model = gs.Exponential(dim=3, var=2.0, len_scale=10, nugget=0.5)
ax = model.plot("variogram")
model.plot("covariance", ax=ax)
model.plot("correlation", ax=ax)
```

Total running time of the script: (0 minutes 0.099 seconds)

Anisotropy and Rotation

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

```
import gstools as gs

model = gs.Gaussian(dim=3, var=2.0, len_scale=10, anis=0.5)
print(model.anis)
print(model.len_scale_vec)
```

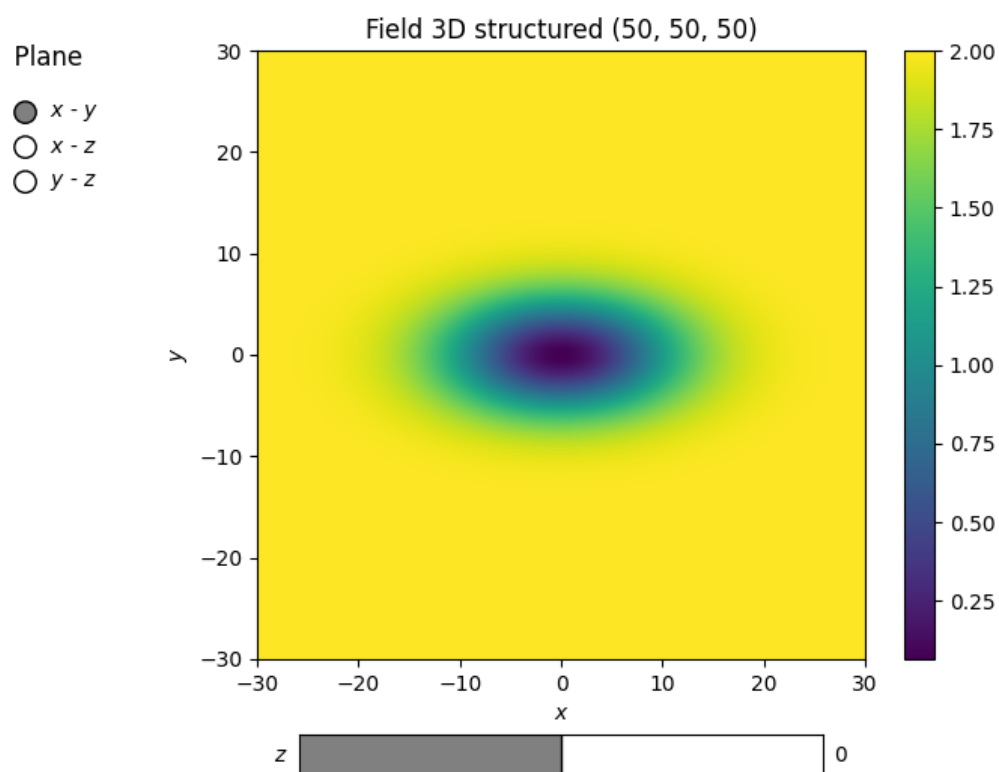
Out:

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

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

Alternatively you can provide a list of length-scales:

```
model = gs.Gaussian(dim=3, var=2.0, len_scale=[10, 5, 4])
model.plot("vario_spatial")
print("Anisotropy representations:")
print("Anis. ratios:", model.anis)
print("Main length scale", model.len_scale)
print("All length scales", model.len_scale_vec)
```



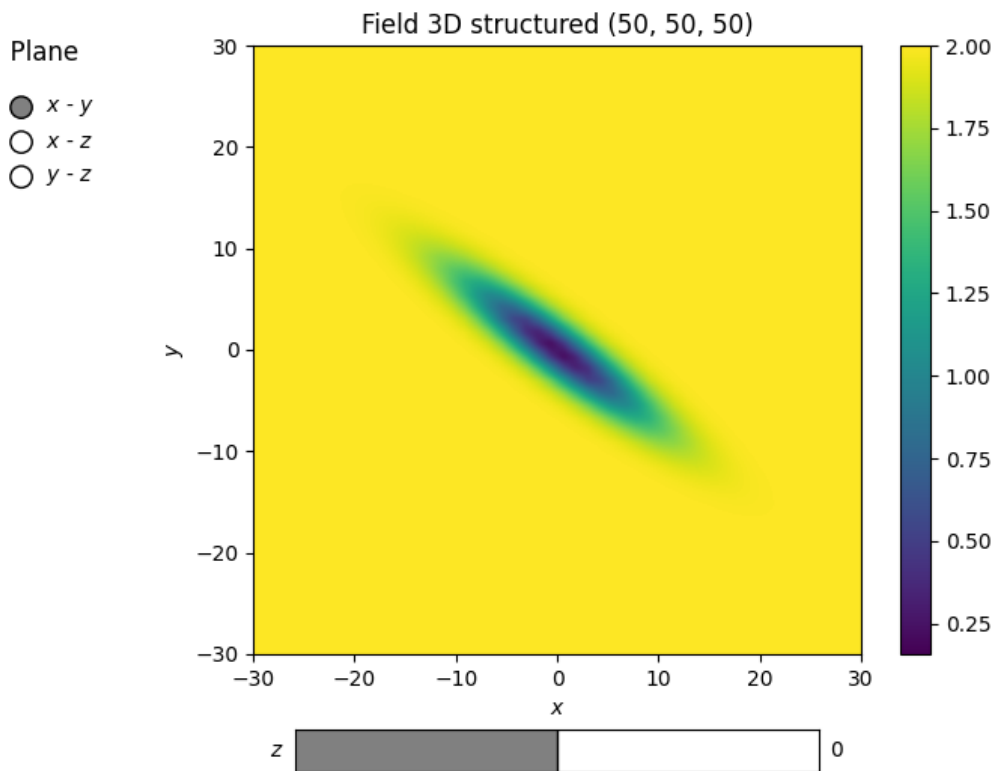
Out:

```
Anisotropy representations:
Anis. ratios: [0.5 0.4]
Main length scale 10.0
All length scales [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 = gs.Gaussian(dim=3, var=2.0, len_scale=[10, 2], angles=2.5)
model.plot("vario_spatial")
print("Rotation angles", model.angles)
```



Out:

```
Rotation angles [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](#))
- in nD: See the random field example about higher dimensions

Total running time of the script: (0 minutes 0.927 seconds)

Spectral methods

The spectrum of a covariance model is given by:

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

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

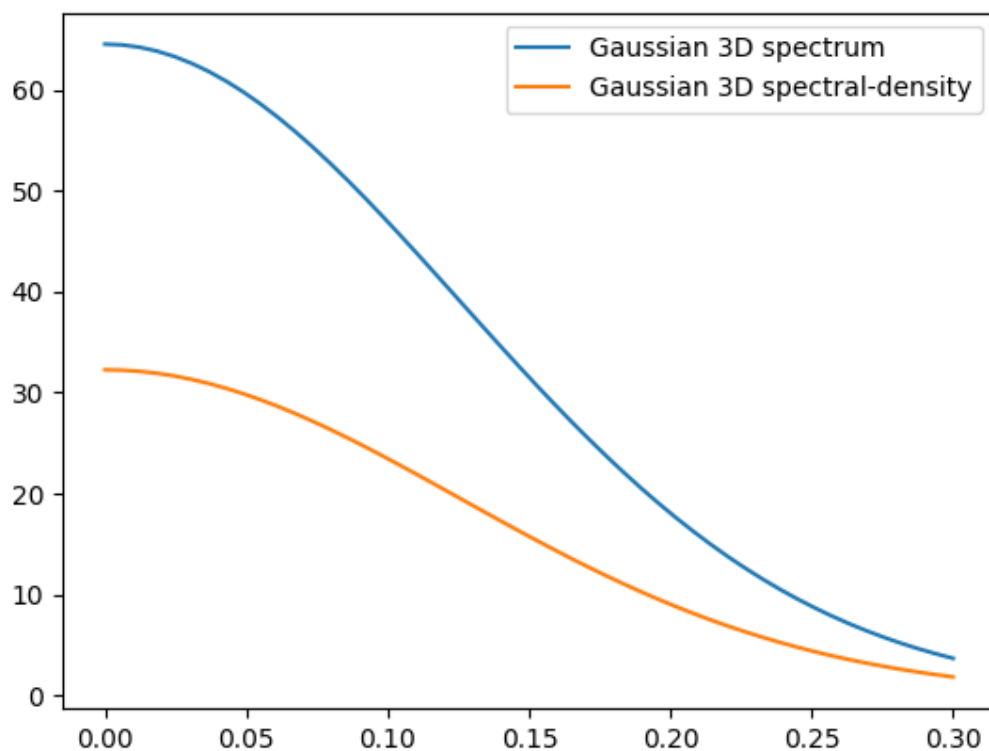
Depending on the spectrum, the spectral-density is defined by:

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

You can access these methods by:

```
import gstools as gs

model = gs.Gaussian(dim=3, var=2.0, len_scale=10)
ax = model.plot("spectrum")
model.plot("spectral_density", ax=ax)
```



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 [CovModel.spectral_rad_pdf](#) or [CovModel.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 [CovModel.has_cdf](#) and [CovModel.has_ppf](#) will check for that.

Total running time of the script: (0 minutes 0.095 seconds)

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:

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

You can access it by:

```
import gstools as gs

model = gs.Stable(dim=3, var=2.0, len_scale=10)
print("Main integral scale:", model.integral_scale)
print("All integral scales:", model.integral_scale_vec)
```

Out:

```
Main integral scale: 9.027452929509336
All integral scales: [9.02745293 9.02745293 9.02745293]
```

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

```
model = gs.Stable(dim=3, var=2.0, integral_scale=[10, 4, 2])
print("Anisotropy ratios:", model.anis)
print("Main length scale:", model.len_scale)
print("All length scales:", model.len_scale_vec)
print("Main integral scale:", model.integral_scale)
print("All integral scales:", model.integral_scale_vec)
```

Out:

```
Anisotropy ratios: [0.4 0.2]
Main length scale: 11.077321674324725
All length scales: [11.07732167 4.43092867 2.21546433]
Main integral scale: 10.0
All integral scales: [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 = gs.Stable(dim=3, var=2.0, len_scale=10)
per_scale = model.percentile_scale(0.9)
int_scale = model.integral_scale
len_scale = model.len_scale
print("90% Percentile scale:", per_scale)
print("Integral scale:", int_scale)
print("Length scale:", len_scale)
```

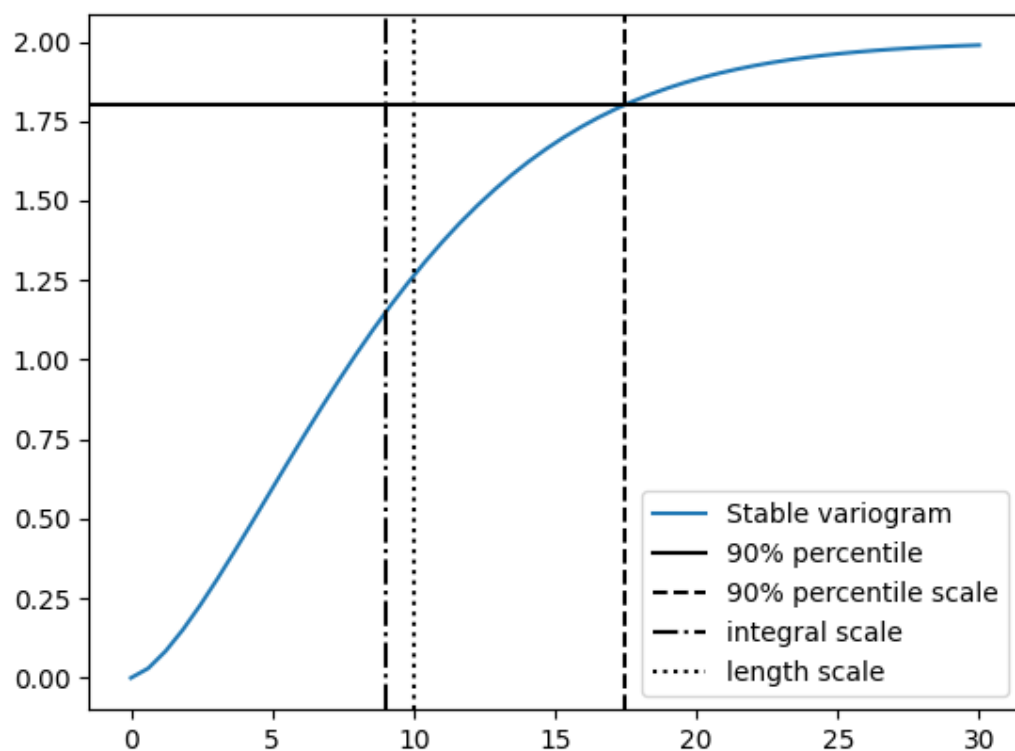
Out:

```
90% Percentile scale: 17.437215135964117
Integral scale: 9.027452929509336
Length scale: 10.0
```

Note: The nugget is neglected by the percentile scale.

Comparison

```
ax = model.plot()
ax.axhline(1.8, color="k", label=r"90% percentile")
ax.axvline(per_scale, color="k", linestyle="--", label=r"90% percentile scale")
ax.axvline(int_scale, color="k", linestyle="-.", label=r"integral scale")
ax.axvline(len_scale, color="k", linestyle=":", label=r"length scale")
ax.legend()
```



Total running time of the script: (0 minutes 0.162 seconds)

Additional Parameters

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

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

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

```
import numpy as np
import gstools as gs

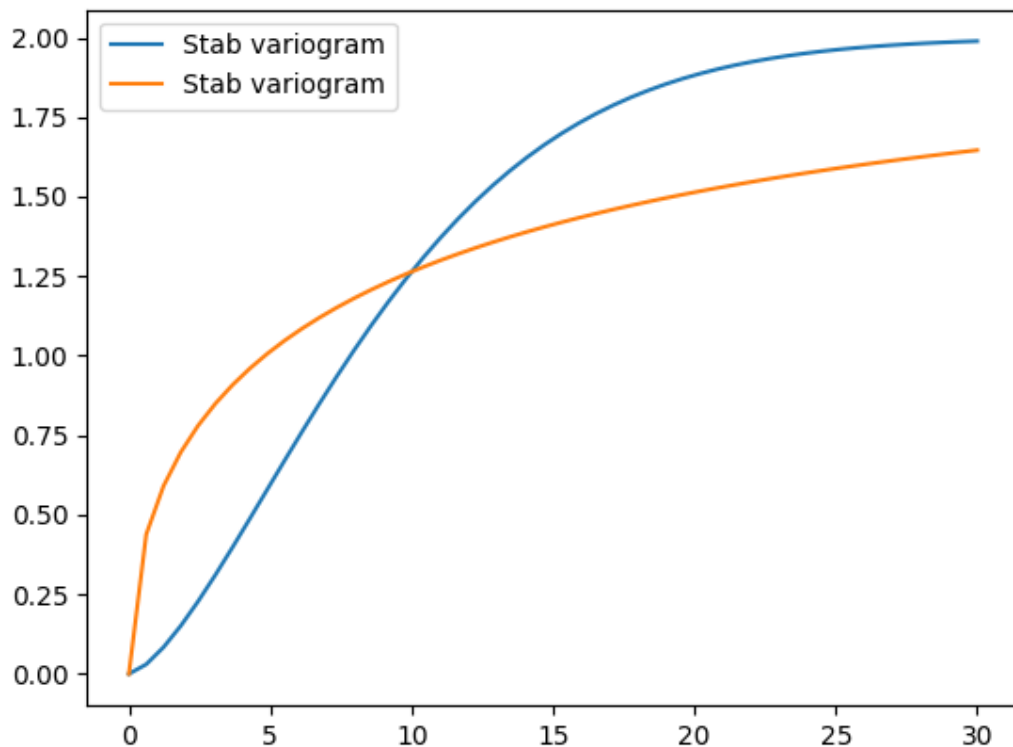
class Stab(gs.CovModel):
    def default_opt_arg(self):
        return {"alpha": 1.5}

    def cor(self, h):
        return np.exp(-(h ** self.alpha))
```

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

Now we can instantiate this model by either setting `alpha` implicitly with the default value or explicitly:

```
model1 = Stab(dim=2, var=2.0, len_scale=10)
model2 = Stab(dim=2, var=2.0, len_scale=10, alpha=0.5)
ax = model1.plot()
model2.plot(ax=ax)
```



Apparently, the parameter `alpha` controls the slope of the variogram and consequently the roughness of a generated random field.

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

Total running time of the script: (0 minutes 0.107 seconds)

Fitting variogram data

The model class comes with a routine to fit the model-parameters to given variogram data. In the following we will use the self defined stable model from a previous example.

```
import numpy as np
import gstools as gs

class Stab(gs.CovModel):
    def default_opt_arg(self):
        return {"alpha": 1.5}

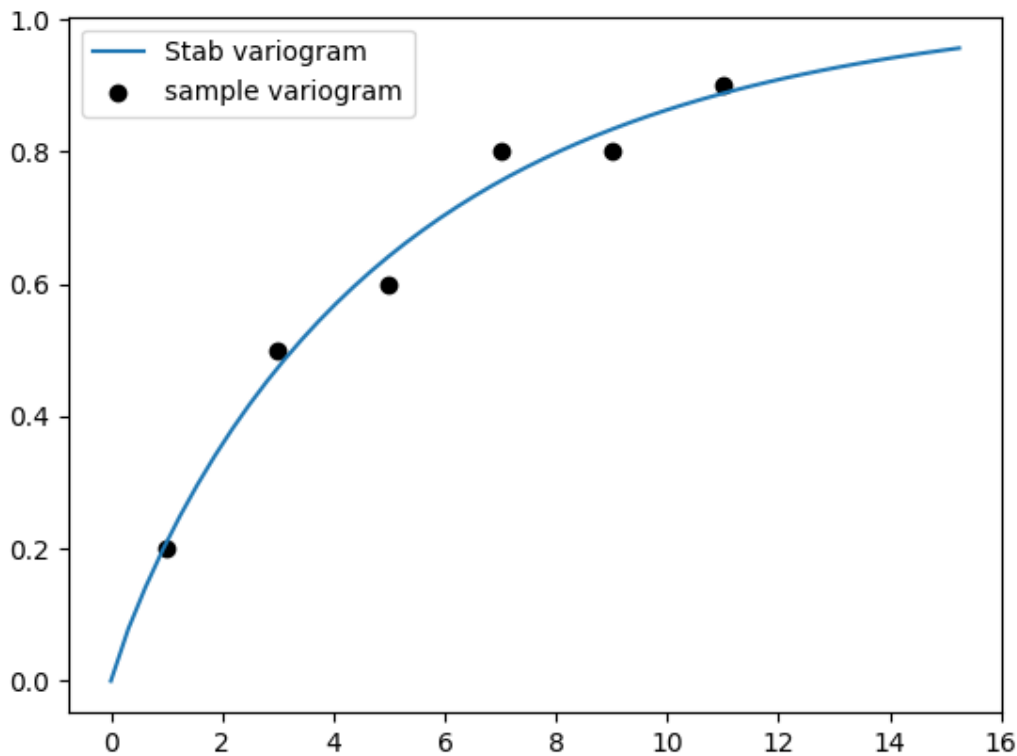
    def cor(self, h):
        return np.exp(-(h ** self.alpha))

# Exemplary variogram data (e.g. estimated from field observations)
bins = [1.0, 3.0, 5.0, 7.0, 9.0, 11.0]
est_vario = [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])
results, pcov = model.fit_variogram(bins, est_vario, nugget=False)
print("Results:", results)
```

Out:

```
Results: {'var': 1.0245739533522298, 'len_scale': 5.081591843438552, 'nugget': 0.0,
↪ 'alpha': 0.9067041165465699}
```

```
ax = model.plot()
ax.scatter(bins, est_vario, color="k", label="sample variogram")
ax.legend()
```



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.optimize.curve_fit`.

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

<code>CovModel.default_opt_arg_bounds()</code>	Provide default boundaries for optional arguments.
<code>CovModel.default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>CovModel.set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>CovModel.check_arg_bounds()</code>	Check arguments to be within their 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:

<code>CovModel.var_bounds</code>	Bounds for the variance.
<code>CovModel.len_scale_bounds</code>	Bounds for the length scale.
<code>CovModel.nugget_bounds</code>	Bounds for the nugget.
<code>CovModel.opt_arg_bounds</code>	Bounds for the optional arguments.

continues on next page

Table 4 – continued from previous page

<code>CovModel.arg_bounds</code>	Bounds for all parameters.
----------------------------------	----------------------------

Total running time of the script: (0 minutes 0.127 seconds)

2.3 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.

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

Examples

Fit Variogram

```
import numpy as np
import gstools as gs
```

Generate a synthetic field with an exponential model.

```
x = np.random.RandomState(19970221).rand(1000) * 100.0
y = np.random.RandomState(20011012).rand(1000) * 100.0
model = gs.Exponential(dim=2, var=2, len_scale=8)
srf = gs.SRF(model, mean=0, seed=19970221)
field = srf((x, y))
```

Estimate the variogram of the field with 40 bins.

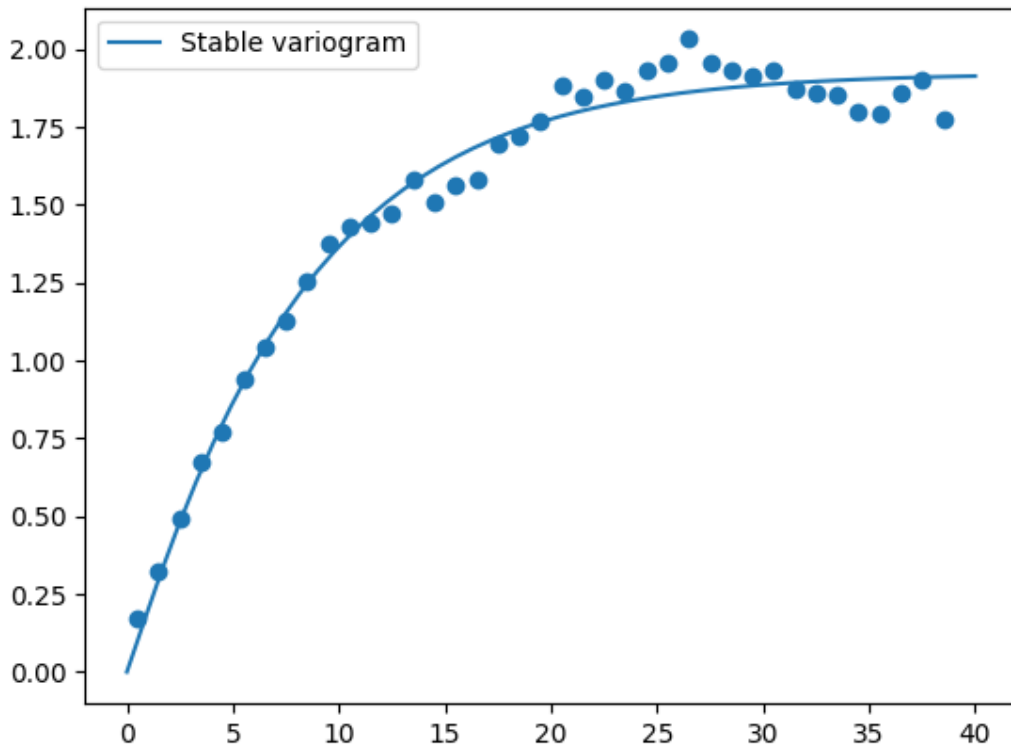
```
bins = np.arange(40)
bin_center, gamma = gs.vario_estimate((x, y), field, bins)
```

Fit the variogram with a stable model (no nugget fitted).

```
fit_model = gs.Stable(dim=2)
fit_model.fit_variogram(bin_center, gamma, nugget=False)
```

Plot the fitting result.

```
ax = fit_model.plot(x_max=40)
ax.scatter(bin_center, gamma)
print(fit_model)
```



Out:

```
Stable(dim=2, var=1.92, len_scale=8.15, nugget=0.0, alpha=1.05)
```

Total running time of the script: (0 minutes 0.493 seconds)

Finding the best fitting variogram model

```
import numpy as np
import gstools as gs
from matplotlib import pyplot as plt
```

Generate a synthetic field with an exponential model.

```
x = np.random.RandomState(19970221).rand(1000) * 100.0
y = np.random.RandomState(20011012).rand(1000) * 100.0
model = gs.Exponential(dim=2, var=2, len_scale=8)
srf = gs.SRF(model, mean=0, seed=19970221)
field = srf((x, y))
```

Estimate the variogram of the field with 40 bins and plot the result.

```
bins = np.arange(40)
bin_center, gamma = gs.vario_estimate((x, y), field, bins)
```

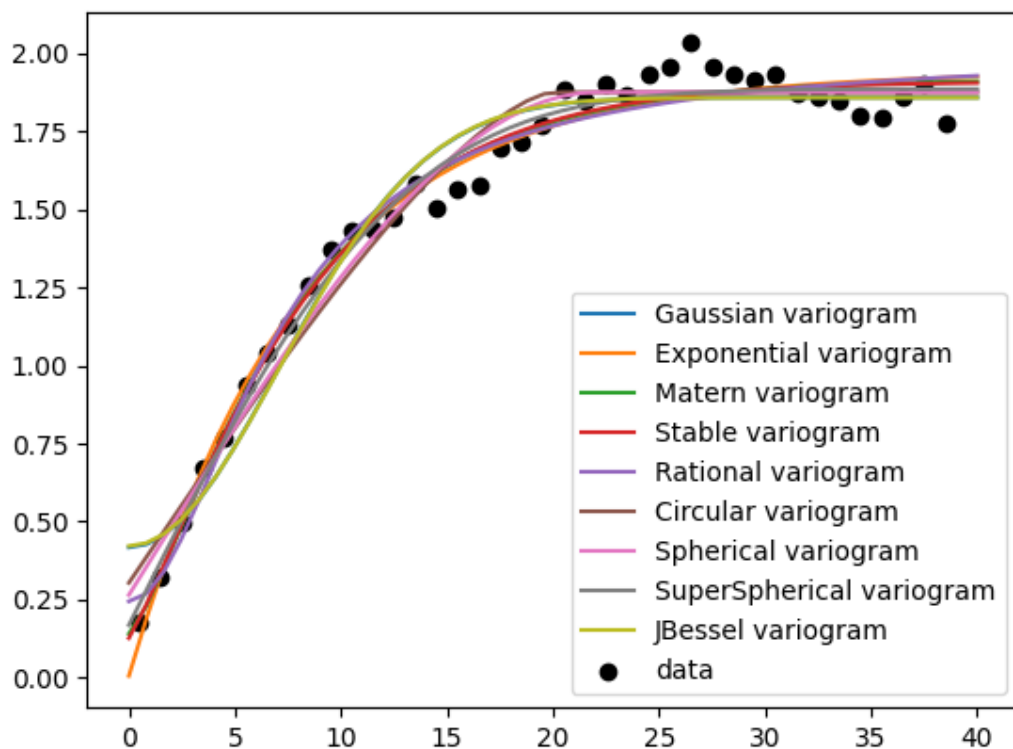
Define a set of models to test.

```
models = {
    "Gaussian": gs.Gaussian,
    "Exponential": gs.Exponential,
    "Matern": gs.Matern,
    "Stable": gs.Stable,
    "Rational": gs.Rational,
    "Circular": gs.Circular,
    "Spherical": gs.Spherical,
    "SuperSpherical": gs.SuperSpherical,
    "JBessel": gs.JBessel,
}
scores = {}
```

Iterate over all models, fit their variogram and calculate the r2 score.

```
# plot the estimated variogram
plt.scatter(bin_center, gamma, color="k", label="data")
ax = plt.gca()

# fit all models to the estimated variogram
for model in models:
    fit_model = models[model](dim=2)
    para, pcov, r2 = fit_model.fit_variogram(bin_center, gamma, return_r2=True)
    fit_model.plot(x_max=40, ax=ax)
    scores[model] = r2
```



Create a ranking based on the score and determine the best models

```

ranking = [
    (k, v)
    for k, v in sorted(scores.items(), key=lambda item: item[1], reverse=True)
]
print("RANKING")
for i, (model, score) in enumerate(ranking, 1):
    print(i, model, score)

plt.show()

```

Out:

```

RANKING
1 Stable 0.9821836193000343
2 Matern 0.9817602690672453
3 SuperSpherical 0.9814051618626767
4 Exponential 0.980407470735337
5 Rational 0.9771067080321653
6 Spherical 0.9733371670897375
7 Circular 0.9672526098783125
8 Gaussian 0.9592818084007272
9 JBessel 0.9583119496108051

```

Total running time of the script: (0 minutes 1.007 seconds)

Multi-field variogram estimation

In this example, we demonstrate how to estimate a variogram from multiple fields on the same point-set that should have the same statistical properties.

```

import numpy as np
import gstools as gs
import matplotlib.pyplot as plt

x = np.random.RandomState(19970221).rand(1000) * 100.0
y = np.random.RandomState(20011012).rand(1000) * 100.0
model = gs.Exponential(dim=2, var=2, len_scale=8)
srf = gs.SRF(model, mean=0)

```

Generate two synthetic fields with an exponential model.

```

field1 = srf((x, y), seed=19970221)
field2 = srf((x, y), seed=20011012)
fields = [field1, field2]

```

Now we estimate the variograms for both fields individually and then again simultaneously with only one call.

```

bins = np.arange(40)
bin_center, gamma1 = gs.vario_estimate((x, y), field1, bins)
bin_center, gamma2 = gs.vario_estimate((x, y), field2, bins)
bin_center, gamma = gs.vario_estimate((x, y), fields, bins)

```

Now we demonstrate that the mean variogram from both fields coincides with the joined estimated one.

```

plt.plot(bin_center, gamma1, label="field 1")
plt.plot(bin_center, gamma2, label="field 2")

```

(continues on next page)

(continued from previous page)

```
plt.plot(bin_center, gamma, label="joined fields")
plt.plot(bin_center, 0.5 * (gamma1 + gamma2), ":", label="field 1+2 mean")
plt.legend()
plt.show()
```



Total running time of the script: (0 minutes 1.112 seconds)

Directional variogram estimation and fitting in 2D

In this example, we demonstrate how to estimate a directional variogram by setting the direction angles in 2D.

Afterwards we will fit a model to this estimated variogram and show the result.

```
import numpy as np
import gstools as gs
from matplotlib import pyplot as plt
```

Generating synthetic field with anisotropy and a rotation of 22.5 degree.

```
angle = np.pi / 8
model = gs.Exponential(dim=2, len_scale=[10, 5], angles=angle)
x = y = range(101)
srf = gs.SRF(model, seed=123456)
field = srf((x, y), mesh_type="structured")
```

Now we are going to estimate a directional variogram with an angular tolerance of 11.25 degree and a bandwidth of 8.


```

bins = range(0, 40, 2)
bin_center, dir_vario, counts = gs.vario_estimate(
    *((x, y), field, bins),
    direction=gs.rotated_main_axes(dim=2, angles=angle),
    angles_tol=np.pi / 16,
    bandwidth=8,
    mesh_type="structured",
    return_counts=True,
)

```

Afterwards we can use the estimated variogram to fit a model to it:

```

print("Original:")
print(model)
model.fit_variogram(bin_center, dir_vario)
print("Fitted:")
print(model)

```

Out:

```

Original:
Exponential(dim=2, var=1.0, len_scale=10.0, nugget=0.0, anis=[0.5], angles=[0.393])
Fitted:
Exponential(dim=2, var=0.942, len_scale=9.14, nugget=1.1e-17, anis=[0.529], angles=[0.
↪ 393])

```

Plotting.

```

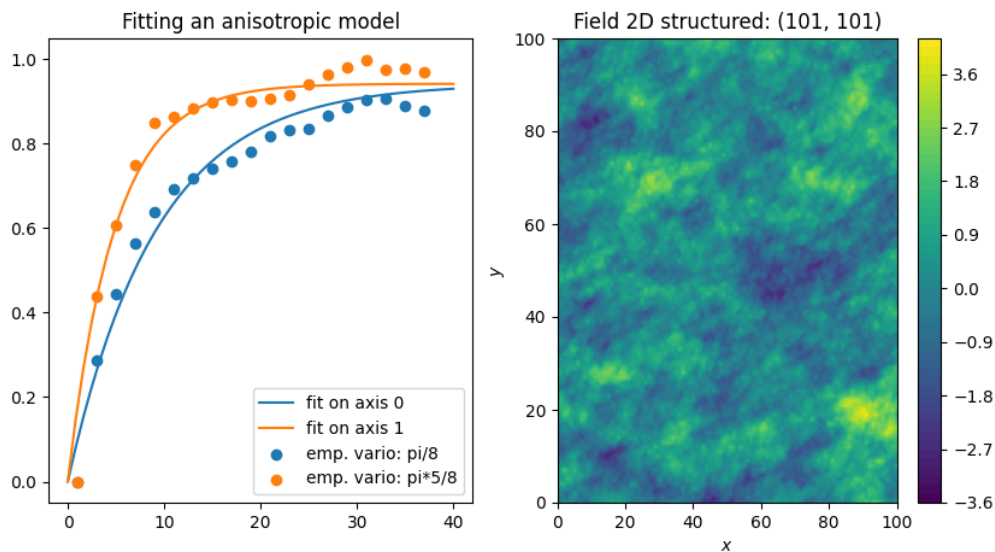
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=[10, 5])

ax1.scatter(bin_center, dir_vario[0], label="emp. vario: pi/8")
ax1.scatter(bin_center, dir_vario[1], label="emp. vario: pi*5/8")
ax1.legend(loc="lower right")

model.plot("vario_axis", axis=0, ax=ax1, x_max=40, label="fit on axis 0")
model.plot("vario_axis", axis=1, ax=ax1, x_max=40, label="fit on axis 1")
ax1.set_title("Fitting an anisotropic model")

srf.plot(ax=ax2)
plt.show()

```



Without fitting a model, we see that the correlation length in the main direction is greater than the transversal one.

Total running time of the script: (0 minutes 8.308 seconds)

Directional variogram estimation and fitting in 3D

In this example, we demonstrate how to estimate a directional variogram by setting the estimation directions in 3D. Afterwards we will fit a model to this estimated variogram and show the result.

```
import numpy as np
import gstools as gs
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Generating synthetic field with anisotropy and rotation by Tait-Bryan angles.

```
dim = 3
# rotation around z, y, x
angles = [np.deg2rad(90), np.deg2rad(45), np.deg2rad(22.5)]
model = gs.Gaussian(dim=3, len_scale=[16, 8, 4], angles=angles)
x = y = z = range(50)
pos = (x, y, z)
srf = gs.SRF(model, seed=1001)
field = srf.structured(pos)
```

Here we generate the axes of the rotated coordinate system to get an impression what the rotation angles do.

```
# All 3 axes of the rotated coordinate-system
main_axes = gs.rotated_main_axes(dim, angles)
axis1, axis2, axis3 = main_axes
```

Now we estimate the variogram along the main axes. When the main axes are unknown, one would need to sample multiple directions and look for the one with the longest correlation length (flattest gradient). Then check the transversal directions and so on.

```
bin_center, dir_vario, counts = gs.vario_estimate(
    pos,
    field,
```

(continues on next page)

(continued from previous page)

```

direction=main_axes,
bandwidth=10,
sampling_size=2000,
sampling_seed=1001,
mesh_type="structured",
return_counts=True,
)

```

Afterwards we can use the estimated variogram to fit a model to it. Note, that the rotation angles need to be set beforehand.

```

print("Original:")
print(model)
model.fit_variogram(bin_center, dir_vario)
print("Fitted:")
print(model)

```

Out:

```

Original:
Gaussian(dim=3, var=1.0, len_scale=16.0, nugget=0.0, anis=[0.5, 0.25], angles=[1.57,
↪0.785, 0.393])
Fitted:
Gaussian(dim=3, var=0.972, len_scale=13.0, nugget=0.0138, anis=[0.542, 0.281],
↪angles=[1.57, 0.785, 0.393])

```

Plotting main axes and the fitted directional variogram.

```

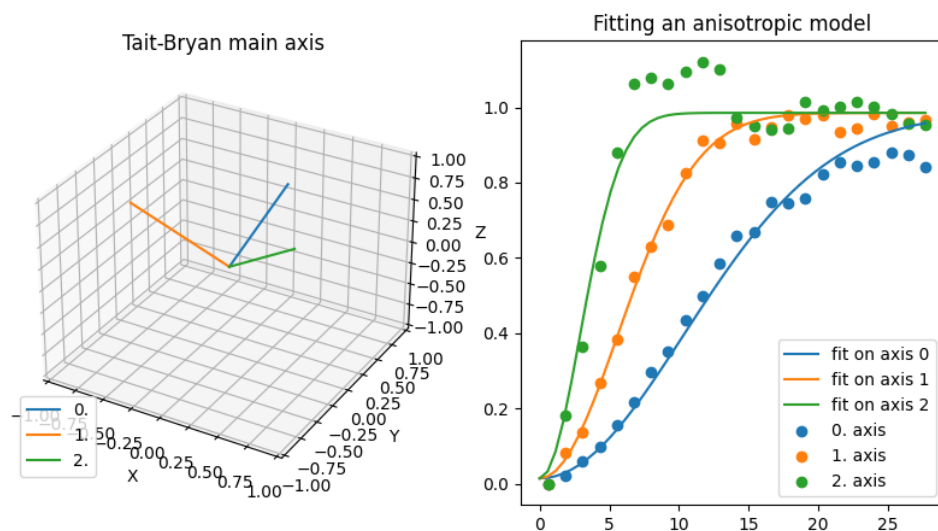
fig = plt.figure(figsize=[10, 5])
ax1 = fig.add_subplot(121, projection=Axes3D.name)
ax2 = fig.add_subplot(122)

ax1.plot([0, axis1[0]], [0, axis1[1]], [0, axis1[2]], label="0.")
ax1.plot([0, axis2[0]], [0, axis2[1]], [0, axis2[2]], label="1.")
ax1.plot([0, axis3[0]], [0, axis3[1]], [0, axis3[2]], label="2.")
ax1.set_xlim(-1, 1)
ax1.set_ylim(-1, 1)
ax1.set_zlim(-1, 1)
ax1.set_xlabel("X")
ax1.set_ylabel("Y")
ax1.set_zlabel("Z")
ax1.set_title("Tait-Bryan main axis")
ax1.legend(loc="lower left")

x_max = max(bin_center)
ax2.scatter(bin_center, dir_vario[0], label="0. axis")
ax2.scatter(bin_center, dir_vario[1], label="1. axis")
ax2.scatter(bin_center, dir_vario[2], label="2. axis")
model.plot("vario_axis", axis=0, ax=ax2, x_max=x_max, label="fit on axis 0")
model.plot("vario_axis", axis=1, ax=ax2, x_max=x_max, label="fit on axis 1")
model.plot("vario_axis", axis=2, ax=ax2, x_max=x_max, label="fit on axis 2")
ax2.set_title("Fitting an anisotropic model")
ax2.legend()

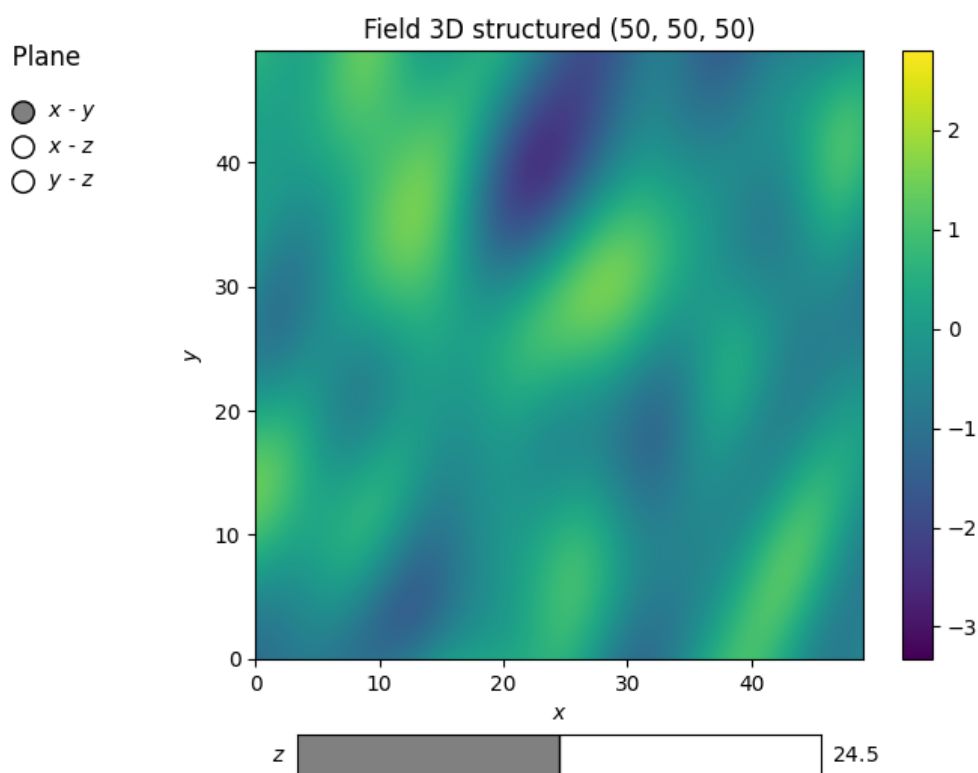
plt.show()

```



Also, let's have a look at the field.

```
srf.plot()
```



Total running time of the script: (0 minutes 8.532 seconds)

Fit Variogram with automatic binning

```
import numpy as np
import gstools as gs
```

Generate a synthetic field with an exponential model.

```
x = np.random.RandomState(19970221).rand(1000) * 100.0
y = np.random.RandomState(20011012).rand(1000) * 100.0
model = gs.Exponential(dim=2, var=2, len_scale=8)
srf = gs.SRF(model, mean=0, seed=19970221)
field = srf((x, y))
print(field.var())
```

Out:

```
1.6791948750716688
```

Estimate the variogram of the field with automatic binning.

```
bin_center, gamma = gs.vario_estimate((x, y), field)
print("estimated bin number:", len(bin_center))
print("maximal bin distance:", max(bin_center))
```

Out:

```
estimated bin number: 21
maximal bin distance: 45.88516574202333
```

Fit the variogram with a stable model (no nugget fitted).

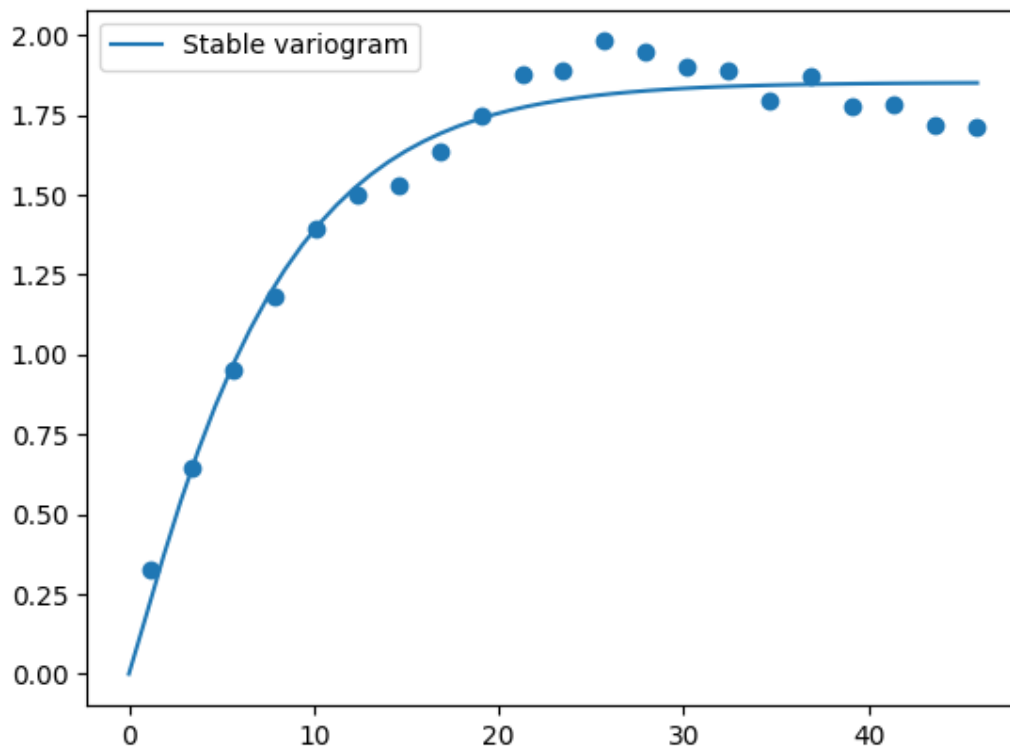
```
fit_model = gs.Stable(dim=2)
fit_model.fit_variogram(bin_center, gamma, nugget=False)
print(fit_model)
```

Out:

```
Stable(dim=2, var=1.85, len_scale=7.42, nugget=0.0, alpha=1.09)
```

Plot the fitting result.

```
ax = fit_model.plot(x_max=max(bin_center))
ax.scatter(bin_center, gamma)
```



Total running time of the script: (0 minutes 0.359 seconds)

Automatic binning with lat-lon data

In this example we demonstrate automatic binning for a tiny data set containing temperature records from Germany (See the detailed DWD example for more information on the data).

We use a data set from 20 meteo-stations choosen randomly.

```
import numpy as np
import gstools as gs

# lat, lon, temperature
data = np.array([
    [52.9336, 8.237, 15.7],
    [48.6159, 13.0506, 13.9],
    [52.4853, 7.9126, 15.1],
    [50.7446, 9.345, 17.0],
    [52.9437, 12.8518, 21.9],
    [53.8633, 8.1275, 11.9],
    [47.8342, 10.8667, 11.4],
    [51.0881, 12.9326, 17.2],
    [48.406, 11.3117, 12.9],
    [49.7273, 8.1164, 17.2],
    [49.4691, 11.8546, 13.4],
    [48.0197, 12.2925, 13.9],
    [50.4237, 7.4202, 18.1],
```

(continues on next page)

(continued from previous page)

```

    [53.0316, 13.9908, 21.3],
    [53.8412, 13.6846, 21.3],
    [54.6792, 13.4343, 17.4],
    [49.9694, 9.9114, 18.6],
    [51.3745, 11.292, 20.2],
    [47.8774, 11.3643, 12.7],
    [50.5908, 12.7139, 15.8],
]
)
pos = data.T[:2] # lat, lon
field = data.T[2] # temperature

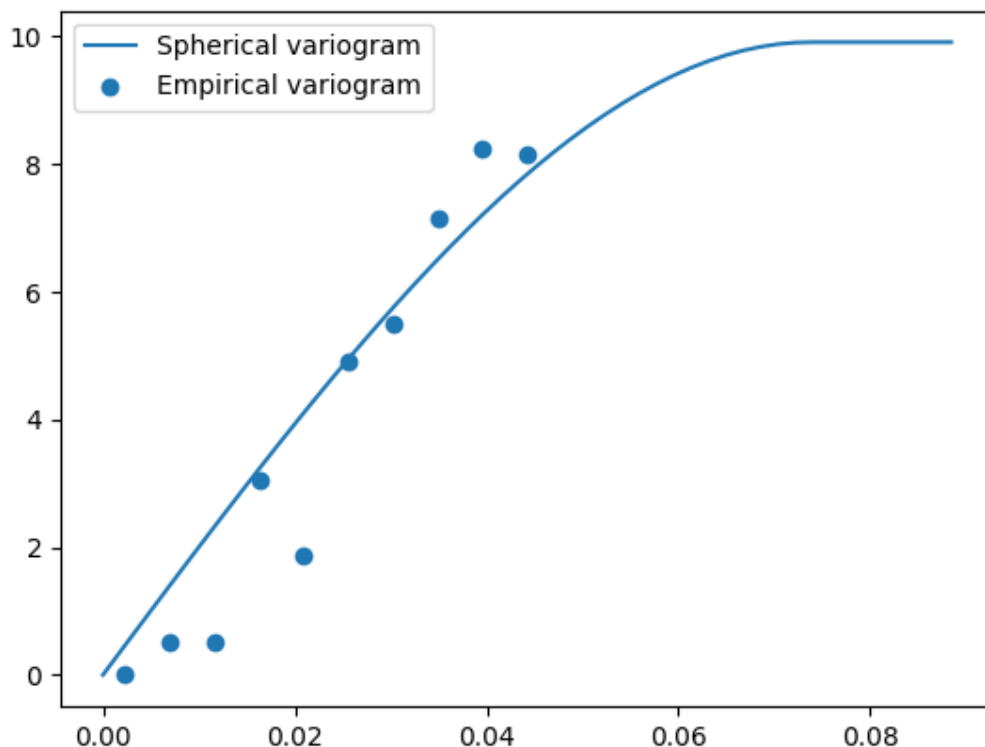
```

Since the overall range of these meteo-stations is too low, we can use the data-variance as additional information during the fit of the variogram.

```

emp_v = gs.vario_estimate(pos, field, latlon=True)
sph = gs.Spherical(latlon=True, rescale=gs.EARTH_RADIUS)
sph.fit_variogram(*emp_v, sill=np.var(field))
ax = sph.plot(x_max=2 * np.max(emp_v[0]))
ax.scatter(*emp_v, label="Empirical variogram")
ax.legend()
print(sph)

```



Out:

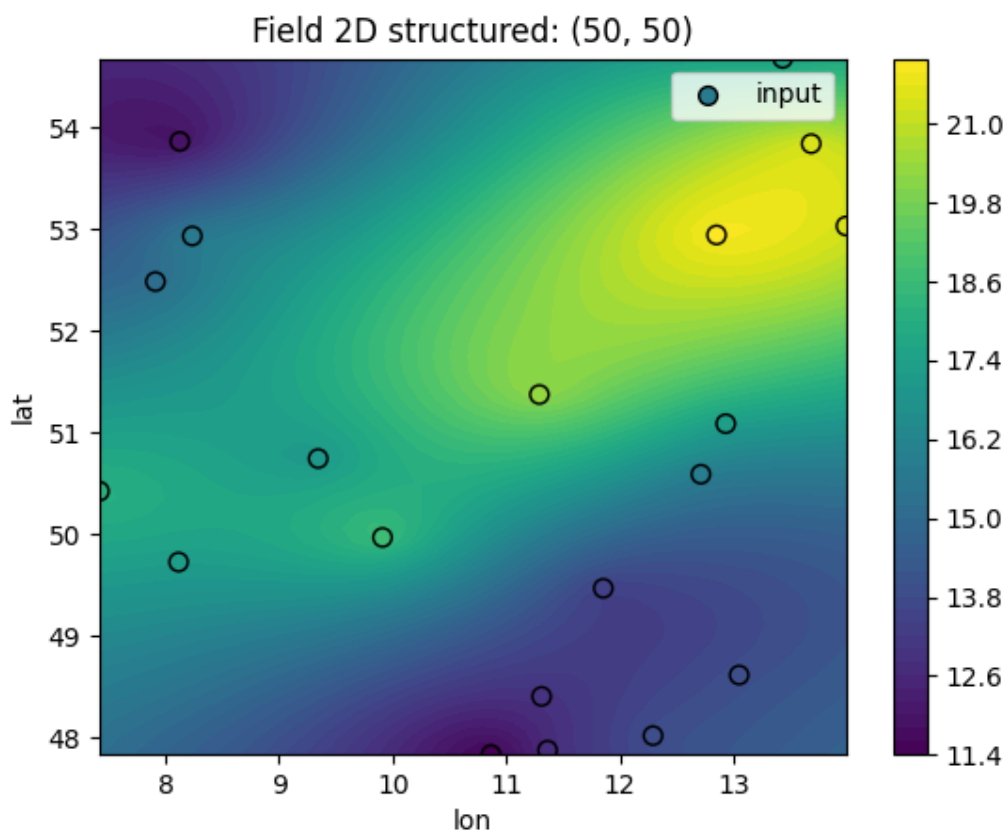
```
Spherical(latlon=True, var=9.91, len_scale=4.7e+02, nugget=1.78e-15, rescale=6.37e+03)
```

As we can see, the variogram fitting was successful and providing the data variance helped finding the right length-

scale.

Now, we'll use this covariance model to interpolate the given data with ordinary kriging.

```
# enclosing box for data points
grid_lat = np.linspace(np.min(pos[0]), np.max(pos[0]))
grid_lon = np.linspace(np.min(pos[1]), np.max(pos[1]))
# ordinary kriging
krige = gs.krige.Ordinary(sph, pos, field)
krige((grid_lat, grid_lon), mesh_type="structured")
ax = krige.plot()
# plotting lat on y-axis and lon on x-axis
ax.scatter(pos[1], pos[0], 50, c=field, edgecolors="k", label="input")
ax.legend()
```

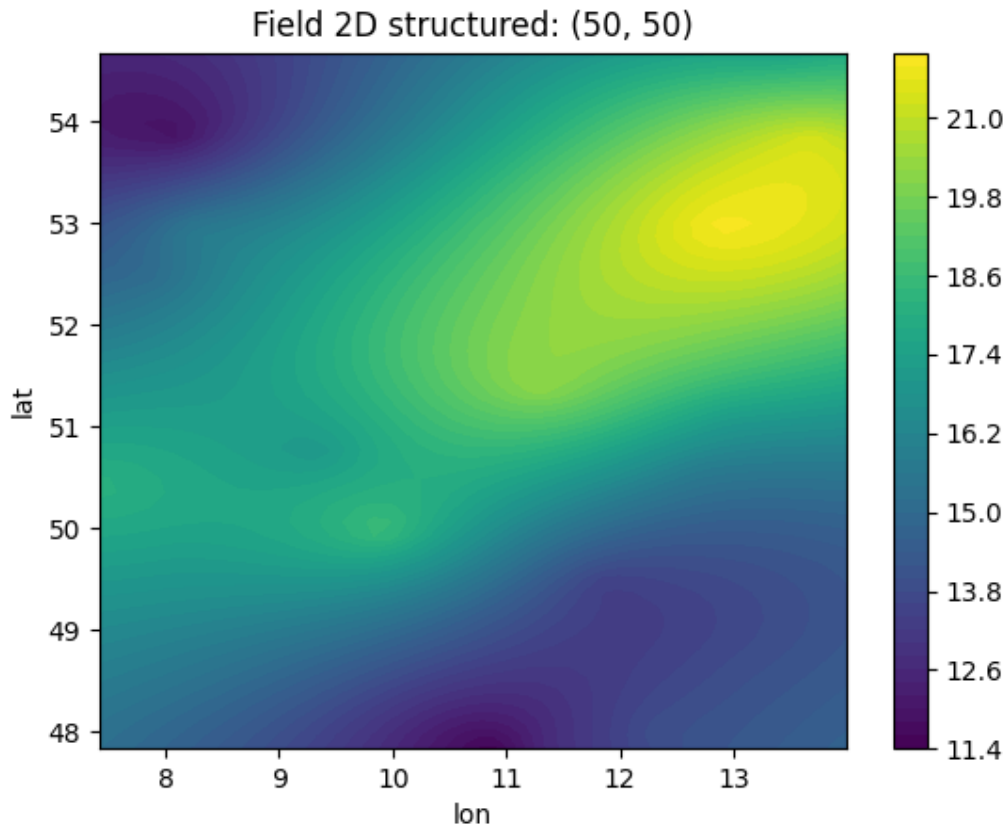


Looks good, doesn't it?

This workflow is also implemented in the [Kriging](#) class, by setting `fit_variogram=True`. Then the whole procedure shortens:

```
krige = gs.krige.Ordinary(sph, pos, field, fit_variogram=True)
krige.structured((grid_lat, grid_lon))

# plot the result
krige.plot()
# show the fitting results
print(krige.model)
```

Out:

```
Spherical(latlon=True, var=9.91, len_scale=4.7e+02, nugget=1.78e-15, rescale=6.37e+03)
```

This example shows, that setting up variogram estimation and kriging routines is straight forward with GSTools!

Total running time of the script: (0 minutes 0.575 seconds)

2.4 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.

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:

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

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.

Examples

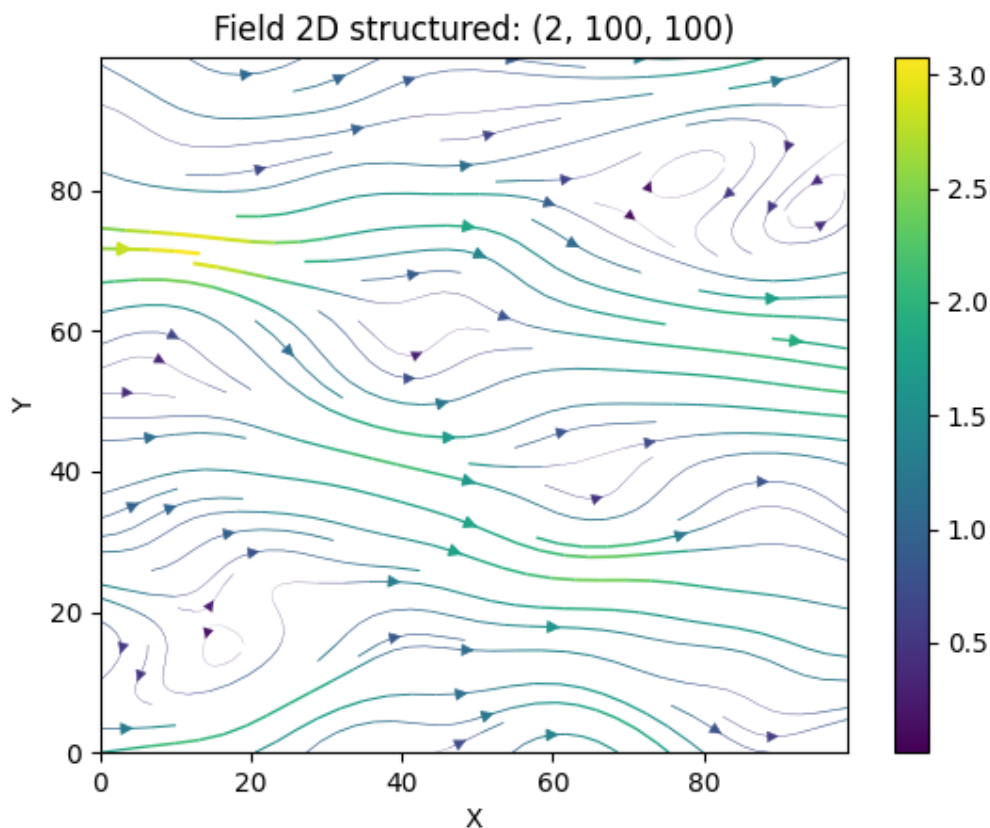
Generating a Random 2D Vector Field

As a first example we are going to generate a 2d vector field with a Gaussian covariance model on a structured grid:

```
import numpy as np
import gstools as gs

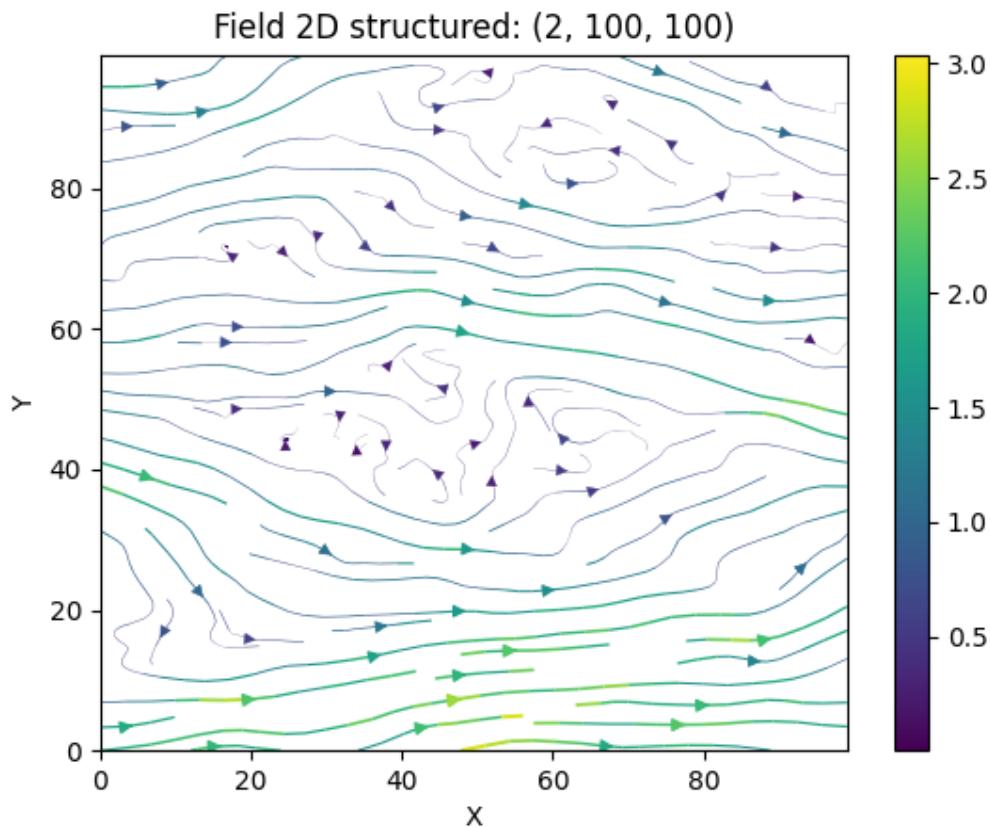
# the grid
x = np.arange(100)
y = np.arange(100)

# a smooth Gaussian covariance model
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.SRF(model, generator="VectorField", seed=19841203)
srf((x, y), mesh_type="structured")
srf.plot()
```



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

```
# a rougher exponential covariance model
model2 = gs.Exponential(dim=2, var=1, len_scale=10)
srf.model = model2
srf((x, y), mesh_type="structured", seed=19841203)
srf.plot()
```



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.

Total running time of the script: (0 minutes 2.953 seconds)

Generating a Random 3D Vector Field

In this example we are going to generate a random 3D vector field with a Gaussian covariance model. The mesh on which we generate the field will be externally defined and it will be generated by PyVista.

```
import gstools as gs
import pyvista as pv

# mainly for setting a white background
pv.set_plot_theme("document")
```

create a uniform grid with PyVista

```
dim, spacing, origin = (40, 30, 10), (1, 1, 1), (-10, 0, 0)
mesh = pv.UniformGrid(dim, spacing, origin)
```

create an incompressible random 3d velocity field on the given mesh with added mean velocity in x-direction

```
model = gs.Gaussian(dim=3, var=3, len_scale=1.5)
srf = gs.SRF(model, mean=(0.5, 0, 0), generator="VectorField", seed=198412031)
srf.mesh(mesh, points="points", name="Velocity")
```

Now, we can do the plotting

```
streamlines = mesh.streamlines(
    "Velocity",
    terminal_speed=0.0,
    n_points=800,
    source_radius=2.5,
)

# set a fancy camera position
cpos = [(25, 23, 17), (0, 10, 0), (0, 0, 1)]

p = pv.Plotter()
# adding an outline might help navigating in 3D space
# p.add_mesh(mesh.outline(), color="k")
p.add_mesh(
    streamlines.tube(radius=0.005),
    show_scalar_bar=False,
    diffuse=0.5,
    ambient=0.5,
)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/gstools/envs/v1.3.0/lib/python3.7/
↳ site-packages/pyvista/plotting/plotting.py:97: UserWarning:
This system does not appear to be running an xserver.
PyVista will likely segfault when rendering.

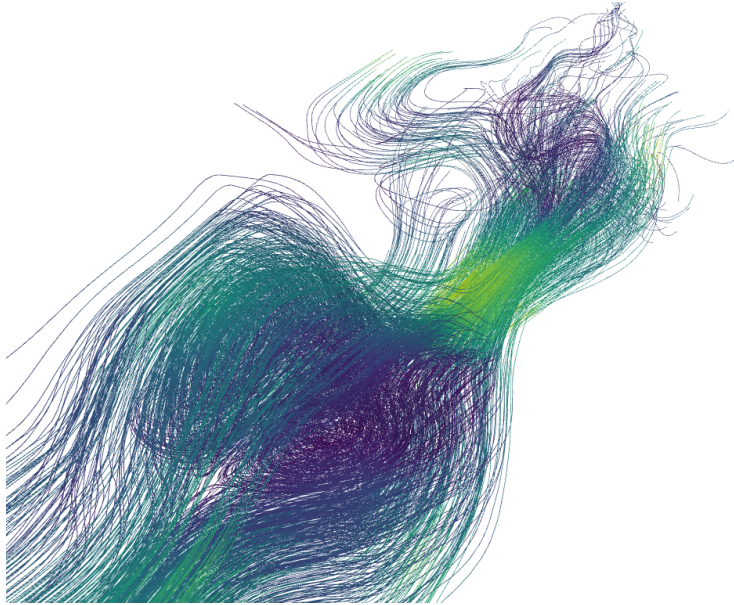
Try starting a virtual frame buffer with xvfb, or using
``pyvista.start_xvfb()``

warnings.warn('\n'
```

Note: PyVista is not working on readthedocs, but you can try it out yourself by uncommenting the following line of code.

```
# p.show(cpos=cpos)
```

The result should look like this:



Total running time of the script: (0 minutes 4.998 seconds)

2.5 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.

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

The different kriging approaches provide different ways of calculating W .

The `Krige` class provides everything in one place and you can switch on/off the features you want:

- *unbiased*: the weights have to sum up to 1. If true, this results in *Ordinary* kriging, where the mean is estimated, otherwise it will result in *Simple* kriging, where the mean has to be given.
- *drift_functions*: you can give a polynomial order or a list of self defined functions representing the internal drift of the given values. This drift will be fitted internally during the kriging interpolation. This results in *Universal* kriging.
- *ext_drift*: You can also give an external drift per point to the routine. In contrast to the internal drift, that is evaluated at the desired points with the given functions, the external drift has to be given for each point from an “external” source. This results in *ExtDrift* kriging.
- *trend, mean, normalizer*: These are used to pre- and post-process data. If you already have fitted a trend model that is provided as a callable function, you can give it to the kriging routine. Normalizer are power-transformations to gain normality. *mean* behaves similar to *trend* but is applied at another position:
 1. conditioning data is de-trended (subtracting trend)
 2. detrended conditioning data is then normalized (in order to follow a normal distribution)
 3. normalized conditioning data is set to zero mean (subtracting mean)

Cosequently, when there is no normalizer given, trend and mean are the same thing and only one should be used. *Detrended* kriging is a shortcut to provide only a trend and simple kriging with normal data.

- *exact* and *cond_err*: To incorporate the nugget effect and/or measurement errors, one can set *exact* to *False* and provide either individual measurement errors for each point or set the nugget as a constant measurement error everywhere.
- *pseudo_inv*: Sometimes the inversion of the kriging matrix can be numerically unstable. This occurs for examples in cases of redundant input values. In this case we provide a switch to use the pseudo-inverse of the matrix. Then redundant conditional values will automatically be averaged.

Note: All mentioned features can be combined within the *Krige* class. All other kriging classes are just shortcuts to this class with a limited list of input parameters.

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

```
import gstools as gs
# conditions
cond_pos = [...]
cond_val = [...]
model = gs.Gaussian(dim=1, var=0.5, len_scale=2)
krig = gs.krige.Simple(model, cond_pos=cond_pos, cond_val=cond_val, mean=1)
```

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.

Provided Kriging Methods

The following kriging methods are provided within the submodule *gstools.krige*.

<i>Krige</i> (model, cond_pos, cond_val[, ...])	A Swiss Army knife for kriging.
<i>Simple</i> (model, cond_pos, cond_val[, mean, ...])	Simple kriging.
<i>Ordinary</i> (model, cond_pos, cond_val[, ...])	Ordinary kriging.
<i>Universal</i> (model, cond_pos, cond_val, ...[, ...])	Universal kriging.
<i>ExtDrift</i> (model, cond_pos, cond_val, ext_drift)	External drift kriging (EDK).
<i>Detrended</i> (model, cond_pos, cond_val, trend)	Detrended simple kriging.

Examples

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

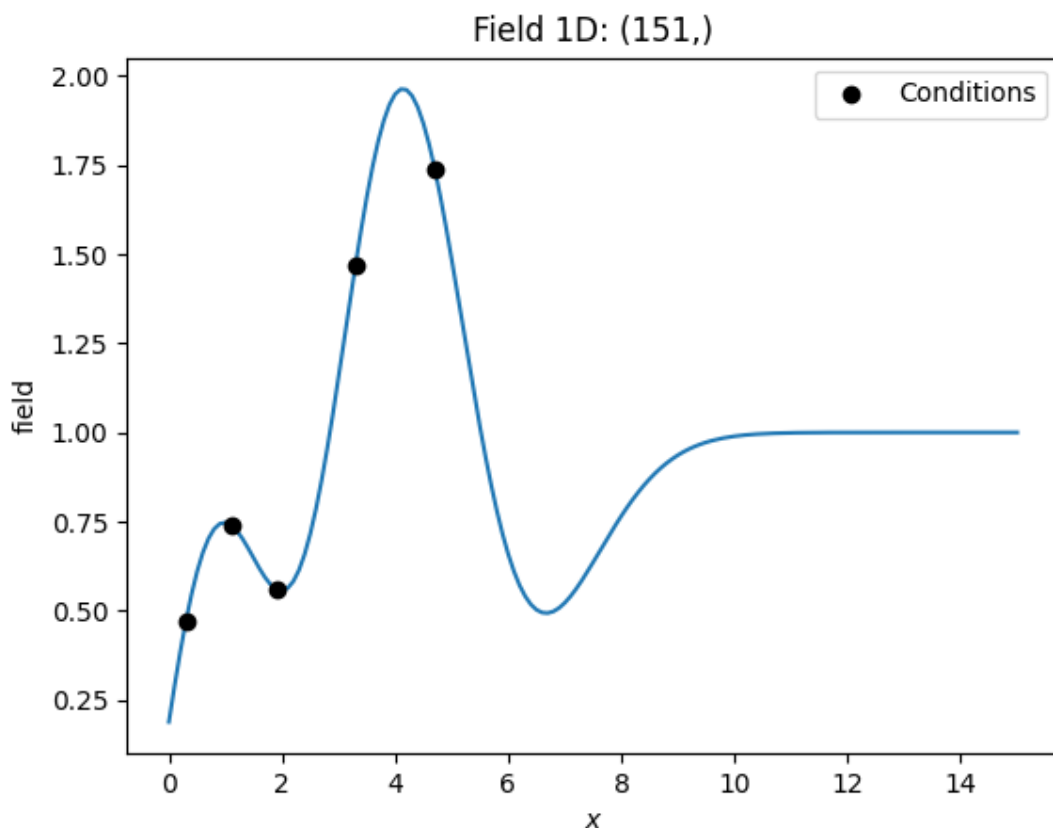
Here we use simple kriging in 1D (for plotting reasons) with 5 given observations/conditions. The mean of the field has to be given beforehand.

```
import numpy as np
from gstools import Gaussian, krig

# 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 = krig.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()
```



Total running time of the script: (0 minutes 0.111 seconds)

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:

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

Thereby $c(x_i, x_j)$ is the covariance of the given observations and μ is a Lagrange multiplier to minimize the kriging error and estimate the mean.

Example

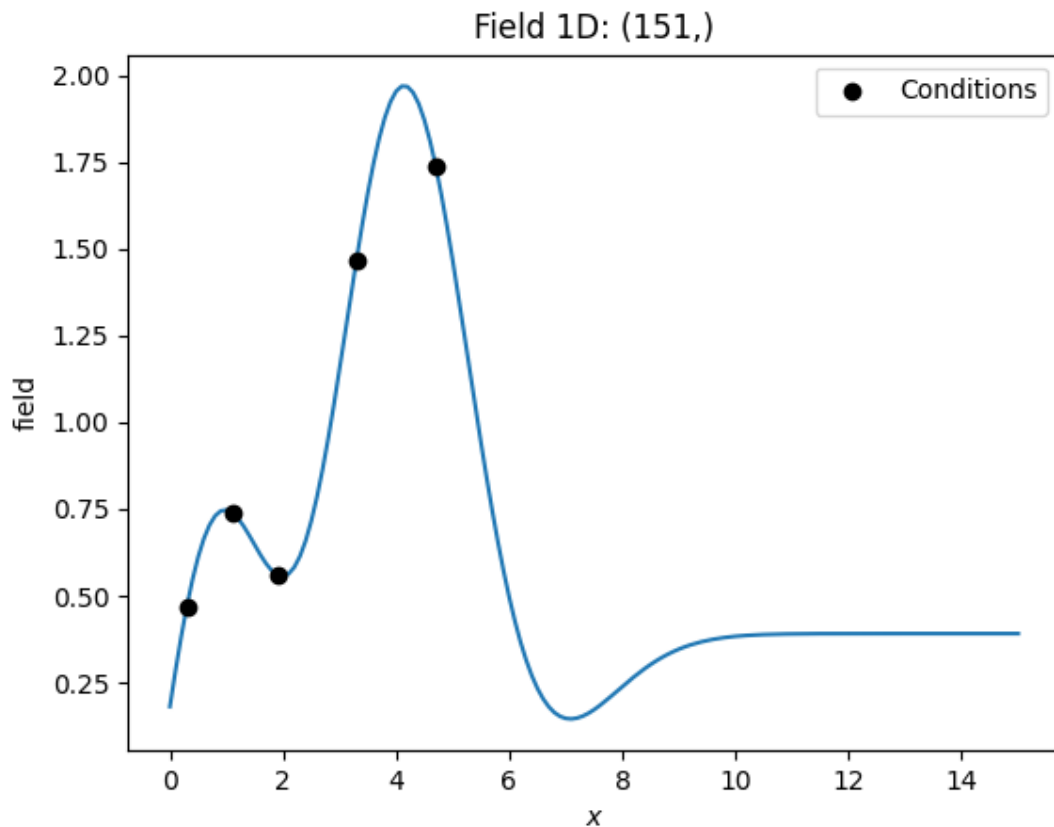
Here we use ordinary kriging in 1D (for plotting reasons) with 5 given observations/conditions. The estimated mean can be accessed by `krig.mean`.

```
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()
```

Total running time of the script: (0 minutes 0.108 seconds)

Interface to PyKrige

To use fancier methods like [regression kriging](#), we provide an interface to [PyKrige](#) (>v1.5), which means you can pass a GStools covariance model to the kriging routines of PyKrige.

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

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

# conditioning data
cond_x = [0.3, 1.9, 1.1, 3.3, 4.7]
cond_y = [1.2, 0.6, 3.2, 4.4, 3.8]
cond_val = [0.47, 0.56, 0.74, 1.47, 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)
```

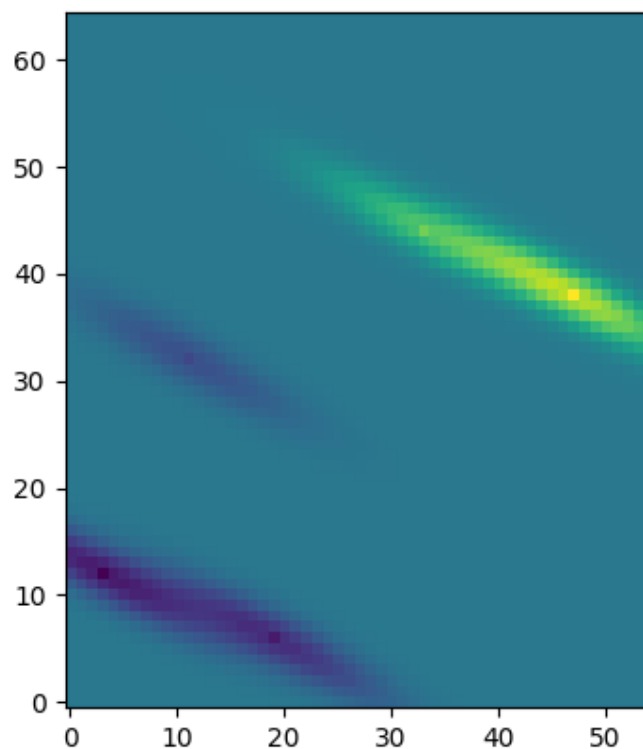
A GStools based [Gaussian](#) covariance model:

```
model = gs.Gaussian(
    dim=2, len_scale=1, anis=0.2, angles=-0.5, var=0.5, nugget=0.1
)
```

Ordinary Kriging with PyKrige

One can pass the defined GStools model as variogram model, which will *not* be fitted to the given data. By providing the GStools model, rotation and anisotropy are also automatically defined:

```
OK1 = OrdinaryKriging(cond_x, cond_y, cond_val, variogram_model=model)
z1, ss1 = OK1.execute("grid", gridx, gridy)
plt.imshow(z1, origin="lower")
plt.show()
```

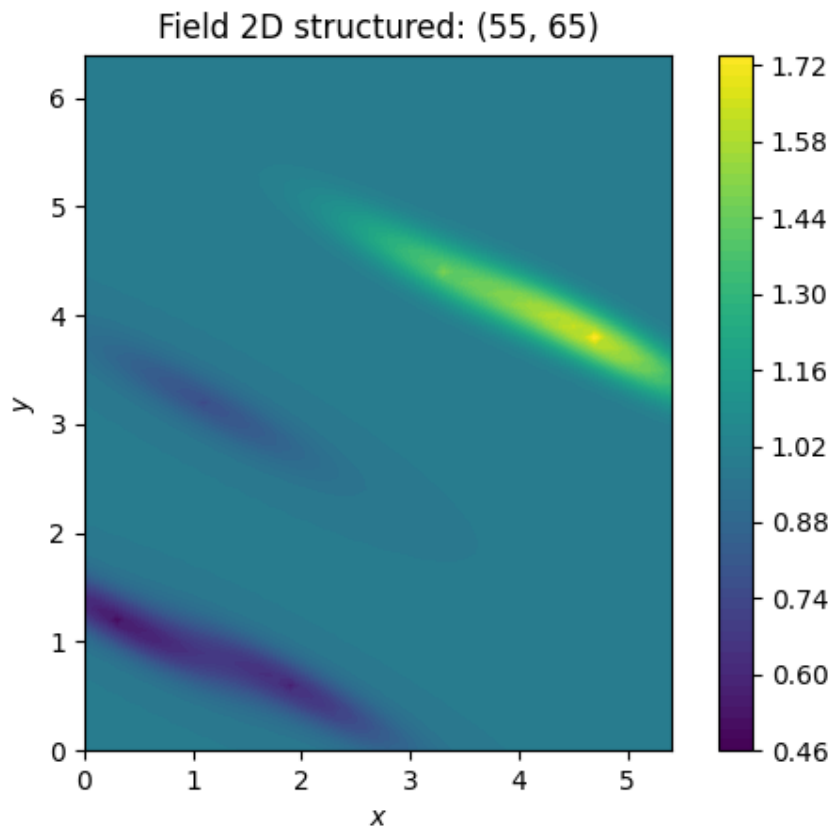


Ordinary Kriging with GStools

The *Ordinary* kriging class is provided by GStools as a shortcut to define ordinary kriging with the general *Krige* class.

PyKrige's routines are using exact kriging by default (when given a nugget). To reproduce this behavior in GStools, we have to set `exact=True`.

```
OK2 = gs.krige.Ordinary(model, [cond_x, cond_y], cond_val, exact=True)
OK2.structured([gridx, gridy])
ax = OK2.plot()
ax.set_aspect("equal")
```



Total running time of the script: (0 minutes 0.317 seconds)

Compare Kriging

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

# conditons
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)
```

A gaussian variogram model.

```
model = Gaussian(dim=1, var=0.5, len_scale=2)
```

Two kriged fields. One with simple and one with ordinary kriging.

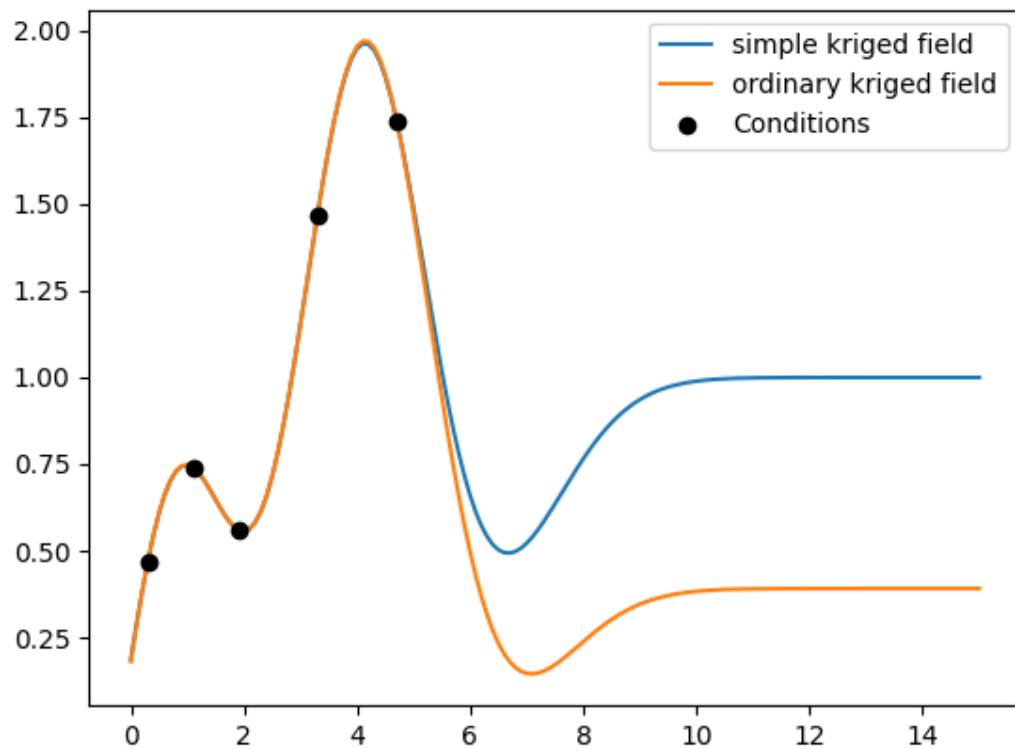
```
kr1 = krige.Simple(model=model, mean=1, cond_pos=cond_pos, cond_val=cond_val)
kr2 = krige.Ordinary(model=model, cond_pos=cond_pos, cond_val=cond_val)
kr1(gridx)
kr2(gridx)
```

```
plt.plot(gridx, kr1.field, label="simple kriged field")
plt.plot(gridx, kr2.field, label="ordinary kriged field")
```

(continues on next page)

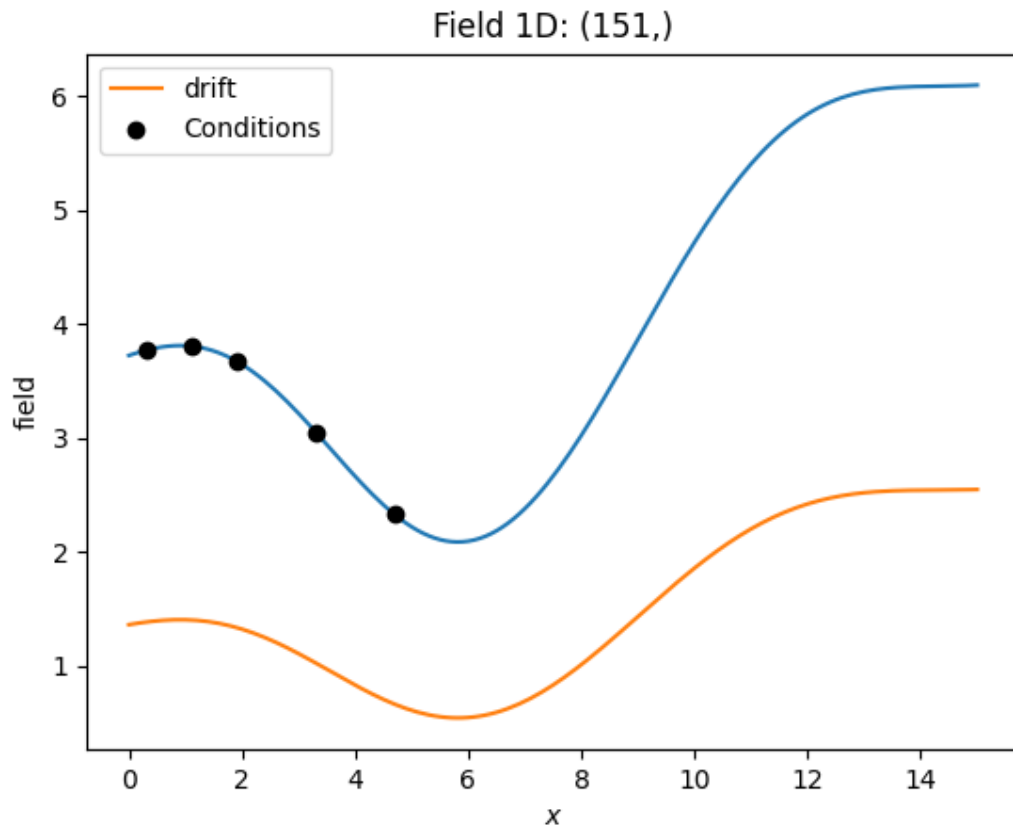
(continued from previous page)

```
plt.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")  
plt.legend()  
plt.show()
```



Total running time of the script: (0 minutes 0.113 seconds)

External Drift Kriging



```
import numpy as np
from gstools import SRF, Gaussian, krige

# synthetic condtions with a drift
drift_model = Gaussian(dim=1, len_scale=4)
drift = SRF(drift_model, seed=1010)
cond_pos = [0.3, 1.9, 1.1, 3.3, 4.7]
ext_drift = drift(cond_pos)
cond_val = ext_drift * 2 + 1
# resulting grid
gridx = np.linspace(0.0, 15.0, 151)
grid_drift = drift(gridx)
# kriging
model = Gaussian(dim=1, var=2, len_scale=4)
krig = krige.ExtDrift(model, cond_pos, cond_val, ext_drift)
krig(gridx, ext_drift=grid_drift)
ax = krig.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
ax.plot(gridx, grid_drift, label="drift")
ax.legend()
```

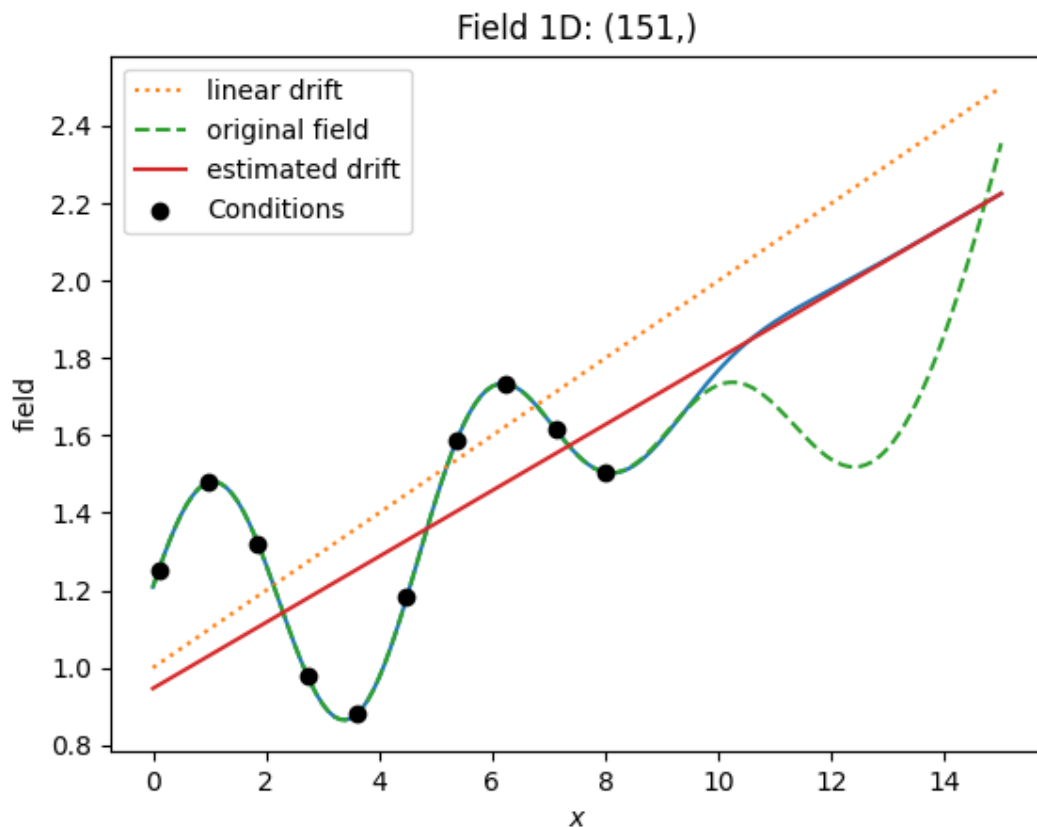
Total running time of the script: (0 minutes 0.117 seconds)

Universal Kriging

You can give a polynomial order or a list of self defined functions representing the internal drift of the given values. This drift will be fitted internally during the kriging interpolation.

In the following we are creating artificial data, where a linear drift was added. The resulting samples are then used as input for Universal kriging.

The “linear” drift is then estimated during the interpolation. To access only the estimated mean/drift, we provide a switch *only_mean* in the call routine.



```
import numpy as np
from gstools import SRF, Gaussian, krig

# synthetic conditons with a drift
drift_model = Gaussian(dim=1, var=0.1, len_scale=2)
drift = SRF(drift_model, seed=101)
cond_pos = np.linspace(0.1, 8, 10)
cond_val = drift(cond_pos) + cond_pos * 0.1 + 1
# resulting grid
gridx = np.linspace(0.0, 15.0, 151)
drift_field = drift(gridx) + gridx * 0.1 + 1
# kriging
model = Gaussian(dim=1, var=0.1, len_scale=2)
krig = krig.Universal(model, cond_pos, cond_val, "linear")
krig(gridx)
ax = krig.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
ax.plot(gridx, gridx * 0.1 + 1, ":", label="linear drift")
```

(continues on next page)

(continued from previous page)

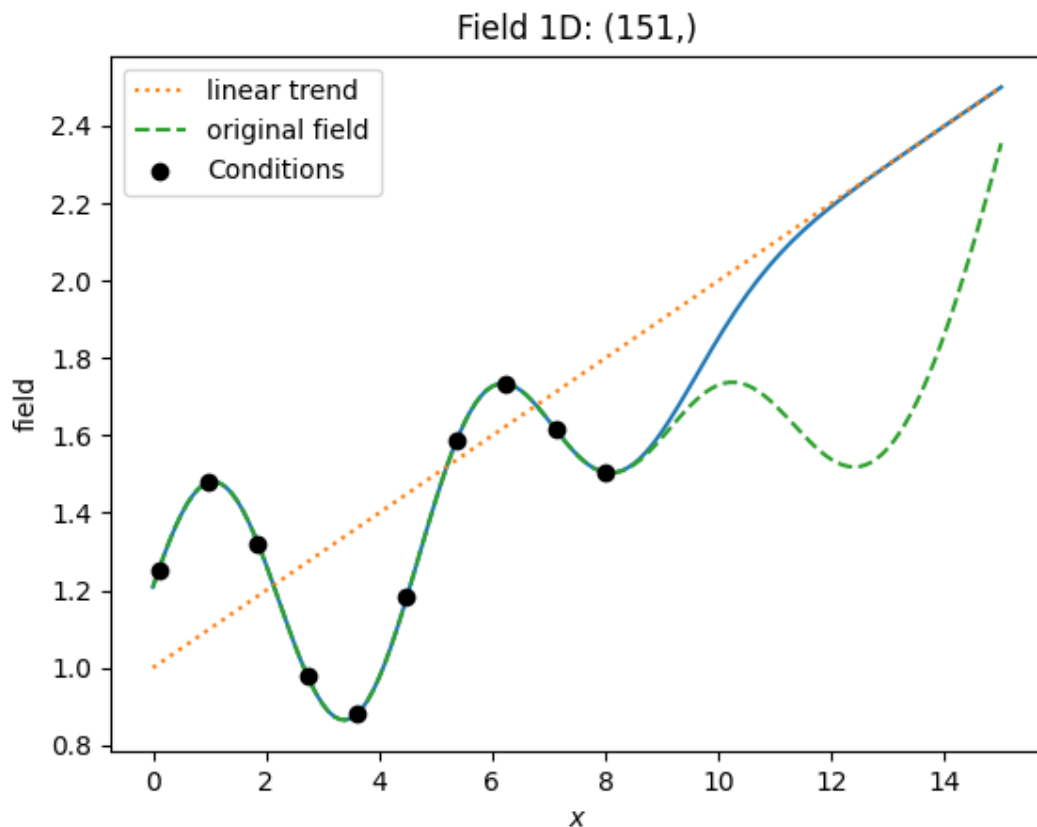
```
ax.plot(gridx, drift_field, "--", label="original field")

mean = krig(gridx, only_mean=True)
ax.plot(gridx, mean, label="estimated drift")

ax.legend()
```

Total running time of the script: (0 minutes 0.140 seconds)

Detrended Kriging



```
import numpy as np
from gstools import SRF, Gaussian, krige

def trend(x):
    """Example for a simple linear trend."""
    return x * 0.1 + 1

# synthetic conditons with trend/drift
drift_model = Gaussian(dim=1, var=0.1, len_scale=2)
drift = SRF(drift_model, seed=101)
cond_pos = np.linspace(0.1, 8, 10)
cond_val = drift(cond_pos) + trend(cond_pos)
# resulting grid
```

(continues on next page)

(continued from previous page)

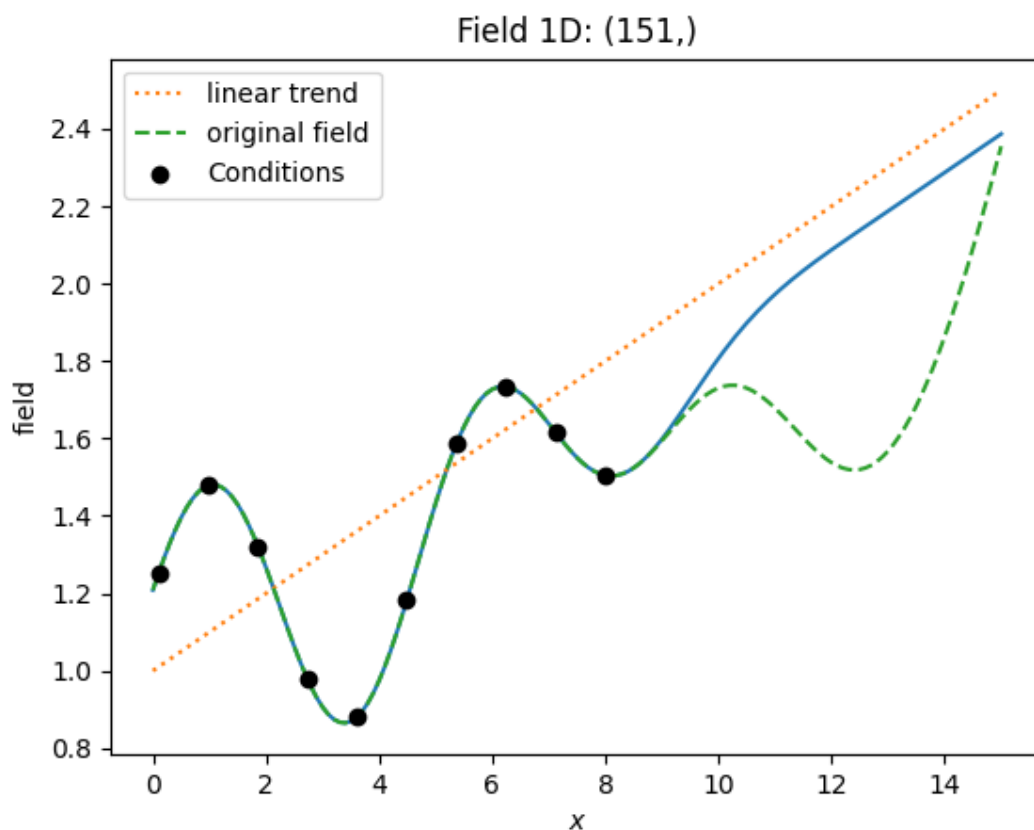
```

gridx = np.linspace(0.0, 15.0, 151)
drift_field = drift(gridx) + trend(gridx)
# kriging
model = Gaussian(dim=1, var=0.1, len_scale=2)
krig_trend = krige.Detrended(model, cond_pos, cond_val, trend)
krig_trend(gridx)
ax = krig_trend.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
ax.plot(gridx, trend(gridx), ":", label="linear trend")
ax.plot(gridx, drift_field, "--", label="original field")
ax.legend()

```

Total running time of the script: (0 minutes 0.131 seconds)

Detrended Ordinary Kriging



```

import numpy as np
from gstools import SRF, Gaussian, krige

def trend(x):
    """Example for a simple linear trend."""
    return x * 0.1 + 1

# synthetic conditons with trend/drift

```

(continues on next page)

(continued from previous page)

```

drift_model = Gaussian(dim=1, var=0.1, len_scale=2)
drift = SRF(drift_model, seed=101)
cond_pos = np.linspace(0.1, 8, 10)
cond_val = drift(cond_pos) + trend(cond_pos)
# resulting grid
gridx = np.linspace(0.0, 15.0, 151)
drift_field = drift(gridx) + trend(gridx)
# kriging
model = Gaussian(dim=1, var=0.1, len_scale=2)
krig_trend = krige.Ordinary(model, cond_pos, cond_val, trend=trend)
krig_trend(gridx)
ax = krig_trend.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
ax.plot(gridx, trend(gridx), ":", label="linear trend")
ax.plot(gridx, drift_field, "--", label="original field")
ax.legend()

```

Total running time of the script: (0 minutes 0.133 seconds)

Incorporating measurement errors

To incorporate the nugget effect and/or given measurement errors, one can set *exact* to *False* and provide either individual measurement errors for each point or set the nugget as a constant measurement error everywhere.

In the following we will show the influence of the nugget and measurement errors.

```

import numpy as np
import gstools as gs

# condtions
cond_pos = [0.3, 1.1, 1.9, 3.3, 4.7]
cond_val = [0.47, 0.74, 0.56, 1.47, 1.74]
cond_err = [0.01, 0.0, 0.1, 0.05, 0]
# resulting grid
gridx = np.linspace(0.0, 15.0, 151)
# spatial random field class
model = gs.Gaussian(dim=1, var=0.9, len_scale=1, nugget=0.1)

```

Here we will use Simple kriging (*unbiased=False*) to interpolate the given conditions.

```

krig = gs.Krige(
    model=model,
    cond_pos=cond_pos,
    cond_val=cond_val,
    mean=1,
    unbiased=False,
    exact=False,
    cond_err=cond_err,
)
krig(gridx)

```

Let's plot the data. You can see, that the estimated values differ more from the input, when the given measurement errors get bigger. In addition we plot the standard deviation.

```

ax = krig.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")

```

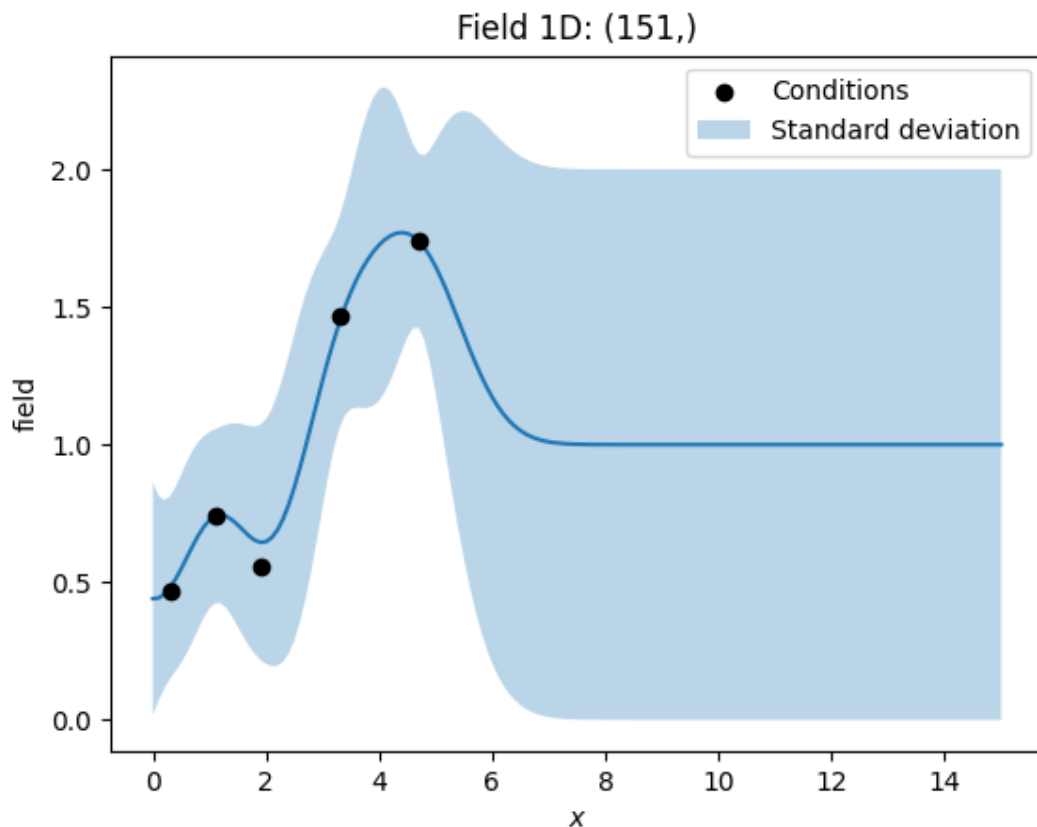
(continues on next page)

(continued from previous page)

```

ax.fill_between(
    gridx,
    # plus/minus standard deviation (70 percent confidence interval)
    krig.field - np.sqrt(krig.krige_var),
    krig.field + np.sqrt(krig.krige_var),
    alpha=0.3,
    label="Standard deviation",
)
ax.legend()

```



Total running time of the script: (0 minutes 0.106 seconds)

Redundant data and pseudo-inverse

It can happen, that the kriging system gets numerically unstable. One reason could be, that the input data contains redundant conditioning points that hold different values.

To smoothly deal with such situations, you can use the pseudo inverse for the kriging matrix, which is enabled by default.

This will result in the average value for the redundant data.

Example

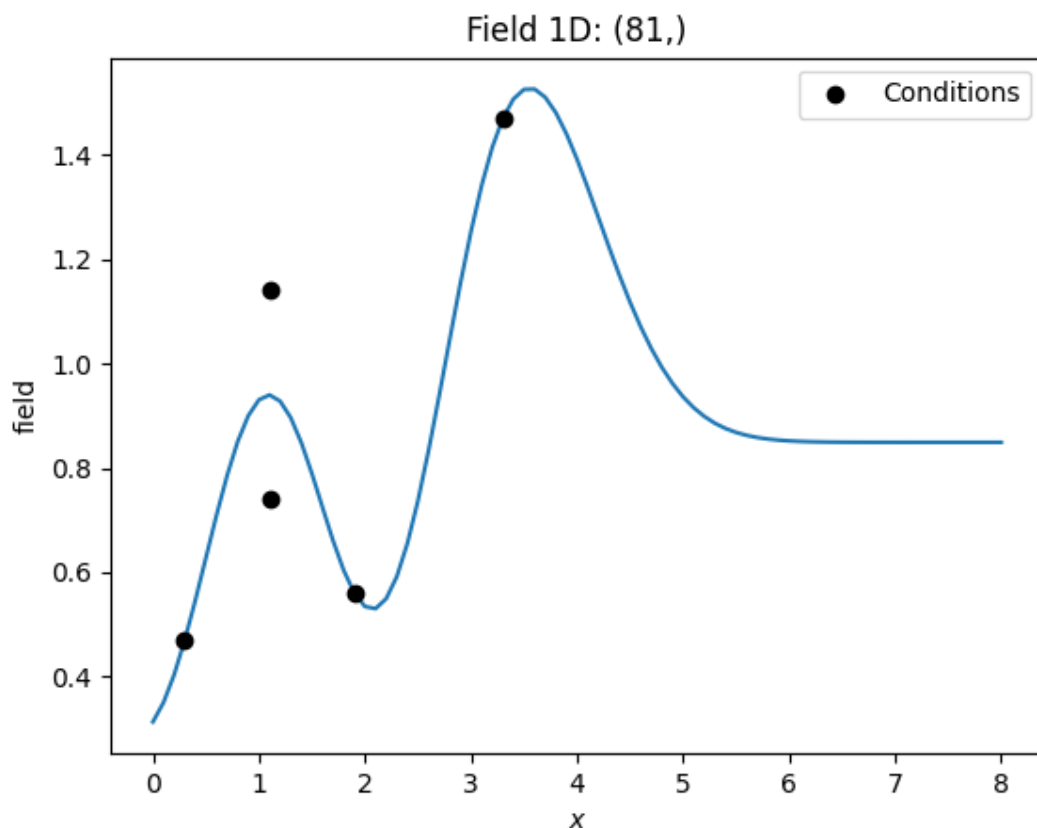
In the following we have two different values at the same location. The resulting kriging field will hold the average at this point.

```
import numpy as np
from gstools import Gaussian, krig

# condtions
cond_pos = [0.3, 1.9, 1.1, 3.3, 1.1]
cond_val = [0.47, 0.56, 0.74, 1.47, 1.14]
# resulting grid
gridx = np.linspace(0.0, 8.0, 81)
# spatial random field class
model = Gaussian(dim=1, var=0.5, len_scale=1)
```

```
krig = krig.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()
```



Total running time of the script: (0 minutes 0.106 seconds)

2.6 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.

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, with 0 as mean and 1 as variance that will be multiplied with the kriging standard deviation.

To do so, you can instantiate a `CondSRF` class with a configured `Krige` class.

The setup of the a conditioned random field should be as follows:

```
krige = gs.Krige(model, cond_pos, cond_val)
cond_srf = gs.CondSRF(krige)
field = cond_srf(grid)
```

Examples

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.

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

# 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)
```

The conditioned spatial random field class depends on a Krige class in order to handle the conditions. This is created as described in the kriging tutorial.

Here we use a Gaussian covariance model and ordinary kriging for conditioning the spatial random field.

```
model = gs.Gaussian(dim=1, var=0.5, len_scale=1.5)
krige = gs.krige.Ordinary(model, cond_pos, cond_val)
cond_srf = gs.CondSRF(krige)
```

```
fields = []
for i in range(100):
    fields.append(cond_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, cond_srf.krige(gridx, only_mean=True), label="estimated mean")
plt.plot(gridx, np.mean(fields, axis=0), linestyle=":", label="Ensemble mean")
plt.plot(gridx, cond_srf.krige.field, linestyle="dashed", label="kriged field")
plt.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
# 99 percent confidence interval
conf = gs.tools.confidence_scaling(0.99)
plt.fill_between(
    gridx,
    cond_srf.krige.field - conf * np.sqrt(cond_srf.krige.krige_var),
```

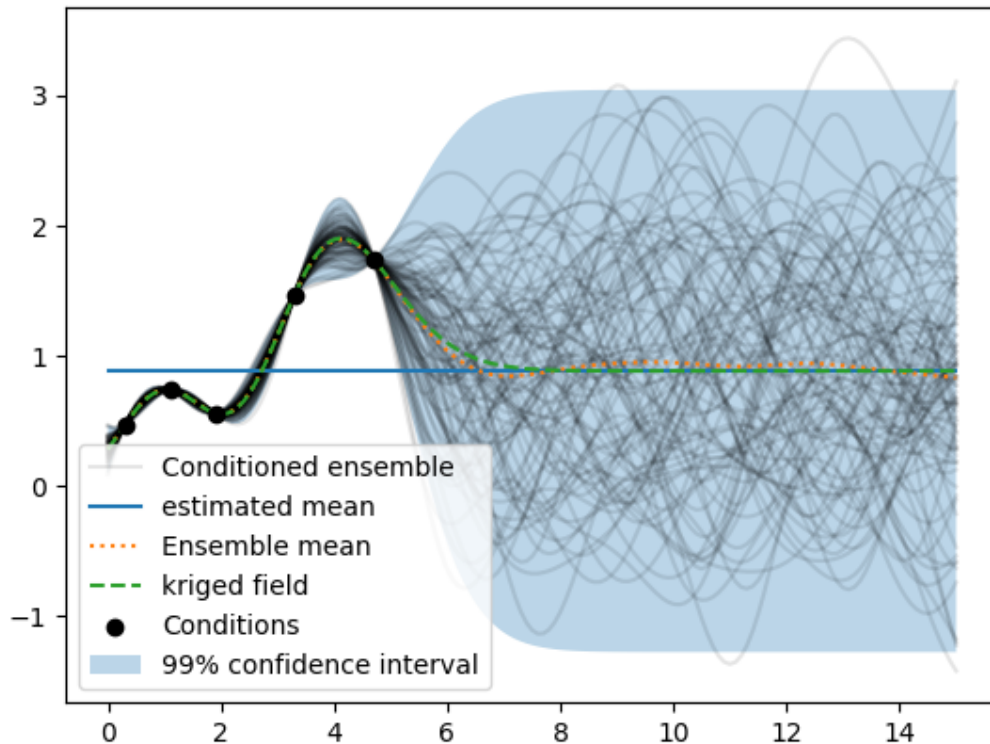
(continues on next page)

(continued from previous page)

```

cond_srf.krige.field + conf * np.sqrt(cond_srf.krige.krige_var),
alpha=0.3,
label="99% confidence interval",
)
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.

Total running time of the script: (0 minutes 1.312 seconds)

Creating an Ensemble of conditioned 2D Fields

Let's create an ensemble of conditioned random fields in 2D.

```

import numpy as np
import matplotlib.pyplot as plt
import gstools as gs

# conditioning data (x, y, value)
cond_pos = [[0.3, 1.9, 1.1, 3.3, 4.7], [1.2, 0.6, 3.2, 4.4, 3.8]]
cond_val = [0.47, 0.56, 0.74, 1.47, 1.74]

# grid definition for output field
x = np.arange(0, 5, 0.1)

```

(continues on next page)

(continued from previous page)

```
y = np.arange(0, 5, 0.1)

model = gs.Gaussian(dim=2, var=0.5, len_scale=5, anis=0.5, angles=-0.5)
krige = gs.Krige(model, cond_pos=cond_pos, cond_val=cond_val)
cond_srf = gs.CondSRF(krige)
```

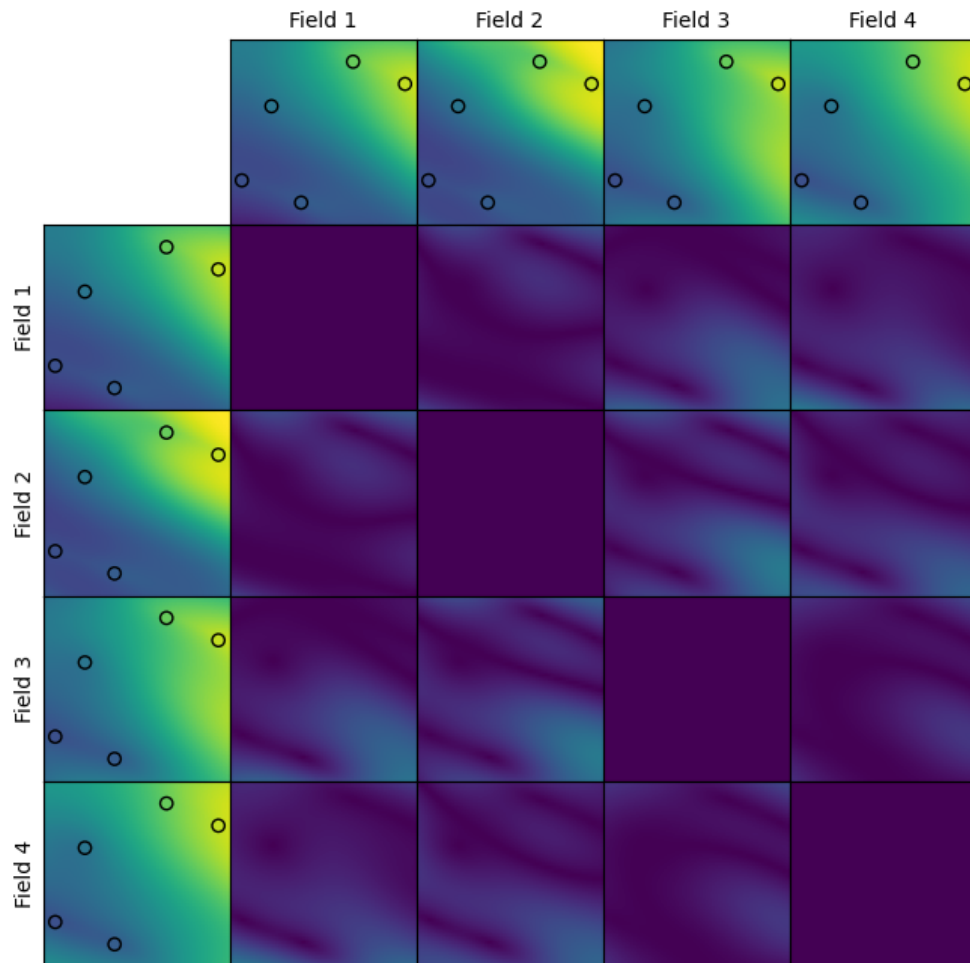
We create a list containing the generated conditioned fields.

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

Now let's have a look at the pairwise differences between the generated fields. We will see, that they coincide at the given conditions.

```
fig, ax = plt.subplots(ens_no + 1, ens_no + 1, figsize=(8, 8))
# plotting kwargs for scatter and image
sc_kwargs = dict(c=cond_val, edgecolors="k", vmin=0, vmax=np.max(field))
im_kwargs = dict(extent=2 * [0, 5], origin="lower", vmin=0, vmax=np.max(field))
for i in range(ens_no):
    # conditioned fields and conditions
    ax[i + 1, 0].imshow(field[i].T, **im_kwargs)
    ax[i + 1, 0].scatter(*cond_pos, **sc_kwargs)
    ax[i + 1, 0].set_ylabel(f"Field {i+1}", fontsize=10)
    ax[0, i + 1].imshow(field[i].T, **im_kwargs)
    ax[0, i + 1].scatter(*cond_pos, **sc_kwargs)
    ax[0, i + 1].set_title(f"Field {i+1}", fontsize=10)
    # absolute differences
    for j in range(ens_no):
        ax[i + 1, j + 1].imshow(np.abs(field[i] - field[j]).T, **im_kwargs)

# beautify plots
ax[0, 0].axis("off")
for a in ax.flatten():
    a.set_xticklabels([], a.set_yticklabels([]))
    a.set_xticks([], a.set_yticks([]))
fig.subplots_adjust(wspace=0, hspace=0)
fig.show()
```



Total running time of the script: (0 minutes 1.250 seconds)

2.7 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:

<code>binary(fld[, divide, upper, lower])</code>	Binary transformation.
<code>discrete(fld, values[, thresholds])</code>	Discrete transformation.
<code>boxcox(fld[, lmbda, shift])</code>	(Inverse) Box-Cox transformation to denormalize data.
<code>zinnharvey(fld[, conn])</code>	Zinn and Harvey transformation to connect low or high values.
<code>normal_force_moments(fld)</code>	Force moments of a normal distributed field.

continues on next page

Table 6 – continued from previous page

<code>normal_to_lognormal(fld)</code>	Transform normal distribution to log-normal distribution.
<code>normal_to_uniform(fld)</code>	Transform normal distribution to uniform distribution on [0, 1].
<code>normal_to_arcsin(fld[, a, b])</code>	Transform normal distribution to the bimodal arcsin distribution.
<code>normal_to_uquad(fld[, a, b])</code>	Transform normal distribution to U-quadratic distribution.

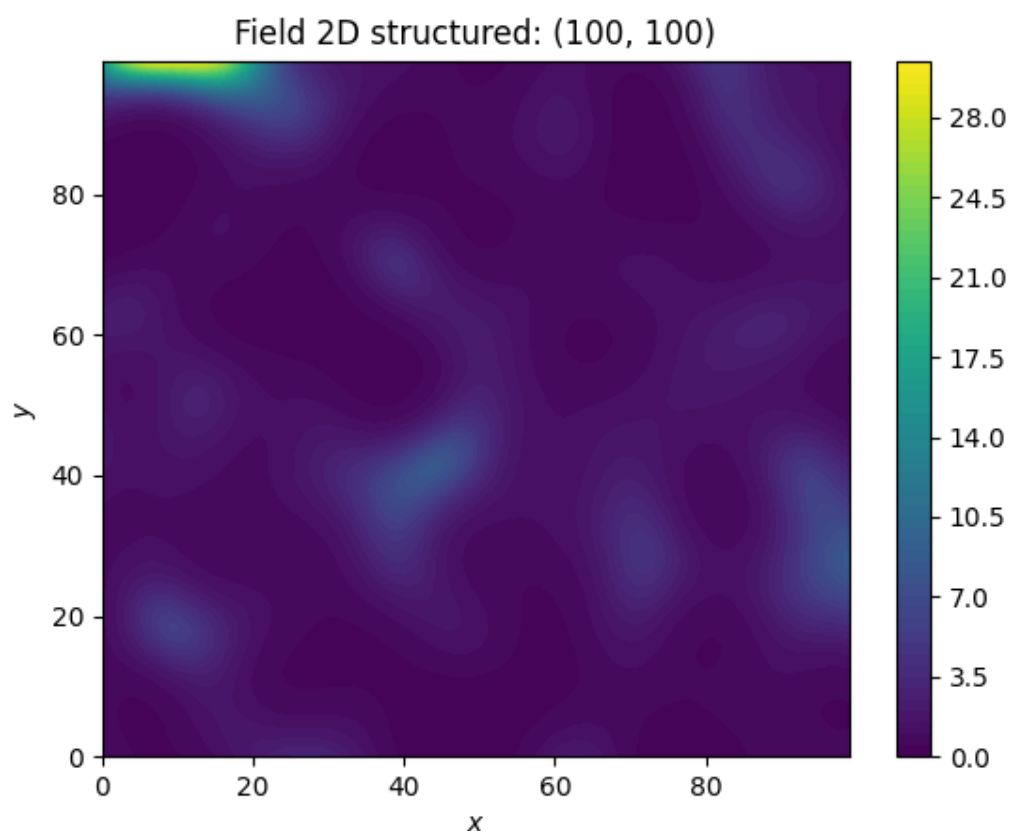
All the transformations take a field class, that holds a generated field, as input and will manipulate this field inplace. Simply import the transform submodule and apply a transformation to the srf class:

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

Examples

log-normal fields

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



```
import gstools as gs

# structured field with a size of 100x100 and a grid-size of 1x1
```

(continues on next page)

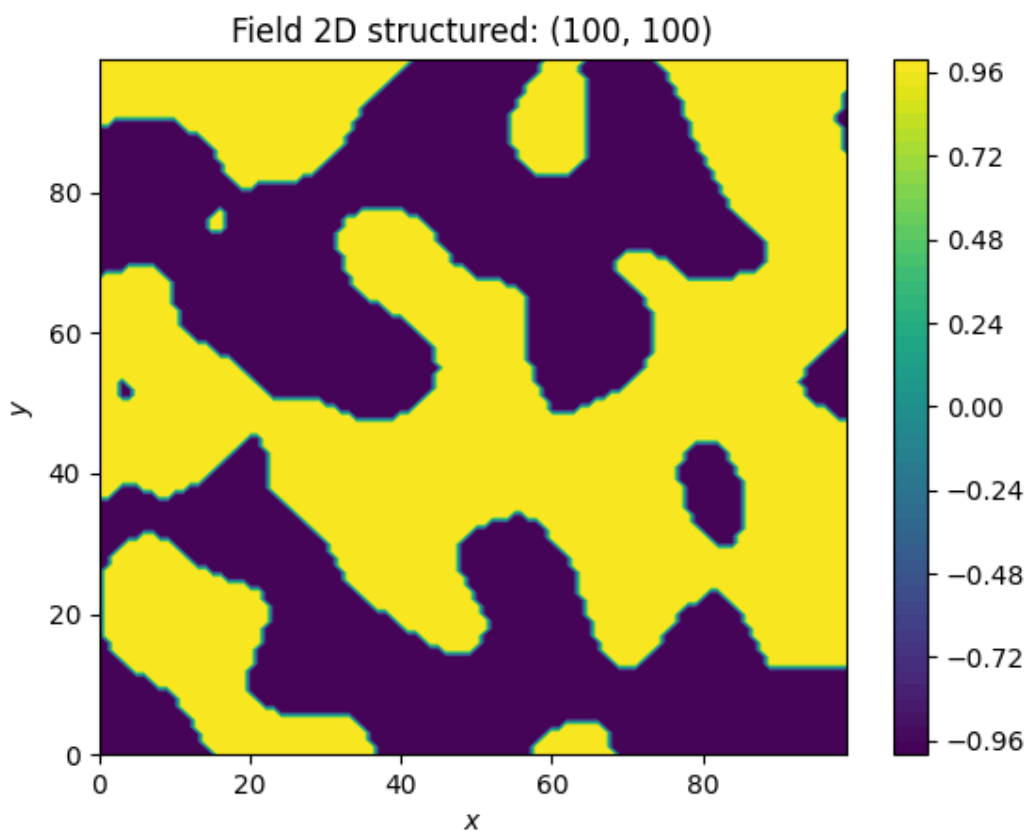
(continued from previous page)

```
x = y = range(100)
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.SRF(model, seed=20170519)
srf.structured([x, y])
gs.transform.normal_to_lognormal(srf)
srf.plot()
```

Total running time of the script: (0 minutes 0.705 seconds)

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.



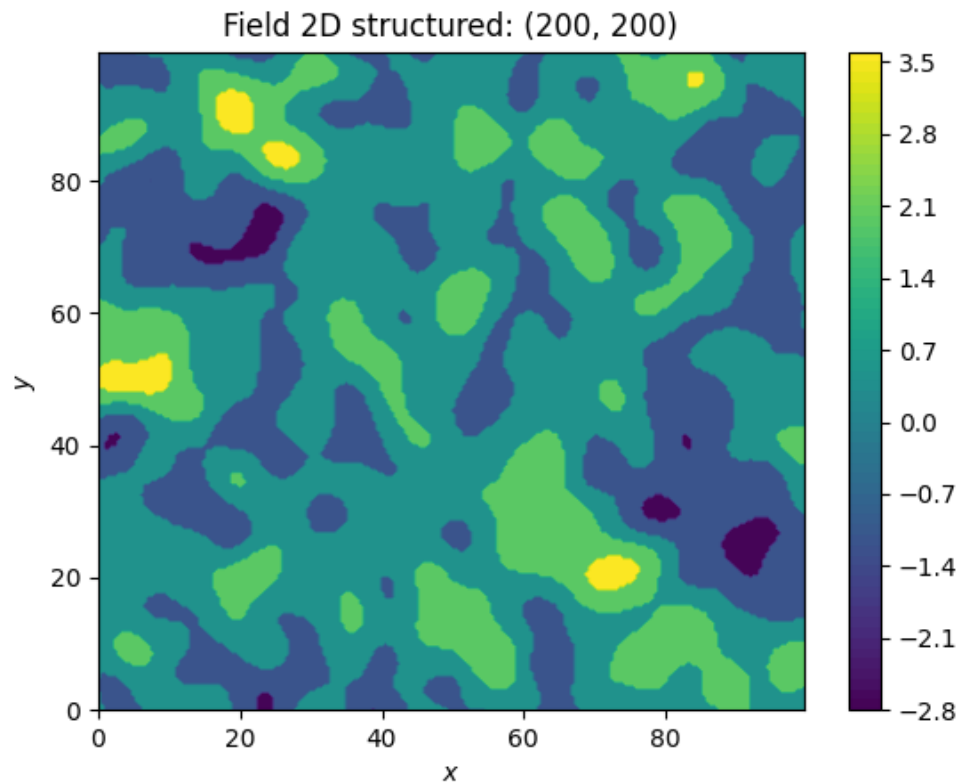
```
import gstools as gs

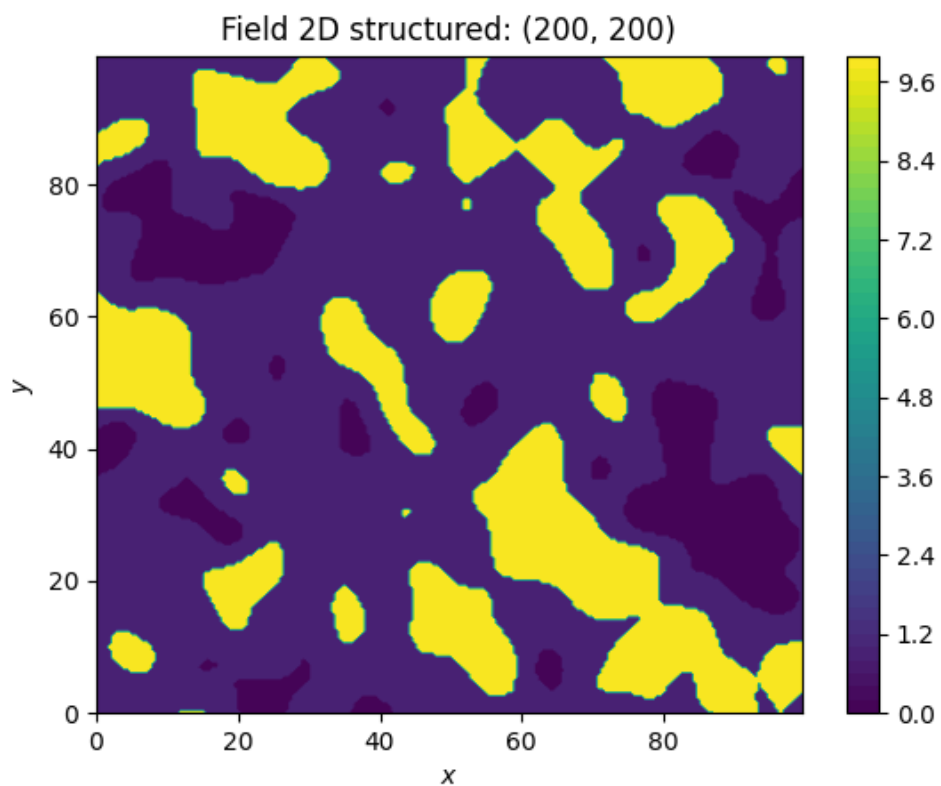
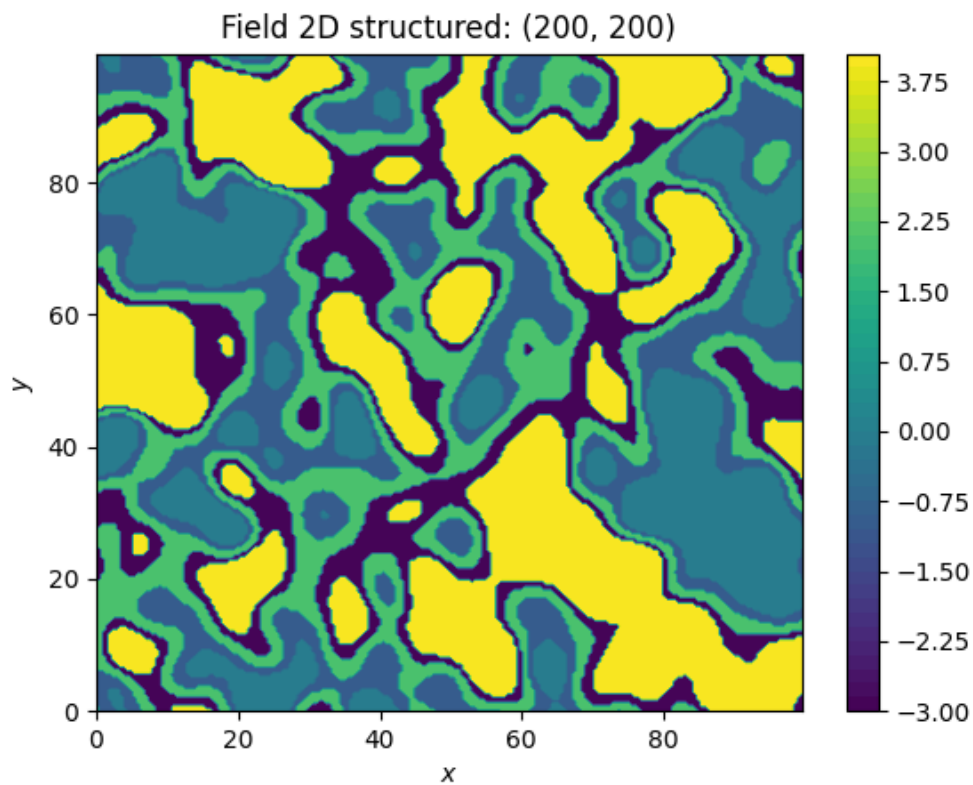
# structured field with a size of 100x100 and a grid-size of 1x1
x = y = range(100)
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.SRF(model, seed=20170519)
srf.structured([x, y])
gs.transform.binary(srf)
srf.plot()
```

Total running time of the script: (0 minutes 0.750 seconds)

Discrete fields

Here we transform a field to a discrete field with values. If we do not give thresholds, the pairwise means of the given values are taken as thresholds. If thresholds are given, arbitrary values can be applied to the field.





```
import numpy as np
import gstools as gs
```

(continues on next page)

(continued from previous page)

```
# structured field with a size of 100x100 and a grid-size of 0.5x0.5
x = y = np.arange(200) * 0.5
model = gs.Gaussian(dim=2, var=1, len_scale=5)
srf = gs.SRF(model, seed=20170519)

# create 5 equidistantly spaced values, thresholds are the arithmetic means
srf.structured([x, y])
discrete_values = np.linspace(np.min(srf.field), np.max(srf.field), 5)
gs.transform.discrete(srf, discrete_values)
srf.plot()

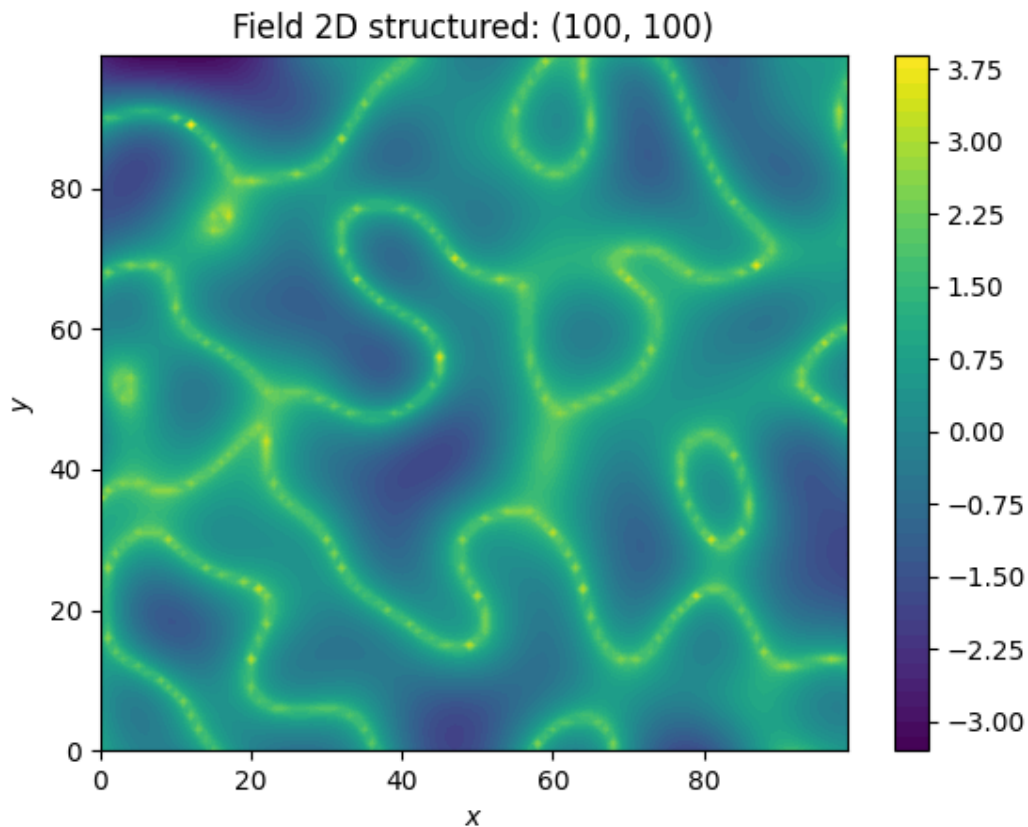
# calculate thresholds for equal shares
# but apply different values to the separated classes
discrete_values2 = [0, -1, 2, -3, 4]
srf.structured([x, y])
gs.transform.discrete(srf, discrete_values2, thresholds="equal")
srf.plot()

# user defined thresholds
thresholds = [-1, 1]
# apply different values to the separated classes
discrete_values3 = [0, 1, 10]
srf.structured([x, y])
gs.transform.discrete(srf, discrete_values3, thresholds=thresholds)
srf.plot()
```

Total running time of the script: (0 minutes 6.892 seconds)

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.



```
import gstools as gs

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

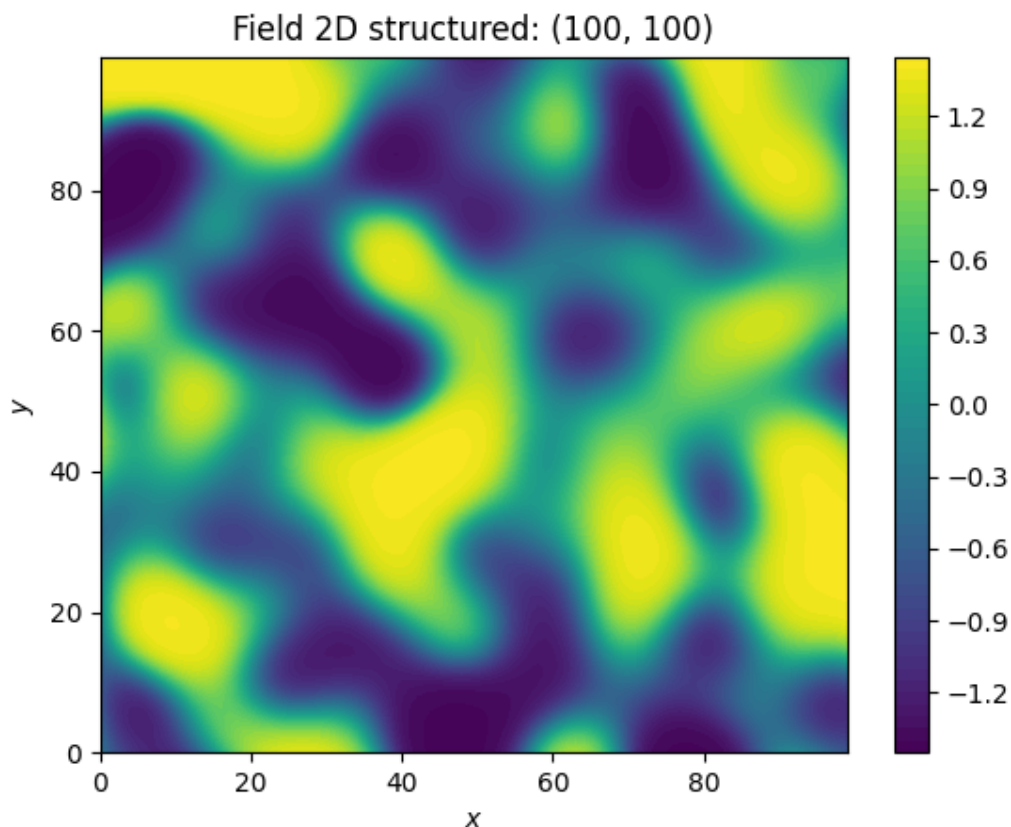
Total running time of the script: (0 minutes 0.769 seconds)

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.



```
import gstools as gs

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

Total running time of the script: (0 minutes 0.785 seconds)

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.

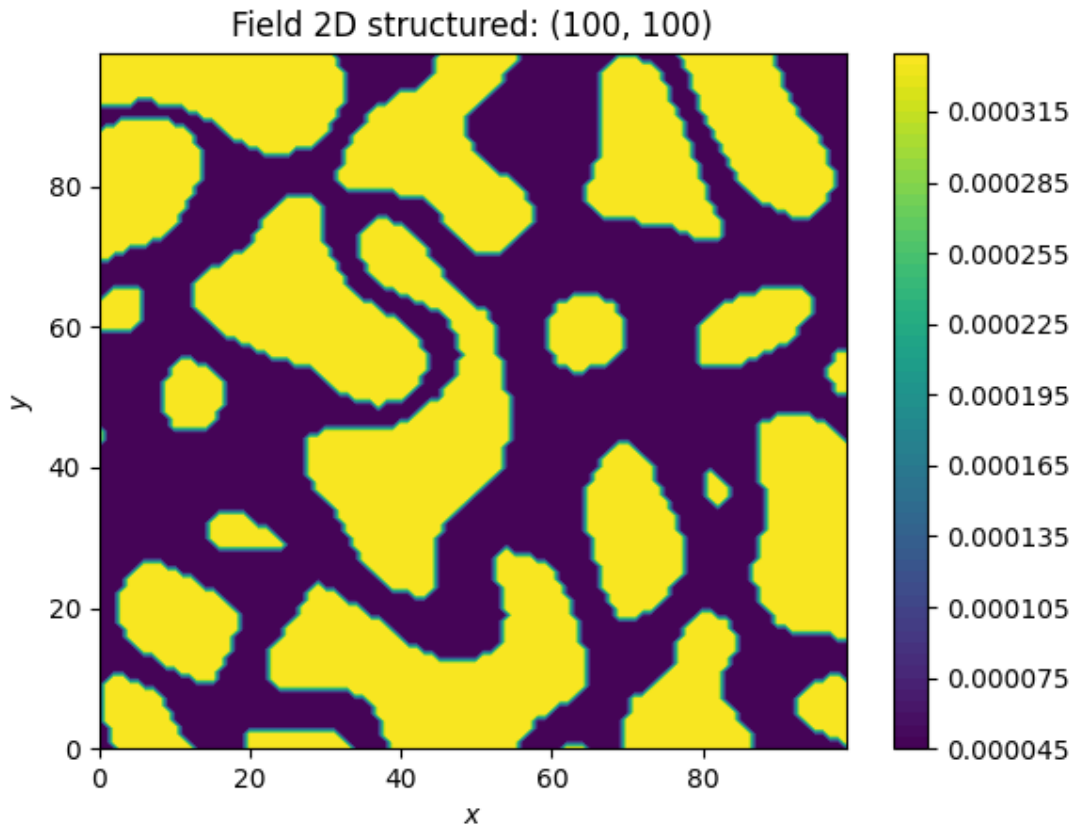
```
import gstools as gs

# structured field with a size of 100x100 and a grid-size of 1x1
x = y = range(100)
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.SRF(model, mean=-9, seed=20170519)
srf.structured([x, y])
gs.transform.normal_force_moments(srf)
```

(continues on next page)

(continued from previous page)

```
gs.transform.zinnharvey(srf, conn="low")
gs.transform.binary(srf)
gs.transform.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.

Total running time of the script: (0 minutes 0.835 seconds)

2.8 Geographic Coordinates

GSTools provides support for [geographic coordinates](#) given by:

- latitude `lat`: specifies the north–south position of a point on the Earth’s surface
- longitude `lon`: specifies the east–west position of a point on the Earth’s surface

If you want to use this feature for field generation or Kriging, you have to set up a geographical covariance Model by setting `latlon=True` in your desired model (see [CovModel](#)):

```
import numpy as np
import gstools as gs

model = gs.Gaussian(latlon=True, var=2, len_scale=np.pi / 16)
```

By doing so, the model will use the associated *Yadrenko* model on a sphere (see [here](#)). The `len_scale` is given in radians to scale the arc-length. In order to have a more meaningful length scale, one can use the `rescale` argument:

```
import gstools as gs

model = gs.Gaussian(latlon=True, var=2, len_scale=500, rescale=gs.EARTH_RADIUS)
```

Then `len_scale` can be interpreted as given in km.

A *Yadrenko* model C is derived from a valid isotropic covariance model in 3D C_{3D} by the following relation:

$$C(\zeta) = C_{3D} \left(2 \cdot \sin \left(\frac{\zeta}{2} \right) \right)$$

Where ζ is the great-circle distance.

Note: `lat` and `lon` are given in degree, whereas the great-circle distance *zeta* is given in radians.

Note, that $2 \cdot \sin(\frac{\zeta}{2})$ is the chordal distance of two points on a sphere, which means we simply think of the earth surface as a sphere, that is cut out of the surrounding three dimensional space, when using the *Yadrenko* model.

Note: Anisotropy is not available with the geographical models, since their geometry is not euclidean. When passing values for `CovModel.anis` or `CovModel.angles`, they will be ignored.

Since the *Yadrenko* model comes from a 3D model, the model dimension will be 3 (see `CovModel.dim`) but the `field_dim` will be 2 in this case (see `CovModel.field_dim`).

Examples

Working with lat-lon random fields

In this example, we demonstrate how to generate a random field on geographical coordinates.

First we setup a model, with `latlon=True`, to get the associated *Yadrenko* model.

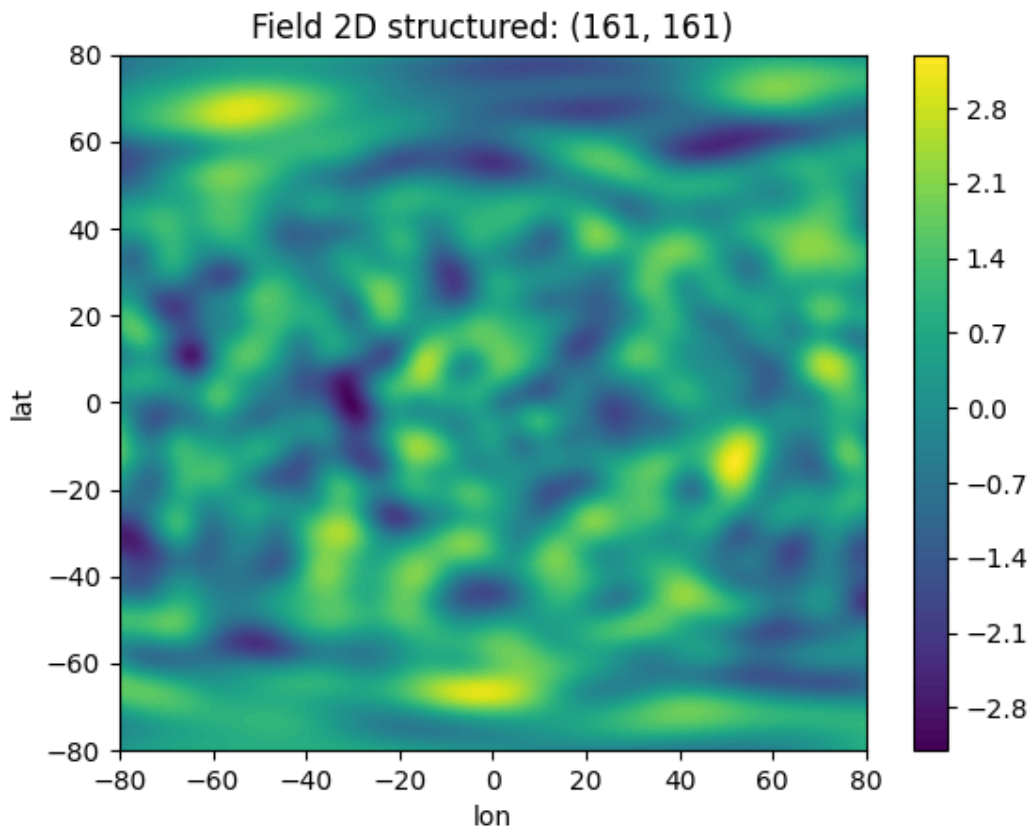
In addition, we will use the earth radius provided by `EARTH_RADIUS`, to have a meaningful length scale in km.

To generate the field, we simply pass `(lat, lon)` as the position tuple to the *SRF* class.

```
import gstools as gs

model = gs.Gaussian(latlon=True, var=1, len_scale=777, rescale=gs.EARTH_RADIUS)

lat = lon = range(-80, 81)
srf = gs.SRF(model, seed=1234)
field = srf.structured((lat, lon))
srf.plot()
```

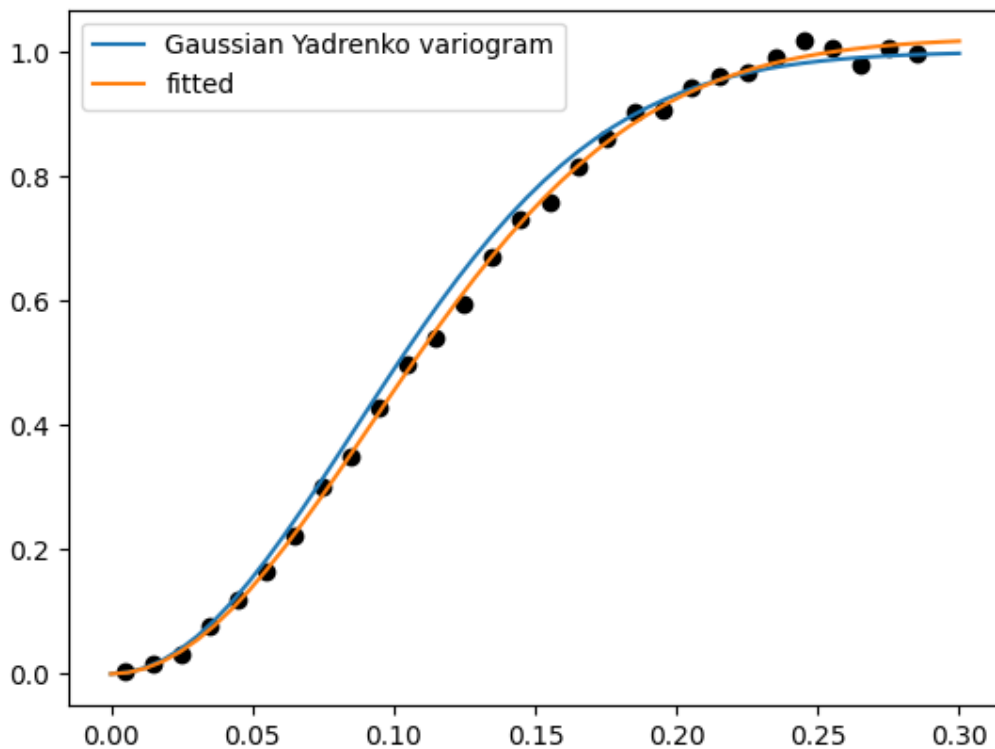



This was easy as always! Now we can use this field to estimate the empirical variogram in order to prove, that the generated field has the correct geo-statistical properties. The `vario_estimate` routine also provides a `latlon` switch to indicate, that the given field is defined on geographical variables.

As we will see, everything went well... phew!

```
bin_edges = [0.01 * i for i in range(30)]
bin_center, emp_vario = gs.vario_estimate(
    (lat, lon),
    field,
    bin_edges,
    latlon=True,
    mesh_type="structured",
    sampling_size=2000,
    sampling_seed=12345,
)

ax = model.plot("vario_yadrenko", x_max=0.3)
model.fit_variogram(bin_center, emp_vario, nugget=False)
model.plot("vario_yadrenko", ax=ax, label="fitted", x_max=0.3)
ax.scatter(bin_center, emp_vario, color="k")
print(model)
```



Out:

```
Gaussian(latlon=True, var=1.02, len_scale=8.3e+02, nugget=0.0, rescale=6.37e+03)
```

Note: Note, that the estimated variogram coincides with the yadrenko variogram, which means it depends on the great-circle distance given in radians.

Keep that in mind when defining bins: The range is at most $\pi \approx 3.14$, which corresponds to the half globe.

Total running time of the script: (0 minutes 9.923 seconds)

Kriging geographical data

In this example we are going to interpolate actual temperature data from the German weather service [DWD](#).

Data is retrieved utilizing the beautiful package [wetterdienst](#), which serves as an API for the DWD data.

For better visualization, we also download a simple shapefile of the German borderline with [cartopy](#).

In order to keep the number of dependencies low, the calls of both functions shown beneath are commented out.

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

def get_borders_germany():
    """Download simple german shape file with cartopy."""
    from cartopy.io import shapereader as shp_read # version 0.18.0
```

(continues on next page)

(continued from previous page)

```

import geopandas as gp # 0.8.1

shpfile = shp_read.natural_earth("50m", "cultural", "admin_0_countries")
df = gp.read_file(shpfile) # only use the simplest polygon
poly = df.loc[df["ADMIN"] == "Germany"]["geometry"].values[0][0]
np.savetxt("de_borders.txt", list(poly.exterior.coords))

def get_dwd_temperature(date="2020-06-09 12:00:00"):
    """Get air temperature from german weather stations from 9.6.20 12:00."""
    from wetterdienst.dwd import observations as obs # version 0.13.0

    settings = dict(
        resolution=obs.DWDObservationResolution.HOURLY,
        start_date=date,
        end_date=date,
    )
    sites = obs.DWDObservationStations(
        parameter_set=obs.DWDObservationParameterSet.TEMPERATURE_AIR,
        period=obs.DWDObservationPeriod.RECENT,
        **settings,
    )
    ids, lat, lon = sites.all().loc[:, ["STATION_ID", "LAT", "LON"]].values.T
    observations = obs.DWDObservationData(
        station_ids=ids,
        parameters=obs.DWDObservationParameter.HOURLY.TEMPERATURE_AIR_200,
        periods=obs.DWDObservationPeriod.RECENT,
        **settings,
    )
    temp = observations.all().VALUE.values
    sel = np.isfinite(temp)
    # select only valid temperature data
    ids, lat, lon, temp = ids.astype(float)[sel], lat[sel], lon[sel], temp[sel]
    head = "id, lat, lon, temp" # add a header to the file
    np.savetxt("temp_obs.txt", np.array([ids, lat, lon, temp]).T, header=head)

```

If you want to download the data again, uncomment the two following lines. We will simply load the resulting files to gain the border polygon and the observed temperature along with the station locations given by lat-lon values.

```

# get_borders_germany()
# get_dwd_temperature(date="2020-06-09 12:00:00")

border = np.loadtxt("de_borders.txt")
ids, lat, lon, temp = np.loadtxt("temp_obs.txt").T

```

First we will estimate the variogram of our temperature data. As the maximal bin distance we choose 8 degrees, which corresponds to a chordal length of about 900 km.

```

bins = gs.standard_bins((lat, lon), max_dist=np.deg2rad(8), latlon=True)
bin_c, vario = gs.vario_estimate((lat, lon), temp, bins, latlon=True)

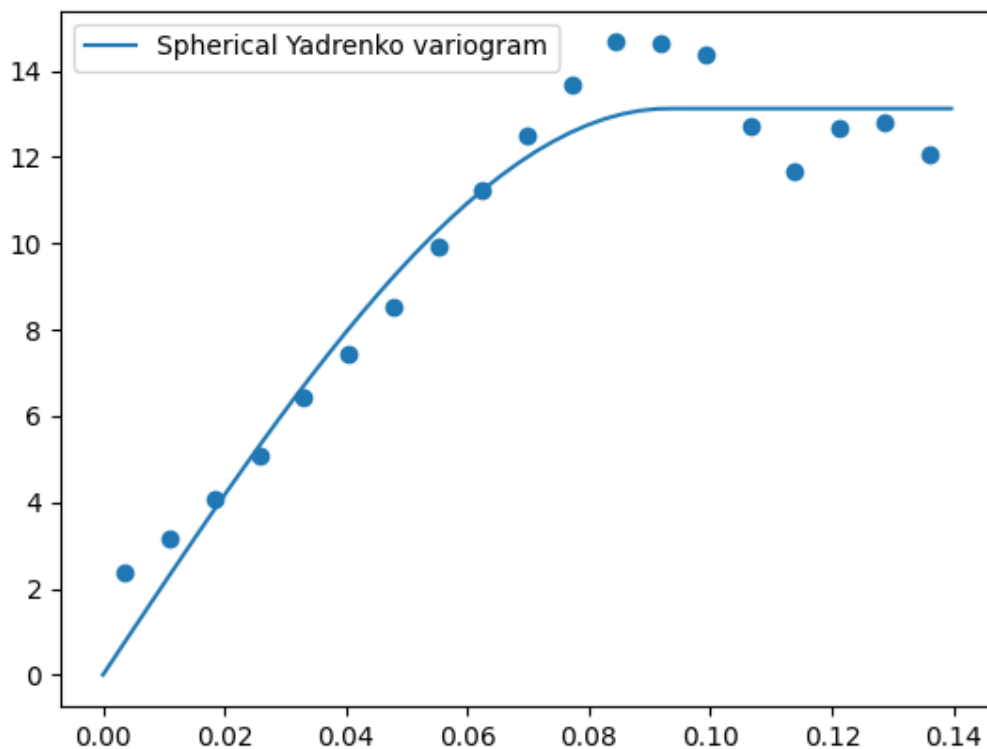
```

Now we can use this estimated variogram to fit a model to it. Here we will use a *Spherical* model. We select the `latlon` option to use the *Yadrenko* variant of the model to gain a valid model for lat-lon coordinates and we rescale it to the earth-radius. Otherwise the length scale would be given in radians representing the great-circle distance.

We deselect the nugget from fitting and plot the result afterwards.

Note: You need to plot the Yadrenko variogram, since the standard variogram still holds the ordinary routine that is not respecting the great-circle distance.

```
model = gs.Spherical(latlon=True, rescale=gs.EARTH_RADIUS)
model.fit_variogram(bin_c, vario, nugget=False)
ax = model.plot("vario_yadrenko", x_max=bins[-1])
ax.scatter(bin_c, vario)
print(model)
```



Out:

```
Spherical(latlon=True, var=13.1, len_scale=5.93e+02, nugget=0.0, rescale=6.37e+03)
```

As we see, we have a rather large correlation length of 600 km.

Now we want to interpolate the data using *Universal* kriging. In order to tinker around with the data, we will use a north-south drift by assuming a linear correlation with the latitude. This can be done as follows:

```
def north_south_drift(lat, lon):
    return lat

uk = gs.krige.Universal(
    model=model,
    cond_pos=(lat, lon),
    cond_val=temp,
    drift_functions=north_south_drift,
)
```

Now we generate the kriging field, by defining a lat-lon grid that covers the whole of Germany. The *Krige* class provides the option to only krig the mean field, so one can have a glimpse at the estimated drift.

```
g_lat = np.arange(47, 56.1, 0.1)
g_lon = np.arange(5, 16.1, 0.1)

field, k_var = uk((g_lat, g_lon), mesh_type="structured")
mean = uk((g_lat, g_lon), mesh_type="structured", only_mean=True)
```

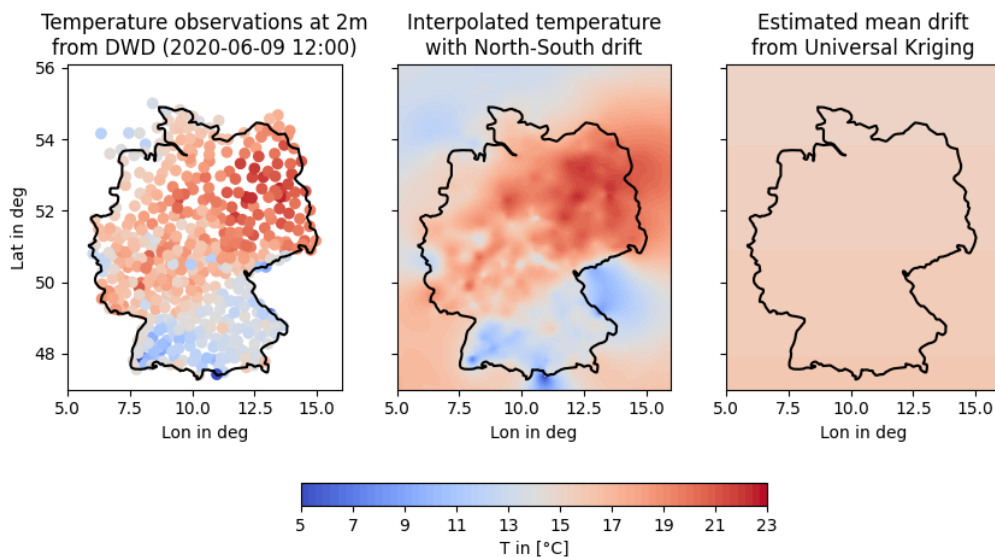
And that's it. Now let's have a look at the generated field and the input data along with the estimated mean:

```
levels = np.linspace(5, 23, 64)
fig, ax = plt.subplots(1, 3, figsize=[10, 5], sharey=True)
sca = ax[0].scatter(lon, lat, c=temp, vmin=5, vmax=23, cmap="coolwarm")
co1 = ax[1].contourf(g_lon, g_lat, field, levels, cmap="coolwarm")
co2 = ax[2].contourf(g_lon, g_lat, mean, levels, cmap="coolwarm")

[ax[i].plot(border[:, 0], border[:, 1], color="k") for i in range(3)]
[ax[i].set_xlim([5, 16]) for i in range(3)]
[ax[i].set_xlabel("Lon in deg") for i in range(3)]
ax[0].set_ylabel("Lat in deg")

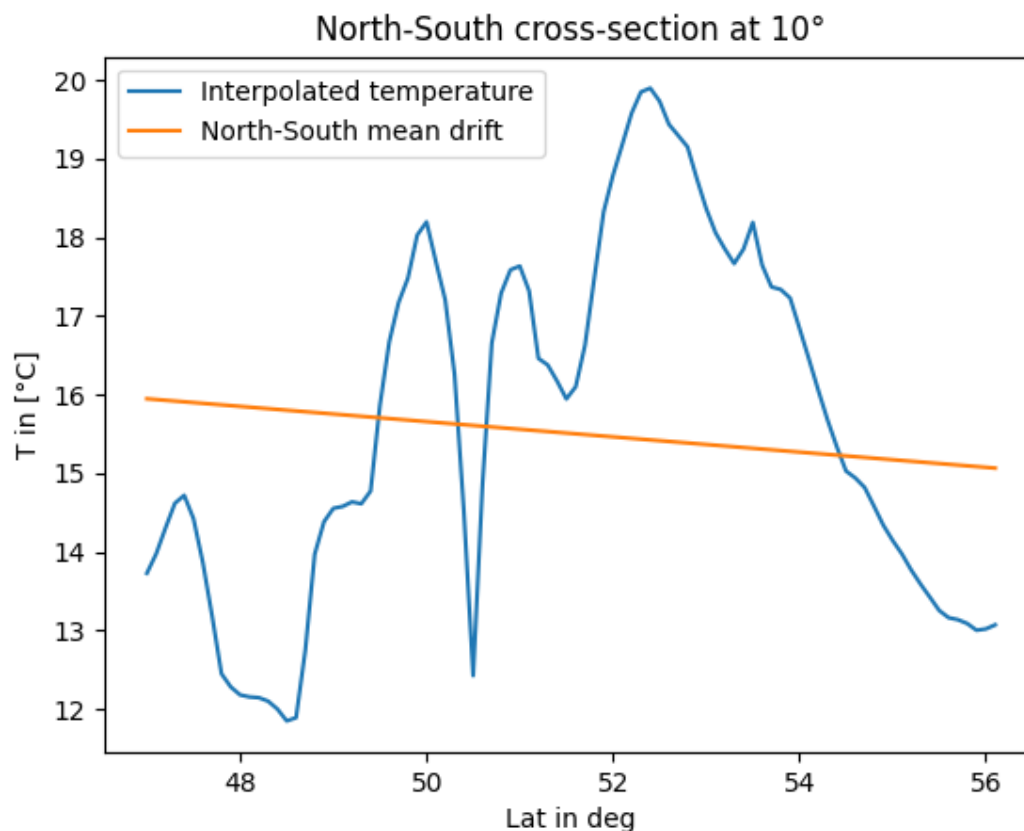
ax[0].set_title("Temperature observations at 2m\nfrom DWD (2020-06-09 12:00)")
ax[1].set_title("Interpolated temperature\nwith North-South drift")
ax[2].set_title("Estimated mean drift\nfrom Universal Kriging")

fmt = dict(orientation="horizontal", shrink=0.5, fraction=0.1, pad=0.2)
fig.colorbar(co2, ax=ax, **fmt).set_label("T in [°C]")
```



To get a better impression of the estimated north-south drift, we'll take a look at a cross-section at a longitude of 10 degree:

```
fig, ax = plt.subplots()
ax.plot(g_lat, field[:, 50], label="Interpolated temperature")
ax.plot(g_lat, mean[:, 50], label="North-South mean drift")
ax.set_xlabel("Lat in deg")
ax.set_ylabel("T in [°C]")
ax.set_title("North-South cross-section at 10°")
ax.legend()
```



Interpretation of the results is now up to you! ;-)

Total running time of the script: (0 minutes 8.188 seconds)

2.9 Spatio-Temporal Modeling

Spatio-Temporal modelling can provide insights into time dependent processes like rainfall, air temperature or crop yield.

GSTools provides the metric spatio-temporal model for all covariance models by enhancing the spatial model dimension with a time dimension to result in the spatio-temporal dimension `st_dim` and setting a spatio-temporal anisotropy ratio with `st_anis`:

```
import gstools as gs
dim = 3 # spatial dimension
st_dim = dim + 1
st_anis = 0.4
st_model = gs.Exponential(dim=st_dim, anis=st_anis)
```

Since it is given in the name “spatio-temporal”, we will always treat the time as last dimension. This enables us to have spatial anisotropy and rotation defined as in non-temporal models, without altering the behavior in the time dimension:

```
anis = [0.4, 0.2] # spatial anisotropy in 3D
angles = [0.5, 0.4, 0.3] # spatial rotation in 3D
st_model = gs.Exponential(dim=st_dim, anis=anis+[st_anis], angles=angles)
```

In order to generate spatio-temporal position tuples, GSTools provides a convenient function `generate_st_grid`. The output can be used for spatio-temporal random field generation (or kriging resp. conditioned fields):

```
pos = dim * [1, 2, 3] # 3 points in space (1,1,1), (2,2,2) and (3,3,3)
time = range(10) # 10 time steps
st_grid = gs.generate_st_grid(pos, time)
st_rsf = gs.SRF(st_model)
st_field = st_rsf(st_grid).reshape(-1, len(time))
```

Then we can access the different time-steps by the last array index.

Examples

Creating a 1D Synthetic Precipitation Field

In this example we will create a time series of a 1D synthetic precipitation field.

We'll start off by creating a Gaussian random field with an exponential variogram, which seems to reproduce the spatial correlations of precipitation fields quite well. We'll create a daily timeseries over a one dimensional cross section of 50km. This workflow is suited for sub daily precipitation time series.

```
import copy
import numpy as np
import matplotlib.pyplot as plt
import gstools as gs

# fix the seed for reproducibility
seed = 20170521
# spatial axis of 50km with a resolution of 1km
x = np.arange(0, 50, 1.0)
# half daily timesteps over three months
t = np.arange(0.0, 90.0, 0.5)

# total spatio-temporal dimension
st_dim = 1 + 1
# space-time anisotropy ratio given in units d / km
st_anis = 0.4

# an exponential variogram with a corr. lengths of 2d and 5km
model = gs.Exponential(dim=st_dim, var=1.0, len_scale=5.0, anis=st_anis)
# create a spatial random field instance
srf = gs.SRF(model, seed=seed)

pos, time = [x], [t]

# a Gaussian random field which is also saved internally for the transformations
srf.structured(pos + time)
P_gau = copy.deepcopy(srf.field)
```

Next, we could take care of the dry periods. Therefore we would simply introduce a lower threshold value. But we will combine this step with the next one. Anyway, for demonstration purposes, we will also do it with the threshold value now.

```
threshold = 0.85
P_cut = copy.deepcopy(srf.field)
P_cut[P_cut <= threshold] = 0.0
```

With the above lines of code we have created a cut off Gaussian spatial random field with an exponential variogram. But precipitation fields are not distributed Gaussian. Thus, we will now transform the field with an inverse box-cox transformation (create a non-Gaussian field), which is often used to account for the skewness of precipitation

fields. Different values have been suggested for the transformation parameter lambda, but we will stick to 1/2. As already mentioned, we will perform the cutoff for the dry periods with this transformation implicitly with the shift. The warning will tell you that values have indeed been cut off and it can be ignored. We call the resulting field Gaussian anamorphosis.

```
# the lower this value, the more will be cut off, a value of 0.2 cuts off
# nearly everything in this example.
cutoff = 0.55
gs.transform.boxcox(srf, lmbda=0.5, shift=-1.0 / cutoff)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/gstools/envs/v1.3.0/lib/python3.7/
↳ site-packages/gstools/transform/field.py:164: UserWarning: Box-Cox: Some values
↳ will be cut off!
  warn("Box-Cox: Some values will be cut off!")
```

As a last step, the amount of precipitation is set. This should of course be calibrated towards observations (the same goes for the threshold, the variance, correlation length, and so on).

```
amount = 2.0
srf.field *= amount
P_ana = srf.field
```

Finally we can have a look at the fields resulting from each step. Note, that the cutoff of the cut Gaussian only approximates the cutoff values from the box-cox transformation. For a closer look, we will examine a cross section at an arbitrary location. And afterwards we will create a contour plot for visual candy.

```
fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)

axs[0, 0].set_title("Gaussian")
axs[0, 0].plot(t, P_gau[20, :])
axs[0, 0].set_ylabel(r"$P$ / mm")

axs[0, 1].set_title("Cut Gaussian")
axs[0, 1].plot(t, P_cut[20, :])

axs[1, 0].set_title("Cut Gaussian Anamorphosis")
axs[1, 0].plot(t, P_ana[20, :])
axs[1, 0].set_xlabel(r"$t$ / d")
axs[1, 0].set_ylabel(r"$P$ / mm")

axs[1, 1].set_title("Different Cross Section")
axs[1, 1].plot(t, P_ana[10, :])
axs[1, 1].set_xlabel(r"$t$ / d")

plt.tight_layout()

fig, axs = plt.subplots(2, 2, sharex=True, sharey=True)

axs[0, 0].set_title("Gaussian")
cont = axs[0, 0].contourf(t, x, P_gau, cmap="PuBu", levels=10)
cbar = fig.colorbar(cont, ax=axs[0, 0])
cbar.ax.set_ylabel(r"$P$ / mm")
axs[0, 0].set_ylabel(r"$x$ / km")

axs[0, 1].set_title("Cut Gaussian")
cont = axs[0, 1].contourf(t, x, P_cut, cmap="PuBu", levels=10)
```

(continues on next page)

(continued from previous page)

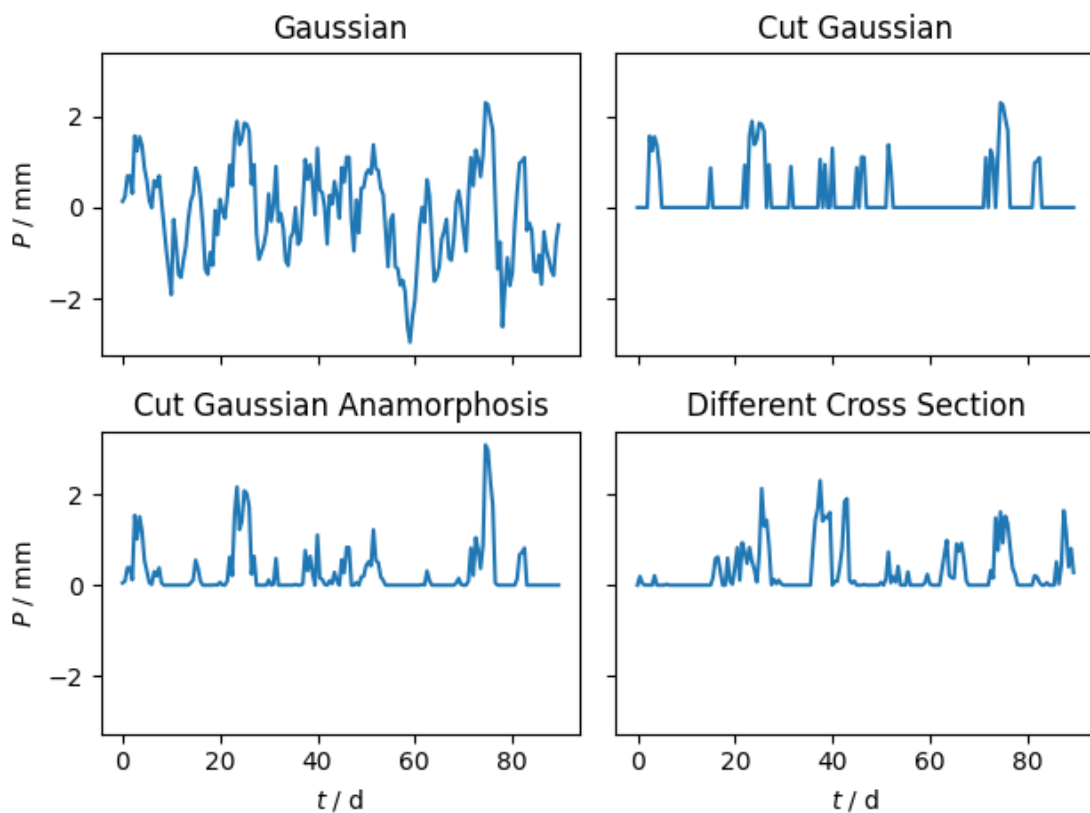
```

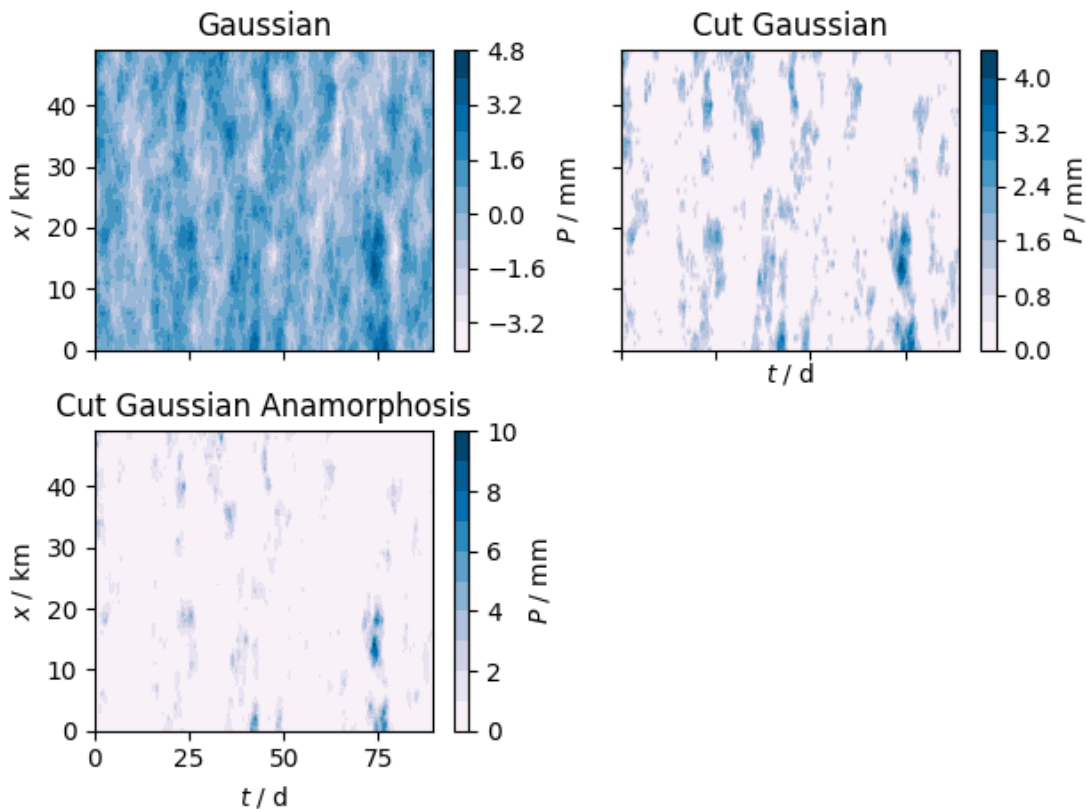
cbar = fig.colorbar(cont, ax=axes[0, 1])
cbar.ax.set_ylabel(r"$P$ / mm")
axes[0, 1].set_xlabel(r"$t$ / d")

axes[1, 0].set_title("Cut Gaussian Anamorphosis")
cont = axes[1, 0].contourf(t, x, P_ana, cmap="PuBu", levels=10)
cbar = fig.colorbar(cont, ax=axes[1, 0])
cbar.ax.set_ylabel(r"$P$ / mm")
axes[1, 0].set_xlabel(r"$t$ / d")
axes[1, 0].set_ylabel(r"$x$ / km")

fig.delaxes(axes[1, 1])
plt.tight_layout()

```





Total running time of the script: (0 minutes 1.180 seconds)

Creating a 2D Synthetic Precipitation Field

In this example we'll create a time series of a 2D synthetic precipitation field.

Very similar to the previous tutorial, we'll start off by creating a Gaussian random field with an exponential variogram, which seems to reproduce the spatial correlations of precipitation fields quite well. We'll create a daily timeseries over a two dimensional domain of 50km x 40km. This workflow is suited for sub daily precipitation time series.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import gstools as gs

# fix the seed for reproducibility
seed = 20170521
# 1st spatial axis of 50km with a resolution of 1km
x = np.arange(0, 50, 1.0)
# 2nd spatial axis of 40km with a resolution of 1km
y = np.arange(0, 40, 1.0)
# half daily timesteps over three months
t = np.arange(0.0, 90.0, 0.5)

# total spatio-temporal dimension
st_dim = 2 + 1
# space-time anisotropy ratio given in units d / km
st_anis = 0.4
```

(continues on next page)

(continued from previous page)

```

# an exponential variogram with a corr. lengths of 5km, 5km, and 2d
model = gs.Exponential(dim=st_dim, var=1.0, len_scale=5.0, anis=st_anis)
# create a spatial random field instance
srf = gs.SRF(model, seed=seed)

pos, time = [x, y], [t]

# the Gaussian random field
srf.structured(pos + time)

# account for the skewness and the dry periods
cutoff = 0.55
gs.transform.boxcox(srf, lmbda=0.5, shift=-1.0 / cutoff)

# adjust the amount of precipitation
amount = 4.0
srf.field *= amount

```

Out:

```

/home/docs/checkouts/readthedocs.org/user_builds/gstools/envs/v1.3.0/lib/python3.7/
↳ site-packages/gstools/transform/field.py:164: UserWarning: Box-Cox: Some values
↳ will be cut off!
    warn("Box-Cox: Some values will be cut off!")

```

plot the 2d precipitation field over time as an animation.

```

def _update_ani(time_step):
    im.set_array(srf.field[:, :, time_step].T)
    return (im,)

fig, ax = plt.subplots()
im = ax.imshow(
    srf.field[:, :, 0].T,
    cmap="Blues",
    interpolation="bicubic",
    origin="lower",
)
cbar = fig.colorbar(im)
cbar.ax.set_ylabel(r"Precipitation $$ / mm")
ax.set_xlabel(r"$x$ / km")
ax.set_ylabel(r"$y$ / km")

ani = animation.FuncAnimation(
    fig, _update_ani, len(t), interval=100, blit=True
)

```

Total running time of the script: (1 minutes 29.897 seconds)

2.10 Normalizing Data

When dealing with real-world data, one can't assume it to be normal distributed. In fact, many properties are modeled by applying different transformations, for example conductivity is often assumed to be log-normal or precipitation is transformed using the famous box-cox power transformation.

These “normalizers” are often represented as parametric power transforms and one is interested in finding the best parameter to gain normality in the input data.

This is of special interest when kriging should be applied, since the target variable of the kriging interpolation is assumed to be normal distributed.

GSTools provides a set of Normalizers and routines to automatically fit these to input data by minimizing the likelihood function.

Mean, Trend and Normalizers

All Field classes (*SRF*, *Krige* or *CondSRF*) provide the input of *mean*, *normalizer* and *trend*:

- A *trend* can be a callable function, that represents a trend in input data. For example a linear decrease of temperature with height.
- The *normalizer* will be applied after the data was detrended, i.e. the trend was subtracted from the data, in order to gain normality.
- The *mean* is now interpreted as the mean of the normalized data. The user could also provide a callable mean, but it is mostly meant to be constant.

When no normalizer is given, *trend* and *mean* basically behave the same. We just decided that a trend is associated with raw data and a mean is used in the context of normally distributed data.

Provided Normalizers

The following normalizers can be passed to all Field-classes and variogram estimation routines or can be used as standalone tools to analyse data.

<i>LogNormal</i> ([data])	Log-normal fields.
<i>BoxCox</i> ([data])	Box-Cox (1964) transformed fields.
<i>BoxCoxShift</i> ([data])	Box-Cox (1964) transformed fields including shifting.
<i>YeoJohnson</i> ([data])	Yeo-Johnson (2000) transformed fields.
<i>Modulus</i> ([data])	Modulus or John-Draper (1980) transformed fields.
<i>Manly</i> ([data])	Manly (1971) transformed fields.

Examples

Log-Normal Kriging

Log Normal kriging is a term to describe a special workflow for kriging to deal with log-normal data, like conductivity or transmissivity in hydrogeology.

It simply means to first convert the input data to a normal distribution, i.e. applying a logarithmic function, then interpolating these values with kriging and transforming the result back with the exponential function.

The resulting kriging variance describes the error variance of the log-values of the target variable.

In this example we will use ordinary kriging.

```
import numpy as np
import gstools as gs

# conditons
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)
# stable covariance model
model = gs.Stable(dim=1, var=0.5, len_scale=2.56, alpha=1.9)
```

In order to result in log-normal kriging, we will use the [LogNormal](#) Normalizer. This is a parameter-less normalizer, so we don't have to fit it.

```
normalizer = gs.normalizer.LogNormal
```

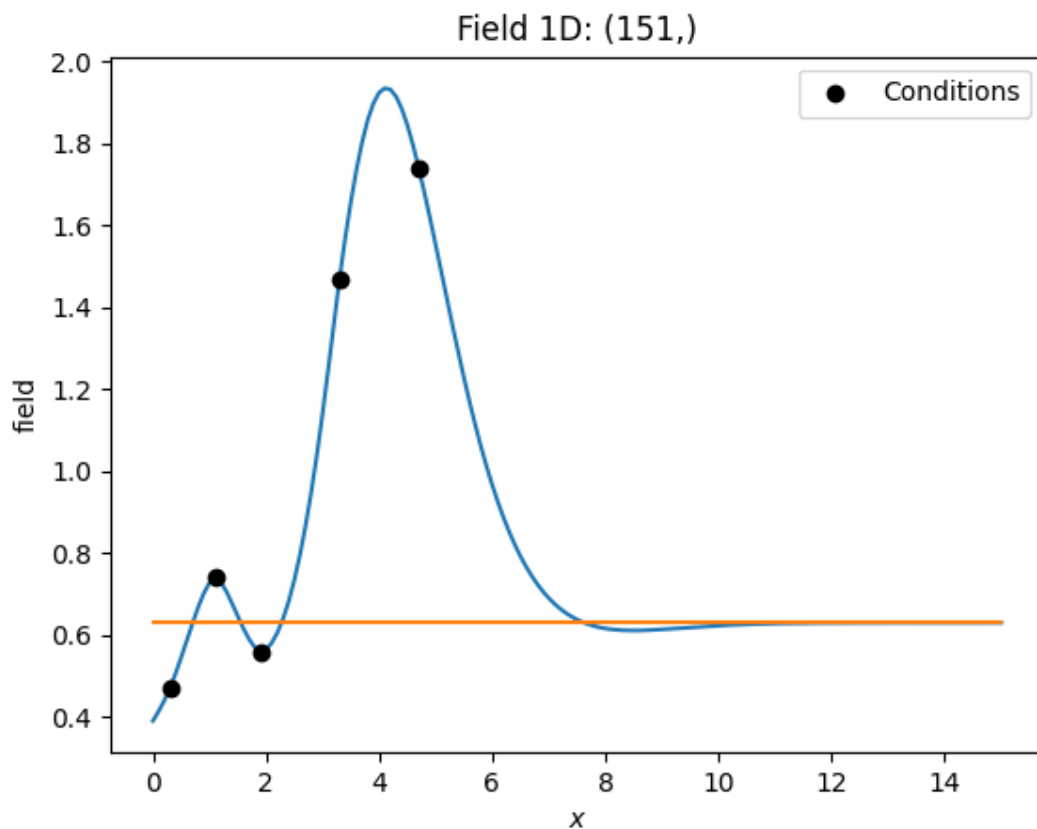
Now we generate the interpolated field as well as the mean field. This can be done by setting *only_mean=True* in *Krige.__call__*. The result is then stored as *mean_field*.

In terms of log-normal kriging, this mean represents the geometric mean of the field.

```
krige = gs.krige.Ordinary(model, cond_pos, cond_val, normalizer=normalizer)
# interpolate the field
krige(gridx)
# also generate the mean field
krige(gridx, only_mean=True)
```

And that's it. Let's have a look at the results.

```
ax = krige.plot()
# plotting the geometric mean
krige.plot("mean_field", ax=ax)
# plotting the conditioning data
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
ax.legend()
```



Total running time of the script: (0 minutes 0.114 seconds)

Automatic fitting

In order to demonstrate how to automatically fit normalizer and variograms, we generate synthetic log-normal data, that should be interpolated with ordinary kriging.

Normalizers are fitted by minimizing the likelihood function and variograms are fitted by estimating the empirical variogram with automatic binning and fitting the theoretical model to it. Thereby the sill is constrained to match the field variance.

Artificial data

Here we generate log-normal data following a Gaussian covariance model. We will generate the “original” field on a 60x60 mesh, from which we will take samples in order to pretend a situation of data-scarcity.

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

# structured field with edge length of 50
x = y = range(51)
pos = gs.generate_grid([x, y])
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.SRF(model, seed=20170519, normalizer=gs.normalizer.LogNormal())
# generate the original field
srf(pos)
```

Here, we sample 60 points and set the conditioning points and values.

```
ids = np.arange(srf.field.size)
samples = np.random.RandomState(20210201).choice(ids, size=60, replace=False)

# sample conditioning points from generated field
cond_pos = pos[:, samples]
cond_val = srf.field[samples]
```

Fitting and Interpolation

Now we want to interpolate the “measured” samples and we want to normalize the given data with the BoxCox transformation.

Here we set up the kriging routine and use a *Stable* model, that should be fitted automatically to the given data and we pass the *BoxCox* normalizer in order to gain normality.

The normalizer will be fitted automatically to the data, by setting `fit_normalizer=True`.

The covariance/variogram model will be fitted by an automatic workflow by setting `fit_variogram=True`.

```
krige = gs.krige.Ordinary(
    model=gs.Stable(dim=2),
    cond_pos=cond_pos,
    cond_val=cond_val,
    normalizer=gs.normalizer.BoxCox(),
    fit_normalizer=True,
    fit_variogram=True,
)
```

First, let’s have a look at the fitting results:

```
print(krige.model)
print(krige.normalizer)
```

Out:

```
Stable(dim=2, var=0.576, len_scale=8.85, nugget=0.00682, alpha=2.0)
BoxCox(lmbda=-0.0754)
```

As we see, it went quite well. Variance is a bit underestimated, but length scale and nugget are good. The shape parameter of the stable model is correctly estimated to be close to 2, so we result in a Gaussian like model.

The BoxCox parameter *lmbda* was estimated to be almost 0, which means, the log-normal distribution was correctly fitted.

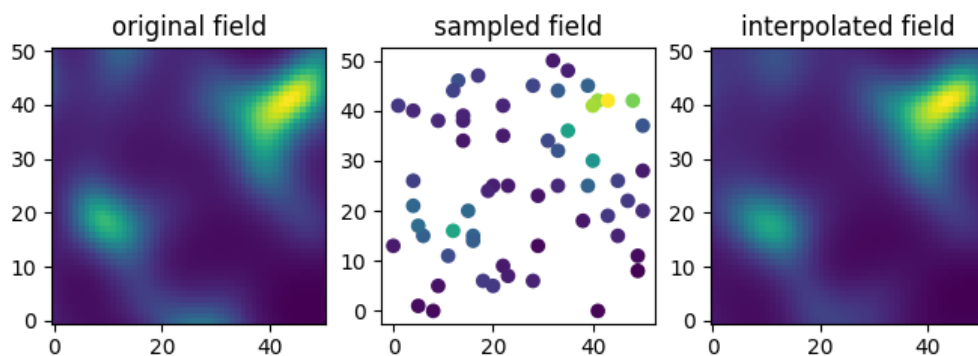
Now let’s run the kriging interpolation.

```
krige(pos)
```

Plotting

Finally let's compare the original, sampled and interpolated fields. As we'll see, there is a lot of information in the covariance structure of the measurement samples and the field is reconstructed quite accurately.

```
fig, ax = plt.subplots(1, 3, figsize=[8, 3])
ax[0].imshow(srf.field.reshape(len(x), len(y)).T, origin="lower")
ax[1].scatter(*cond_pos, c=cond_val)
ax[2].imshow(krige.field.reshape(len(x), len(y)).T, origin="lower")
# titles
ax[0].set_title("original field")
ax[1].set_title("sampled field")
ax[2].set_title("interpolated field")
# set aspect ratio to equal in all plots
[ax[i].set_aspect("equal") for i in range(3)]
```



Total running time of the script: (0 minutes 0.398 seconds)

Normalizer Comparison

Let's compare the transformation behavior of the provided normalizers.

But first, we define a convenience routine and make some imports as always.

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

def dashes(i=1, max_n=12, width=1):
    """Return line dashes."""
    return i * [width, width] + [max_n * 2 * width - 2 * i * width, width]
```

We select 4 normalizers depending on a single parameter λ and plot their transformation behavior within the interval $[-5, 5]$.

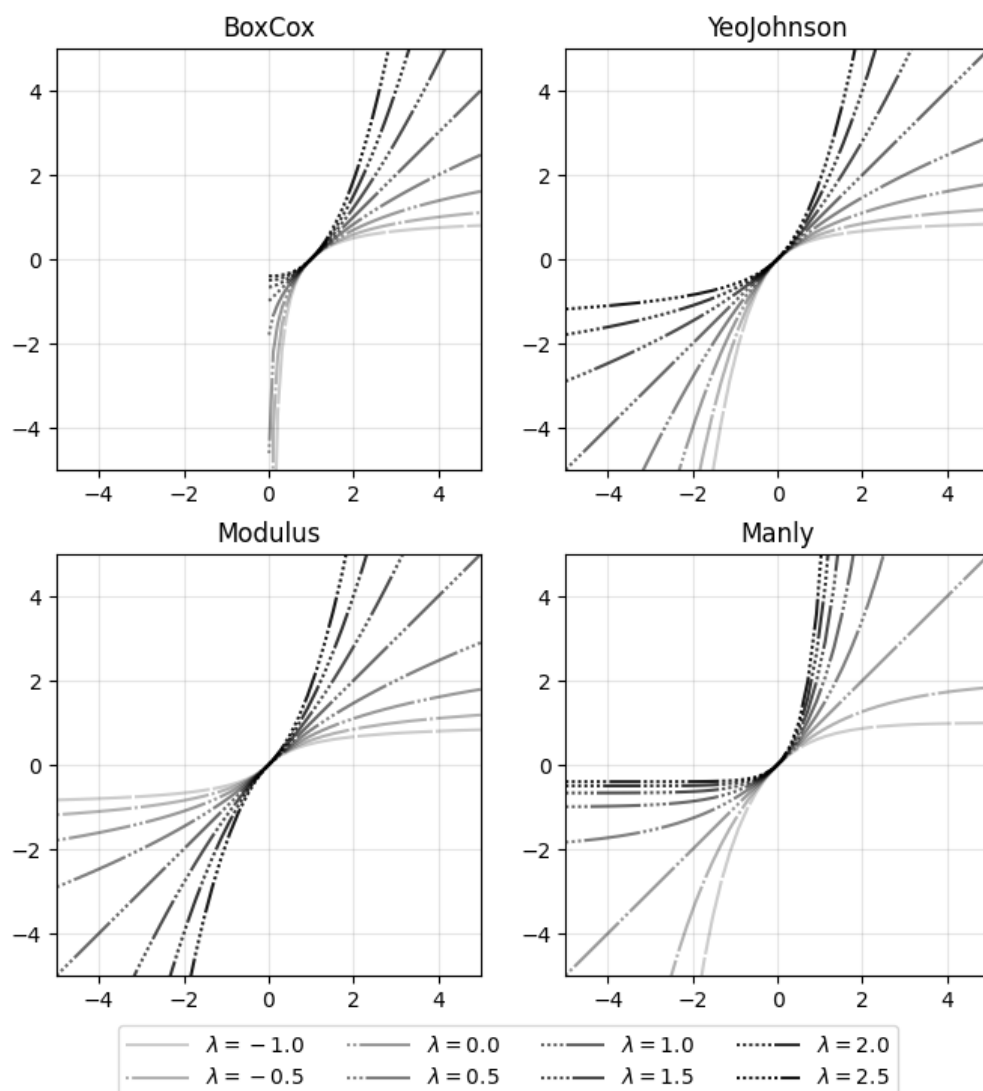
For the shape parameter λ , we create a list of 8 values ranging from -1 to 2.5.


```
lmbdas = [i * 0.5 for i in range(-2, 6)]
normalizers = [
    gs.normalizer.BoxCox,
    gs.normalizer.YeoJohnson,
    gs.normalizer.Modulus,
    gs.normalizer.Manly,
]
```

Let's plot them!

```
fig, ax = plt.subplots(2, 2, figsize=[8, 8])
for i, norm in enumerate(normalizers):
    # correctly setting the data range
    x_rng = norm().normalize_range
    x = np.linspace(max(-5, x_rng[0] + 0.01), min(5, x_rng[1] - 0.01))
    for j, lambda in enumerate(lmbdas):
        ax.flat[i].plot(
            x,
            norm(lmbda=lambda).normalize(x),
            label=r"$\lambda=" + str(lambda) + "$",
            color="k",
            alpha=0.2 + j * 0.1,
            dashes=dashes(j),
        )
    # axis formatting
    ax.flat[i].grid(which="both", color="grey", linestyle="-", alpha=0.2)
    ax.flat[i].set_ylim((-5, 5))
    ax.flat[i].set_xlim((-5, 5))
    ax.flat[i].set_title(norm().name)
# figure formatting
handles, labels = ax.flat[-1].get_legend_handles_labels()
fig.legend(handles, labels, loc="lower center", ncol=4, handlelength=3.0)
fig.suptitle("Normalizer Comparison", fontsize=20)
fig.show()
```

Normalizer Comparison



The missing *LogNormal* transformation is covered by the *BoxCox* transformation for $\lambda=0$. The *BoxCoxShift* transformation is simply the *BoxCox* transformation shifted on the X-axis.

Total running time of the script: (0 minutes 0.340 seconds)

2.11 Miscellaneous Tutorials

More examples which do not really fit into other categories. Some are not more than a code snippet, while others are more complex and more than one part of GSTools is involved.

Examples

Truncated Power Law Variograms

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

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

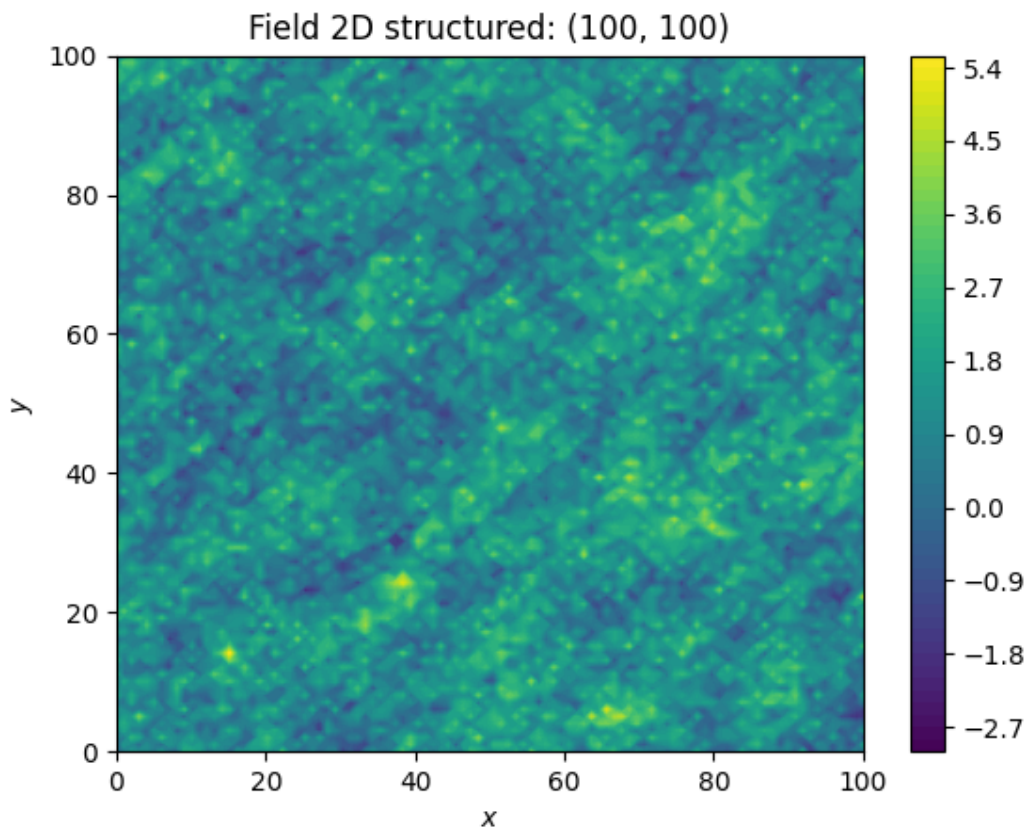
with *Stable* modes on each scale:

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

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

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

$$\begin{aligned} \gamma_{\ell_{\text{up}}}(r) &= \sigma_{\ell_{\text{up}}}^2 \cdot \left(1 - \frac{2H}{\alpha} \cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell_{\text{up}}}\right)^\alpha\right]\right) \\ \sigma_{\ell_{\text{up}}}^2 &= C \cdot \frac{\ell_{\text{up}}^{2H}}{2H} \end{aligned}$$



```
import numpy as np
import gstools as gs

x = y = np.linspace(0, 100, 100)
model = gs.TPLStable(
    dim=2, # spatial dimension
    var=1, # variance (C is calculated internally, so variance is actually 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 = gs.SRF(model, mean=1.0, seed=19970221)
srf.structured([x, y])
srf.plot()
```

Total running time of the script: (0 minutes 15.298 seconds)

Exporting Fields

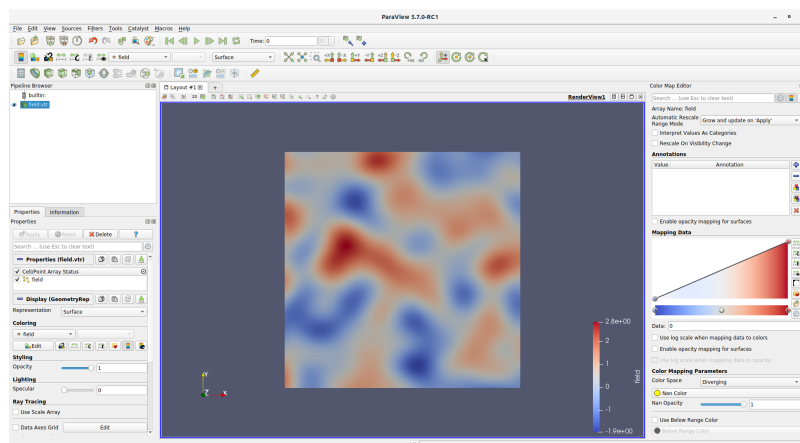
GSTools provides simple exporting routines to convert generated fields to VTK files.

These can be viewed for example with Paraview.

```
import gstools as gs

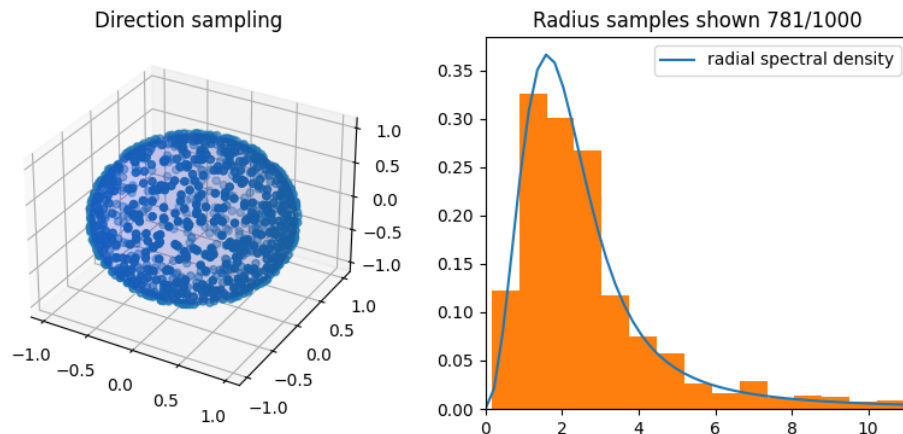
x = y = range(100)
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.SRF(model)
field = srf((x, y), mesh_type="structured")
srf.vtk_export(filename="field")
```

The result displayed with Paraview:



Total running time of the script: (0 minutes 0.482 seconds)

Check Random Sampling



```
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import gstools as gs

def norm_rad(vec):
    """Direction on the unit sphere."""
    vec = np.array(vec, ndmin=2)
    norm = np.zeros(vec.shape[1])
    for i in range(vec.shape[0]):
        norm += vec[i] ** 2
    norm = np.sqrt(norm)
    return np.einsum("j,ij->ij", 1 / norm, vec), norm

def plot_rand_meth_samples(generator):
    """Plot the samples of the rand meth class."""
    norm, rad = norm_rad(generator._cov_sample)

    fig = plt.figure(figsize=(10, 4))

    if generator.model.dim == 3:
        ax = fig.add_subplot(121, projection=Axes3D.name)
        u = np.linspace(0, 2 * np.pi, 100)
        v = np.linspace(0, np.pi, 100)
        x = np.outer(np.cos(u), np.sin(v))
        y = np.outer(np.sin(u), np.sin(v))
        z = np.outer(np.ones(np.size(u)), np.cos(v))
        ax.plot_surface(x, y, z, rstride=4, cstride=4, color="b", alpha=0.1)
        ax.scatter(norm[0], norm[1], norm[2])
    elif generator.model.dim == 2:
        ax = fig.add_subplot(121)
        u = np.linspace(0, 2 * np.pi, 100)
        x = np.cos(u)
        y = np.sin(u)
        ax.plot(x, y, color="b", alpha=0.1)
        ax.scatter(norm[0], norm[1])
        ax.set_aspect("equal")
```

(continues on next page)

(continued from previous page)

```
else:
    ax = fig.add_subplot(121)
    ax.bar(-1, np.sum(np.isclose(norm, -1)), color="C0")
    ax.bar(1, np.sum(np.isclose(norm, 1)), color="C0")
    ax.set_xticks([-1, 1])
    ax.set_xticklabels(["-1", "1"])
    ax.set_title("Direction sampling")

    ax = fig.add_subplot(122)
    x = np.linspace(0, 10 / generator.model.integral_scale)
    y = generator.model.spectral_rad_pdf(x)
    ax.plot(x, y, label="radial spectral density")
    sample_in = np.sum(rad <= np.max(x))
    ax.hist(rad[rad <= np.max(x)], bins=sample_in // 50, density=True)
    ax.set_xlim([0, np.max(x)])
    ax.set_title("Radius samples shown {}/{}".format(sample_in, len(rad)))
    ax.legend()
    plt.show()

model = gs.Stable(dim=3, alpha=1.5)
srf = gs.SRF(model, seed=2020)
plot_rand_meth_samples(srf.generator)
```

Total running time of the script: (0 minutes 3.630 seconds)

Analyzing the Herten Aquifer with GSTools

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 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.

You don't have to execute the `download_herten` and `generate_transmissivity` functions, since the only produce the `herten_transmissivity.gz` and `grid_dim_origin_spacing.txt`, which are already present.

```
import os
import numpy as np
import matplotlib.pyplot as plt
import gstools as gs

VTK_PATH = os.path.join("Herten-analog", "sim-big_1000x1000x140", "sim.vtk")
```

```
def download_herten():
    """Download the data, warning: its about 250MB."""
    import zipfile
    import urllib.request

    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 "big" simulation
    with zipfile.ZipFile(data_filename, "r") as zf:
        zf.extract(VTK_PATH)
```

```
def generate_transmissivity():
    """Generate a file with a transmissivity field from the HERTEN data."""
    import pyvista as pv
    import shutil

    print("Loading Herten data with pyvista")
    mesh = pv.read(VTK_PATH)
    herten = mesh.point_arrays["facies"].reshape(mesh.dimensions, order="F")
    # conductivity values per fazies from the supplementary data
    cond = 1e-4 * np.array(
        [2.5, 2.3, 0.61, 260, 1300, 950, 0.43, 0.006, 23, 1.4]
    )
    # assign the conductivities to the facies
    herten_cond = cond[herten]
    # Next, we are going to calculate the transmissivity,
    # by integrating over the vertical axis
    herten_trans = np.sum(herten_cond, axis=2) * mesh.spacing[2]
    # saving some grid informations
    grid = [mesh.dimensions[:2], mesh.origin[:2], mesh.spacing[:2]]
    print("Saving the transmissivity field and grid information")
    np.savetxt("herten_transmissivity.gz", herten_trans)
    np.savetxt("grid_dim_origin_spacing.txt", grid)
    # Some cleanup. You can comment out these lines to keep the downloaded data
    os.remove("data.zip")
    shutil.rmtree("Herten-analog")
```

Downloading and Preprocessing

You can uncomment the following two calls, so the data is downloaded and processed again.

```
# download_herten()
# generate_transmissivity()
```

Analyzing the data

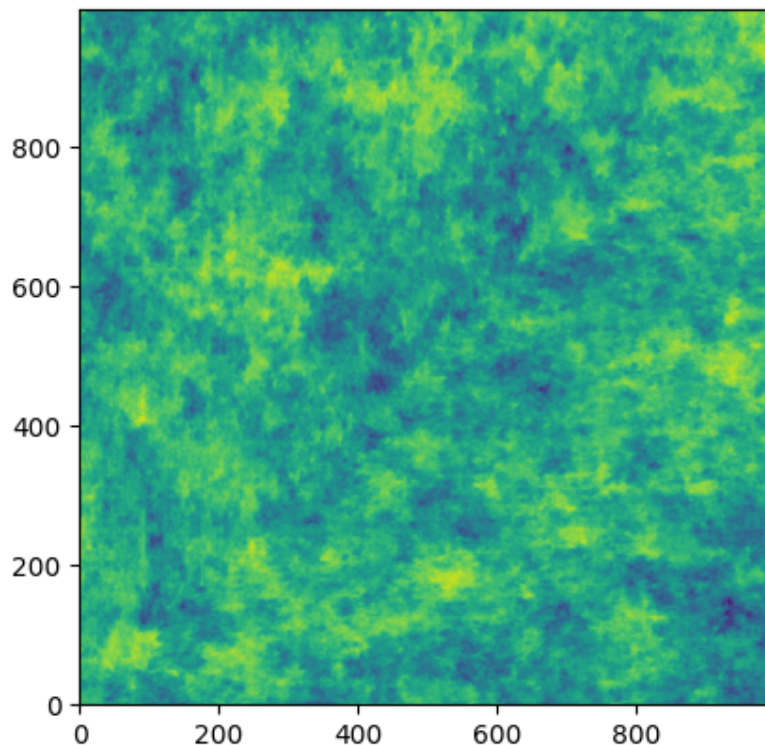
The Herten data provides information about the grid, which was already used in the previous code block. From this information, we can create our own grid on which we can estimate the variogram. As a first step, we are going to estimate an isotropic variogram, meaning that we will take point pairs from 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

```
herten_log_trans = np.log(np.loadtxt("herten_transmissivity.gz"))
dim, origin, spacing = np.loadtxt("grid_dim_origin_spacing.txt")

# create a structured grid on which the data is defined
x_s = np.arange(origin[0], origin[0] + dim[0] * spacing[0], spacing[0])
y_s = np.arange(origin[1], origin[1] + dim[1] * spacing[1], spacing[1])
# create the corresponding unstructured grid for the variogram estimation
x_u, y_u = np.meshgrid(x_s, y_s)
```

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

```
plt.imshow(herten_log_trans.T, origin="lower", aspect="equal")
plt.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.

```
bins = gs.standard_bins(pos=(x_u, y_u), max_dist=10)
bin_center, gamma = gs.vario_estimate(
    (x_u, y_u),
    herten_log_trans.reshape(-1),
    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.

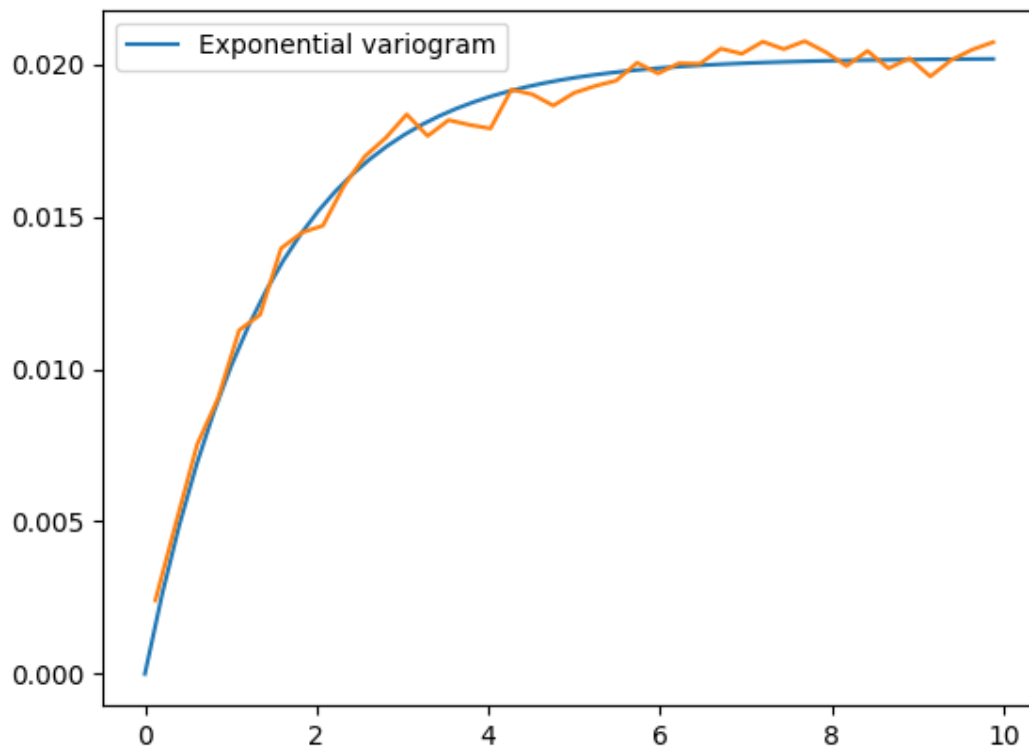
Fitting the Variogram

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

```
# fit an exponential model
fit_model = gs.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.

```
ax = fit_model.plot(x_max=max(bin_center))
ax.plot(bin_center, gamma)
```



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

```
print(fit_model)
```

Out:

```
Exponential(dim=2, var=0.0202, len_scale=1.45, nugget=0.0)
```

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.

```
# estimate the variogram on a structured grid
# use only every 10th value, otherwise calculations would take very long
x_s_skip = np.ravel(x_s)[::10]
y_s_skip = np.ravel(y_s)[::10]
herten_trans_skip = herten_log_trans[::10, ::10]
```

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

```
gamma_x = gs.vario_estimate_axis(herten_trans_skip, direction="x")
gamma_y = gs.vario_estimate_axis(herten_trans_skip, direction="y")
```

With these two estimated variograms, we can start fitting [Exponential](#) covariance models

```

x_plot = x_s_skip[:21]
y_plot = y_s_skip[:21]
# fit an exponential model
fit_model_x = gs.Exponential(dim=2)
fit_model_x.fit_variogram(x_plot, gamma_x[:21], nugget=False)
fit_model_y = gs.Exponential(dim=2)
fit_model_y.fit_variogram(y_plot, gamma_y[:21], 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.

```

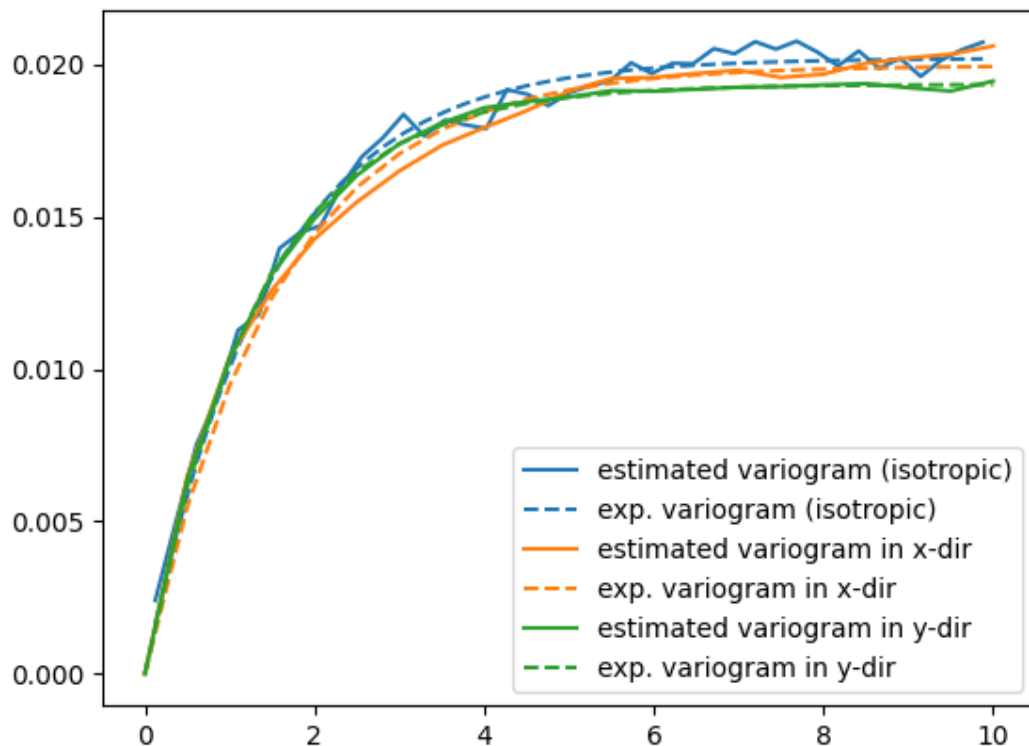
plt.figure() # new figure
(line,) = plt.plot(bin_center, gamma, label="estimated variogram (isotropic)")
plt.plot(
    bin_center,
    fit_model.variogram(bin_center),
    color=line.get_color(),
    linestyle="--",
    label="exp. variogram (isotropic)",
)

(line,) = plt.plot(x_plot, gamma_x[:21], label="estimated variogram in x-dir")
plt.plot(
    x_plot,
    fit_model_x.variogram(x_plot),
    color=line.get_color(),
    linestyle="--",
    label="exp. variogram in x-dir",
)

(line,) = plt.plot(y_plot, gamma_y[:21], label="estimated variogram in y-dir")
plt.plot(
    y_plot,
    fit_model_y.variogram(y_plot),
    color=line.get_color(),
    linestyle="--",
    label="exp. variogram in y-dir",
)

plt.legend()
plt.show()

```



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.

```
print("semivariogram model (isotropic):\n", fit_model)
print("semivariogram model (in x-dir):\n", fit_model_x)
print("semivariogram model (in y-dir):\n", fit_model_y)
```

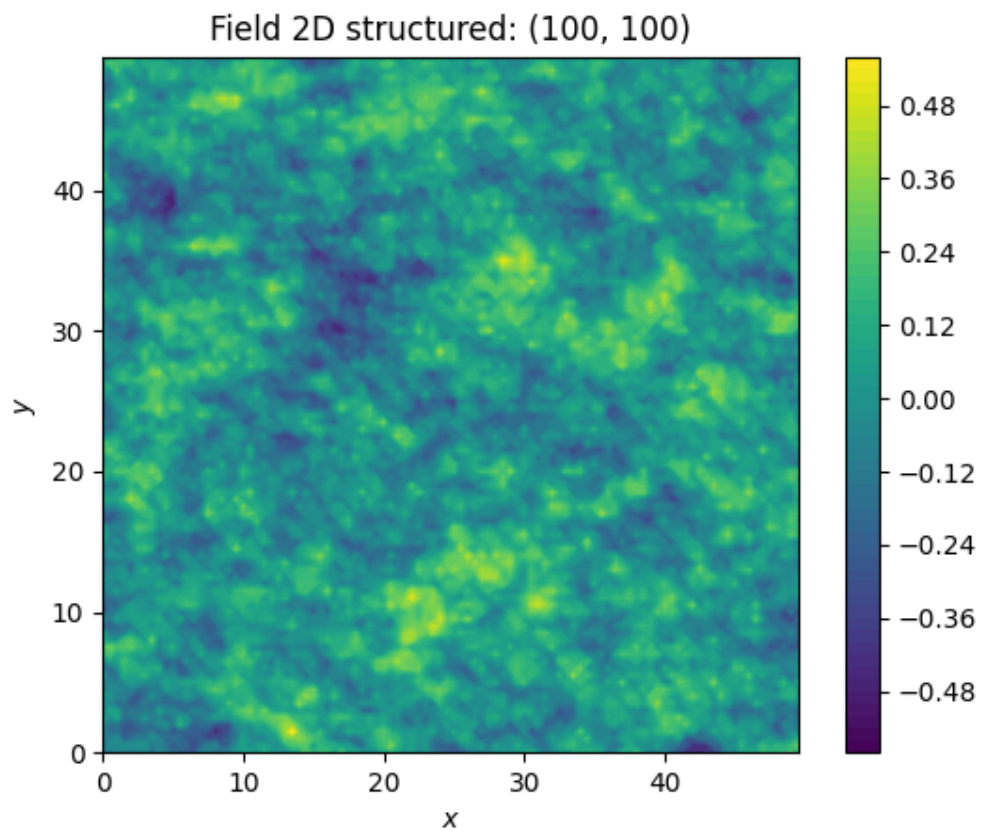
Out:

```
semivariogram model (isotropic):
  Exponential(dim=2, var=0.0202, len_scale=1.45, nugget=0.0)
semivariogram model (in x-dir.):
  Exponential(dim=2, var=0.0199, len_scale=1.55, nugget=0.0)
semivariogram model (in y-dir.):
  Exponential(dim=2, var=0.0193, len_scale=1.31, nugget=0.0)
```

Creating a Spatial Random Field from the Herten Parameters

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

```
# create a spatial random field on the low-resolution grid
srf = gs.SRF(fit_model, seed=19770928)
srf.structured([x_s_skip, y_s_skip])
ax = srf.plot()
ax.set_aspect("equal")
```



That's pretty neat!

Total running time of the script: (0 minutes 3.573 seconds)

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

<i>covmodel</i>	GStools subpackage providing a set of handy covariance models.
<i>field</i>	GStools subpackage providing tools for spatial random fields.
<i>variogram</i>	GStools subpackage providing tools for estimating and fitting variograms.
<i>krige</i>	GStools subpackage providing kriging.
<i>random</i>	GStools subpackage for random number generation.
<i>tools</i>	GStools subpackage providing miscellaneous tools.
<i>transform</i>	GStools subpackage providing transformations to post-process normal fields.
<i>normalizer</i>	GStools subpackage providing normalization routines.

3.3 Classes

Kriging

Swiss-Army-Knife for Kriging. For short cut classes see: [*gstools.krige*](#)

<i>Krige</i> (model, cond_pos, cond_val[, ...])	A Swiss Army knife for kriging.
---	---------------------------------

Spatial Random Field

Classes for (conditioned) random field generation

<i>SRF</i> (model[, mean, normalizer, trend, ...])	A class to generate spatial random fields (SRF).
<i>CondSRF</i> (krige[, generator])	A class to generate conditioned spatial random fields (SRF).

Covariance Base-Class

Class to construct user defined covariance models

<i>CovModel</i> ([dim, var, len_scale, nugget, ...])	Base class for the GStools covariance models.
--	---

Covariance Models

Standard Covariance Models

<i>Gaussian</i> ([dim, var, len_scale, nugget, ...])	The Gaussian covariance model.
<i>Exponential</i> ([dim, var, len_scale, nugget, ...])	The Exponential covariance model.
<i>Matern</i> ([dim, var, len_scale, nugget, anis, ...])	The Matérn covariance model.
<i>Stable</i> ([dim, var, len_scale, nugget, anis, ...])	The stable covariance model.
<i>Rational</i> ([dim, var, len_scale, nugget, ...])	The rational quadratic covariance model.
<i>Cubic</i> ([dim, var, len_scale, nugget, anis, ...])	The Cubic covariance model.
<i>Linear</i> ([dim, var, len_scale, nugget, anis, ...])	The bounded linear covariance model.
<i>Circular</i> ([dim, var, len_scale, nugget, ...])	The circular covariance model.
<i>Spherical</i> ([dim, var, len_scale, nugget, ...])	The Spherical covariance model.
<i>HyperSpherical</i> ([dim, var, len_scale, ...])	The Hyper-Spherical covariance model.
<i>SuperSpherical</i> ([dim, var, len_scale, ...])	The Super-Spherical covariance model.
<i>JBessel</i> ([dim, var, len_scale, nugget, anis, ...])	The J-Bessel hole model.

Truncated Power Law Covariance Models

<i>TPLGaussian</i> ([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Gaussian modes.
<i>TPLExponential</i> ([dim, var, len_scale, ...])	Truncated-Power-Law with Exponential modes.
<i>TPLStable</i> ([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Stable modes.
<i>TPLSimple</i> ([dim, var, len_scale, nugget, ...])	The simply truncated power law model.

3.4 Functions

VTK-Export

Routines to export fields to the vtk format

<i>vtk_export</i> (filename, pos, fields[, mesh_type])	Export a field to vtk.
<i>to_vtk</i> (pos, fields[, mesh_type])	Create a VTK/PyVista grid.

Geometric

Some convenient functions for geometric operations

<code>rotated_main_axes(dim, angles)</code>	Create list of the main axis defined by the given system rotations.
<code>generate_grid(pos)</code>	Generate grid from a structured position tuple.
<code>generate_st_grid(pos, time[, mesh_type])</code>	Generate spatio-temporal grid from a position tuple and time array.

Variogram Estimation

Estimate the variogram of a given field with these routines

<code>vario_estimate(pos, field[, bin_edges, ...])</code>	Estimates the empirical variogram.
<code>vario_estimate_axis(field[, direction, ...])</code>	Estimates the variogram along array axis.
<code>standard_bins([pos, dim, latlon, mesh_type, ...])</code>	Get standard binning.

3.5 Misc

<code>EARTH_RADIUS</code>	earth radius for WGS84 ellipsoid in km
---------------------------	--

3.6 gstools.covmodel

GStools subpackage providing a set of handy covariance models.

Subpackages

<code>plot</code>	GStools subpackage providing plotting routines for the covariance models.
-------------------	---

Covariance Base-Class

Class to construct user defined covariance models

<code>CovModel</code> ([dim, var, len_scale, nugget, ...])	Base class for the GStools covariance models.
--	---

gstools.covmodel.CovModel

```
class gstools.covmodel.CovModel(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                                integral_scale=None, rescale=None, latlon=False, 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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** (`float`, optional) – nugget of the model. Default: 0.0
- **anis** (`float` or `list`, optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - $e_y = l_y / l_x$
 - $e_z = l_z / l_x$If only one value is given in 3D, *e_y* will be set to 1. This value will be ignored, if multiple *len_scales* are given. Default: 1.0
- **angles** (`float` or `list`, optional) – angles of rotation (given in rad):
 - 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`

- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, `dim` will be set to 3 and anisotropy will be disabled. `rescale` can be set to e.g. earth's radius, to have a meaningful `len_scale` parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<i>anisometrize</i> (pos)	Bring a position tuple into the anisotropic coordinate-system.
<i>calc_integral_scale</i> ()	Calculate the integral scale of the isotrope model.
<i>check_arg_bounds</i> ()	Check arguments to be within their given bounds.
<i>check_dim</i> (dim)	Check the given dimension.
<i>check_opt_arg</i> ()	Run checks for the optional arguments.
<i>cor_axis</i> (r[, axis])	Correlation along axis of anisotropy.
<i>cor_spatial</i> (pos)	Spatial correlation respecting anisotropy and rotation.
<i>cor_yadrenko</i> (zeta)	Yadrenko correlation for great-circle distance from latlon-pos.
<i>cov_axis</i> (r[, axis])	Covariance along axis of anisotropy.
<i>cov_nugget</i> (r)	Isotropic covariance of the model respecting the nugget at r=0.
<i>cov_spatial</i> (pos)	Spatial covariance respecting anisotropy and rotation.
<i>cov_yadrenko</i> (zeta)	Yadrenko covariance for great-circle distance from latlon-pos.
<i>default_arg_bounds</i> ()	Provide default boundaries for arguments.
<i>default_opt_arg</i> ()	Provide default optional arguments by the user.
<i>default_opt_arg_bounds</i> ()	Provide default boundaries for optional arguments.
<i>default_rescale</i> ()	Provide default rescaling factor.
<i>fit_variogram</i> (x_data, y_data[, anis, sill, ...])	Fitting the variogram-model to an empirical variogram.
<i>fix_dim</i> ()	Set a fix dimension for the model.
<i>isometrize</i> (pos)	Make a position tuple ready for isotropic operations.

continues on next page

Table 13 – continued from previous page

<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(r)

Isotropic covariance of the model respecting the nugget at r=0.

cov_spatial(pos)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(zeta)

Yadrenko covariance for great-circle distance from latlon-pos.

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 when overridden.

default_opt_arg_bounds()

Provide default boundaries for optional arguments.

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis=True*, *sill=None*, *init_guess='default'*, *weights=None*,
method='trf', *loss='soft_l1'*, *max_eval=None*, *return_r2=False*,
curve_fit_kwargs=None, ***para_select*)

Fiting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The messured variogram If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios wont be altered during fitting. Default: True
- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:
 - $\text{sill} = \text{var} + \text{nugget}$
 - if the variance is bigger than the sill, nugget will bet set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deslected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can bet set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:
 - ‘inv’: inverse distance $1 / (\text{x_data} + 1)$
 - list: weights given per bin
 - callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, *optional*) – Algorithm to perform minimization.
 - `'trf'` : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - `'dogbox'` : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: `'trf'`

- **loss** (`str` or callable, *optional*) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:
 - `'linear'` (default) : $\rho(z) = z$. Gives a standard least-squares problem.
 - `'soft_l1'` : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
 - `'huber'` : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to `'soft_l1'`.
 - `'cauchy'` : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
 - `'arctan'` : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to `'cauchy'`.

If callable, it must take a 1-d ndarray $z=f^2$ and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: `'soft_l1'`

- **max_eval** (`int` or `None`, *optional*) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, *optional*) – Whether to return the r2 score of the estimation. Default: `False`
- **curve_fit_kwargs** (`dict`, *optional*) – Other keyword arguments passed to `scipys curve_fit`. Default: `None`
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
- **r2_score** (`float`, *optional*) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(*pos*)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(*r*)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool*, optional) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r*, *axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

property_angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property_anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property_anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [*a*, *b*] or [*a*, *b*, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list` of `str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list` of `float`

property dim

The dimension of the model.

Type `int`

property dist_func

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property do_rotation

State if a rotation is performed.

Type `bool`

property field_dim

The field dimension of the model.

Type `int`

property hankel_kw

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property has_cdf

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property `is_isotropic`

State if a model is isotropic.

Type `bool`

property `iso_arg`

Names of isotropic arguments.

Type `list` of `str`

property `iso_arg_list`

Values of isotropic arguments.

Type `list` of `float`

property `latlon`

Whether the model depends on geographical coords.

Type `bool`

property `len_rescaled`

The rescaled main length scale of the model.

Type `float`

property `len_scale`

The main length scale of the model.

Type `float`

property `len_scale_bounds`

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("`o`") or closed ("`c`").

Type `list`

property `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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$
-

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

Covariance Models

Standard Covariance Models

<code>Gaussian</code> ([dim, var, len_scale, nugget, ...])	The Gaussian covariance model.
<code>Exponential</code> ([dim, var, len_scale, nugget, ...])	The Exponential covariance model.
<code>Matern</code> ([dim, var, len_scale, nugget, anis, ...])	The Matérn covariance model.
<code>Stable</code> ([dim, var, len_scale, nugget, anis, ...])	The stable covariance model.
<code>Rational</code> ([dim, var, len_scale, nugget, ...])	The rational quadratic covariance model.
<code>Cubic</code> ([dim, var, len_scale, nugget, anis, ...])	The Cubic covariance model.
<code>Linear</code> ([dim, var, len_scale, nugget, anis, ...])	The bounded linear covariance model.
<code>Circular</code> ([dim, var, len_scale, nugget, ...])	The circular covariance model.
<code>Spherical</code> ([dim, var, len_scale, nugget, ...])	The Spherical covariance model.
<code>HyperSpherical</code> ([dim, var, len_scale, ...])	The Hyper-Spherical covariance model.
<code>SuperSpherical</code> ([dim, var, len_scale, ...])	The Super-Spherical covariance model.
<code>JBessel</code> ([dim, var, len_scale, nugget, anis, ...])	The J-Bessel hole model.

`gstools.covmodel.Gaussian`

```
class gstools.covmodel.Gaussian(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,  
                                integral_scale=None, rescale=None, latlon=False, 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 variogram [Webster2007]:

$$\gamma(r) = \sigma^2 \left(1 - \exp \left(- \left(s \cdot \frac{r}{\ell} \right)^2 \right) \right) + n$$

Where the standard rescale factor is $s = \frac{\sqrt{\pi}}{2}$.

References

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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** (`float`, optional) – nugget of the model. Default: 0.0
- **anis** (`float` or `list`, optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - *e_y* = *l_y* / *l_x*
 - *e_z* = *l_z* / *l_x*

If only one value is given in 3D, `e_y` will be set to 1. This value will be ignored, if multiple `len_scales` are given. Default: 1.0

- **angles** (`float` or `list`, optional) – angles of rotation (given in rad):
 - 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`
- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, `dim` will be set to 3 and anisotropy will be disabled. `rescale` can be set to e.g. earth's radius, to have a meaningful `len_scale` parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg list of *str*: Names of isotropic arguments.

iso_arg_list list of *float*: Values of isotropic arguments.

latlon *bool*: Whether the model depends on geographical coords.

len_rescaled *float*: The rescaled main length scale of the model.

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

len_scale_bounds *list*: Bounds for the length 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.

rescale *float*: Rescale factor for the length scale of the model.

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

<i>anisometrize</i> (pos)	Bring a position tuple into the anisotropic coordinate-system.
<i>calc_integral_scale</i> ()	Calculate the integral scale of the isotrope model.
<i>check_arg_bounds</i> ()	Check arguments to be within their given bounds.
<i>check_dim</i> (dim)	Check the given dimension.
<i>check_opt_arg</i> ()	Run checks for the optional arguments.
<i>cor</i> (h)	Gaussian normalized correlation function.
<i>cor_axis</i> (r[, axis])	Correlation along axis of anisotropy.
<i>cor_spatial</i> (pos)	Spatial correlation respecting anisotropy and rotation.
<i>cor_yadrenko</i> (zeta)	Yadrenko correlation for great-circle distance from latlon-pos.
<i>correlation</i> (r)	Correlation function of the model.
<i>cov_axis</i> (r[, axis])	Covariance along axis of anisotropy.

continues on next page

Table 15 – continued from previous page

<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at $r=0$.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Provide default optional arguments by the user.
<code>default_opt_arg_bounds()</code>	Provide default boundaries for optional arguments.
<code>default_rescale()</code>	Gaussian rescaling factor to result in integral scale.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fitting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_cdf(r)</code>	Gaussian radial spectral cdf.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectral_rad_ppf(u)</code>	Gaussian radial spectral ppf.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at $r=0$.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(*h*)

Gaussian normalized correlation function.

cor_axis(*r*, *axis*=0)

Correlation along axis of anisotropy.

cor_spatial(*pos*)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(*zeta*)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(*r*)

Correlation function of the model.

cov_axis(*r*, *axis*=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

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 when overridden.

default_opt_arg_bounds()

Provide default boundaries for optional arguments.

default_rescale()

Gaussian rescaling factor to result in integral scale.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True

- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

- $\text{sill} = \text{var} + \text{nugget}$
- if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: `None`

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (\mathbf{x_data} + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: `None`

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’: Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’: dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipy.curve_fit`. The following keyword values are allowed:

- ‘linear’ (default): $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’: $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’: $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’: $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’: $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r^2 score of the estimation. Default: `False`
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: `None`
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters won't be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
- **r2_score** (`float`, optional) – r^2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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”
 - “vario_yadrenko”
 - “cov_yadrenko”

- "cor_yadrenko"
- "vario_axis"
- "cov_axis"
- "cor_axis"
- "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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_cdf(*r*)

Gaussian radial spectral cdf.

spectral_rad_pdf(*r*)

Radial spectral density of the model.

spectral_rad_ppf(*u*)

Gaussian radial spectral ppf.

Notes

Not defined for 3D.

spectrum(*k*)

Spectrum of the covariance model.

This is given by:

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

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_axis(*r*, *axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [*a*, *b*] or [*a*, *b*, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list` of `str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [*a*, *b*] or [*a*, *b*, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property `has_ppf`

State if a ppf is defined by the user.

Type `bool`

property `integral_scale`

The main integral scale of the model.

Raises **`ValueError`** – If integral scale is not settable.

Type `float`

property `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`

property `is_isotropic`

State if a model is isotropic.

Type `bool`

property `iso_arg`

Names of isotropic arguments.

Type `list` of `str`

property iso_arg_list

Values of isotropic arguments.

Type `list` of `float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the length scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `opt_arg`

Names of the optional arguments.

Type `list` of `str`

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

Type `dict`

property `pykrige_angle`

2D rotation angle for pykrige.

property `pykrige_angle_x`

3D rotation angle around x for pykrige.

property `pykrige_angle_y`

3D rotation angle around y for pykrige.

property `pykrige_angle_z`

3D rotation angle around z for pykrige.

property `pykrige_anis`

2D anisotropy ratio for pykrige.

property `pykrige_anis_y`

3D anisotropy ratio in y direction for pykrige.

property `pykrige_anis_z`

3D anisotropy ratio in z direction for pykrige.

property `pykrige_kwargs`

Keyword arguments for pykrige routines.

property `rescale`

Rescale factor for the length scale of the model.

Type `float`

property `sill`

The sill of the variogram.

Notes

This is calculated by:

- `sill = variance + nugget`
-

Type `float`

property `var`

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.Exponential

```
class gstools.covmodel.Exponential(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0,
                                   angles=0.0, integral_scale=None, rescale=None, latlon=False,
                                   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 variogram [Webster2007]:

$$\gamma(r) = \sigma^2 \left(1 - \exp \left(-s \cdot \frac{r}{\ell} \right) \right) + n$$

Where the standard rescale factor is $s = 1$.

References

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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [e_y, e_z].
 - $e_y = l_y / l_x$
 - $e_z = l_z / l_x$

If only one value is given in 3D, e_y will be set to 1. This value will be ignored, if multiple len_scales are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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](#)
- **rescale** ([float](#) or [None](#), optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: [None](#)
- **latlon** ([bool](#), optional) – Whether the model is describing 2D fields on earths surface described by latitude and longitude. When using this, the model will internally use the associated ‘Yadrenko’ model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal

to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth's radius, to have a meaningful *len_scale* parameter. Default: False

- **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
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds *list*: Bounds for the nugget.

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

arg_bounds *dict*: Bounds for all parameters.

arg_list *list* of *float*: Values of all arguments.

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.

field_dim *int*: The field dimension of the model.

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.

is_isotropic *bool*: State if a model is isotropic.

iso_arg *list* of *str*: Names of isotropic arguments.

iso_arg_list *list* of *float*: Values of isotropic arguments.

latlon *bool*: Whether the model depends on geographical coords.

len_rescaled *float*: The rescaled main length scale of the model.

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

len_scale_bounds *list*: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<i>anisometrize</i> (pos)	Bring a position tuple into the anisotropic coordinate-system.
<i>calc_integral_scale</i> ()	Calculate the integral scale of the isotrope model.
<i>check_arg_bounds</i> ()	Check arguments to be within their given bounds.
<i>check_dim</i> (dim)	Check the given dimension.
<i>check_opt_arg</i> ()	Run checks for the optional arguments.
<i>cor</i> (h)	Exponential normalized correlation function.
<i>cor_axis</i> (r[, axis])	Correlation along axis of anisotropy.
<i>cor_spatial</i> (pos)	Spatial correlation respecting anisotropy and rotation.
<i>cor_yadrenko</i> (zeta)	Yadrenko correlation for great-circle distance from latlon-pos.
<i>correlation</i> (r)	Correlation function of the model.
<i>cov_axis</i> (r[, axis])	Covariance along axis of anisotropy.
<i>cov_nugget</i> (r)	Isotropic covariance of the model respecting the nugget at r=0.
<i>cov_spatial</i> (pos)	Spatial covariance respecting anisotropy and rotation.
<i>cov_yadrenko</i> (zeta)	Yadrenko covariance for great-circle distance from latlon-pos.
<i>covariance</i> (r)	Covariance of the model.
<i>default_arg_bounds</i> ()	Provide default boundaries for arguments.
<i>default_opt_arg</i> ()	Provide default optional arguments by the user.
<i>default_opt_arg_bounds</i> ()	Provide default boundaries for optional arguments.
<i>default_rescale</i> ()	Provide default rescaling factor.
<i>fit_variogram</i> (x_data, y_data[, anis, sill, ...])	Fiting the variogram-model to an empirical variogram.
<i>fix_dim</i> ()	Set a fix dimension for the model.
<i>isometrize</i> (pos)	Make a position tuple ready for isotropic operations.
<i>ln_spectral_rad_pdf</i> (r)	Log radial spectral density of the model.
<i>main_axes</i> ()	Axes of the rotated coordinate-system.
<i>percentile_scale</i> ([per])	Calculate the percentile scale of the isotrope model.
<i>plot</i> ([func])	Plot a function of a the CovModel.

continues on next page

Table 16 – continued from previous page

<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_cdf(r)</code>	Exponential radial spectral cdf.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectral_rad_ppf(u)</code>	Exponential radial spectral ppf.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at $r=0$.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(h)

Exponential normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(r)

Isotropic covariance of the model respecting the nugget at $r=0$.

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

cov_yadrenko(*zeta*)
Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)
Covariance of the model.

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 when overridden.

default_opt_arg_bounds()
Provide default boundaries for optional arguments.

default_rescale()
Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis=True*, *sill=None*, *init_guess='default'*, *weights=None*, *method='trf'*, *loss='soft_l1'*, *max_eval=None*, *return_r2=False*, *curve_fit_kwargs=None*, ***para_select*)
Fiting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The messured variogram If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios wont be altered during fitting. Default: True
- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:
 - $\text{sill} = \text{var} + \text{nugget}$
 - if the variance is bigger than the sill, nugget will bet set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deslected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can bet set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$

- list: weights given per bin

- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.

- ‘trf’: Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.

- ‘dogbox’: dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default): $\rho(z) = z$. Gives a standard least-squares problem.

- ‘soft_l1’: $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.

- ‘huber’: $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.

- ‘cauchy’: $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.

- ‘arctan’: $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If None (default), the value is chosen automatically: $100 * n$.

- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False

- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None

- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values

- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** (`float`, optional) – r2 score of the curve fitting results. Only if `return_r2` is True.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(check_args=True, **kwargs)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool*, *optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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_cdf(r)

Exponential radial spectral cdf.

spectral_rad_pdf(r)

Radial spectral density of the model.

spectral_rad_ppf(u)

Exponential radial spectral ppf.

Notes

Not defined for 3D.

spectrum(k)

Spectrum of the covariance model.

This is given by:

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

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_axis(r, axis=0)

Variogram along axis of anisotropy.

vario_nugget(r)

Isotropic variogram of the model respecting the nugget at r=0.

vario_spatial(pos)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(zeta)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(r)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list of str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list of float`

property dim

The dimension of the model.

Type `int`

property dist_func

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple of callable`

property do_rotation

State if a rotation is performed.

Type `bool`

property field_dim

The field dimension of the model.

Type `int`

property hankel_kw

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property **has_cdf**

State if a cdf is defined by the user.

Type `bool`

property **has_ppf**

State if a ppf is defined by the user.

Type `bool`

property **integral_scale**

The main integral scale of the model.

Raises **ValueError** – If integral scale is not settable.

Type `float`

property **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`

property **is_isotropic**

State if a model is isotropic.

Type `bool`

property **iso_arg**

Names of isotropic arguments.

Type `list of str`

property **iso_arg_list**

Values of isotropic arguments.

Type `list of float`

property **latlon**

Whether the model depends on geographical coords.

Type `bool`

property **len_rescaled**

The rescaled main length scale of the model.

Type `float`

property **len_scale**

The main length scale of the model.

Type `float`

property **len_scale_bounds**

Bounds for the length scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list` of `str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.Matern

```
class gstools.covmodel.Matern(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                              integral_scale=None, rescale=None, latlon=False, 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 [Rasmussen2003]:

$$\rho(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \cdot \left(\sqrt{\nu} \cdot s \cdot \frac{r}{\ell} \right)^\nu \cdot K_\nu \left(\sqrt{\nu} \cdot s \cdot \frac{r}{\ell} \right)$$

Where the standard rescale factor is $s = 1$. 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 represents the limiting case:

$$\rho(r) = \exp \left(- \left(s \cdot \frac{r}{2\ell} \right)^2 \right)$$

References**Parameters**

- **nu** ([float](#), optional) – Shape parameter. Standard range: $[0.2, 30]$ Default: `1.0`
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: `1.0`
- **nugget** ([float](#), optional) – nugget of the model. Default: `0.0`
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - *e_y* = *l_y* / *l_x*
 - *e_z* = *l_z* / *l_x*

If only one value is given in 3D, *e_y* will be set to 1. This value will be ignored, if multiple *len_scales* are given. Default: `1.0`
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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`

- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, `dim` will be set to 3 and anisotropy will be disabled. `rescale` can be set to e.g. earth's radius, to have a meaningful `len_scale` parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	Matérn normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Defaults for the optional arguments.
<code>default_opt_arg_bounds()</code>	Defaults for boundaries of the optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fitting the variogram-model to an empirical variogram.

continues on next page

Table 17 – continued from previous page

<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(h)

Matérn normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(*r*, *axis*=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

default_arg_bounds()

Provide default boundaries for arguments.

Given as a dictionary.

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": [0.2, 30.0, "cc"]}

Returns Boundaries for optional arguments

Return type dict

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (numpy.ndarray) – The bin-centers of the empirical variogram.
- **y_data** (numpy.ndarray) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (bool, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it "False". You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (float or bool, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:
 - $\text{sill} = \text{var} + \text{nugget}$
 - if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: `None`

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:
 - ‘inv’: inverse distance $1 / (x_data + 1)$
 - list: weights given per bin
 - callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: `None`

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’: Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’: dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:
 - ‘linear’ (default): $\rho(z) = z$. Gives a standard least-squares problem.
 - ‘soft_l1’: $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
 - ‘huber’: $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
 - ‘cauchy’: $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
 - ‘arctan’: $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: `False`
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: `None`

- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters won't be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
- **r2_score** (`float`, optional) – r2 score of the curve fitting results. Only if `return_r2` is True.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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”
 - “vario_yadrenko”
 - “cov_yadrenko”
 - “cor_yadrenko”
 - “vario_axis”
 - “cov_axis”
 - “cor_axis”
 - “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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list` of `str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list` of `float`

property dim

The dimension of the model.

Type `int`

property dist_func

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property do_rotation

State if a rotation is performed.

Type `bool`

property field_dim

The field dimension of the model.

Type `int`

property **hankel_kw**

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property **has_cdf**

State if a cdf is defined by the user.

Type `bool`

property **has_ppf**

State if a ppf is defined by the user.

Type `bool`

property **integral_scale**

The main integral scale of the model.

Raises **ValueError** – If integral scale is not settable.

Type `float`

property **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`

property **is_isotropic**

State if a model is isotropic.

Type `bool`

property **iso_arg**

Names of isotropic arguments.

Type `list` of `str`

property **iso_arg_list**

Values of isotropic arguments.

Type `list` of `float`

property **latlon**

Whether the model depends on geographical coords.

Type `bool`

property **len_rescaled**

The rescaled main length scale of the model.

Type `float`

property **len_scale**

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.Stable

```
class gstools.covmodel.Stable(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                             integral_scale=None, rescale=None, latlon=False, 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 [Wackernagel2003]:

$$\rho(r) = \exp\left(-\left(s \cdot \frac{r}{\ell}\right)^\alpha\right)$$

Where the standard rescale factor is $s = 1$. α is a shape parameter with $\alpha \in (0, 2]$

References**Parameters**

- **alpha** ([float](#), optional) – Shape parameter. Standard range: $(0, 2]$ Default: 1.5
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [e_y, e_z].
 - $e_y = l_y / l_x$
 - $e_z = l_z / l_x$

If only one value is given in 3D, e_y will be set to 1. This value will be ignored, if multiple len_scales are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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](#)
- **rescale** ([float](#) or [None](#), optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: [None](#)
- **latlon** ([bool](#), optional) – Whether the model is describing 2D fields on earths surface described by latitude and longitude. When using this, the model will internally use the associated ‘Yadrenko’ model to represent a valid model. This means, the spatial distance

r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, `dim` will be set to 3 and anisotropy will be disabled. `rescale` can be set to e.g. earth's radius, to have a meaningful `len_scale` parameter. Default: False

- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<i>anisometrize</i> (pos)	Bring a position tuple into the anisotropic coordinate-system.
<i>calc_integral_scale</i> ()	Calculate the integral scale of the isotrope model.
<i>check_arg_bounds</i> ()	Check arguments to be within their given bounds.
<i>check_dim</i> (dim)	Check the given dimension.
<i>check_opt_arg</i> ()	Check the optional arguments.
<i>cor</i> (h)	Stable normalized correlation function.
<i>cor_axis</i> (r[, axis])	Correlation along axis of anisotropy.
<i>cor_spatial</i> (pos)	Spatial correlation respecting anisotropy and rotation.
<i>cor_yadrenko</i> (zeta)	Yadrenko correlation for great-circle distance from latlon-pos.
<i>correlation</i> (r)	Correlation function of the model.
<i>cov_axis</i> (r[, axis])	Covariance along axis of anisotropy.
<i>cov_nugget</i> (r)	Isotropic covariance of the model respecting the nugget at r=0.
<i>cov_spatial</i> (pos)	Spatial covariance respecting anisotropy and rotation.
<i>cov_yadrenko</i> (zeta)	Yadrenko covariance for great-circle distance from latlon-pos.
<i>covariance</i> (r)	Covariance of the model.
<i>default_arg_bounds</i> ()	Provide default boundaries for arguments.
<i>default_opt_arg</i> ()	Defaults for the optional arguments.
<i>default_opt_arg_bounds</i> ()	Defaults for boundaries of the optional arguments.
<i>default_rescale</i> ()	Provide default rescaling factor.
<i>fit_variogram</i> (x_data, y_data[, anis, sill, ...])	Fitting the variogram-model to an empirical variogram.
<i>fix_dim</i> ()	Set a fix dimension for the model.
<i>isometrize</i> (pos)	Make a position tuple ready for isotropic operations.
<i>ln_spectral_rad_pdf</i> (r)	Log radial spectral density of the model.
<i>main_axes</i> ()	Axes of the rotated coordinate-system.
<i>percentile_scale</i> ([per])	Calculate the percentile scale of the isotrope model.

continues on next page

Table 18 – continued from previous page

<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at $r=0$.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

`anisometrize(pos)`

Bring a position tuple into the anisotropic coordinate-system.

`calc_integral_scale()`

Calculate the integral scale of the isotrope model.

`check_arg_bounds()`

Check arguments to be within their given bounds.

`check_dim(dim)`

Check the given dimension.

`check_opt_arg()`

Check the optional arguments.

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

`cor(h)`

Stable normalized correlation function.

`cor_axis(r, axis=0)`

Correlation along axis of anisotropy.

`cor_spatial(pos)`

Spatial correlation respecting anisotropy and rotation.

`cor_yadrenko(zeta)`

Yadrenko correlation for great-circle distance from latlon-pos.

`correlation(r)`

Correlation function of the model.

`cov_axis(r, axis=0)`

Covariance along axis of anisotropy.

`cov_nugget(r)`

Isotropic covariance of the model respecting the nugget at $r=0$.

`cov_spatial(pos)`

Spatial covariance respecting anisotropy and rotation.

`cov_yadrenko(zeta)`

Yadrenko covariance for great-circle distance from latlon-pos.

`covariance(r)`

Covariance of the model.

default_arg_bounds()

Provide default boundaries for arguments.

Given as a dictionary.

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

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fiting the variogram-model to an empirical variogram.

Parameters

- **x_data** (numpy.ndarray) – The bin-centers of the empirical variogram.
- **y_data** (numpy.ndarray) – The messured variogram If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (bool, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios wont be altered during fitting. Default: True
- **sill** (float or bool, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

– $\text{sill} = \text{var} + \text{nugget}$

– if the variance is bigger than the sill, nugget will bet set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a ValueError comes up. Same for nugget. If sill=False, it will be deslected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (str or dict, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can bet set to “default” or “current” for unspecified values to get same behavior as

given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’ : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’ : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default) : $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’ : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’ : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’ : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’ : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of `popt` from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** ([float](#), optional) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type *numpy.ndarray*

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property `anis_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `arg`

Names of all arguments.

Type `list` of `str`

property `arg_bounds`

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property `arg_list`

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list` of `str`

property iso_arg_list

Values of isotropic arguments.

Type `list` of `float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list` of `str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.Rational

```
class gstools.covmodel.Rational(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                                integral_scale=None, rescale=None, latlon=False, 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 [Rasmussen2003]:

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

Where the standard rescale factor is $s = 1$. α is a shape parameter and should be > 0.5 .

For $\alpha \rightarrow \infty$ this model converges to the Gaussian model:

$$\rho(r) = \exp\left(-\left(s \cdot \frac{r}{\ell}\right)^2\right)$$

References

Parameters

- **alpha** ([float](#), optional) – Shape parameter. Standard range: $[0.5, 50]$ Default: `1.0`
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: `1.0`
- **nugget** ([float](#), optional) – nugget of the model. Default: `0.0`
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - $e_y = l_y / l_x$
 - $e_z = l_z / l_x$

If only one value is given in 3D, *e_y* will be set to 1. This value will be ignored, if multiple *len_scales* are given. Default: `1.0`
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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`

- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, `dim` will be set to 3 and anisotropy will be disabled. `rescale` can be set to e.g. earth's radius, to have a meaningful `len_scale` parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	Rational normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Defaults for the optional arguments.
<code>default_opt_arg_bounds()</code>	Defaults for boundaries of the optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fitting the variogram-model to an empirical variogram.

continues on next page

Table 19 – continued from previous page

<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(h)

Rational normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(*r*, *axis*=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

default_arg_bounds()

Provide default boundaries for arguments.

Given as a dictionary.

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, 50.0]}

Returns Boundaries for optional arguments

Return type dict

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (numpy.ndarray) – The bin-centers of the empirical variogram.
- **y_data** (numpy.ndarray) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (bool, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it "False". You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (float or bool, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:
 - $\text{sill} = \text{var} + \text{nugget}$
 - if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: `None`

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:
 - ‘inv’: inverse distance $1 / (x_data + 1)$
 - list: weights given per bin
 - callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: `None`

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’: Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’: dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:
 - ‘linear’ (default): $\rho(z) = z$. Gives a standard least-squares problem.
 - ‘soft_l1’: $\rho(z) = 2 * ((1 + z)**0.5 - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
 - ‘huber’: $\rho(z) = z$ if $z \leq 1$ else $2*z**0.5 - 1$. Works similarly to ‘soft_l1’.
 - ‘cauchy’: $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
 - ‘arctan’: $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: `False`
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: `None`

- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
- **r2_score** (`float`, optional) – r2 score of the curve fitting results. Only if `return_r2` is True.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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”
 - “vario_yadrenko”
 - “cov_yadrenko”
 - “cor_yadrenko”
 - “vario_axis”
 - “cov_axis”
 - “cor_axis”
 - “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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list` of `str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list` of `float`

property dim

The dimension of the model.

Type `int`

property dist_func

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property do_rotation

State if a rotation is performed.

Type `bool`

property field_dim

The field dimension of the model.

Type `int`

property **hankel_kw**

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property **has_cdf**

State if a cdf is defined by the user.

Type `bool`

property **has_ppf**

State if a ppf is defined by the user.

Type `bool`

property **integral_scale**

The main integral scale of the model.

Raises **ValueError** – If integral scale is not settable.

Type `float`

property **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`

property **is_isotropic**

State if a model is isotropic.

Type `bool`

property **iso_arg**

Names of isotropic arguments.

Type `list` of `str`

property **iso_arg_list**

Values of isotropic arguments.

Type `list` of `float`

property **latlon**

Whether the model depends on geographical coords.

Type `bool`

property **len_rescaled**

The rescaled main length scale of the model.

Type `float`

property **len_scale**

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.Cubic

```
class gstools.covmodel.Cubic(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                             integral_scale=None, rescale=None, latlon=False, var_raw=None,
                             hankel_kw=None, **opt_arg)
```

Bases: [gstools.covmodel.base.CovModel](#)

The Cubic covariance model.

A model with reverse curvature near the origin and a finite range of correlation.

Notes

This model is given by the following correlation function [Chiles2009]:

$$\rho(r) = \begin{cases} 1 - 7 \left(s \cdot \frac{r}{\ell}\right)^2 + \frac{35}{4} \left(s \cdot \frac{r}{\ell}\right)^3 - \frac{7}{2} \left(s \cdot \frac{r}{\ell}\right)^5 + \frac{3}{4} \left(s \cdot \frac{r}{\ell}\right)^7 & r < \frac{\ell}{s} \\ 0 & r \geq \frac{\ell}{s} \end{cases}$$

Where the standard rescale factor is $s = 1$.

References**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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [e_y, e_z].
 - e_y = l_y / l_x
 - e_z = l_z / l_x

If only one value is given in 3D, e_y will be set to 1. This value will be ignored, if multiple len_scales are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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](#)
- **rescale** ([float](#) or [None](#), optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: [None](#)

- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth's radius, to have a meaningful *len_scale* parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	Spherical normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Provide default optional arguments by the user.
<code>default_opt_arg_bounds()</code>	Provide default boundaries for optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fiting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.

continues on next page

Table 20 – continued from previous page

<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(h)

Spherical normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

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 when overridden.

default_opt_arg_bounds()

Provide default boundaries for optional arguments.

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis=True*, *sill=None*, *init_guess='default'*, *weights=None*,
method='trf', *loss='soft_l1'*, *max_eval=None*, *return_r2=False*,
curve_fit_kwargs=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

– $\text{sill} = \text{var} + \text{nugget}$

– if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as

given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’ : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’ : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default) : $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’ : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’ : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’ : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’ : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of `popt` from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** ([float](#), optional) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type *numpy.ndarray*

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property `anis_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `list`

property `arg`

Names of all arguments.

Type `list` of `str`

property `arg_bounds`

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `dict`

property `arg_list`

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list` of `str`

property iso_arg_list

Values of isotropic arguments.

Type `list` of `float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list` of `str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.Linear

```
class gstools.covmodel.Linear(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                             integral_scale=None, rescale=None, latlon=False, 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 [Webster2007]:

$$\rho(r) = \begin{cases} 1 - s \cdot \frac{r}{\ell} & r < \frac{\ell}{s} \\ 0 & r \geq \frac{\ell}{s} \end{cases}$$

Where the standard rescale factor is $s = 1$.

References**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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [e_y, e_z].
 - e_y = l_y / l_x
 - e_z = l_z / l_x

If only one value is given in 3D, e_y will be set to 1. This value will be ignored, if multiple len_scales are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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](#)
- **rescale** ([float](#) or [None](#), optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: [None](#)

- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth's radius, to have a meaningful *len_scale* parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Linear model is only valid in 1D.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	Linear normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Provide default optional arguments by the user.
<code>default_opt_arg_bounds()</code>	Provide default boundaries for optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fiting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.

continues on next page

Table 21 – continued from previous page

<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Linear model is only valid in 1D.

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(h)

Linear normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

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 when overridden.

default_opt_arg_bounds()

Provide default boundaries for optional arguments.

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis=True*, *sill=None*, *init_guess='default'*, *weights=None*,
method='trf', *loss='soft_l1'*, *max_eval=None*, *return_r2=False*,
curve_fit_kwargs=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

– $\text{sill} = \text{var} + \text{nugget}$

– if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as

given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’ : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’ : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default) : $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’ : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’ : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’ : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’ : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of `popt` from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** ([float](#), optional) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type *numpy.ndarray*

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property `anis_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `arg`

Names of all arguments.

Type `list` of `str`

property `arg_bounds`

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property `arg_list`

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list` of `str`

property iso_arg_list

Values of isotropic arguments.

Type `list` of `float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open (“o”) or closed (“c”).

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list` of `str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.Circular

```
class gstools.covmodel.Circular(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                                integral_scale=None, rescale=None, latlon=False, 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 [Webster2007]:

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

Where the standard rescale factor is $s = 1$.

References**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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** (**float**, optional) – nugget of the model. Default: 0.0
- **anis** (**float** or **list**, optional) – anisotropy ratios in the transversal directions [e_y, e_z].
 - e_y = l_y / l_x
 - e_z = l_z / l_x

If only one value is given in 3D, e_y will be set to 1. This value will be ignored, if multiple len_scales are given. Default: 1.0
- **angles** (**float** or **list**, optional) – angles of rotation (given in rad):
 - 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**
- **rescale** (**float** or **None**, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: **None**

- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth's radius, to have a meaningful *len_scale* parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Circular model is only valid in 1D and 2D.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	Circular normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Provide default optional arguments by the user.
<code>default_opt_arg_bounds()</code>	Provide default boundaries for optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fiting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.

continues on next page

Table 22 – continued from previous page

<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Circular model is only valid in 1D and 2D.

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(h)

Circular normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

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 when overridden.

default_opt_arg_bounds()

Provide default boundaries for optional arguments.

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis=True*, *sill=None*, *init_guess='default'*, *weights=None*,
method='trf', *loss='soft_l1'*, *max_eval=None*, *return_r2=False*,
curve_fit_kwargs=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

– $\text{sill} = \text{var} + \text{nugget}$

– if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as

given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’: Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’: dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default): $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’: $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’: $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’: $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’: $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of `popt` from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** ([float](#), optional) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type *numpy.ndarray*

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property `anis_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `list`

property `arg`

Names of all arguments.

Type `list` of `str`

property `arg_bounds`

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `dict`

property `arg_list`

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list` of `str`

property iso_arg_list

Values of isotropic arguments.

Type `list` of `float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open (“o”) or closed (“c”).

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list` of `str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.Spherical

```
class gstools.covmodel.Spherical(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                                integral_scale=None, rescale=None, latlon=False, 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 [Webster2007]:

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

Where the standard rescale factor is $s = 1$.

References

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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [e_y, e_z].
 - e_y = l_y / l_x
 - e_z = l_z / l_x

If only one value is given in 3D, e_y will be set to 1. This value will be ignored, if multiple len_scales are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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](#)
- **rescale** ([float](#) or [None](#), optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: [None](#)

- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth's radius, to have a meaningful *len_scale* parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<i>anisometrize</i> (pos)	Bring a position tuple into the anisotropic coordinate-system.
<i>calc_integral_scale</i> ()	Calculate the integral scale of the isotrope model.
<i>check_arg_bounds</i> ()	Check arguments to be within their given bounds.
<i>check_dim</i> (dim)	Spherical model is only valid in 1D, 2D and 3D.
<i>check_opt_arg</i> ()	Run checks for the optional arguments.
<i>cor</i> (h)	Spherical normalized correlation function.
<i>cor_axis</i> (r[, axis])	Correlation along axis of anisotropy.
<i>cor_spatial</i> (pos)	Spatial correlation respecting anisotropy and rotation.
<i>cor_yadrenko</i> (zeta)	Yadrenko correlation for great-circle distance from latlon-pos.
<i>correlation</i> (r)	Correlation function of the model.
<i>cov_axis</i> (r[, axis])	Covariance along axis of anisotropy.
<i>cov_nugget</i> (r)	Isotropic covariance of the model respecting the nugget at r=0.
<i>cov_spatial</i> (pos)	Spatial covariance respecting anisotropy and rotation.
<i>cov_yadrenko</i> (zeta)	Yadrenko covariance for great-circle distance from latlon-pos.
<i>covariance</i> (r)	Covariance of the model.
<i>default_arg_bounds</i> ()	Provide default boundaries for arguments.
<i>default_opt_arg</i> ()	Provide default optional arguments by the user.
<i>default_opt_arg_bounds</i> ()	Provide default boundaries for optional arguments.
<i>default_rescale</i> ()	Provide default rescaling factor.
<i>fit_variogram</i> (x_data, y_data[, anis, sill, ...])	Fiting the variogram-model to an empirical variogram.
<i>fix_dim</i> ()	Set a fix dimension for the model.
<i>isometrize</i> (pos)	Make a position tuple ready for isotropic operations.

continues on next page

Table 23 – continued from previous page

<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

`anisometrize(pos)`

Bring a position tuple into the anisotropic coordinate-system.

`calc_integral_scale()`

Calculate the integral scale of the isotrope model.

`check_arg_bounds()`

Check arguments to be within their given bounds.

`check_dim(dim)`

Spherical model is only valid in 1D, 2D and 3D.

`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(h)`

Spherical normalized correlation function.

`cor_axis(r, axis=0)`

Correlation along axis of anisotropy.

`cor_spatial(pos)`

Spatial correlation respecting anisotropy and rotation.

`cor_yadrenko(zeta)`

Yadrenko correlation for great-circle distance from latlon-pos.

`correlation(r)`

Correlation function of the model.

`cov_axis(r, axis=0)`

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

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 when overridden.

default_opt_arg_bounds()

Provide default boundaries for optional arguments.

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis=True*, *sill=None*, *init_guess='default'*, *weights=None*,
method='trf', *loss='soft_l1'*, *max_eval=None*, *return_r2=False*,
curve_fit_kwargs=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

– $\text{sill} = \text{var} + \text{nugget}$

– if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as

given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’ : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’ : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default) : $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’ : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’ : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’ : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’ : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray $z=f^2$ and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** (`float`, optional) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property `anis_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `list`

property `arg`

Names of all arguments.

Type `list` of `str`

property `arg_bounds`

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `dict`

property `arg_list`

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list` of `str`

property iso_arg_list

Values of isotropic arguments.

Type `list` of `float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list` of `str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.HyperSpherical

```
class gstools.covmodel.HyperSpherical(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0,
                                     angles=0.0, integral_scale=None, rescale=None,
                                     latlon=False, var_raw=None, hankel_kw=None, **opt_arg)
```

Bases: [gstools.covmodel.base.CovModel](#)

The Hyper-Spherical covariance model.

This model is derived from the relative intersection area of two d-dimensional hyperspheres, 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 the Circular model and in 3D the Spherical model.

Notes

This model is given by the following correlation function [Matern1960]:

$$\rho(r) = \begin{cases} 1 - s \cdot \frac{r}{\ell} \cdot \frac{{}_2F_1\left(\frac{1}{2}, -\frac{d-1}{2}, \frac{3}{2}, \left(s \cdot \frac{r}{\ell}\right)^2\right)}{{}_2F_1\left(\frac{1}{2}, -\frac{d-1}{2}, \frac{3}{2}, 1\right)} & r < \frac{\ell}{s} \\ 0 & r \geq \frac{\ell}{s} \end{cases}$$

Where the standard rescale factor is $s = 1$. d is the dimension.

References**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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - *e_y* = *l_y* / *l_x*
 - *e_z* = *l_z* / *l_x*

If only one value is given in 3D, *e_y* will be set to 1. This value will be ignored, if multiple *len_scales* are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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](#)
- **rescale** ([float](#) or [None](#), optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: [None](#)

- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth's radius, to have a meaningful *len_scale* parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	Hyper-Spherical normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Provide default optional arguments by the user.
<code>default_opt_arg_bounds()</code>	Provide default boundaries for optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fiting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.

continues on next page

Table 24 – continued from previous page

<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(h)

Hyper-Spherical normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

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 when overridden.

default_opt_arg_bounds()

Provide default boundaries for optional arguments.

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis=True*, *sill=None*, *init_guess='default'*, *weights=None*, *method='trf'*, *loss='soft_l1'*, *max_eval=None*, *return_r2=False*, *curve_fit_kwargs=None*, *para_select*)**

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

– $\text{sill} = \text{var} + \text{nugget}$

– if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as

given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’ : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’ : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default) : $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’ : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’ : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’ : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’ : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of `popt` from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** ([float](#), optional) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property `anis_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `arg`

Names of all arguments.

Type `list` of `str`

property `arg_bounds`

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property `arg_list`

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list` of `str`

property iso_arg_list

Values of isotropic arguments.

Type `list` of `float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list` of `str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.SuperSpherical

```
class gstools.covmodel.SuperSpherical(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0,
                                     angles=0.0, integral_scale=None, rescale=None,
                                     latlon=False, var_raw=None, hankel_kw=None, **opt_arg)
```

Bases: [`gstools.covmodel.base.CovModel`](#)

The Super-Spherical covariance model.

This model is derived from the relative intersection area of two d-dimensional hyperspheres, where the middle points have a distance of r and the diameters are given by ℓ . It is than valid in all lower dimensions. By default it coincides with the Hyper-Spherical model.

Notes

This model is given by the following correlation function [Matern1960]:

$$\rho(r) = \begin{cases} 1 - s \cdot \frac{r}{\ell} \cdot \frac{{}_2F_1\left(\frac{1}{2}, -\nu, \frac{3}{2}, \left(s \cdot \frac{r}{\ell}\right)^2\right)}{{}_2F_1\left(\frac{1}{2}, -\nu, \frac{3}{2}, 1\right)} & r < \frac{\ell}{s} \\ 0 & r \geq \frac{\ell}{s} \end{cases}$$

Where the standard rescale factor is $s = 1$. $\nu \geq \frac{d-1}{2}$ is a shape parameter.

References**Parameters**

- **nu** ([float](#), optional) – Shape parameter. Standard range: $[(\text{dim}-1)/2, 50]$ Default: $(\text{dim}-1)/2$
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [e_y, e_z].
 - e_y = l_y / l_x
 - e_z = l_z / l_x

If only one value is given in 3D, e_y will be set to 1. This value will be ignored, if multiple len_scales are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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](#)

- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, `dim` will be set to 3 and anisotropy will be disabled. `rescale` can be set to e.g. earth's radius, to have a meaningful `len_scale` parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	Super-Spherical normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Defaults for the optional arguments.
<code>default_opt_arg_bounds()</code>	Defaults for boundaries of the optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fitting the variogram-model to an empirical variogram.

continues on next page

Table 25 – continued from previous page

<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(h)

Super-Spherical normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(*r*, *axis*=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

default_arg_bounds()

Provide default boundaries for arguments.

Given as a dictionary.

default_opt_arg()

Defaults for the optional arguments.

- {"nu": (dim-1)/2}

Returns Defaults for optional arguments

Return type dict

default_opt_arg_bounds()

Defaults for boundaries of the optional arguments.

- {"nu": [(dim-1)/2, 50.0]}

Returns Boundaries for optional arguments

Return type dict

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (numpy.ndarray) – The bin-centers of the empirical variogram.
- **y_data** (numpy.ndarray) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (bool, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it "False". You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (float or bool, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:
 - sill = var + nugget
 - if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: `None`

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:
 - ‘inv’: inverse distance $1 / (x_data + 1)$
 - list: weights given per bin
 - callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: `None`

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’: Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’: dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:
 - ‘linear’ (default): $\rho(z) = z$. Gives a standard least-squares problem.
 - ‘soft_l1’: $\rho(z) = 2 * ((1 + z)**0.5 - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
 - ‘huber’: $\rho(z) = z$ if $z \leq 1$ else $2*z**0.5 - 1$. Works similarly to ‘soft_l1’.
 - ‘cauchy’: $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
 - ‘arctan’: $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: `False`
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: `None`

- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters won't be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
- **r2_score** (`float`, optional) – r2 score of the curve fitting results. Only if `return_r2` is True.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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”
 - “vario_yadrenko”
 - “cov_yadrenko”
 - “cor_yadrenko”
 - “vario_axis”
 - “cov_axis”
 - “cor_axis”
 - “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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list` of `str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list` of `float`

property dim

The dimension of the model.

Type `int`

property dist_func

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property do_rotation

State if a rotation is performed.

Type `bool`

property field_dim

The field dimension of the model.

Type `int`

property **hankel_kw**

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property **has_cdf**

State if a cdf is defined by the user.

Type `bool`

property **has_ppf**

State if a ppf is defined by the user.

Type `bool`

property **integral_scale**

The main integral scale of the model.

Raises **ValueError** – If integral scale is not settable.

Type `float`

property **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`

property **is_isotropic**

State if a model is isotropic.

Type `bool`

property **iso_arg**

Names of isotropic arguments.

Type `list` of `str`

property **iso_arg_list**

Values of isotropic arguments.

Type `list` of `float`

property **latlon**

Whether the model depends on geographical coords.

Type `bool`

property **len_rescaled**

The rescaled main length scale of the model.

Type `float`

property **len_scale**

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.JBessel

```
class gstools.covmodel.JBessel(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                              integral_scale=None, rescale=None, latlon=False, var_raw=None,
                              hankel_kw=None, **opt_arg)
```

Bases: [gstools.covmodel.base.CovModel](#)

The J-Bessel hole model.

This covariance model is a valid hole model, meaning it has areas of negative correlation but a valid spectral density.

Notes

This model is given by the following correlation function [Chiles2009]:

$$\rho(r) = \Gamma(\nu + 1) \cdot \frac{J_\nu\left(s \cdot \frac{r}{\ell}\right)}{\left(s \cdot \frac{r}{2\ell}\right)^\nu}$$

Where the standard rescale factor is $s = 1$. Γ is the gamma function and J_ν is the Bessel functions of the first kind. $\nu \geq \frac{d}{2} - 1$ is a shape parameter, which defaults to $\nu = \frac{d}{2}$, since the spectrum of the model gets instable for $\nu \rightarrow \frac{d}{2} - 1$.

For $\nu = \frac{1}{2}$ (valid in d=1,2,3) we get the so-called ‘Wave’ model:

$$\rho(r) = \frac{\sin\left(s \cdot \frac{r}{\ell}\right)}{s \cdot \frac{r}{\ell}}$$

References**Parameters**

- **nu** ([float](#), optional) – Shape parameter. Standard range: [dim/2 - 1, 50] Default: dim/2
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [e_y, e_z].
 - e_y = l_y / l_x
 - e_z = l_z / l_x

If only one value is given in 3D, e_y will be set to 1. This value will be ignored, if multiple len_scales are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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`
- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, `dim` will be set to 3 and anisotropy will be disabled. `rescale` can be set to e.g. earth's radius, to have a meaningful `len_scale` parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds list: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Check the optional arguments.
<code>cor(h)</code>	J-Bessel correlation.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Defaults for the optional arguments.

continues on next page

Table 26 – continued from previous page

<code>default_opt_arg_bounds()</code>	Defaults for boundaries of the optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fitting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

check_opt_arg()

Check the optional arguments.

Warns nu – If nu is close to dim/2 - 1, the model tends to get unstable.

cor(h)

J-Bessel correlation.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

Correlation function of the model.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

default_arg_bounds()

Provide default boundaries for arguments.

Given as a dictionary.

default_opt_arg()

Defaults for the optional arguments.

- {"nu": dim/2}

Returns Defaults for optional arguments

Return type dict

default_opt_arg_bounds()

Defaults for boundaries of the optional arguments.

- {"nu": [dim/2 - 1, 50.0]}

Returns Boundaries for optional arguments

Return type dict

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis=True*, *sill=None*, *init_guess='default'*, *weights=None*, *method='trf'*, *loss='soft_l1'*, *max_eval=None*, *return_r2=False*, *curve_fit_kwargs=None*, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (numpy.ndarray) – The bin-centers of the empirical variogram.
- **y_data** (numpy.ndarray) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (bool, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (float or bool, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:
 - $\text{sill} = \text{var} + \text{nugget}$
 - if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a ValueError comes up. Same for nugget. If sill=False, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (*str* or *dict*, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: {"len_scale": 10, "default": "current"}

Default: “default”

- **weights** (*str*, *numpy.ndarray*, *callable*, optional) – Weights applied to each point in the estimation. Either:
 - ‘inv’: inverse distance $1 / (x_data + 1)$
 - list: weights given per bin
 - callable: function applied to *x_data*

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** ({‘trf’, ‘dogbox’}, *optional*) – Algorithm to perform minimization.
 - ‘trf’: Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’: dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (*str* or *callable*, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:
 - ‘linear’ (default): $\rho(z) = z$. Gives a standard least-squares problem.
 - ‘soft_l1’: $\rho(z) = 2 * ((1 + z)**0.5 - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
 - ‘huber’: $\rho(z) = z$ if $z \leq 1$ else $2*z**0.5 - 1$. Works similarly to ‘soft_l1’.
 - ‘cauchy’: $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
 - ‘arctan’: $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray $z=f**2$ and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (*int* or *None*, optional) – Maximum number of function evaluations before the termination. If None (default), the value is chosen automatically: $100 * n$.
- **return_r2** (*bool*, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (*dict*, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None

- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters won't be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
- **r2_score** (`float`, optional) – r2 score of the curve fitting results. Only if `return_r2` is True.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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”
 - “vario_yadrenko”
 - “cov_yadrenko”
 - “cor_yadrenko”
 - “vario_axis”
 - “cov_axis”
 - “cor_axis”
 - “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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list` of `str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list` of `float`

property dim

The dimension of the model.

Type `int`

property dist_func

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property do_rotation

State if a rotation is performed.

Type `bool`

property field_dim

The field dimension of the model.

Type `int`

property **hankel_kw**

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property **has_cdf**

State if a cdf is defined by the user.

Type `bool`

property **has_ppf**

State if a ppf is defined by the user.

Type `bool`

property **integral_scale**

The main integral scale of the model.

Raises **ValueError** – If integral scale is not settable.

Type `float`

property **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`

property **is_isotropic**

State if a model is isotropic.

Type `bool`

property **iso_arg**

Names of isotropic arguments.

Type `list` of `str`

property **iso_arg_list**

Values of isotropic arguments.

Type `list` of `float`

property **latlon**

Whether the model depends on geographical coords.

Type `bool`

property **len_rescaled**

The rescaled main length scale of the model.

Type `float`

property **len_scale**

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

Truncated Power Law Covariance Models

<code>TPLGaussian</code> ([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Gaussian modes.
<code>TPLExponential</code> ([dim, var, len_scale, ...])	Truncated-Power-Law with Exponential modes.
<code>TPLStable</code> ([dim, var, len_scale, nugget, ...])	Truncated-Power-Law with Stable modes.
<code>TPLSimple</code> ([dim, var, len_scale, nugget, ...])	The simply truncated power law model.

gstools.covmodel.TPLGaussian

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

Bases: `gstools.covmodel.tpl_models.TPLCovModel`

Truncated-Power-Law with Gaussian modes.

Notes

The truncated power law is given by a superposition of scale-dependent variograms [Federico1997]:

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

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:

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

$$\sigma_{\ell_{\text{low}}, \ell_{\text{up}}}^2 = \frac{C \cdot (\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H})}{2H}$$

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

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

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

The following Parameters occur:

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

This will be calculated internally by:

$$- \text{len_up} = \text{len_low} + \text{len_scale}$$

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.

References

Parameters

- **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, inf]. Default: 0.0
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** (`float`, optional) – nugget of the model. Default: 0.0
- **anis** (`float` or `list`, optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - $e_y = l_y / l_x$
 - $e_z = l_z / l_x$If only one value is given in 3D, *e_y* will be set to 1. This value will be ignored, if multiple *len_scales* are given. Default: 1.0
- **angles** (`float` or `list`, optional) – angles of rotation (given in rad):
 - 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`
- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earths surface described by latitude and longitude. When using this, the model will internally use the associated ‘Yadrenko’ model to represent a valid model. This means, the spatial distance *r* will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth’s radius, to have a meaningful *len_scale* parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_low_rescaled `float`: Lower length scale truncation rescaled.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length scale.

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

len_up `float`: Upper length scale truncation of the model.

len_up_rescaled `float`: Upper length scale truncation rescaled.

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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	TPL with Gaussian modes - normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	TPL with Gaussian modes - correlation function.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Defaults for the optional arguments.
<code>default_opt_arg_bounds()</code>	Defaults for boundaries of the optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fiting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for C (intensity of variation) to result in variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.

continues on next page

Table 28 – continued from previous page

<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at $r=0$.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(*pos*)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(*dim*)

Check the given dimension.

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(*h*)

TPL with Gaussian modes - normalized correlation function.

cor_axis(*r*, *axis*=0)

Correlation along axis of anisotropy.

cor_spatial(*pos*)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(*zeta*)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(*r*)

TPL with Gaussian modes - correlation function.

cov_axis(*r*, *axis*=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

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, inf, "cc"]}

Returns Boundaries for optional arguments

Return type dict

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fiting the variogram-model to an empirical variogram.

Parameters

- **x_data** (numpy.ndarray) – The bin-centers of the empirical variogram.
- **y_data** (numpy.ndarray) – The messured variogram If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (bool, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios wont be altered during fitting. Default: True
- **sill** (float or bool, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

– $\text{sill} = \text{var} + \text{nugget}$

– if the variance is bigger than the sill, nugget will bet set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a ValueError comes up. Same for nugget. If sill=False, it will be deslected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (str or dict, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can bet set to “default” or “current” for unspecified values to get same behavior as

given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’ : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’ : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default) : $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’ : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’ : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’ : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’ : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of `popt` from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** ([float](#), optional) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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 C (intensity of variation) to result in variance.

vario_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property `anis_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `arg`

Names of all arguments.

Type `list` of `str`

property `arg_bounds`

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property `arg_list`

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list of str`

property iso_arg_list

Values of isotropic arguments.

Type `list of float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_low_rescaled

Lower length scale truncation rescaled.

- `len_low_rescaled = len_low / rescale`

Type `float`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the length scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property len_up

Upper length scale truncation of the model.

- `len_up = len_low + len_scale`

Type `float`

property len_up_rescaled

Upper length scale truncation rescaled.

- `len_up_rescaled = (len_low + len_scale) / rescale`

Type `float`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$
-

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `var_raw`

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.TPLExponential

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

Bases: `gstools.covmodel.tpl_models.TPLCovModel`

Truncated-Power-Law with Exponential modes.

Notes

The truncated power law is given by a superposition of scale-dependent variograms [Federico1997]:

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

with *Exponential* modes on each scale:

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

This results in:

$$\begin{aligned} \gamma_{\ell_{\text{low}}, \ell_{\text{up}}}(r) &= \sigma_{\ell_{\text{low}}, \ell_{\text{up}}}^2 \cdot \left(1 - 2H \cdot \frac{\ell_{\text{up}}^{2H} \cdot E_{1+2H}\left[\frac{r}{\ell_{\text{up}}}\right] - \ell_{\text{low}}^{2H} \cdot E_{1+2H}\left[\frac{r}{\ell_{\text{low}}}\right]}{\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H}}\right) \\ \sigma_{\ell_{\text{low}}, \ell_{\text{up}}}^2 &= \frac{C \cdot (\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H})}{2H} \end{aligned}$$

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

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

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

The following Parameters occur:

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

This will be calculated internally by:

$$\text{len_up} = \text{len_low} + \text{len_scale}$$

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.

References

Parameters

- **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, inf]. Default: 0.0
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** (`float`, optional) – nugget of the model. Default: 0.0
- **anis** (`float` or `list`, optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - $e_y = l_y / l_x$
 - $e_z = l_z / l_x$If only one value is given in 3D, *e_y* will be set to 1. This value will be ignored, if multiple *len_scales* are given. Default: 1.0
- **angles** (`float` or `list`, optional) – angles of rotation (given in rad):
 - 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`
- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earths surface described by latitude and longitude. When using this, the model will internally use the associated ‘Yadrenko’ model to represent a valid model. This means, the spatial distance *r* will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth’s radius, to have a meaningful *len_scale* parameter. Default: `False`
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_low_rescaled `float`: Lower length scale truncation rescaled.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length scale.

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

len_up `float`: Upper length scale truncation of the model.

len_up_rescaled `float`: Upper length scale truncation rescaled.

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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	TPL with Exponential modes - normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	TPL with Exponential modes - correlation function.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Defaults for the optional arguments.
<code>default_opt_arg_bounds()</code>	Defaults for boundaries of the optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fiting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for C (intensity of variation) to result in variance.

continues on next page

Table 29 – continued from previous page

<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at $r=0$.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(h)

TPL with Exponential modes - normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

TPL with Exponential modes - correlation function.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(r)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(pos)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(zeta)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(r)

Covariance of the model.

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, inf, "cc"]}

Returns Boundaries for optional arguments

Return type dict

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fiting the variogram-model to an empirical variogram.

Parameters

- **x_data** (numpy.ndarray) – The bin-centers of the empirical variogram.
- **y_data** (numpy.ndarray) – The messured variogram If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (bool, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios wont be altered during fitting. Default: True
- **sill** (float or bool, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

– $\text{sill} = \text{var} + \text{nugget}$

– if the variance is bigger than the sill, nugget will bet set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a ValueError comes up. Same for nugget. If sill=False, it will be deslected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (str or dict, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can bet set to “default” or “current” for unspecified values to get same behavior as

given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:

- ‘inv’: inverse distance $1 / (x_data + 1)$
- list: weights given per bin
- callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, `optional`) – Algorithm to perform minimization.
 - ‘trf’ : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’ : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:

- ‘linear’ (default) : $\rho(z) = z$. Gives a standard least-squares problem.
- ‘soft_l1’ : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
- ‘huber’ : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
- ‘cauchy’ : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
- ‘arctan’ : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipys curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of `popt` from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.

- **r2_score** ([float](#), optional) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword (“var”, “len_scale”, “nugget”, <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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 C (intensity of variation) to result in variance.

vario_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property `anis_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `arg`

Names of all arguments.

Type `list` of `str`

property `arg_bounds`

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property `arg_list`

Values of all arguments.

Type `list` of `float`

property `dim`

The dimension of the model.

Type `int`

property `dist_func`

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property `do_rotation`

State if a rotation is performed.

Type `bool`

property `field_dim`

The field dimension of the model.

Type `int`

property `hankel_kw`

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property `has_cdf`

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list of str`

property iso_arg_list

Values of isotropic arguments.

Type `list of float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_low_rescaled

Lower length scale truncation rescaled.

- `len_low_rescaled = len_low / rescale`

Type `float`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the length scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property len_up

Upper length scale truncation of the model.

- `len_up = len_low + len_scale`

Type `float`

property len_up_rescaled

Upper length scale truncation rescaled.

- `len_up_rescaled = (len_low + len_scale) / rescale`

Type `float`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$
-

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `var_raw`

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.TPLStable

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

Bases: `gstools.covmodel.tpl_models.TPLCovModel`

Truncated-Power-Law with Stable modes.

Notes

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

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

with *Stable* modes on each scale:

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

This results in:

$$\begin{aligned} \gamma_{\ell_{\text{low}}, \ell_{\text{up}}}(r) &= \sigma_{\ell_{\text{low}}, \ell_{\text{up}}}^2 \cdot \left(1 - \frac{2H}{\alpha} \cdot \frac{\ell_{\text{up}}^{2H} \cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell_{\text{up}}}\right)^\alpha\right] - \ell_{\text{low}}^{2H} \cdot E_{1+\frac{2H}{\alpha}}\left[\left(\frac{r}{\ell_{\text{low}}}\right)^\alpha\right]}{\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H}}\right) \\ \sigma_{\ell_{\text{low}}, \ell_{\text{up}}}^2 &= \frac{C \cdot (\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H})}{2H} \end{aligned}$$

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

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

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

The following Parameters occur:

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

This will be calculated internally by:

$$\text{len_up} = \text{len_low} + \text{len_scale}$$

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.

Parameters

- **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, inf]. Default: 0.0
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** (`float`, optional) – nugget of the model. Default: 0.0
- **anis** (`float` or `list`, optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - $e_y = l_y / l_x$
 - $e_z = l_z / l_x$If only one value is given in 3D, *e_y* will be set to 1. This value will be ignored, if multiple *len_scales* are given. Default: 1.0
- **angles** (`float` or `list`, optional) – angles of rotation (given in rad):
 - 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`
- **rescale** (`float` or `None`, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: `None`
- **latlon** (`bool`, optional) – Whether the model is describing 2D fields on earths surface described by latitude and longitude. When using this, the model will internally use the associated ‘Yadrenko’ model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth’s radius, to have a meaningful *len_scale* parameter. Default: False
- **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`
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds `list`: Bounds for the nugget.

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

arg_bounds `dict`: Bounds for all parameters.

arg_list `list` of `float`: Values of all arguments.

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.

field_dim `int`: The field dimension of the model.

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.

is_isotropic `bool`: State if a model is isotropic.

iso_arg `list` of `str`: Names of isotropic arguments.

iso_arg_list `list` of `float`: Values of isotropic arguments.

latlon `bool`: Whether the model depends on geographical coords.

len_low_rescaled `float`: Lower length scale truncation rescaled.

len_rescaled `float`: The rescaled main length scale of the model.

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

len_scale_bounds `list`: Bounds for the length scale.

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

len_up `float`: Upper length scale truncation of the model.

len_up_rescaled `float`: Upper length scale truncation rescaled.

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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Check the optional arguments.
<code>cor(h)</code>	TPL with Stable modes - normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	TPL with Stable modes - correlation function.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Defaults for the optional arguments.
<code>default_opt_arg_bounds()</code>	Defaults for boundaries of the optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fitting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for C (intensity of variation) to result in variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.

continues on next page

Table 30 – continued from previous page

<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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.

cor(h)

TPL with Stable modes - normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(zeta)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(r)

TPL with Stable modes - correlation function.

cov_axis(r, axis=0)

Covariance along axis of anisotropy.

cov_nugget(r)

Isotropic covariance of the model respecting the nugget at r=0.

cov_spatial(pos)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(zeta)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(r)

Covariance of the model.

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, inf, "cc"]}

Returns Boundaries for optional arguments

Return type `dict`

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (`numpy.ndarray`) – The bin-centers of the empirical variogram.
- **y_data** (`numpy.ndarray`) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (`bool`, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it “False”. You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (`float` or `bool`, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:
 - $\text{sill} = \text{var} + \text{nugget}$
 - if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: None

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: {"len_scale": 10, "default": "current"}

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:
 - ‘inv’: inverse distance $1 / (\text{x_data} + 1)$
 - list: weights given per bin
 - callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: None

- **method** (`{'trf', 'dogbox'}`, *optional*) – Algorithm to perform minimization.
 - `'trf'` : Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - `'dogbox'` : dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: `'trf'`

- **loss** (`str` or callable, *optional*) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:
 - `'linear'` (default) : $\rho(z) = z$. Gives a standard least-squares problem.
 - `'soft_l1'` : $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
 - `'huber'` : $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to `'soft_l1'`.
 - `'cauchy'` : $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
 - `'arctan'` : $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to `'cauchy'`.

If callable, it must take a 1-d ndarray $z=f^2$ and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: `'soft_l1'`

- **max_eval** (`int` or `None`, *optional*) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.
- **return_r2** (`bool`, *optional*) – Whether to return the r2 score of the estimation. Default: `False`
- **curve_fit_kwargs** (`dict`, *optional*) – Other keyword arguments passed to `scipys curve_fit`. Default: `None`
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
- **r2_score** (`float`, *optional*) – r2 score of the curve fitting results. Only if `return_r2` is `True`.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(*r*)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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"
 - "vario_yadrenko"
 - "cov_yadrenko"
 - "cor_yadrenko"
 - "vario_axis"
 - "cov_axis"
 - "cor_axis"
 - "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool*, optional) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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 C (intensity of variation) to result in variance.

vario_axis(*r*, *axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [*a*, *b*] or [*a*, *b*, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list` of `str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list` of `float`

property dim

The dimension of the model.

Type `int`

property dist_func

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property do_rotation

State if a rotation is performed.

Type `bool`

property field_dim

The field dimension of the model.

Type `int`

property hankel_kw

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property has_cdf

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list` of `str`

property iso_arg_list

Values of isotropic arguments.

Type `list` of `float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_low_rescaled

Lower length scale truncation rescaled.

- `len_low_rescaled = len_low / rescale`

Type `float`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property len_scale

The main length scale of the model.

Type `float`

property len_scale_bounds

Bounds for the lenght scale.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of `"oo"`, `"cc"`, `"oc"` or `"co"` to define if the bounds are open ("o") or closed ("c").

Type `list`

property 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`

property len_up

Upper length scale truncation of the model.

- `len_up = len_low + len_scale`

Type `float`

property len_up_rescaled

Upper length scale truncation rescaled.

- `len_up_rescaled = (len_low + len_scale) / rescale`

Type `float`

property name

The name of the CovModel class.

Type `str`

property nugget

The nugget of the model.

Type `float`

property nugget_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: `[a, b]` or `[a, b, <type>]` where `<type>` is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property opt_arg

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$
-

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property var_raw

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.TPLSimple

```
class gstools.covmodel.TPLSimple(dim=3, var=1.0, len_scale=1.0, nugget=0.0, anis=1.0, angles=0.0,
                                integral_scale=None, rescale=None, latlon=False, var_raw=None,
                                hankel_kw=None, **opt_arg)
```

Bases: [gstools.covmodel.base.CovModel](#)

The simply truncated power law model.

This model describes a simple truncated power law with a finite length scale. In contrast to other models, this one is not derived from super-positioning modes.

Notes

This model is given by the following correlation function [Wendland1995]:

$$\rho(r) = \begin{cases} (1 - s \cdot \frac{r}{\ell})^\nu & r < \frac{\ell}{s} \\ 0 & r \geq \frac{\ell}{s} \end{cases}$$

Where the standard rescale factor is $s = 1$. $\nu \geq \frac{d+1}{2}$ is a shape parameter, which defaults to $\nu = \frac{d+1}{2}$,

For $\nu = 1$ (valid only in $d=1$) this coincides with the truncated linear model:

$$\rho(r) = \begin{cases} 1 - s \cdot \frac{r}{\ell} & r < \frac{\ell}{s} \\ 0 & r \geq \frac{\ell}{s} \end{cases}$$

References

Parameters

- **nu** ([float](#), optional) – Shape parameter. Standard range: $[(\text{dim}+1)/2, 50]$ Default: $\text{dim}/2$
- **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. If only two values are given in 3D, the latter one will be used for both transversal directions. Default: 1.0
- **nugget** ([float](#), optional) – nugget of the model. Default: 0.0
- **anis** ([float](#) or [list](#), optional) – anisotropy ratios in the transversal directions [*e_y*, *e_z*].
 - $e_y = l_y / l_x$
 - $e_z = l_z / l_x$If only one value is given in 3D, *e_y* will be set to 1. This value will be ignored, if multiple *len_scales* are given. Default: 1.0
- **angles** ([float](#) or [list](#), optional) – angles of rotation (given in rad):
 - 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
- **rescale** (float or None, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit conversion or rescaling the length scale to coincide with e.g. the integral scale. Will be set by each model individually. Default: None
- **latlon** (bool, optional) – Whether the model is describing 2D fields on earth's surface described by latitude and longitude. When using this, the model will internally use the associated 'Yadrenko' model to represent a valid model. This means, the spatial distance r will be replaced by $2 \sin(\alpha/2)$, where α is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, *dim* will be set to 3 and anisotropy will be disabled. *rescale* can be set to e.g. earth's radius, to have a meaningful *len_scale* parameter. Default: False
- **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
- ****opt_arg** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

Attributes

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

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

anis_bounds list: Bounds for the nugget.

arg list of str: Names of all arguments.

arg_bounds dict: Bounds for all parameters.

arg_list list of float: Values of all arguments.

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.

field_dim int: The field dimension of the model.

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.

is_isotropic bool: State if a model is isotropic.

iso_arg list of str: Names of isotropic arguments.

iso_arg_list list of float: Values of isotropic arguments.

latlon bool: Whether the model depends on geographical coords.

len_rescaled float: The rescaled main length scale of the model.

len_scale float: The main length scale of the model.

len_scale_bounds list: Bounds for the length 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.

rescale float: Rescale factor for the length scale of the model.

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

<code>anisometrize(pos)</code>	Bring a position tuple into the anisotropic coordinate-system.
<code>calc_integral_scale()</code>	Calculate the integral scale of the isotrope model.
<code>check_arg_bounds()</code>	Check arguments to be within their given bounds.
<code>check_dim(dim)</code>	Check the given dimension.
<code>check_opt_arg()</code>	Run checks for the optional arguments.
<code>cor(h)</code>	TPL Simple - normalized correlation function.
<code>cor_axis(r[, axis])</code>	Correlation along axis of anisotropy.
<code>cor_spatial(pos)</code>	Spatial correlation respecting anisotropy and rotation.
<code>cor_yadrenko(zeta)</code>	Yadrenko correlation for great-circle distance from latlon-pos.
<code>correlation(r)</code>	Correlation function of the model.
<code>cov_axis(r[, axis])</code>	Covariance along axis of anisotropy.
<code>cov_nugget(r)</code>	Isotropic covariance of the model respecting the nugget at r=0.
<code>cov_spatial(pos)</code>	Spatial covariance respecting anisotropy and rotation.
<code>cov_yadrenko(zeta)</code>	Yadrenko covariance for great-circle distance from latlon-pos.
<code>covariance(r)</code>	Covariance of the model.
<code>default_arg_bounds()</code>	Provide default boundaries for arguments.
<code>default_opt_arg()</code>	Defaults for the optional arguments.

continues on next page

Table 31 – continued from previous page

<code>default_opt_arg_bounds()</code>	Defaults for boundaries of the optional arguments.
<code>default_rescale()</code>	Provide default rescaling factor.
<code>fit_variogram(x_data, y_data[, anis, sill, ...])</code>	Fitting the variogram-model to an empirical variogram.
<code>fix_dim()</code>	Set a fix dimension for the model.
<code>isometrize(pos)</code>	Make a position tuple ready for isotropic operations.
<code>ln_spectral_rad_pdf(r)</code>	Log radial spectral density of the model.
<code>main_axes()</code>	Axes of the rotated coordinate-system.
<code>percentile_scale([per])</code>	Calculate the percentile scale of the isotrope model.
<code>plot([func])</code>	Plot a function of a the CovModel.
<code>pykrige_vario([args, r])</code>	Isotropic variogram of the model for pykrige.
<code>set_arg_bounds([check_args])</code>	Set bounds for the parameters of the model.
<code>spectral_density(k)</code>	Spectral density of the covariance model.
<code>spectral_rad_pdf(r)</code>	Radial spectral density of the model.
<code>spectrum(k)</code>	Spectrum of the covariance model.
<code>var_factor()</code>	Factor for the variance.
<code>vario_axis(r[, axis])</code>	Variogram along axis of anisotropy.
<code>vario_nugget(r)</code>	Isotropic variogram of the model respecting the nugget at r=0.
<code>vario_spatial(pos)</code>	Spatial variogram respecting anisotropy and rotation.
<code>vario_yadrenko(zeta)</code>	Yadrenko variogram for great-circle distance from latlon-pos.
<code>variogram(r)</code>	Isotropic variogram of the model.

anisometrize(pos)

Bring a position tuple into the anisotropic coordinate-system.

calc_integral_scale()

Calculate the integral scale of the isotrope model.

check_arg_bounds()

Check arguments to be within their given bounds.

check_dim(dim)

Check the given dimension.

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(h)

TPL Simple - normalized correlation function.

cor_axis(r, axis=0)

Correlation along axis of anisotropy.

cor_spatial(pos)

Spatial correlation respecting anisotropy and rotation.

cor_yadrenko(*zeta*)

Yadrenko correlation for great-circle distance from latlon-pos.

correlation(*r*)

Correlation function of the model.

cov_axis(*r*, *axis*=0)

Covariance along axis of anisotropy.

cov_nugget(*r*)

Isotropic covariance of the model respecting the nugget at $r=0$.

cov_spatial(*pos*)

Spatial covariance respecting anisotropy and rotation.

cov_yadrenko(*zeta*)

Yadrenko covariance for great-circle distance from latlon-pos.

covariance(*r*)

Covariance of the model.

default_arg_bounds()

Provide default boundaries for arguments.

Given as a dictionary.

default_opt_arg()

Defaults for the optional arguments.

- {"nu": dim/2}

Returns Defaults for optional arguments

Return type dict

default_opt_arg_bounds()

Defaults for boundaries of the optional arguments.

- {"nu": [dim/2 - 1, 50.0]}

Returns Boundaries for optional arguments

Return type dict

default_rescale()

Provide default rescaling factor.

fit_variogram(*x_data*, *y_data*, *anis*=True, *sill*=None, *init_guess*='default', *weights*=None, *method*='trf', *loss*='soft_l1', *max_eval*=None, *return_r2*=False, *curve_fit_kwargs*=None, ***para_select*)

Fitting the variogram-model to an empirical variogram.

Parameters

- **x_data** (numpy.ndarray) – The bin-centers of the empirical variogram.
- **y_data** (numpy.ndarray) – The measured variogram. If multiple are given, they are interpreted as the directional variograms along the main axis of the associated rotated coordinate system. Anisotropy ratios will be estimated in that case.
- **anis** (bool, optional) – In case of a directional variogram, you can control anisotropy by this argument. Deselect the parameter from fitting, by setting it "False". You could also pass a fixed value to be set in the model. Then the anisotropy ratios won't be altered during fitting. Default: True
- **sill** (float or bool, optional) – Here you can provide a fixed sill for the variogram. It needs to be in a fitting range for the var and nugget bounds. If variance or nugget are not selected for estimation, the nugget will be recalculated to fulfill:

- $\text{sill} = \text{var} + \text{nugget}$
- if the variance is bigger than the sill, nugget will be set to its lower bound and the variance will be set to the fitting partial sill.

If variance is deselected, it needs to be less than the sill, otherwise a `ValueError` comes up. Same for nugget. If `sill=False`, it will be deselected from estimation and set to the current sill of the model. Then, the procedure above is applied. Default: `None`

- **init_guess** (`str` or `dict`, optional) – Initial guess for the estimation. Either:
 - “default”: using the default values of the covariance model (“len_scale” will be mean of given bin centers; “var” and “nugget” will be mean of given variogram values (if in given bounds))
 - “current”: using the current values of the covariance model
 - dict: dictionary with parameter names and given value (separate “default” can be set to “default” or “current” for unspecified values to get same behavior as given above (“default” by default)) Example: `{"len_scale": 10, "default": "current"}`

Default: “default”

- **weights** (`str`, `numpy.ndarray`, callable, optional) – Weights applied to each point in the estimation. Either:
 - ‘inv’: inverse distance $1 / (\mathbf{x_data} + 1)$
 - list: weights given per bin
 - callable: function applied to `x_data`

If callable, it must take a 1-d ndarray. Then `weights = f(x_data)`. Default: `None`

- **method** (`{'trf', 'dogbox'}`, optional) – Algorithm to perform minimization.
 - ‘trf’: Trust Region Reflective algorithm, particularly suitable for large sparse problems with bounds. Generally robust method.
 - ‘dogbox’: dogleg algorithm with rectangular trust regions, typical use case is small problems with bounds. Not recommended for problems with rank-deficient Jacobian.

Default: ‘trf’

- **loss** (`str` or callable, optional) – Determines the loss function in `scipys curve_fit`. The following keyword values are allowed:
 - ‘linear’ (default): $\rho(z) = z$. Gives a standard least-squares problem.
 - ‘soft_l1’: $\rho(z) = 2 * ((1 + z)^{0.5} - 1)$. The smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares.
 - ‘huber’: $\rho(z) = z$ if $z \leq 1$ else $2*z^{0.5} - 1$. Works similarly to ‘soft_l1’.
 - ‘cauchy’: $\rho(z) = \ln(1 + z)$. Severely weakens outliers influence, but may cause difficulties in optimization process.
 - ‘arctan’: $\rho(z) = \arctan(z)$. Limits a maximum loss on a single residual, has properties similar to ‘cauchy’.

If callable, it must take a 1-d ndarray `z=f**2` and return an array_like with shape (3, m) where row 0 contains function values, row 1 contains first derivatives and row 2 contains second derivatives. Default: ‘soft_l1’

- **max_eval** (`int` or `None`, optional) – Maximum number of function evaluations before the termination. If `None` (default), the value is chosen automatically: $100 * n$.

- **return_r2** (`bool`, optional) – Whether to return the r2 score of the estimation. Default: False
- **curve_fit_kwargs** (`dict`, optional) – Other keyword arguments passed to `scipy.optimize.curve_fit`. Default: None
- ****para_select** – You can deselect parameters from fitting, by setting them “False” using their names as keywords. You could also pass fixed values for each parameter. Then these values will be applied and the involved parameters wont be fitted. By default, all parameters are fitted.

Returns

- **fit_para** (`dict`) – Dictionary with the fitted parameter values
- **pcov** (`numpy.ndarray`) – The estimated covariance of *popt* from `scipy.optimize.curve_fit`. To compute one standard deviation errors on the parameters use `perr = np.sqrt(np.diag(pcov))`.
- **r2_score** (`float`, optional) – r2 score of the curve fitting results. Only if `return_r2` is True.

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.

isometrize(pos)

Make a position tuple ready for isotropic operations.

ln_spectral_rad_pdf(r)

Log radial spectral density of the model.

main_axes()

Axes of the rotated coordinate-system.

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”
 - “vario_yadrenko”
 - “cov_yadrenko”
 - “cor_yadrenko”

- "vario_axis"
- "cov_axis"
- "cor_axis"
- "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.

set_arg_bounds(*check_args=True, **kwargs*)

Set bounds for the parameters of the model.

Parameters

- **check_args** (*bool, optional*) – Whether to check if the arguments are in their valid bounds. In case not, a proper default value will be determined. Default: True
- ****kwargs** – Parameter name as keyword ("var", "len_scale", "nugget", <opt_arg>) and a list of 2 or 3 values: [a, b] or [a, b, <type>] where <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:

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

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_axis(*r, axis=0*)

Variogram along axis of anisotropy.

vario_nugget(*r*)

Isotropic variogram of the model respecting the nugget at $r=0$.

vario_spatial(*pos*)

Spatial variogram respecting anisotropy and rotation.

vario_yadrenko(*zeta*)

Yadrenko variogram for great-circle distance from latlon-pos.

variogram(*r*)

Isotropic variogram of the model.

property angles

Rotation angles (in rad) of the model.

Type `numpy.ndarray`

property anis

The anisotropy factors of the model.

Type `numpy.ndarray`

property anis_bounds

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property arg

Names of all arguments.

Type `list` of `str`

property arg_bounds

Bounds for all parameters.

Notes

Keys are the arg names and values are lists of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `dict`

property arg_list

Values of all arguments.

Type `list` of `float`

property dim

The dimension of the model.

Type `int`

property dist_func

pdf, cdf and ppf.

Spectral distribution info from the model.

Type `tuple` of `callable`

property do_rotation

State if a rotation is performed.

Type `bool`

property field_dim

The field dimension of the model.

Type `int`

property hankel_kw

`hankel.SymmetricFourierTransform` kwargs.

Type `dict`

property has_cdf

State if a cdf is defined by the user.

Type `bool`

property has_ppf

State if a ppf is defined by the user.

Type `bool`

property integral_scale

The main integral scale of the model.

Raises `ValueError` – If integral scale is not settable.

Type `float`

property 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`

property is_isotropic

State if a model is isotropic.

Type `bool`

property iso_arg

Names of isotropic arguments.

Type `list of str`

property iso_arg_list

Values of isotropic arguments.

Type `list of float`

property latlon

Whether the model depends on geographical coords.

Type `bool`

property len_rescaled

The rescaled main length scale of the model.

Type `float`

property `len_scale`

The main length scale of the model.

Type `float`

property `len_scale_bounds`

Bounds for the length scale.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `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`

property `name`

The name of the CovModel class.

Type `str`

property `nugget`

The nugget of the model.

Type `float`

property `nugget_bounds`

Bounds for the nugget.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `opt_arg`

Names of the optional arguments.

Type `list of str`

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

Type `dict`

property pykrige_angle

2D rotation angle for pykrige.

property pykrige_angle_x

3D rotation angle around x for pykrige.

property pykrige_angle_y

3D rotation angle around y for pykrige.

property pykrige_angle_z

3D rotation angle around z for pykrige.

property pykrige_anis

2D anisotropy ratio for pykrige.

property pykrige_anis_y

3D anisotropy ratio in y direction for pykrige.

property pykrige_anis_z

3D anisotropy ratio in z direction for pykrige.

property pykrige_kwargs

Keyword arguments for pykrige routines.

property rescale

Rescale factor for the length scale of the model.

Type `float`

property sill

The sill of the variogram.

Notes

This is calculated by:

- $\text{sill} = \text{variance} + \text{nugget}$
-

Type `float`

property var

The variance of the model.

Type `float`

property var_bounds

Bounds for the variance.

Notes

Is a list of 2 or 3 values: [a, b] or [a, b, <type>] where <type> is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type `list`

property `var_raw`

The raw variance of the model without factor.

(See. `CovModel.var_factor`)

Type `float`

gstools.covmodel.plot

GStools subpackage providing plotting routines for the covariance models.

The following classes and functions are provided

<code>plot_variogram(model[, x_min, x_max, fig, ax])</code>	Plot variogram of a given CovModel.
<code>plot_covariance(model[, x_min, x_max, fig, ax])</code>	Plot covariance of a given CovModel.
<code>plot_correlation(model[, x_min, x_max, fig, ax])</code>	Plot correlation function of a given CovModel.
<code>plot_vario_yadrenko(model[, x_min, x_max, ...])</code>	Plot Yadrenko variogram of a given CovModel.
<code>plot_cov_yadrenko(model[, x_min, x_max, fig, ax])</code>	Plot Yadrenko covariance of a given CovModel.
<code>plot_cor_yadrenko(model[, x_min, x_max, fig, ax])</code>	Plot Yadrenko correlation function of a given CovModel.
<code>plot_vario_axis(model[, axis, x_min, x_max, ...])</code>	Plot variogram of a given CovModel.
<code>plot_cov_axis(model[, axis, x_min, x_max, ...])</code>	Plot variogram of a given CovModel.
<code>plot_vario_spatial(model[, x_min, x_max, ...])</code>	Plot spatial variogram of a given CovModel.
<code>plot_cov_spatial(model[, x_min, x_max, fig, ax])</code>	Plot spatial covariance of a given CovModel.
<code>plot_cor_spatial(model[, x_min, x_max, fig, ax])</code>	Plot spatial correlation of a given CovModel.
<code>plot_spectrum(model[, x_min, x_max, fig, ax])</code>	Plot specturm of a given CovModel.
<code>plot_spectral_density(model[, x_min, x_max, ...])</code>	Plot spectral density of a given CovModel.
<code>plot_spectral_rad_pdf(model[, x_min, x_max, ...])</code>	Plot radial spectral pdf of a given CovModel.

`gstools.covmodel.plot.plot_cor_axis(model, axis=0, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)`

Plot variogram of a given CovModel.

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

Plot spatial correlation of a given CovModel.

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

Plot Yadrenko correlation function of a given CovModel.

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

Plot correlation function of a given CovModel.

`gstools.covmodel.plot.plot_cov_axis(model, axis=0, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)`

Plot variogram of a given CovModel.

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

Plot spatial covariance of a given CovModel.

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

Plot Yadrenko covariance of a given CovModel.

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

Plot covariance of a given CovModel.

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

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,  
                                             **kwargs)
```

Plot radial spectral pdf of a given CovModel.

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

Plot specturm of a given CovModel.

```
gstools.covmodel.plot.plot_vario_axis(model, axis=0, x_min=0.0, x_max=None, fig=None, ax=None,  
                                       **kwargs)
```

Plot variogram of a given CovModel.

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

Plot spatial variogram of a given CovModel.

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

Plot Yadrenko variogram of a given CovModel.

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

Plot variogram of a given CovModel.

3.7 gstools.field

GStools subpackage providing tools for spatial random fields.

Subpackages

<i>generator</i>	GStools subpackage providing generators for spatial random fields.
<i>upsampling</i>	GStools subpackage providing upscaling routines for the spatial random field.

Spatial Random Field

<i>SRF</i> (model[, mean, normalizer, trend, ...])	A class to generate spatial random fields (SRF).
<i>CondSRF</i> (krige[, generator])	A class to generate conditioned spatial random fields (SRF).

gstools.field.SRF

class `gstools.field.SRF`(*model*, *mean*=0.0, *normalizer*=None, *trend*=None, *upsampling*='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* or *callable*, optional) – Mean of the SRF (in normal form). Could also be a callable. The default is 0.0.
- **normalizer** (*None* or *Normalizer*, optional) – Normalizer to be applied to the SRF to transform the field values. The default is None.
- **trend** (*None* or *float* or *callable*, optional) – Trend of the SRF (in transformed form). If no normalizer is applied, this behaves equal to ‘mean’. The default is None.
- **upsampling** (*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

dim `int`: Dimension of the field.
generator `callable`: The generator of the field.
mean `float` or `callable`: The mean of the field.
model `CovModel`: The covariance model of the field.
name `str`: The name of the class.
normalizer `Normalizer`: Normalizer of the field.
trend `float` or `callable`: The trend of the field.
upscaling `str`: Name of the upscaling method.
value_type `str`: Type of the field values (scalar, vector).

Methods

<code>__call__(pos[, seed, point_volumes, mesh_type])</code>	Generate the spatial random field.
<code>mesh(mesh[, points, direction, name])</code>	Generate a field on a given meshio or ogs5py mesh.
<code>plot([field, fig, ax])</code>	Plot the spatial random field.
<code>post_field(field[, name, process, save])</code>	Postprocessing field values.
<code>pre_pos(pos[, mesh_type])</code>	Preprocessing positions and mesh_type.
<code>set_generator(generator, **generator_kwargs)</code>	Set the generator for the field.
<code>structured(*args, **kwargs)</code>	Generate a field on a structured mesh.
<code>to_pyvista([field_select, fieldname])</code>	Create a VTK/PyVista grid of the stored field.
<code>unstructured(*args, **kwargs)</code>	Generate a field on an unstructured mesh.
<code>upscaling_func(*args, **kwargs)</code>	Upscaling method applied to the field variance.
<code>vtk_export(filename[, field_select, fieldname])</code>	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 resetting. 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`

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

Parameters

- **mesh** (*meshio.Mesh* or *ogs5py.MSH* or *PyVista mesh*) – The given meshio, ogs5py, or PyVista 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* or *list*, optional) – Here you can state which direction should be chosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass “xz”. By default, all directions are used. One can also pass a list of indices. Default: “all”
- **name** (*str* or *list* of *str*, optional) – Name(s) to store the field(s) in the given mesh as *point_data* or *cell_data*. If to few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to *Field.__call__*.

Notes

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

See: <https://github.com/nschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: *Field.__call__*

plot(*field='field', fig=None, ax=None, **kwargs*)

Plot the spatial random field.

Parameters

- **field** (*str*, optional) – Field that should be plotted. 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*
- ****kwargs** – Forwarded to the plotting routine.

post_field(*field, name='field', process=True, save=True*)

Postprocessing field values.

Parameters

- **field** (*numpy.ndarray*) – Field values.
- **name** (*str*, optional) – Name. to store the field. The default is “field”.
- **process** (*bool*, optional) – Whether to process field to apply mean, normalizer and trend. The default is True.
- **save** (*bool*, optional) – Whether to store the field under the given name. The default is True.

Returns *field* – Processed field values.

Return type *numpy.ndarray*

pre_pos(*pos, mesh_type='unstructured'*)

Preprocessing positions and mesh_type.

Parameters

- **pos** (*iterable*) – the position tuple, containing main direction and transversal directions
- **mesh_type** (*str*, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((d, n), [numpy.ndarray](#)) – the isometrized position tuple
- **shape** ([tuple](#)) – Shape of the resulting field.

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”

property dim

Dimension of the field.

Type [int](#)

property generator

The generator of the field.

Default: [RandMeth](#)

Type [callable](#)

property mean

The mean of the field.

Type [float](#) or [callable](#)

property model

The covariance model of the field.

Type *CovModel*

property name

The name of the class.

Type *str*

property normalizer

Normalizer of the field.

Type *Normalizer*

property trend

The trend of the field.

Type *float* or *callable*

property upscaling

Name of the upscaling method.

See the `point_volumes` keyword in the *SRF.__call__* routine. Default: “no_scaling”

Type *str*

property value_type

Type of the field values (scalar, vector).

Type *str*

gstools.field.CondSRF

class `gstools.field.CondSRF(krige, generator='RandMeth', **generator_kwargs)`

Bases: `gstools.field.base.Field`

A class to generate conditioned spatial random fields (SRF).

Parameters

- **krige** (*Krige*) – Kriging setup to condition the spatial random field.
- **generator** (*str*, optional) – Name of the field generator to be used. At the moment, only the following generator is provided:
 - "RandMeth" : The Randomization Method. See: [RandMeth](#)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

dim *int*: Dimension of the field.

generator *callable*: The generator of the field.

krige *Krige*: The underlying kriging class.

mean *float* or *callable*: The mean of the field.

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

name *str*: The name of the class.

normalizer *Normalizer*: Normalizer of the field.

trend *float* or *callable*: The trend of the field.

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

Methods

<code>__call__(pos[, seed, mesh_type])</code>	Generate the conditioned spatial random field.
<code>get_scaling(krige_var, shape)</code>	Get scaling coefficients for the random field.
<code>mesh(mesh[, points, direction, name])</code>	Generate a field on a given meshio or ogs5py mesh.
<code>plot([field, fig, ax])</code>	Plot the spatial random field.
<code>post_field(field[, name, process, save])</code>	Postprocessing field values.
<code>pre_pos(pos[, mesh_type])</code>	Preprocessing positions and mesh_type.
<code>set_generator(generator, **generator_kwargs)</code>	Set the generator for the field.
<code>structured(*args, **kwargs)</code>	Generate a field on a structured mesh.
<code>to_pyvista([field_select, fieldname])</code>	Create a VTK/PyVista grid of the stored field.
<code>unstructured(*args, **kwargs)</code>	Generate a field on an unstructured mesh.
<code>vtk_export(filename[, field_select, fieldname])</code>	Export the stored field to vtk.

`__call__(pos, seed=nan, mesh_type='unstructured', **kwargs)`

Generate the conditioned 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 resetting. Default: keep seed from generator
- **mesh_type** (*str*) – 'structured' / 'unstructured'

- ****kwargs** – keyword arguments that are forwarded to the kriging routine in use.

Returns **field** – the conditioned SRF

Return type `numpy.ndarray`

get_scaling(*krige_var, shape*)

Get scaling coefficients for the random field.

Parameters

- **krige_var** (`numpy.ndarray`) – Kriging variance.
- **shape** (`tuple` of `int`) – Field shape.

Returns

- **var_scale** (`numpy.ndarray`) – Variance scaling factor for the random field.
- **nugget** (`numpy.ndarray` or `int`) – Nugget to be added to the field.

mesh(*mesh, points='centroids', direction='all', name='field', **kwargs*)

Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (*meshio.Mesh* or *ogs5py.MSH* or *PyVista mesh*) – The given meshio, ogs5py, or PyVista 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` or `list`, optional) – Here you can state which direction should be chosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass “xz”. By default, all directions are used. One can also pass a list of indices. Default: “all”
- **name** (`str` or `list` of `str`, optional) – Name(s) to store the field(s) in the given mesh as `point_data` or `cell_data`. If to few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to *Field.__call__*.

Notes

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

See: <https://github.com/nschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: *Field.__call__*

plot(*field='field', fig=None, ax=None, **kwargs*)

Plot the spatial random field.

Parameters

- **field** (`str`, optional) – Field that should be plotted. 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*
- ****kwargs** – Forwarded to the plotting routine.

post_field(*field, name='field', process=True, save=True*)

Postprocessing field values.

Parameters

- **field** (`numpy.ndarray`) – Field values.
- **name** (`str`, optional) – Name. to store the field. The default is “field”.
- **process** (`bool`, optional) – Whether to process field to apply mean, normalizer and trend. The default is True.
- **save** (`bool`, optional) – Whether to store the field under the given name. The default is True.

Returns **field** – Processed field values.

Return type `numpy.ndarray`

pre_pos(*pos*, *mesh_type*='unstructured')
Preprocessing positions and mesh_type.

Parameters

- **pos** (`iterable`) – the position tuple, containing main direction and transversal directions
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((d, n), `numpy.ndarray`) – the isometrized position tuple
- **shape** (`tuple`) – Shape of the resulting field.

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__](#)

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”

property dim

Dimension of the field.

Type `int`

property generator

The generator of the field.

Type `callable`

property krig

The underlying kriging class.

Type `Krige`

property mean

The mean of the field.

Type `float` or `callable`

property model

The covariance model of the field.

Type `CovModel`

property name

The name of the class.

Type `str`

property normalizer

Normalizer of the field.

Type `Normalizer`

property trend

The trend of the field.

Type `float` or `callable`

property value_type

Type of the field values (scalar, vector).

Type `str`

Field Base Class

<i>Field</i> (model[, value_type, mean, normalizer, ...])	A base class for random fields, kriging fields, etc.
---	--

gstools.field.Field

class `gstools.field.Field(model, value_type='scalar', mean=None, normalizer=None, trend=None)`

Bases: `object`

A base class for random fields, kriging fields, etc.

Parameters

- **model** (*CovModel*) – Covariance Model related to the field.
- **value_type** (`str`, optional) – Value type of the field. Either “scalar” or “vector”. The default is “scalar”.
- **mean** (`None` or `float` or `callable`, optional) – Mean of the field if wanted. Could also be a callable. The default is `None`.
- **normalizer** (`None` or *Normalizer*, optional) – Normalizer to be applied to the field. The default is `None`.
- **trend** (`None` or `float` or `callable`, optional) – Trend of the denormalized fields. If no normalizer is applied, this behaves equal to ‘mean’. The default is `None`.

Attributes

dim `int`: Dimension of the field.

mean `float` or `callable`: The mean of the field.

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

name `str`: The name of the class.

normalizer *Normalizer*: Normalizer of the field.

trend `float` or `callable`: The trend of the field.

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

Methods

<code>__call__</code> (*args, **kwargs)	Generate the field.
<code>mesh</code> (mesh[, points, direction, name])	Generate a field on a given meshio or ogs5py mesh.
<code>plot</code> ([field, fig, ax])	Plot the spatial random field.
<code>post_field</code> (field[, name, process, save])	Postprocessing field values.
<code>pre_pos</code> (pos[, mesh_type])	Preprocessing positions and mesh_type.
<code>structured</code> (*args, **kwargs)	Generate a field on a structured mesh.
<code>to_pyvista</code> ([field_select, fieldname])	Create a VTK/PyVista grid of the stored field.
<code>unstructured</code> (*args, **kwargs)	Generate a field on an unstructured mesh.
<code>vtk_export</code> (filename[, field_select, fieldname])	Export the stored field to vtk.

`__call__`(*args, **kwargs)
Generate the field.

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

Parameters

- **mesh** (*meshio.Mesh* or *ogs5py.MSH* or *PyVista mesh*) – The given meshio, ogs5py, or PyVista 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* or *list*, 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”. By default, all directions are used. One can also pass a list of indices. Default: “all”
- **name** (*str* or *list* of *str*, optional) – Name(s) to store the field(s) in the given mesh as *point_data* or *cell_data*. If to few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to *Field.__call__*.

Notes

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

See: <https://github.com/nschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: *Field.__call__*

plot(*field='field', fig=None, ax=None, **kwargs*)

Plot the spatial random field.

Parameters

- **field** (*str*, optional) – Field that should be plotted. 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*
- ****kwargs** – Forwarded to the plotting routine.

post_field(*field, name='field', process=True, save=True*)

Postprocessing field values.

Parameters

- **field** (*numpy.ndarray*) – Field values.
- **name** (*str*, optional) – Name. to store the field. The default is “field”.
- **process** (*bool*, optional) – Whether to process field to apply mean, normalizer and trend. The default is True.
- **save** (*bool*, optional) – Whether to store the field under the given name. The default is True.

Returns *field* – Processed field values.

Return type *numpy.ndarray*

pre_pos(*pos, mesh_type='unstructured'*)

Preprocessing positions and mesh_type.

Parameters

- **pos** (*iterable*) – the position tuple, containing main direction and transversal directions
- **mesh_type** (*str*, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((d, n), `numpy.ndarray`) – the isometrized position tuple
- **shape** (`tuple`) – Shape of the resulting field.

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”

property dim

Dimension of the field.

Type `int`

property mean

The mean of the field.

Type `float` or `callable`

property model

The covariance model of the field.

Type `CovModel`

property name

The name of the class.

Type `str`

property normalizer

Normalizer of the field.

Type `Normalizer`

property trend

The trend of the field.

Type `float` or `callable`

property 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

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

```
class gstools.field.generator.IncomprRandMeth(model, mean_velocity=1.0, mode_no=1000,
                                              seed=None, verbose=False, sampling='auto',
                                              **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`
- **sampling** (`str`, optional) – Sampling strategy. Either
 - “auto”: select best strategy depending on given model
 - “inversion”: use inversion method
 - “mcmc”: use mcmc sampling
- ****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 [Kraichnan1970]:

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

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

References

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.

sampling `str`: Sampling strategy.

seed `int`: Seed of the master RNG.

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

verbose `bool`: Verbosity of the generator.

Methods

<code>__call__(pos)</code>	Calculate the random modes for the randomization method.
<code>get_nugget(shape)</code>	Generate normal distributed values for the nugget simulation.
<code>reset_seed([seed])</code>	Recalculate the random amplitudes and wave numbers with the given seed.
<code>update([model, seed])</code>	Update the model and the seed.

`__call__(pos)`

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 **pos** `((d, n), numpy.ndarray)` – the position tuple with d dimensions and n points.

Returns the random modes

Return type `numpy.ndarray`

`get_nugget(shape)`

Generate normal distributed values for the nugget simulation.

Parameters **shape** `(tuple)` – the shape of the summed modes

Returns **nugget** – the nugget in the same shape as the summed 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*

property mode_no

Number of modes in the randomization method.

Type *int*

property model

Covariance model of the spatial random field.

Type *CovModel*

property name

Name of the generator.

Type *str*

property sampling

Sampling strategy.

Type *str*

property 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*

property value_type

Type of the field values (scalar, vector).

Type *str*

property verbose

Verbosity of the generator.

Type *bool*

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

Bases: *object*

Randomization method for calculating isotropic 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*
- **sampling** (*str*, optional) – Sampling strategy. Either
 - “auto”: select best strategy depending on given model

- "inversion": use inversion method
- "mcmc": use mcmc sampling
- ****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 [Hesse2014]:

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

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
-

References

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.
- sampling** `str`: Sampling strategy.
- seed** `int`: Seed of the master RNG.
- value_type** `str`: Type of the field values (scalar, vector).
- verbose** `bool`: Verbosity of the generator.

Methods

<code>__call__(pos[, add_nugget])</code>	Calculate the random modes for the randomization method.
<code>get_nugget(shape)</code>	Generate normal distributed values for the nugget simulation.
<code>reset_seed([seed])</code>	Recalculate the random amplitudes and wave numbers with the given seed.
<code>update([model, seed])</code>	Update the model and the seed.

`__call__(pos, add_nugget=True)`

Calculate the random modes for the randomization method.

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

Parameters

- **pos** ((d, n), `numpy.ndarray`) – the position tuple with d dimensions and n points.
- **add_nugget** (`bool`) – Whether to add nugget noise to the field.

Returns the random modes

Return type `numpy.ndarray`

get_nugget(*shape*)

Generate normal distributed values for the nugget simulation.

Parameters **shape** (`tuple`) – the shape of the summed modes

Returns **nugget** – the nugget in the same shape as the summed 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`

property mode_no

Number of modes in the randomization method.

Type `int`

property model

Covariance model of the spatial random field.

Type `CovModel`

property name

Name of the generator.

Type `str`

property sampling

Sampling strategy.

Type `str`

property 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`

property value_type

Type of the field values (scalar, vector).

Type `str`

property verbose

Verbosity of the generator.

Type `bool`

gstools.field.upsampling

GStools subpackage providing upscaling routines for the spatial random field.

The following functions are provided

<code>var_coarse_graining(model[, point_volumes])</code>	Coarse Graining procedure to upscale the variance for uniform flow.
<code>var_no_scaling(model, *args, **kwargs)</code>	Dummy function to bypass scaling.

`gstools.field.upsampling.var_coarse_graining(model, point_volumes=0.0)`

Coarse Graining 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.upsampling.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*

3.8 gstools.variogram

GStools subpackage providing tools for estimating and fitting variograms.

Variogram estimation

<code>vario_estimate</code> (pos, field[, bin_edges, ...])	Estimates the empirical variogram.
<code>vario_estimate_axis</code> (field[, direction, ...])	Estimates the variogram along array axis.

gstools.variogram.vario_estimate

`gstools.variogram.vario_estimate`(pos, field, bin_edges=None, sampling_size=None, sampling_seed=None, estimator='matheron', latlon=False, direction=None, angles=None, angles_tol=0.39269908169872414, bandwidth=None, no_data=nan, mask=False, mesh_type='unstructured', return_counts=False, mean=None, normalizer=None, trend=None, fit_normalizer=False)

Estimates the empirical variogram.

The algorithm calculates following equation:

$$\gamma(r_k) = \frac{1}{2N(r_k)} \sum_{i=1}^{N(r_k)} (z(\mathbf{x}_i) - z(\mathbf{x}'_i))^2,$$

with $r_k \leq \|\mathbf{x}_i - \mathbf{x}'_i\| < r_{k+1}$ being the bins.

Or if the estimator “cressie” was chosen:

$$\gamma(r_k) = \frac{\frac{1}{2} \left(\frac{1}{N(r_k)} \sum_{i=1}^{N(r_k)} |z(\mathbf{x}_i) - z(\mathbf{x}'_i)|^{0.5} \right)^4}{0.457 + 0.494/N(r_k) + 0.045/N^2(r_k)},$$

with $r_k \leq \|\mathbf{x}_i - \mathbf{x}'_i\| < r_{k+1}$ being the bins. The Cressie estimator is more robust to outliers [Webster2007].

By providing *direction* vector[s] or angles, a directional variogram can be calculated. If multiple directions are given, a set of variograms will be returned. Directional binning is controlled by a given angle tolerance (*angles_tol*) and an optional *bandwidth*, that truncates the width of the search band around the given direction[s].

To reduce the calculation time, *sampling_size* could be passed to sample down the number of field points.

Parameters

- **pos** (`list`) – the position tuple, containing either the point coordinates (x, y, ...) or the axes descriptions (for `mesh_type='structured'`)
- **field** (`numpy.ndarray` or `list` of `numpy.ndarray`) – The spatially distributed data. Can also be of type `numpy.ma.MaskedArray` to use masked values. You can pass a list of fields, that will be used simultaneously. This could be helpful, when there are multiple realizations at the same points, with the same statistical properties.
- **bin_edges** (`numpy.ndarray`, optional) – the bins on which the variogram will be calculated. If `None` are given, standard bins provided by the `standard_bins` routine will be used. Default: `None`
- **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`

- **estimator** (`str`, optional) – the estimator function, possible choices:
 - “matheron”: the standard method of moments of Matheron
 - “cressie”: an estimator more robust to outliers
 Default: “matheron”
- **latlon** (`bool`, optional) – Whether the data is representing 2D fields on earths surface described by latitude and longitude. When using this, the estimator will use great-circle distance for variogram estimation. Note, that only an isotropic variogram can be estimated and a `ValueError` will be raised, if a direction was specified. Bin edges need to be given in radians in this case. Default: `False`
- **direction** (`list` of `numpy.ndarray`, optional) – directions to evaluate a directional variogram. Angular tolerance is given by `angles_tol`. bandwidth to cut off how wide the search for point pairs should be is given by `bandwidth`. You can provide multiple directions at once to get one variogram for each direction. For a single direction you can also use the `angles` parameter, to provide the direction by its spherical coordinates. Default: `None`
- **angles** (`numpy.ndarray`, optional) – the angles of the main axis to calculate the variogram for in radians angle definitions from ISO standard 80000-2:2009 for 1d this parameter will have no effect at all for 2d supply one angle which is azimuth φ (ccw from +x in xy plane) for 3d supply two angles which are azimuth φ (ccw from +x in xy plane) and inclination θ (cw from +z). Can be used instead of direction. Default: `None`
- **angles_tol** (`class:float`, optional) – the tolerance around the variogram angle to count a point as being within this direction from another point (the angular tolerance around the directional vector given by angles) Default: `np.pi/8 = 22.5°`
- **bandwidth** (`class:float`, optional) – bandwidth to cut off the angular tolerance for directional variograms. If `None` is given, only the `angles_tol` parameter will control the point selection. Default: `None`
- **no_data** (`float`, optional) – Value to identify missing data in the given field. Default: `numpy.nan`
- **mask** (`numpy.ndarray` of `bool`, optional) – Mask to deselect data in the given field. Default: `numpy.ma.nomask`
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’, indicates whether the pos tuple describes the axis or the point coordinates. Default: ‘unstructured’
- **return_counts** (`bool`, optional) – if set to true, this function will also return the number of data points found at each lag distance as a third return value Default: `False`
- **mean** (`float`, optional) – mean value used to shift normalized input data. Can also be a callable. The default is `None`.
- **normalizer** (`None` or `Normalizer`, optional) – Normalizer to be applied to the input data to gain normality. The default is `None`.
- **trend** (`None` or `float` or `callable`, optional) – A callable trend function. Should have the signature: `f(x, [y, z, ...])` If no normalizer is applied, this behaves equal to ‘mean’. The default is `None`.
- **fit_normalizer** (`bool`, optional) – Wheater to fit the data-normalizer to the given (detrended) field. Default: `False`

Returns

- **bin_center** ((n), `numpy.ndarray`) – The bin centers.
- **gamma** ((n) or (d, n), `numpy.ndarray`) – The estimated variogram values at bin centers. Is stacked if multiple *directions* (d>1) are given.

- **counts** ((n) or (d, n), `numpy.ndarray`, optional) – The number of point pairs found for each bin. Is stacked if multiple *directions* (d>1) are given. Only provided if *return_counts* is True.
- **normalizer** (`Normalizer`, optional) – The fitted normalizer for the given data. Only provided if *fit_normalizer* is True.

Notes

Internally uses double precision and also returns doubles.

References

`gstools.variogram.vario_estimate_axis`

`gstools.variogram.vario_estimate_axis(field, direction='x', estimator='matheron', no_data=nan)`
Estimates the variogram along array axis.

The indices of the given direction are used for the bins. Uniform spacings along the given axis are assumed.

The algorithm calculates following equation:

$$\gamma(r_k) = \frac{1}{2N(r_k)} \sum_{i=1}^{N(r_k)} (z(\mathbf{x}_i) - z(\mathbf{x}'_i))^2,$$

with $r_k \leq \|\mathbf{x}_i - \mathbf{x}'_i\| < r_{k+1}$ being the bins.

Or if the estimator “cressie” was chosen:

$$\gamma(r_k) = \frac{\frac{1}{2} \left(\frac{1}{N(r_k)} \sum_{i=1}^{N(r_k)} |z(\mathbf{x}_i) - z(\mathbf{x}'_i)|^{0.5} \right)^4}{0.457 + 0.494/N(r_k) + 0.045/N^2(r_k)},$$

with $r_k \leq \|\mathbf{x}_i - \mathbf{x}'_i\| < r_{k+1}$ being the bins. The Cressie estimator is more robust to outliers [Webster2007].

Parameters

- **field** (`numpy.ndarray` or `numpy.ma.MaskedArray`) – the spatially distributed data (can be masked)
- **direction** (`str` or `int`) – the axis over which the variogram will be estimated (x, y, z) or (0, 1, 2, ...)
- **estimator** (`str`, optional) – the estimator function, possible choices:
 - “matheron”: the standard method of moments of Matheron
 - “cressie”: an estimator more robust to outliersDefault: “matheron”
- **no_data** (`float`, optional) – Value to identify missing data in the given field. Default: `numpy.nan`

Returns the estimated variogram along the given direction.

Return type `numpy.ndarray`

Warning: It is assumed that the field is defined on an equidistant Cartesian grid.

Notes

Internally uses double precision and also returns doubles.

References

Binning

<code>standard_bins([pos, dim, latlon, mesh_type, ...])</code>	Get standard binning.
--	-----------------------

`gstools.variogram.standard_bins`

`gstools.variogram.standard_bins(pos=None, dim=2, latlon=False, mesh_type='unstructured', bin_no=None, max_dist=None)`

Get standard binning.

Parameters

- **pos** (`list`, optional) – the position tuple, containing either the point coordinates (x, y, ...) or the axes descriptions (for `mesh_type='structured'`)
- **dim** (`int`, optional) – Field dimension.
- **latlon** (`bool`, optional) – Whether the data is representing 2D fields on earth's surface described by latitude and longitude. When using this, the estimator will use great-circle distance for variogram estimation. Note, that only an isotropic variogram can be estimated and a `ValueError` will be raised, if a direction was specified. Bin edges need to be given in radians in this case. Default: `False`
- **mesh_type** (`str`, optional) – 'structured' / 'unstructured', indicates whether the pos tuple describes the axis or the point coordinates. Default: `'unstructured'`
- **bin_no** (`int`, optional) – number of bins to create. If `None` is given, will be determined by Sturges' rule from the number of points. Default: `None`
- **max_dist** (`float`, optional) – Cut of length for the bins. If `None` is given, it will be set to one third of the box-diameter from the given points. Default: `None`

Returns The generated bin edges.

Return type `numpy.ndarray`

Notes

Internally uses double precision and also returns doubles.

3.9 gstools.krige

GStools subpackage providing kriging.

Kriging Classes

<code>Krige(model, cond_pos, cond_val[, ...])</code>	A Swiss Army knife for kriging.
<code>Simple(model, cond_pos, cond_val[, mean, ...])</code>	Simple kriging.
<code>Ordinary(model, cond_pos, cond_val[, ...])</code>	Ordinary kriging.
<code>Universal(model, cond_pos, cond_val, ...[, ...])</code>	Universal kriging.
<code>ExtDrift(model, cond_pos, cond_val, ext_drift)</code>	External drift kriging (EDK).
<code>Detrended(model, cond_pos, cond_val, trend)</code>	Detrended simple kriging.

gstools.krige.Krige

```
class gstools.krige.Krige(model, cond_pos, cond_val, drift_functions=None, ext_drift=None,
                           mean=None, normalizer=None, trend=None, unbiased=True, exact=False,
                           cond_err='nugget', pseudo_inv=True, pseudo_inv_type='pinv',
                           fit_normalizer=False, fit_variogram=False)
```

Bases: `gstools.field.base.Field`

A Swiss Army knife for kriging.

A Kriging class enabling the basic kriging routines: Simple-, Ordinary-, Universal-, External Drift- and detrended/regression-Kriging as well as Kriging the Mean [Wackernagel2003].

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
- **drift_functions** (`list` of `callable`, `str` or `int`) – Either a list of callable functions, an integer representing the polynomial order of the drift or one of the following strings:
 - “linear” : regional linear drift (equals order=1)
 - “quadratic” : regional quadratic drift (equals order=2)
- **ext_drift** (`numpy.ndarray` or `None`, optional) – the external drift values at the given cond. positions.
- **mean** (`float`, optional) – mean value used to shift normalized conditioning data. Could also be a callable. The default is `None`.
- **normalizer** (`None` or `Normalizer`, optional) – Normalizer to be applied to the input data to gain normality. The default is `None`.
- **trend** (`None` or `float` or `callable`, optional) – A callable trend function. Should have the signature: `f(x, [y, z, ...])` This is used for detrended kriging, where the trended is subtracted from the conditions before kriging is applied. This can be used for regression kriging, where the trend function is determined by an external regression algorithm. If no normalizer is applied, this behaves equal to ‘mean’. The default is `None`.
- **unbiased** (`bool`, optional) – Whether the kriging weights should sum up to 1, so the estimator is unbiased. If unbiased is `False` and no drifts are given, this results in simple kriging. Default: `True`

- **exact** (`bool`, optional) – Whether the interpolator should reproduce the exact input values. If *False*, *cond_err* is interpreted as measurement error at the conditioning points and the result will be more smooth. Default: *False*
 - **cond_err** (`str`, :class `float` or `list`, optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
 - **pseudo_inv** (`bool`, optional) – Whether the kriging system is solved with the pseudo inverted kriging matrix. If *True*, this leads to more numerical stability and redundant points are averaged. But it can take more time. Default: *True*
 - **pseudo_inv_type** (`str` or `callable`, optional) – Here you can select the algorithm to compute the pseudo-inverse matrix:
 - “pinv”: use *pinv* from *scipy* which uses *lstsq*
 - “pinv2”: use *pinv2* from *scipy* which uses *SVD*
 - “pinvh”: use *pinvh* from *scipy* which uses eigen-values
- If you want to use another routine to invert the kriging matrix, you can pass a callable which takes a matrix and returns the inverse. Default: “pinv”
- **fit_normalizer** (`bool`, optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: *False*
 - **fit_variogram** (`bool`, optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the *standard_bins* routine. Default: *False*

Notes

If you have changed any properties in the class, you can update the kriging setup by calling *Krige.set_condition* without any arguments.

References

Attributes

cond_err `list`: The measurement errors at the condition points.
cond_ext_drift `numpy.ndarray`: The ext. drift at the conditions.
cond_mean `numpy.ndarray`: Trend at the conditions.
cond_no `int`: The number of the conditions.
cond_pos `list`: The position tuple of the conditions.
cond_trend `numpy.ndarray`: Trend at the conditions.
cond_val `list`: The values of the conditions.
dim `int`: Dimension of the field.
drift_functions `list` of `callable`: The drift functions.
drift_no `int`: Number of drift values per point.
exact `bool`: Whether the interpolator is exact.
ext_drift_no `int`: Number of external drift values per point.
has_const_mean `bool`: Whether the field has a constant mean or not.

int_drift_no *int*: Number of internal drift values per point.

krige_size *int*: Size of the kriging system.

mean *float* or *callable*: The mean of the field.

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

name *str*: The name of the kriging class.

normalizer *Normalizer*: Normalizer of the field.

pseudo_inv *bool*: Whether pseudo inverse matrix is used.

pseudo_inv_type *str*: Method selector for pseudo inverse calculation.

trend *float* or *callable*: The trend of the field.

unbiased *bool*: Whether the kriging is unbiased or not.

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

Methods

<code>__call__(pos[, mesh_type, ext_drift, ...])</code>	Generate the kriging field.
<code>get_mean([post_process])</code>	Calculate the estimated mean of the detrended field.
<code>mesh(mesh[, points, direction, name])</code>	Generate a field on a given meshio or ogs5py mesh.
<code>plot([field, fig, ax])</code>	Plot the spatial random field.
<code>post_field(field[, name, process, save])</code>	Postprocessing field values.
<code>pre_pos(pos[, mesh_type])</code>	Preprocessing positions and mesh_type.
<code>set_condition([cond_pos, cond_val, ...])</code>	Set the conditions for kriging.
<code>set_drift_functions([drift_functions])</code>	Set the drift functions for universal kriging.
<code>structured(*args, **kwargs)</code>	Generate a field on a structured mesh.
<code>to_pyvista([field_select, fieldname])</code>	Create a VTK/PyVista grid of the stored field.
<code>unstructured(*args, **kwargs)</code>	Generate a field on an unstructured mesh.
<code>vtk_export(filename[, field_select, fieldname])</code>	Export the stored field to vtk.

`__call__(pos, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True)`

Generate the kriging field.

The field is saved as *self.field* and is also returned. The error variance is saved as *self.krige_var* and is also returned.

Parameters

- **pos** (*list*) – the position tuple, containing main direction and transversal directions (x, [y, z])
- **mesh_type** (*str*, optional) – ‘structured’ / ‘unstructured’
- **ext_drift** (*numpy.ndarray* or *None*, optional) – the external drift values at the given positions (only for EDK)
- **chunk_size** (*int*, optional) – Chunk size to cut down the size of the kriging system to prevent memory errors. Default: *None*
- **only_mean** (*bool*, optional) – Whether to only calculate the mean of the kriging field. Default: *False*
- **return_var** (*bool*, optional) – Whether to return the variance along with the field. Default: *True*
- **post_process** (*bool*, optional) – Whether to apply mean, normalizer and trend to the field. Default: *True*

Returns

- **field** (`numpy.ndarray`) – the kriged field or `mean_field`
- **krige_var** (`numpy.ndarray`, optional) – the kriging error variance (if `return_var` is `True` and `only_mean` is `False`)

get_mean(*post_process=True*)

Calculate the estimated mean of the detrended field.

Parameters **post_process** (`bool`, optional) – Whether to apply field-mean and normalizer.
Default: `True`

Returns **mean** – Mean of the Kriging System.

Return type `float` or `None`

Notes

Only not `None` if the Kriging System has a constant mean. This means, no drift is given and the given field-mean is constant. The result is neglecting a potential given trend.

mesh(*mesh, points='centroids', direction='all', name='field', **kwargs*)

Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (`meshio.Mesh` or `ogs5py.MSH` or `PyVista mesh`) – The given meshio, ogs5py, or PyVista 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` or `list`, optional) – Here you can state which direction should be chosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass “xz”. By default, all directions are used. One can also pass a list of indices. Default: “all”
- **name** (`str` or `list of str`, optional) – Name(s) to store the field(s) in the given mesh as `point_data` or `cell_data`. If too few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to `Field.__call__`.

Notes

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

See: <https://github.com/nenschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: `Field.__call__`

plot(*field='field', fig=None, ax=None, **kwargs*)

Plot the spatial random field.

Parameters

- **field** (`str`, optional) – Field that should be plotted. 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`
- ****kwargs** – Forwarded to the plotting routine.

post_field(*field*, *name*='field', *process*=True, *save*=True)

Postprocessing field values.

Parameters

- **field** ([numpy.ndarray](#)) – Field values.
- **name** ([str](#), optional) – Name. to store the field. The default is “field”.
- **process** ([bool](#), optional) – Whether to process field to apply mean, normalizer and trend. The default is True.
- **save** ([bool](#), optional) – Whether to store the field under the given name. The default is True.

Returns **field** – Processed field values.

Return type [numpy.ndarray](#)

pre_pos(*pos*, *mesh_type*='unstructured')

Preprocessing positions and mesh_type.

Parameters

- **pos** ([iterable](#)) – the position tuple, containing main direction and transversal directions
- **mesh_type** ([str](#), optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((d, n), [numpy.ndarray](#)) – the isometrized position tuple
- **shape** ([tuple](#)) – Shape of the resulting field.

set_condition(*cond_pos*=None, *cond_val*=None, *ext_drift*=None, *cond_err*=None, *fit_normalizer*=False, *fit_variogram*=False)

Set the conditions for kriging.

This method could also be used to update the kriging setup, when properties were changed. Then you can call it without arguments.

Parameters

- **cond_pos** ([list](#), optional) – the position tuple of the conditions (x, [y, z]). Default: current.
- **cond_val** ([numpy.ndarray](#), optional) – the values of the conditions. Default: current.
- **ext_drift** ([numpy.ndarray](#) or *None*, optional) – the external drift values at the given conditions (only for EDK) For multiple external drifts, the first dimension should be the index of the drift term. When passing *None*, the existing external drift will be used.
- **cond_err** ([str](#), :class [float](#), [list](#), optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be <= nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **fit_normalizer** ([bool](#), optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: False
- **fit_variogram** ([bool](#), optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the [standard_bins](#) routine. Default: False

set_drift_functions(*drift_functions*=None)

Set the drift functions for universal kriging.

Parameters `drift_functions` (`list` of `callable`, `str` or `int`) – Either a list of callable functions, an integer representing the polynomial order of the drift or one of the following strings:

- "linear" : regional linear drift (equals order=1)
- "quadratic" : regional quadratic drift (equals order=2)

Raises `ValueError` – If the given drift functions are not callable.

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"

property cond_err

The measurement errors at the condition points.

Type `list`

property cond_ext_drift

The ext. drift at the conditions.

Type `numpy.ndarray`

property cond_mean

Trend at the conditions.

Type `numpy.ndarray`

property cond_no

The number of the conditions.

Type `int`

property cond_pos

The position tuple of the conditions.

Type `list`

property cond_trend

Trend at the conditions.

Type `numpy.ndarray`

property cond_val

The values of the conditions.

Type `list`

property dim

Dimension of the field.

Type `int`

property drift_functions

The drift functions.

Type `list` of `callable`

property drift_no

Number of drift values per point.

Type `int`

property exact

Whether the interpolator is exact.

Type `bool`

property ext_drift_no

Number of external drift values per point.

Type `int`

property has_const_mean

Whether the field has a constant mean or not.

Type `bool`

property int_drift_no

Number of internal drift values per point.

Type `int`

property krige_size

Size of the kriging system.

Type `int`

property mean

The mean of the field.

Type `float` or `callable`

property model

The covariance model of the field.

Type `CovModel`

property name

The name of the kriging class.

Type `str`

property normalizer

Normalizer of the field.

Type `Normalizer`

property pseudo_inv

Whether pseudo inverse matrix is used.

Type `bool`

property pseudo_inv_type

Method selector for pseudo inverse calculation.

Type `str`

property trend

The trend of the field.

Type `float` or `callable`

property unbiased

Whether the kriging is unbiased or not.

Type `bool`

property value_type

Type of the field values (scalar, vector).

Type `str`

gstools.krige.Simple

```
class gstools.krige.Simple(model, cond_pos, cond_val, mean=0.0, normalizer=None, trend=None,
                           exact=False, cond_err='nugget', pseudo_inv=True, pseudo_inv_type='pinv',
                           fit_normalizer=False, fit_variogram=False)
```

Bases: [gstools.krige.base.Krige](#)

Simple kriging.

Simple kriging is used to interpolate data with a given mean.

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
- **mean** ([float](#), optional) – mean value used to shift normalized conditioning data. Could also be a callable. The default is None.
- **normalizer** ([None](#) or [Normalizer](#), optional) – Normalizer to be applied to the input data to gain normality. The default is None.
- **trend** ([None](#) or [float](#) or [callable](#), optional) – A callable trend function. Should have the signature: `f(x, [y, z, ...])` This is used for detrended kriging, where the trended is subtracted from the conditions before kriging is applied. This can be used for regression kriging, where the trend function is determined by an external regression algorithm. If no normalizer is applied, this behaves equal to ‘mean’. The default is None.
- **exact** ([bool](#), optional) – Whether the interpolator should reproduce the exact input values. If *False*, *cond_err* is interpreted as measurement error at the conditioning points and the result will be more smooth. Default: *False*
- **cond_err** ([str](#), :class [float](#) or [list](#), optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **pseudo_inv** ([bool](#), optional) – Whether the kriging system is solved with the pseudo inverted kriging matrix. If *True*, this leads to more numerical stability and redundant points are averaged. But it can take more time. Default: *True*
- **pseudo_inv_type** ([str](#) or [callable](#), optional) – Here you can select the algorithm to compute the pseudo-inverse matrix:
 - “pinv”: use *pinv* from *scipy* which uses *lstsq*
 - “pinv2”: use *pinv2* from *scipy* which uses *SVD*
 - “pinvh”: use *pinvh* from *scipy* which uses eigen-values

If you want to use another routine to invert the kriging matrix, you can pass a callable which takes a matrix and returns the inverse. Default: “pinv”

- **fit_normalizer** ([bool](#), optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: *False*
- **fit_variogram** ([bool](#), optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the [standard_bins](#) routine. Default: *False*

Attributes

cond_err [list](#): The measurement errors at the condition points.

cond_ext_drift `numpy.ndarray`: The ext. drift at the conditions.

cond_mean `numpy.ndarray`: Trend at the conditions.

cond_no `int`: The number of the conditions.

cond_pos `list`: The position tuple of the conditions.

cond_trend `numpy.ndarray`: Trend at the conditions.

cond_val `list`: The values of the conditions.

dim `int`: Dimension of the field.

drift_functions `list` of `callable`: The drift functions.

drift_no `int`: Number of drift values per point.

exact `bool`: Whether the interpolator is exact.

ext_drift_no `int`: Number of external drift values per point.

has_const_mean `bool`: Whether the field has a constant mean or not.

int_drift_no `int`: Number of internal drift values per point.

krige_size `int`: Size of the kriging system.

mean `float` or `callable`: The mean of the field.

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

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

normalizer `Normalizer`: Normalizer of the field.

pseudo_inv `bool`: Whether pseudo inverse matrix is used.

pseudo_inv_type `str`: Method selector for pseudo inverse calculation.

trend `float` or `callable`: The trend of the field.

unbiased `bool`: Whether the kriging is unbiased or not.

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

Methods

<code>__call__(pos[, mesh_type, ext_drift, ...])</code>	Generate the kriging field.
<code>get_mean([post_process])</code>	Calculate the estimated mean of the detrended field.
<code>mesh(mesh[, points, direction, name])</code>	Generate a field on a given meshio or ogs5py mesh.
<code>plot([field, fig, ax])</code>	Plot the spatial random field.
<code>post_field(field[, name, process, save])</code>	Postprocessing field values.
<code>pre_pos(pos[, mesh_type])</code>	Preprocessing positions and mesh_type.
<code>set_condition([cond_pos, cond_val, ...])</code>	Set the conditions for kriging.
<code>set_drift_functions([drift_functions])</code>	Set the drift functions for universal kriging.
<code>structured(*args, **kwargs)</code>	Generate a field on a structured mesh.
<code>to_pyvista([field_select, fieldname])</code>	Create a VTK/PyVista grid of the stored field.
<code>unstructured(*args, **kwargs)</code>	Generate a field on an unstructured mesh.
<code>vtk_export(filename[, field_select, fieldname])</code>	Export the stored field to vtk.

`__call__(pos, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True)`

Generate the kriging field.

The field is saved as `self.field` and is also returned. The error variance is saved as `self.krige_var` and is also returned.

Parameters

- **pos** (*list*) – the position tuple, containing main direction and transversal directions (x, [y, z])
- **mesh_type** (*str*, optional) – ‘structured’ / ‘unstructured’
- **ext_drift** (*numpy.ndarray* or *None*, optional) – the external drift values at the given positions (only for EDK)
- **chunk_size** (*int*, optional) – Chunk size to cut down the size of the kriging system to prevent memory errors. Default: *None*
- **only_mean** (*bool*, optional) – Whether to only calculate the mean of the kriging field. Default: *False*
- **return_var** (*bool*, optional) – Whether to return the variance along with the field. Default: *True*
- **post_process** (*bool*, optional) – Whether to apply mean, normalizer and trend to the field. Default: *True*

Returns

- **field** (*numpy.ndarray*) – the kriged field or mean_field
- **krige_var** (*numpy.ndarray*, optional) – the kriging error variance (if return_var is *True* and only_mean is *False*)

get_mean(*post_process=True*)

Calculate the estimated mean of the detrended field.

Parameters **post_process** (*bool*, optional) – Whether to apply field-mean and normalizer. Default: *True*

Returns **mean** – Mean of the Kriging System.

Return type *float* or *None*

Notes

Only not *None* if the Kriging System has a constant mean. This means, no drift is given and the given field-mean is constant. The result is neglecting a potential given trend.

mesh(*mesh*, *points='centroids'*, *direction='all'*, *name='field'*, ***kwargs*)

Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (*meshio.Mesh* or *ogs5py.MSH* or *PyVista mesh*) – The given meshio, ogs5py, or PyVista 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* or *list*, 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”. By default, all directions are used. One can also pass a list of indices. Default: “all”
- **name** (*str* or *list of str*, optional) – Name(s) to store the field(s) in the given mesh as point_data or cell_data. If to few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to *Field.__call__*.

Notes

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

See: <https://github.com/nschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: `Field.__call__`

plot(*field*='field', *fig*=None, *ax*=None, ***kwargs*)

Plot the spatial random field.

Parameters

- **field** (`str`, optional) – Field that should be plotted. 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`
- ****kwargs** – Forwarded to the plotting routine.

post_field(*field*, *name*='field', *process*=True, *save*=True)

Postprocessing field values.

Parameters

- **field** (`numpy.ndarray`) – Field values.
- **name** (`str`, optional) – Name. to store the field. The default is “field”.
- **process** (`bool`, optional) – Whether to process field to apply mean, normalizer and trend. The default is True.
- **save** (`bool`, optional) – Whether to store the field under the given name. The default is True.

Returns `field` – Processed field values.

Return type `numpy.ndarray`

pre_pos(*pos*, *mesh_type*='unstructured')

Preprocessing positions and mesh_type.

Parameters

- **pos** (`iterable`) – the position tuple, containing main direction and transversal directions
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((`d`, `n`), `numpy.ndarray`) – the isometrized position tuple
- **shape** (`tuple`) – Shape of the resulting field.

set_condition(*cond_pos*=None, *cond_val*=None, *ext_drift*=None, *cond_err*=None, *fit_normalizer*=False, *fit_variogram*=False)

Set the conditions for kriging.

This method could also be used to update the kriging setup, when properties were changed. Then you can call it without arguments.

Parameters

- **cond_pos** (`list`, optional) – the position tuple of the conditions (x, [y, z]). Default: current.

- **cond_val** (`numpy.ndarray`, optional) – the values of the conditions. Default: current.
- **ext_drift** (`numpy.ndarray` or `None`, optional) – the external drift values at the given conditions (only for EDK) For multiple external drifts, the first dimension should be the index of the drift term. When passing `None`, the existing external drift will be used.
- **cond_err** (`str`, :class `float`, `list`, optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err='nugget'”. Default: “nugget”
- **fit_normalizer** (`bool`, optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: False
- **fit_variogram** (`bool`, optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the `standard_bins` routine. Default: False

set_drift_functions(*drift_functions=None*)

Set the drift functions for universal kriging.

Parameters **drift_functions** (`list` of `callable`, `str` or `int`) – Either a list of callable functions, an integer representing the polynomial order of the drift or one of the following strings:

- “linear” : regional linear drift (equals order=1)
- “quadratic” : regional quadratic drift (equals order=2)

Raises `ValueError` – If the given drift functions are not callable.

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”

property cond_err

The measurement errors at the condition points.

Type `list`

property cond_ext_drift

The ext. drift at the conditions.

Type `numpy.ndarray`

property cond_mean

Trend at the conditions.

Type `numpy.ndarray`

property cond_no

The number of the conditions.

Type `int`

property cond_pos

The position tuple of the conditions.

Type `list`

property cond_trend

Trend at the conditions.

Type `numpy.ndarray`

property cond_val

The values of the conditions.

Type `list`

property dim

Dimension of the field.

Type `int`

property drift_functions

The drift functions.

Type `list` of `callable`

property drift_no

Number of drift values per point.

Type `int`

property exact

Whether the interpolator is exact.

Type `bool`

property ext_drift_no

Number of external drift values per point.

Type `int`

property has_const_mean

Whether the field has a constant mean or not.

Type `bool`

property int_drift_no

Number of internal drift values per point.

Type `int`

property krige_size

Size of the kriging system.

Type `int`

property mean

The mean of the field.

Type `float` or `callable`

property model

The covariance model of the field.

Type `CovModel`

property name

The name of the kriging class.

Type `str`

property normalizer

Normalizer of the field.

Type `Normalizer`

property pseudo_inv

Whether pseudo inverse matrix is used.

Type `bool`

property pseudo_inv_type

Method selector for pseudo inverse calculation.

Type `str`

property trend

The trend of the field.

Type `float` or `callable`

property unbiased

Whether the kriging is unbiased or not.

Type `bool`

property value_type

Type of the field values (scalar, vector).

Type `str`

gstools.krige.Ordinary

```
class gstools.krige.Ordinary(model, cond_pos, cond_val, normalizer=None, trend=None, exact=False,
                             cond_err='nugget', pseudo_inv=True, pseudo_inv_type='pinv',
                             fit_normalizer=False, fit_variogram=False)
```

Bases: [gstools.krige.base.Krige](#)

Ordinary kriging.

Ordinary kriging is used to interpolate data and estimate a proper mean.

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
- **normalizer** ([None](#) or [Normalizer](#), optional) – Normalizer to be applied to the input data to gain normality. The default is [None](#).
- **trend** ([None](#) or [float](#) or [callable](#), optional) – A callable trend function. Should have the signature: `f(x, [y, z, ...])` This is used for detrended kriging, where the trended is subtracted from the conditions before kriging is applied. This can be used for regression kriging, where the trend function is determined by an external regression algorithm. If no normalizer is applied, this behaves equal to ‘mean’. The default is [None](#).
- **exact** ([bool](#), optional) – Whether the interpolator should reproduce the exact input values. If [False](#), `cond_err` is interpreted as measurement error at the conditioning points and the result will be more smooth. Default: [False](#)
- **cond_err** ([str](#), :class [float](#) or [list](#), optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err=‘nugget’”. Default: “nugget”
- **pseudo_inv** ([bool](#), optional) – Whether the kriging system is solved with the pseudo inverted kriging matrix. If [True](#), this leads to more numerical stability and redundant points are averaged. But it can take more time. Default: [True](#)
- **pseudo_inv_type** ([str](#) or [callable](#), optional) – Here you can select the algorithm to compute the pseudo-inverse matrix:
 - “pinv”: use `pinv` from `scipy` which uses `lstsq`
 - “pinv2”: use `pinv2` from `scipy` which uses `SVD`
 - “pinvh”: use `pinvh` from `scipy` which uses eigen-values

If you want to use another routine to invert the kriging matrix, you can pass a callable which takes a matrix and returns the inverse. Default: “pinv”

- **fit_normalizer** ([bool](#), optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: [False](#)
- **fit_variogram** ([bool](#), optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the [standard_bins](#) routine. Default: [False](#)

Attributes

[cond_err](#) [list](#): The measurement errors at the condition points.

[cond_ext_drift](#) [numpy.ndarray](#): The ext. drift at the conditions.

[cond_mean](#) [numpy.ndarray](#): Trend at the conditions.

cond_no *int*: The number of the conditions.

cond_pos *list*: The position tuple of the conditions.

cond_trend *numpy.ndarray*: Trend at the conditions.

cond_val *list*: The values of the conditions.

dim *int*: Dimension of the field.

drift_functions *list of callable*: The drift functions.

drift_no *int*: Number of drift values per point.

exact *bool*: Whether the interpolator is exact.

ext_drift_no *int*: Number of external drift values per point.

has_const_mean *bool*: Whether the field has a constant mean or not.

int_drift_no *int*: Number of internal drift values per point.

krige_size *int*: Size of the kriging system.

mean *float or callable*: The mean of the field.

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

name *str*: The name of the kriging class.

normalizer *Normalizer*: Normalizer of the field.

pseudo_inv *bool*: Whether pseudo inverse matrix is used.

pseudo_inv_type *str*: Method selector for pseudo inverse calculation.

trend *float or callable*: The trend of the field.

unbiased *bool*: Whether the kriging is unbiased or not.

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

Methods

<code>__call__(pos[, mesh_type, ext_drift, ...])</code>	Generate the kriging field.
<code>get_mean([post_process])</code>	Calculate the estimated mean of the detrended field.
<code>mesh(mesh[, points, direction, name])</code>	Generate a field on a given meshio or ogs5py mesh.
<code>plot([field, fig, ax])</code>	Plot the spatial random field.
<code>post_field(field[, name, process, save])</code>	Postprocessing field values.
<code>pre_pos(pos[, mesh_type])</code>	Preprocessing positions and mesh_type.
<code>set_condition([cond_pos, cond_val, ...])</code>	Set the conditions for kriging.
<code>set_drift_functions([drift_functions])</code>	Set the drift functions for universal kriging.
<code>structured(*args, **kwargs)</code>	Generate a field on a structured mesh.
<code>to_pyvista([field_select, fieldname])</code>	Create a VTK/PyVista grid of the stored field.
<code>unstructured(*args, **kwargs)</code>	Generate a field on an unstructured mesh.
<code>vtk_export(filename[, field_select, fieldname])</code>	Export the stored field to vtk.

`__call__(pos, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True)`

Generate the kriging field.

The field is saved as *self.field* and is also returned. The error variance is saved as *self.krige_var* and is also returned.

Parameters

- **pos** (*list*) – the position tuple, containing main direction and transversal directions (x, [y, z])
- **mesh_type** (*str*, optional) – ‘structured’ / ‘unstructured’
- **ext_drift** (*numpy.ndarray* or *None*, optional) – the external drift values at the given positions (only for EDK)
- **chunk_size** (*int*, optional) – Chunk size to cut down the size of the kriging system to prevent memory errors. Default: *None*
- **only_mean** (*bool*, optional) – Whether to only calculate the mean of the kriging field. Default: *False*
- **return_var** (*bool*, optional) – Whether to return the variance along with the field. Default: *True*
- **post_process** (*bool*, optional) – Whether to apply mean, normalizer and trend to the field. Default: *True*

Returns

- **field** (*numpy.ndarray*) – the kriged field or mean_field
- **krige_var** (*numpy.ndarray*, optional) – the kriging error variance (if return_var is *True* and only_mean is *False*)

get_mean(*post_process=True*)

Calculate the estimated mean of the detrended field.

Parameters **post_process** (*bool*, optional) – Whether to apply field-mean and normalizer. Default: *True*

Returns **mean** – Mean of the Kriging System.

Return type *float* or *None*

Notes

Only not *None* if the Kriging System has a constant mean. This means, no drift is given and the given field-mean is constant. The result is neglecting a potential given trend.

mesh(*mesh*, *points='centroids'*, *direction='all'*, *name='field'*, ***kwargs*)

Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (*meshio.Mesh* or *ogs5py.MSH* or *PyVista mesh*) – The given meshio, ogs5py, or PyVista 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* or *list*, optional) – Here you can state which direction should be chosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass “xz”. By default, all directions are used. One can also pass a list of indices. Default: “all”
- **name** (*str* or *list of str*, optional) – Name(s) to store the field(s) in the given mesh as point_data or cell_data. If to few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to *Field.__call__*.

Notes

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

See: <https://github.com/nenschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: `Field.__call__`

plot(*field*='field', *fig*=None, *ax*=None, ***kwargs*)

Plot the spatial random field.

Parameters

- **field** (`str`, optional) – Field that should be plotted. 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`
- ****kwargs** – Forwarded to the plotting routine.

post_field(*field*, *name*='field', *process*=True, *save*=True)

Postprocessing field values.

Parameters

- **field** (`numpy.ndarray`) – Field values.
- **name** (`str`, optional) – Name. to store the field. The default is “field”.
- **process** (`bool`, optional) – Whether to process field to apply mean, normalizer and trend. The default is True.
- **save** (`bool`, optional) – Whether to store the field under the given name. The default is True.

Returns **field** – Processed field values.

Return type `numpy.ndarray`

pre_pos(*pos*, *mesh_type*='unstructured')

Preprocessing positions and mesh_type.

Parameters

- **pos** (`iterable`) – the position tuple, containing main direction and transversal directions
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((`d`, `n`), `numpy.ndarray`) – the isometrized position tuple
- **shape** (`tuple`) – Shape of the resulting field.

set_condition(*cond_pos*=None, *cond_val*=None, *ext_drift*=None, *cond_err*=None, *fit_normalizer*=False, *fit_variogram*=False)

Set the conditions for kriging.

This method could also be used to update the kriging setup, when properties were changed. Then you can call it without arguments.

Parameters

- **cond_pos** (`list`, optional) – the position tuple of the conditions (x, [y, z]). Default: current.
- **cond_val** (`numpy.ndarray`, optional) – the values of the conditions. Default: current.

- **ext_drift** (`numpy.ndarray` or `None`, optional) – the external drift values at the given conditions (only for EDK) For multiple external drifts, the first dimension should be the index of the drift term. When passing `None`, the existing external drift will be used.
- **cond_err** (`str`, :class `float`, `list`, optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **fit_normalizer** (`bool`, optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: False
- **fit_variogram** (`bool`, optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the `standard_bins` routine. Default: False

set_drift_functions(*drift_functions=None*)

Set the drift functions for universal kriging.

Parameters **drift_functions** (`list` of `callable`, `str` or `int`) – Either a list of callable functions, an integer representing the polynomial order of the drift or one of the following strings:

- “linear” : regional linear drift (equals order=1)
- “quadratic” : regional quadratic drift (equals order=2)

Raises **ValueError** – If the given drift functions are not callable.

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”

property cond_err

The measurement errors at the condition points.

Type `list`

property cond_ext_drift

The ext. drift at the conditions.

Type `numpy.ndarray`

property cond_mean

Trend at the conditions.

Type `numpy.ndarray`

property cond_no

The number of the conditions.

Type `int`

property cond_pos

The position tuple of the conditions.

Type `list`

property cond_trend

Trend at the conditions.

Type `numpy.ndarray`

property cond_val

The values of the conditions.

Type `list`

property dim

Dimension of the field.

Type `int`

property drift_functions

The drift functions.

Type `list` of `callable`

property drift_no

Number of drift values per point.

Type `int`

property exact

Whether the interpolator is exact.

Type `bool`

property ext_drift_no

Number of external drift values per point.

Type `int`

property has_const_mean

Whether the field has a constant mean or not.

Type `bool`

property int_drift_no

Number of internal drift values per point.

Type `int`

property krige_size

Size of the kriging system.

Type `int`

property mean

The mean of the field.

Type `float` or `callable`

property model

The covariance model of the field.

Type `CovModel`

property name

The name of the kriging class.

Type `str`

property normalizer

Normalizer of the field.

Type `Normalizer`

property pseudo_inv

Whether pseudo inverse matrix is used.

Type `bool`

property pseudo_inv_type

Method selector for pseudo inverse calculation.

Type `str`

property trend

The trend of the field.

Type `float` or `callable`

property unbiased

Whether the kriging is unbiased or not.

Type `bool`

property value_type

Type of the field values (scalar, vector).

Type `str`

gstools.krige.Universal

```
class gstools.krige.Universal(model, cond_pos, cond_val, drift_functions, normalizer=None,
                             trend=None, exact=False, cond_err='nugget', pseudo_inv=True,
                             pseudo_inv_type='pinv', fit_normalizer=False, fit_variogram=False)
```

Bases: [gstools.krige.base.Krige](#)

Universal kriging.

Universal kriging is used to interpolate given data with a variable mean, that is determined by a functional drift.

This estimator is set to be unbiased by default. This means, that the weights in the kriging equation sum up to 1. Consequently no constant function needs to be given for a constant drift, since the unbiased condition is applied to all given drift functions.

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
- **drift_functions** ([list](#) of [callable](#), [str](#) or [int](#)) – Either a list of callable functions, an integer representing the polynomial order of the drift or one of the following strings:
 - “linear” : regional linear drift (equals order=1)
 - “quadratic” : regional quadratic drift (equals order=2)
- **normalizer** ([None](#) or [Normalizer](#), optional) – Normalizer to be applied to the input data to gain normality. The default is [None](#).
- **trend** ([None](#) or [float](#) or [callable](#), optional) – A callable trend function. Should have the signature: `f(x, [y, z, ...])` This is used for detrended kriging, where the trended is subtracted from the conditions before kriging is applied. This can be used for regression kriging, where the trend function is determined by an external regression algorithm. If no normalizer is applied, this behaves equal to ‘mean’. The default is [None](#).
- **exact** ([bool](#), optional) – Whether the interpolator should reproduce the exact input values. If [False](#), `cond_err` is interpreted as measurement error at the conditioning points and the result will be more smooth. Default: [False](#)
- **cond_err** ([str](#), :class: [float](#) or [list](#), optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **pseudo_inv** ([bool](#), optional) – Whether the kriging system is solved with the pseudo inverted kriging matrix. If [True](#), this leads to more numerical stability and redundant points are averaged. But it can take more time. Default: [True](#)
- **pseudo_inv_type** ([str](#) or [callable](#), optional) – Here you can select the algorithm to compute the pseudo-inverse matrix:
 - “pinv”: use `pinv` from `scipy` which uses `lstsq`
 - “pinv2”: use `pinv2` from `scipy` which uses `SVD`
 - “pinvh”: use `pinvh` from `scipy` which uses eigen-valuesIf you want to use another routine to invert the kriging matrix, you can pass a callable which takes a matrix and returns the inverse. Default: “pinv”
- **fit_normalizer** ([bool](#), optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: [False](#)

- **fit_variogram** (*bool*, optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the *standard_bins* routine. Default: False

Attributes

cond_err *list*: The measurement errors at the condition points.

cond_ext_drift *numpy.ndarray*: The ext. drift at the conditions.

cond_mean *numpy.ndarray*: Trend at the conditions.

cond_no *int*: The number of the conditions.

cond_pos *list*: The position tuple of the conditions.

cond_trend *numpy.ndarray*: Trend at the conditions.

cond_val *list*: The values of the conditions.

dim *int*: Dimension of the field.

drift_functions *list of callable*: The drift functions.

drift_no *int*: Number of drift values per point.

exact *bool*: Whether the interpolator is exact.

ext_drift_no *int*: Number of external drift values per point.

has_const_mean *bool*: Whether the field has a constant mean or not.

int_drift_no *int*: Number of internal drift values per point.

krige_size *int*: Size of the kriging system.

mean *float or callable*: The mean of the field.

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

name *str*: The name of the kriging class.

normalizer *Normalizer*: Normalizer of the field.

pseudo_inv *bool*: Whether pseudo inverse matrix is used.

pseudo_inv_type *str*: Method selector for pseudo inverse calculation.

trend *float or callable*: The trend of the field.

unbiased *bool*: Whether the kriging is unbiased or not.

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

Methods

__call__ (pos[, mesh_type, ext_drift, ...])	Generate the kriging field.
get_mean ([post_process])	Calculate the estimated mean of the detrended 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.
post_field (field[, name, process, save])	Postprocessing field values.
pre_pos (pos[, mesh_type])	Preprocessing positions and mesh_type.
set_condition ([cond_pos, cond_val, ...])	Set the conditions for kriging.
set_drift_functions ([drift_functions])	Set the drift functions for universal 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.

continues on next page

Table 49 – continued from previous page

<code>unstructured(*args, **kwargs)</code>	Generate a field on an unstructured mesh.
<code>vtk_export(filename[, field_select, fieldname])</code>	Export the stored field to vtk.

`__call__(pos, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True)`

Generate the kriging field.

The field is saved as `self.field` and is also returned. The error variance is saved as `self.krige_var` and is also returned.

Parameters

- **pos** (`list`) – the position tuple, containing main direction and transversal directions (x, [y, z])
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’
- **ext_drift** (`numpy.ndarray` or `None`, optional) – the external drift values at the given positions (only for EDK)
- **chunk_size** (`int`, optional) – Chunk size to cut down the size of the kriging system to prevent memory errors. Default: `None`
- **only_mean** (`bool`, optional) – Whether to only calculate the mean of the kriging field. Default: `False`
- **return_var** (`bool`, optional) – Whether to return the variance along with the field. Default: `True`
- **post_process** (`bool`, optional) – Whether to apply mean, normalizer and trend to the field. Default: `True`

Returns

- **field** (`numpy.ndarray`) – the kriged field or `mean_field`
- **krige_var** (`numpy.ndarray`, optional) – the kriging error variance (if `return_var` is `True` and `only_mean` is `False`)

`get_mean(post_process=True)`

Calculate the estimated mean of the detrended field.

Parameters **post_process** (`bool`, optional) – Whether to apply field-mean and normalizer. Default: `True`

Returns **mean** – Mean of the Kriging System.

Return type `float` or `None`

Notes

Only not `None` if the Kriging System has a constant mean. This means, no drift is given and the given field-mean is constant. The result is neglecting a potential given trend.

`mesh(mesh, points='centroids', direction='all', name='field', **kwargs)`

Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (`meshio.Mesh` or `ogs5py.MSH` or `PyVista mesh`) – The given meshio, ogs5py, or PyVista 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` or `list`, 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”. By default, all directions are used. One can also pass a list of indices. Default: “all”
- **name** (`str` or `list` of `str`, optional) – Name(s) to store the field(s) in the given mesh as `point_data` or `cell_data`. If to few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to `Field.__call__`.

Notes

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

See: <https://github.com/nschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: `Field.__call__`

plot(*field*='field', *fig*=None, *ax*=None, **kwargs)

Plot the spatial random field.

Parameters

- **field** (`str`, optional) – Field that should be plotted. 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`
- ****kwargs** – Forwarded to the plotting routine.

post_field(*field*, *name*='field', *process*=True, *save*=True)

Postprocessing field values.

Parameters

- **field** (`numpy.ndarray`) – Field values.
- **name** (`str`, optional) – Name. to store the field. The default is “field”.
- **process** (`bool`, optional) – Whether to process field to apply mean, normalizer and trend. The default is True.
- **save** (`bool`, optional) – Whether to store the field under the given name. The default is True.

Returns **field** – Processed field values.

Return type `numpy.ndarray`

pre_pos(*pos*, *mesh_type*='unstructured')

Preprocessing positions and mesh_type.

Parameters

- **pos** (`iterable`) – the position tuple, containing main direction and transversal directions
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((d, n), `numpy.ndarray`) – the isometrized position tuple
- **shape** (`tuple`) – Shape of the resulting field.

set_condition(*cond_pos=None, cond_val=None, ext_drift=None, cond_err=None, fit_normalizer=False, fit_variogram=False*)

Set the conditions for kriging.

This method could also be used to update the kriging setup, when properties were changed. Then you can call it without arguments.

Parameters

- **cond_pos** (*list*, optional) – the position tuple of the conditions (x, [y, z]). Default: current.
- **cond_val** (*numpy.ndarray*, optional) – the values of the conditions. Default: current.
- **ext_drift** (*numpy.ndarray* or *None*, optional) – the external drift values at the given conditions (only for EDK) For multiple external drifts, the first dimension should be the index of the drift term. When passing *None*, the existing external drift will be used.
- **cond_err** (*str*, :class *float*, *list*, optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **fit_normalizer** (*bool*, optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: False
- **fit_variogram** (*bool*, optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the [standard_bins](#) routine. Default: False

set_drift_functions(*drift_functions=None*)

Set the drift functions for universal kriging.

Parameters **drift_functions** (*list* of *callable*, *str* or *int*) – Either a list of callable functions, an integer representing the polynomial order of the drift or one of the following strings:

- “linear” : regional linear drift (equals order=1)
- “quadratic” : regional quadratic drift (equals order=2)

Raises **ValueError** – If the given drift functions are not callable.

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”

property cond_err

The measurement errors at the condition points.

Type `list`

property cond_ext_drift

The ext. drift at the conditions.

Type `numpy.ndarray`

property cond_mean

Trend at the conditions.

Type `numpy.ndarray`

property cond_no

The number of the conditions.

Type `int`

property cond_pos

The position tuple of the conditions.

Type `list`

property cond_trend

Trend at the conditions.

Type `numpy.ndarray`

property cond_val

The values of the conditions.

Type `list`

property dim

Dimension of the field.

Type `int`

property drift_functions

The drift functions.

Type `list` of `callable`

property drift_no

Number of drift values per point.

Type `int`

property exact

Whether the interpolator is exact.

Type `bool`

property ext_drift_no

Number of external drift values per point.

Type `int`

property has_const_mean

Whether the field has a constant mean or not.

Type `bool`

property int_drift_no

Number of internal drift values per point.

Type `int`

property krige_size

Size of the kriging system.

Type `int`

property mean

The mean of the field.

Type `float` or `callable`

property model

The covariance model of the field.

Type `CovModel`

property name

The name of the kriging class.

Type `str`

property normalizer

Normalizer of the field.

Type `Normalizer`

property pseudo_inv

Whether pseudo inverse matrix is used.

Type `bool`

property pseudo_inv_type

Method selector for pseudo inverse calculation.

Type `str`

property trend

The trend of the field.

Type `float` or `callable`

property unbiased

Whether the kriging is unbiased or not.

Type `bool`

property value_type

Type of the field values (scalar, vector).

Type `str`

gstools.krige.ExtDrift

```
class gstools.krige.ExtDrift(model, cond_pos, cond_val, ext_drift, normalizer=None, trend=None,
                             exact=False, cond_err='nugget', pseudo_inv=True,
                             pseudo_inv_type='pinv', fit_normalizer=False, fit_variogram=False)
```

Bases: [gstools.krige.base.Krige](#)

External drift kriging (EDK).

External drift kriging is used to interpolate given data with a variable mean, that is determined by an external drift.

This estimator is set to be unbiased by default. This means, that the weights in the kriging equation sum up to 1. Consequently no constant external drift needs to be given to estimate a proper mean.

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
- **ext_drift** ([numpy.ndarray](#)) – the external drift values at the given condition positions.
- **normalizer** ([None](#) or [Normalizer](#), optional) – Normalizer to be applied to the input data to gain normality. The default is [None](#).
- **trend** ([None](#) or [float](#) or [callable](#), optional) – A callable trend function. Should have the signature: f(x, [y, z, ...]) This is used for detrended kriging, where the trended is subtracted from the conditions before kriging is applied. This can be used for regression kriging, where the trend function is determined by an external regression algorithm. If no normalizer is applied, this behaves equal to ‘mean’. The default is [None](#).
- **exact** ([bool](#), optional) – Whether the interpolator should reproduce the exact input values. If [False](#), *cond_err* is interpreted as measurement error at the conditioning points and the result will be more smooth. Default: [False](#)
- **cond_err** ([str](#), :class [float](#) or [list](#), optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be <= nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **pseudo_inv** ([bool](#), optional) – Whether the kriging system is solved with the pseudo inverted kriging matrix. If [True](#), this leads to more numerical stability and redundant points are averaged. But it can take more time. Default: [True](#)
- **pseudo_inv_type** ([str](#) or [callable](#), optional) – Here you can select the algorithm to compute the pseudo-inverse matrix:
 - “pinv”: use *pinv* from *scipy* which uses *lstsq*
 - “pinv2”: use *pinv2* from *scipy* which uses *SVD*
 - “pinvh”: use *pinvh* from *scipy* which uses eigen-values

If you want to use another routine to invert the kriging matrix, you can pass a callable which takes a matrix and returns the inverse. Default: “pinv”

- **fit_normalizer** ([bool](#), optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: [False](#)
- **fit_variogram** ([bool](#), optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the [standard_bins](#) routine. Default: [False](#)

Attributes

cond_err list: The measurement errors at the condition points.

cond_ext_drift `numpy.ndarray`: The ext. drift at the conditions.

cond_mean `numpy.ndarray`: Trend at the conditions.

cond_no int: The number of the conditions.

cond_pos list: The position tuple of the conditions.

cond_trend `numpy.ndarray`: Trend at the conditions.

cond_val list: The values of the conditions.

dim int: Dimension of the field.

drift_functions list of callable: The drift functions.

drift_no int: Number of drift values per point.

exact bool: Whether the interpolator is exact.

ext_drift_no int: Number of external drift values per point.

has_const_mean bool: Whether the field has a constant mean or not.

int_drift_no int: Number of internal drift values per point.

krige_size int: Size of the kriging system.

mean float or callable: The mean of the field.

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

name str: The name of the kriging class.

normalizer `Normalizer`: Normalizer of the field.

pseudo_inv bool: Whether pseudo inverse matrix is used.

pseudo_inv_type str: Method selector for pseudo inverse calculation.

trend float or callable: The trend of the field.

unbiased bool: Whether the kriging is unbiased or not.

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

Methods

<code>__call__(pos[, mesh_type, ext_drift, ...])</code>	Generate the kriging field.
<code>get_mean([post_process])</code>	Calculate the estimated mean of the detrended field.
<code>mesh(mesh[, points, direction, name])</code>	Generate a field on a given meshio or ogs5py mesh.
<code>plot([field, fig, ax])</code>	Plot the spatial random field.
<code>post_field(field[, name, process, save])</code>	Postprocessing field values.
<code>pre_pos(pos[, mesh_type])</code>	Preprocessing positions and mesh_type.
<code>set_condition([cond_pos, cond_val, ...])</code>	Set the conditions for kriging.
<code>set_drift_functions([drift_functions])</code>	Set the drift functions for universal kriging.
<code>structured(*args, **kwargs)</code>	Generate a field on a structured mesh.
<code>to_pyvista([field_select, fieldname])</code>	Create a VTK/PyVista grid of the stored field.
<code>unstructured(*args, **kwargs)</code>	Generate a field on an unstructured mesh.
<code>vtk_export(filename[, field_select, fieldname])</code>	Export the stored field to vtk.

```
__call__(pos, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False,
          return_var=True, post_process=True)
```

Generate the kriging field.

The field is saved as *self.field* and is also returned. The error variance is saved as *self.krige_var* and is also returned.

Parameters

- **pos** (**list**) – the position tuple, containing main direction and transversal directions (x, [y, z])
- **mesh_type** (**str**, optional) – ‘structured’ / ‘unstructured’
- **ext_drift** (**numpy.ndarray** or **None**, optional) – the external drift values at the given positions (only for EDK)
- **chunk_size** (**int**, optional) – Chunk size to cut down the size of the kriging system to prevent memory errors. Default: *None*
- **only_mean** (**bool**, optional) – Whether to only calculate the mean of the kriging field. Default: *False*
- **return_var** (**bool**, optional) – Whether to return the variance along with the field. Default: *True*
- **post_process** (**bool**, optional) – Whether to apply mean, normalizer and trend to the field. Default: *True*

Returns

- **field** (**numpy.ndarray**) – the kriged field or mean_field
- **krige_var** (**numpy.ndarray**, optional) – the kriging error variance (if return_var is *True* and only_mean is *False*)

```
get_mean(post_process=True)
```

Calculate the estimated mean of the detrended field.

Parameters **post_process** (**bool**, optional) – Whether to apply field-mean and normalizer. Default: *True*

Returns **mean** – Mean of the Kriging System.

Return type **float** or **None**

Notes

Only not *None* if the Kriging System has a constant mean. This means, no drift is given and the given field-mean is constant. The result is neglecting a potential given trend.

```
mesh(mesh, points='centroids', direction='all', name='field', **kwargs)
```

Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (*meshio.Mesh* or *ogs5py.MSH* or *PyVista mesh*) – The given meshio, ogs5py, or PyVista 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** or **list**, 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”. By default, all directions are used. One can also pass a list of indices. Default: “all”

- **name** (`str` or `list` of `str`, optional) – Name(s) to store the field(s) in the given mesh as `point_data` or `cell_data`. If to few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to `Field.__call__`.

Notes

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

See: <https://github.com/nschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: `Field.__call__`

plot(`field='field'`, `fig=None`, `ax=None`, ****kwargs**)

Plot the spatial random field.

Parameters

- **field** (`str`, optional) – Field that should be plotted. 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`
- ****kwargs** – Forwarded to the plotting routine.

post_field(`field`, `name='field'`, `process=True`, `save=True`)

Postprocessing field values.

Parameters

- **field** (`numpy.ndarray`) – Field values.
- **name** (`str`, optional) – Name. to store the field. The default is “field”.
- **process** (`bool`, optional) – Whether to process field to apply mean, normalizer and trend. The default is `True`.
- **save** (`bool`, optional) – Whether to store the field under the given name. The default is `True`.

Returns `field` – Processed field values.

Return type `numpy.ndarray`

pre_pos(`pos`, `mesh_type='unstructured'`)

Preprocessing positions and `mesh_type`.

Parameters

- **pos** (`iterable`) – the position tuple, containing main direction and transversal directions
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((`d`, `n`), `numpy.ndarray`) – the isometrized position tuple
- **shape** (`tuple`) – Shape of the resulting field.

set_condition(`cond_pos=None`, `cond_val=None`, `ext_drift=None`, `cond_err=None`,
`fit_normalizer=False`, `fit_variogram=False`)

Set the conditions for kriging.

This method could also be used to update the kriging setup, when properties were changed. Then you can call it without arguments.

Parameters

- **cond_pos** (*list*, optional) – the position tuple of the conditions (x, [y, z]). Default: current.
- **cond_val** (*numpy.ndarray*, optional) – the values of the conditions. Default: current.
- **ext_drift** (*numpy.ndarray* or *None*, optional) – the external drift values at the given conditions (only for EDK) For multiple external drifts, the first dimension should be the index of the drift term. When passing *None*, the existing external drift will be used.
- **cond_err** (*str*, :class *float*, *list*, optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **fit_normalizer** (*bool*, optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: False
- **fit_variogram** (*bool*, optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the *standard_bins* routine. Default: False

set_drift_functions(*drift_functions=None*)

Set the drift functions for universal kriging.

Parameters **drift_functions** (*list* of *callable*, *str* or *int*) – Either a list of callable functions, an integer representing the polynomial order of the drift or one of the following strings:

- “linear” : regional linear drift (equals order=1)
- “quadratic” : regional quadratic drift (equals order=2)

Raises **ValueError** – If the given drift functions are not callable.

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”

property cond_err

The measurement errors at the condition points.

Type `list`

property cond_ext_drift

The ext. drift at the conditions.

Type `numpy.ndarray`

property cond_mean

Trend at the conditions.

Type `numpy.ndarray`

property cond_no

The number of the conditions.

Type `int`

property cond_pos

The position tuple of the conditions.

Type `list`

property cond_trend

Trend at the conditions.

Type `numpy.ndarray`

property cond_val

The values of the conditions.

Type `list`

property dim

Dimension of the field.

Type `int`

property drift_functions

The drift functions.

Type `list` of `callable`

property drift_no

Number of drift values per point.

Type `int`

property exact

Whether the interpolator is exact.

Type `bool`

property ext_drift_no

Number of external drift values per point.

Type `int`

property has_const_mean

Whether the field has a constant mean or not.

Type `bool`

property int_drift_no

Number of internal drift values per point.

Type `int`

property krige_size

Size of the kriging system.

Type `int`

property mean

The mean of the field.

Type `float` or `callable`

property model

The covariance model of the field.

Type `CovModel`

property name

The name of the kriging class.

Type `str`

property normalizer

Normalizer of the field.

Type `Normalizer`

property pseudo_inv

Whether pseudo inverse matrix is used.

Type `bool`

property pseudo_inv_type

Method selector for pseudo inverse calculation.

Type `str`

property trend

The trend of the field.

Type `float` or `callable`

property unbiased

Whether the kriging is unbiased or not.

Type `bool`

property value_type

Type of the field values (scalar, vector).

Type `str`

gstools.krige.Detrended

```
class gstools.krige.Detrended(model, cond_pos, cond_val, trend, exact=False, cond_err='nugget',
                             pseudo_inv=True, pseudo_inv_type='pinv', fit_variogram=False)
```

Bases: [gstools.krige.base.Krige](#)

Detrended simple kriging.

In detrended kriging, the data is detrended before interpolation by simple kriging with zero mean.

The trend needs to be a callable function the user has to provide. This can be used for regression kriging, where the trend function is determined by an external regression algorithm.

This is just a shortcut for simple kriging with a given trend function, zero mean and no normalizer.

A trend can be given with EVERY provided kriging routine.

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
- **trend_function** ([callable](#)) – The callable trend function. Should have the signature: `f(x, [y, z])`
- **exact** ([bool](#), optional) – Whether the interpolator should reproduce the exact input values. If *False*, *cond_err* is interpreted as measurement error at the conditioning points and the result will be more smooth. Default: *False*
- **cond_err** ([str](#), :class: [float](#) or [list](#), optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **pseudo_inv** ([bool](#), optional) – Whether the kriging system is solved with the pseudo inverted kriging matrix. If *True*, this leads to more numerical stability and redundant points are averaged. But it can take more time. Default: *True*
- **pseudo_inv_type** ([str](#) or [callable](#), optional) – Here you can select the algorithm to compute the pseudo-inverse matrix:
 - “pinv”: use *pinv* from *scipy* which uses *lstsq*
 - “pinv2”: use *pinv2* from *scipy* which uses *SVD*
 - “pinvh”: use *pinvh* from *scipy* which uses eigen-values

If you want to use another routine to invert the kriging matrix, you can pass a callable which takes a matrix and returns the inverse. Default: “pinv”

- **fit_variogram** ([bool](#), optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the [standard_bins](#) routine. Default: *False*

Attributes

cond_err [list](#): The measurement errors at the condition points.

cond_ext_drift [numpy.ndarray](#): The ext. drift at the conditions.

cond_mean [numpy.ndarray](#): Trend at the conditions.

cond_no [int](#): The number of the conditions.

cond_pos [list](#): The position tuple of the conditions.

cond_trend `numpy.ndarray`: Trend at the conditions.

cond_val `list`: The values of the conditions.

dim `int`: Dimension of the field.

drift_functions `list` of `callable`: The drift functions.

drift_no `int`: Number of drift values per point.

exact `bool`: Whether the interpolator is exact.

ext_drift_no `int`: Number of external drift values per point.

has_const_mean `bool`: Whether the field has a constant mean or not.

int_drift_no `int`: Number of internal drift values per point.

krige_size `int`: Size of the kriging system.

mean `float` or `callable`: The mean of the field.

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

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

normalizer `Normalizer`: Normalizer of the field.

pseudo_inv `bool`: Whether pseudo inverse matrix is used.

pseudo_inv_type `str`: Method selector for pseudo inverse calculation.

trend `float` or `callable`: The trend of the field.

unbiased `bool`: Whether the kriging is unbiased or not.

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

Methods

<code>__call__(pos[, mesh_type, ext_drift, ...])</code>	Generate the kriging field.
<code>get_mean([post_process])</code>	Calculate the estimated mean of the detrended field.
<code>mesh(mesh[, points, direction, name])</code>	Generate a field on a given meshio or ogs5py mesh.
<code>plot([field, fig, ax])</code>	Plot the spatial random field.
<code>post_field(field[, name, process, save])</code>	Postprocessing field values.
<code>pre_pos(pos[, mesh_type])</code>	Preprocessing positions and mesh_type.
<code>set_condition([cond_pos, cond_val, ...])</code>	Set the conditions for kriging.
<code>set_drift_functions([drift_functions])</code>	Set the drift functions for universal kriging.
<code>structured(*args, **kwargs)</code>	Generate a field on a structured mesh.
<code>to_pyvista([field_select, fieldname])</code>	Create a VTK/PyVista grid of the stored field.
<code>unstructured(*args, **kwargs)</code>	Generate a field on an unstructured mesh.
<code>vtk_export(filename[, field_select, fieldname])</code>	Export the stored field to vtk.

`__call__(pos, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True)`

Generate the kriging field.

The field is saved as `self.field` and is also returned. The error variance is saved as `self.krige_var` and is also returned.

Parameters

- **pos** (`list`) – the position tuple, containing main direction and transversal directions (x, [y, z])
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’

- **ext_drift** (`numpy.ndarray` or `None`, optional) – the external drift values at the given positions (only for EDK)
- **chunk_size** (`int`, optional) – Chunk size to cut down the size of the kriging system to prevent memory errors. Default: `None`
- **only_mean** (`bool`, optional) – Whether to only calculate the mean of the kriging field. Default: `False`
- **return_var** (`bool`, optional) – Whether to return the variance along with the field. Default: `True`
- **post_process** (`bool`, optional) – Whether to apply mean, normalizer and trend to the field. Default: `True`

Returns

- **field** (`numpy.ndarray`) – the kriged field or mean_field
- **krige_var** (`numpy.ndarray`, optional) – the kriging error variance (if return_var is `True` and only_mean is `False`)

get_mean(*post_process=True*)

Calculate the estimated mean of the detrended field.

Parameters **post_process** (`bool`, optional) – Whether to apply field-mean and normalizer. Default: `True`

Returns **mean** – Mean of the Kriging System.

Return type `float` or `None`

Notes

Only not `None` if the Kriging System has a constant mean. This means, no drift is given and the given field-mean is constant. The result is neglecting a potential given trend.

mesh(*mesh, points='centroids', direction='all', name='field', **kwargs*)

Generate a field on a given meshio or ogs5py mesh.

Parameters

- **mesh** (`meshio.Mesh` or `ogs5py.MSH` or `PyVista mesh`) – The given meshio, ogs5py, or PyVista 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` or `list`, optional) – Here you can state which direction should be chosen for lower dimension. For example, if you got a 2D mesh in xz direction, you have to pass “xz”. By default, all directions are used. One can also pass a list of indices. Default: “all”
- **name** (`str` or `list` of `str`, optional) – Name(s) to store the field(s) in the given mesh as point_data or cell_data. If too few names are given, digits will be appended. Default: “field”
- ****kwargs** – Keyword arguments forwarded to `Field.__call__`.

Notes

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

See: <https://github.com/nschloe/meshio> See: <https://github.com/pyvista/pyvista>

See: `Field.__call__`

plot(*field*='field', *fig*=None, *ax*=None, ***kwargs*)

Plot the spatial random field.

Parameters

- **field** (`str`, optional) – Field that should be plotted. 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`
- ****kwargs** – Forwarded to the plotting routine.

post_field(*field*, *name*='field', *process*=True, *save*=True)

Postprocessing field values.

Parameters

- **field** (`numpy.ndarray`) – Field values.
- **name** (`str`, optional) – Name. to store the field. The default is “field”.
- **process** (`bool`, optional) – Whether to process field to apply mean, normalizer and trend. The default is True.
- **save** (`bool`, optional) – Whether to store the field under the given name. The default is True.

Returns **field** – Processed field values.

Return type `numpy.ndarray`

pre_pos(*pos*, *mesh_type*='unstructured')

Preprocessing positions and mesh_type.

Parameters

- **pos** (`iterable`) – the position tuple, containing main direction and transversal directions
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns

- **iso_pos** ((`d`, `n`), `numpy.ndarray`) – the isometrized position tuple
- **shape** (`tuple`) – Shape of the resulting field.

set_condition(*cond_pos*=None, *cond_val*=None, *ext_drift*=None, *cond_err*=None, *fit_normalizer*=False, *fit_variogram*=False)

Set the conditions for kriging.

This method could also be used to update the kriging setup, when properties were changed. Then you can call it without arguments.

Parameters

- **cond_pos** (`list`, optional) – the position tuple of the conditions (x, [y, z]). Default: current.
- **cond_val** (`numpy.ndarray`, optional) – the values of the conditions. Default: current.
- **ext_drift** (`numpy.ndarray` or `None`, optional) – the external drift values at the given conditions (only for EDK) For multiple external drifts, the first dimension should be the index of the drift term. When passing `None`, the existing external drift will be used.

- **cond_err** (*str*, :class `float`, *list*, optional) – The measurement error at the conditioning points. Either “nugget” to apply the model-nugget, a single value applied to all points or an array with individual values for each point. The measurement error has to be \leq nugget. The “exact=True” variant only works with “cond_err=’nugget’”. Default: “nugget”
- **fit_normalizer** (*bool*, optional) – Wheater to fit the data-normalizer to the given conditioning data. Default: False
- **fit_variogram** (*bool*, optional) – Wheater to fit the given variogram model to the data. This is done by using isotropy settings of the given model, assuming the sill to be the data variance and with the standard bins provided by the [standard_bins](#) routine. Default: False

set_drift_functions(*drift_functions=None*)

Set the drift functions for universal kriging.

Parameters **drift_functions** (*list* of *callable*, *str* or *int*) – Either a list of callable functions, an integer representing the polynomial order of the drift or one of the following strings:

- “linear” : regional linear drift (equals order=1)
- “quadratic” : regional quadratic drift (equals order=2)

Raises **ValueError** – If the given drift functions are not callable.

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”

property cond_err

The measurement errors at the condition points.

Type `list`

property cond_ext_drift

The ext. drift at the conditions.

Type `numpy.ndarray`

property cond_mean

Trend at the conditions.

Type `numpy.ndarray`

property cond_no

The number of the conditions.

Type `int`

property cond_pos

The position tuple of the conditions.

Type `list`

property cond_trend

Trend at the conditions.

Type `numpy.ndarray`

property cond_val

The values of the conditions.

Type `list`

property dim

Dimension of the field.

Type `int`

property drift_functions

The drift functions.

Type `list` of `callable`

property drift_no

Number of drift values per point.

Type `int`

property exact

Whether the interpolator is exact.

Type `bool`

property ext_drift_no

Number of external drift values per point.

Type `int`

property has_const_mean

Whether the field has a constant mean or not.

Type `bool`

property int_drift_no

Number of internal drift values per point.

Type `int`

property krige_size

Size of the kriging system.

Type `int`

property mean

The mean of the field.

Type `float` or `callable`

property model

The covariance model of the field.

Type *CovModel*

property name

The name of the kriging class.

Type *str*

property normalizer

Normalizer of the field.

Type *Normalizer*

property pseudo_inv

Whether pseudo inverse matrix is used.

Type *bool*

property pseudo_inv_type

Method selector for pseudo inverse calculation.

Type *str*

property trend

The trend of the field.

Type *float* or *callable*

property unbiased

Whether the kriging is unbiased or not.

Type *bool*

property value_type

Type of the field values (scalar, vector).

Type *str*

3.10 gstools.random

GStools subpackage for random number generation.

Random Number Generator

<code>RNG([seed])</code>	A random number generator for different distributions and multiple streams.
--------------------------	---

Seed Generator

<code>MasterRNG(seed)</code>	Master random number generator for generating seeds.
------------------------------	--

Distribution factory

<code>dist_gen([pdf_in, cdf_in, ppf_in])</code>	Distribution Factory.
---	-----------------------

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

<code>__call__()</code>	Return a random seed.
-------------------------	-----------------------

`__call__()`

Return a random seed.

property `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.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.RandomState`: Randomstate.

`seed` `int`: Seed of the master RNG.

Methods

<code>sample_dist(pdf, cdf, ppf, size)</code>	Sample from a distribution given by pdf, cdf and/or ppf.
<code>sample_ln_pdf(ln_pdf[, size, sample_around, ...])</code>	Sample from a distribution given by ln(pdf).
<code>sample_sphere(dim[, size])</code>	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: [50](#)
- **burn_in** ([int](#), optional) – Number of burn-in runs in the mcmc algorithm. Default: [20](#)
- **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`

property random

Randomstate.

Get a stream to the numpy Random number generator. You can use this, to call any provided distribution from `numpy.random.RandomState`.

Type `numpy.random.RandomState`

property 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.

3.11 gstools.tools

GStools subpackage providing miscellaneous tools.

Export

<code>vtk_export(filename, pos, fields[, mesh_type])</code>	Export a field to vtk.
<code>vtk_export_structured(filename, pos, fields)</code>	Export a field to vtk structured rectilinear grid file.
<code>vtk_export_unstructured(filename, pos, fields)</code>	Export a field to vtk unstructured grid file.
<code>to_vtk(pos, fields[, mesh_type])</code>	Create a VTK/PyVista grid.
<code>to_vtk_structured(pos, fields)</code>	Create a vtk structured rectilinear grid from a field.
<code>to_vtk_unstructured(pos, fields)</code>	Export a field to vtk structured rectilinear grid file.

Special functions

<code>confidence_scaling([per])</code>	Scaling of standard deviation to get the desired confidence interval.
<code>inc_gamma(s, x)</code>	Calculate the (upper) incomplete gamma function.
<code>exp_int(s, x)</code>	Calculate the exponential integral $E_s(x)$.
<code>inc_beta(a, b, x)</code>	Calculate the incomplete Beta function.
<code>tplstable_cor(r, len_scale, hurst, alpha)</code>	Calculate the correlation function of the TPLStable model.
<code>tpl_exp_spec_dens(k, dim, len_scale, hurst)</code>	Spectral density of the TPLExponential covariance model.
<code>tpl_gau_spec_dens(k, dim, len_scale, hurst)</code>	Spectral density of the TPLGaussian covariance model.

Geometric

<code>rotated_main_axes(dim, angles)</code>	Create list of the main axis defined by the given system rotations.
<code>set_angles(dim, angles)</code>	Set the angles for the given dimension.
<code>set_anis(dim, anis)</code>	Set the anisotropy ratios for the given dimension.
<code>no_of_angles(dim)</code>	Calculate number of rotation angles depending on the dimension.
<code>rotation_planes(dim)</code>	Get all 2D sub-planes for rotation.
<code>givens_rotation(dim, plane, angle)</code>	Givens rotation matrix in arbitrary dimensions.
<code>matrix_rotate(dim, angles)</code>	Create a matrix to rotate points to the target coordinate-system.
<code>matrix_derotate(dim, angles)</code>	Create a matrix to derotate points to the initial coordinate-system.
<code>matrix_isotropify(dim, anis)</code>	Create a stretching matrix to make things isotrope.
<code>matrix_anisotropify(dim, anis)</code>	Create a stretching matrix to make things anisotrope.
<code>matrix_isometrize(dim, angles, anis)</code>	Create a matrix to derotate points and make them isotrope.
<code>matrix_anisometrize(dim, angles, anis)</code>	Create a matrix to rotate points and make them anisotrope.
<code>ang2dir(angles[, dtype, dim])</code>	Convert n-D spherical coordinates to Euclidean direction vectors.
<code>generate_grid(pos)</code>	Generate grid from a structured position tuple.

continues on next page

Table 59 – continued from previous page

<code>generate_st_grid(pos, time[, mesh_type])</code>	Generate spatio-temporal grid from a position tuple and time array.
---	---

Misc

<code>EARTH_RADIUS</code>	earth radius for WGS84 ellipsoid in km
---------------------------	--

`gstools.tools.ang2dir(angles, dtype=<class 'numpy.float64'>, dim=None)`

Convert n-D spherical coordinates to Euclidean direction vectors.

Parameters

- **angles** (`list` of `numpy.ndarray`) – spherical coordinates given as angles.
- **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
- **dim** (`int`, optional) – Cut of information above the given dimension. Otherwise, dimension is determined by number of angles Default: None

Returns the array of direction vectors

Return type `numpy.ndarray`

`gstools.tools.confidence_scaling(per=0.95)`

Scaling of standard deviation to get the desired confidence interval.

Parameters **per** (`float`, optional) – Confidence level. The default is 0.95.

Returns Scale to multiply the standard deviation with.

Return type `float`

`gstools.tools.exp_int(s, x)`

Calculate the exponential integral $E_s(x)$.

Given by: $E_s(x) = \int_1^\infty \frac{e^{-xt}}{t^s} dt$

Parameters

- **s** (`float`) – exponent in the integral (should be > -100)
- **x** (`numpy.ndarray`) – input values

`gstools.tools.generate_grid(pos)`

Generate grid from a structured position tuple.

Parameters **pos** (`tuple` of `numpy.ndarray`) – The structured position tuple.

Returns Unstructured position tuple.

Return type `numpy.ndarray`

`gstools.tools.generate_st_grid(pos, time, mesh_type='unstructured')`

Generate spatio-temporal grid from a position tuple and time array.

Parameters

- **pos** (`tuple` of `numpy.ndarray`) – The (un-)structured position tuple.
- **time** (`iterable`) – The time array.
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

Returns Unstructured spatio-temporal point tuple.

Return type `numpy.ndarray`

Notes

Time dimension will be the last one.

`gstools.tools.givens_rotation(dim, plane, angle)`

Givens rotation matrix in arbitrary dimensions.

Parameters

- **dim** (`int`) – spatial dimension
- **plane** (`list` of `int`) – the plane to rotate in, given by the indices of the two defining axes. For example the xy plane is defined by $(0,1)$
- **angle** (`float` or `list`) – the rotation angle in the given plane

Returns Rotation matrix.

Return type `numpy.ndarray`

`gstools.tools.inc_beta(a, b, x)`

Calculate 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.inc_gamma(s, x)`

Calculate the (upper) incomplete gamma function.

Given by: $\Gamma(s, x) = \int_x^\infty t^{s-1} e^{-t} dt$

Parameters

- **s** (`float`) – exponent in the integral
- **x** (`numpy.ndarray`) – input values

`gstools.tools.matrix_anisometrize(dim, angles, anis)`

Create a matrix to rotate points and make them anisotrope.

Parameters

- **dim** (`int`) – spatial dimension
- **angles** (`float` or `list`) – the rotation angles of the target coordinate-system
- **anis** (`list` of `float`) – the anisotropy of length scales along the transversal directions

Returns Transformation matrix.

Return type `numpy.ndarray`

`gstools.tools.matrix_anisotropify(dim, anis)`

Create a stretching matrix to make things anisotrope.

Parameters

- **dim** (`int`) – spatial dimension
- **anis** (`list` of `float`) – the anisotropy of length scales along the transversal directions

Returns Stretching matrix.

Return type `numpy.ndarray`

`gstools.tools.matrix_derotate(dim, angles)`

Create a matrix to derotate points to the initial coordinate-system.

Parameters

- **dim** (`int`) – spatial dimension
- **angles** (`float` or `list`) – the rotation angles of the target coordinate-system

Returns Rotation matrix.

Return type `numpy.ndarray`

`gstools.tools.matrix_isometrize(dim, angles, anis)`

Create a matrix to derotate points and make them isotrope.

Parameters

- **dim** (`int`) – spatial dimension
- **angles** (`float` or `list`) – the rotation angles of the target coordinate-system
- **anis** (`list` of `float`) – the anisotropy of length scales along the transversal directions

Returns Transformation matrix.

Return type `numpy.ndarray`

`gstools.tools.matrix_isotropify(dim, anis)`

Create a stretching matrix to make things isotrope.

Parameters

- **dim** (`int`) – spatial dimension
- **anis** (`list` of `float`) – the anisotropy of length scales along the transversal directions

Returns Stretching matrix.

Return type `numpy.ndarray`

`gstools.tools.matrix_rotate(dim, angles)`

Create a matrix to rotate points to the target coordinate-system.

Parameters

- **dim** (`int`) – spatial dimension
- **angles** (`float` or `list`) – the rotation angles of the target coordinate-system

Returns Rotation matrix.

Return type `numpy.ndarray`

`gstools.tools.no_of_angles(dim)`

Calculate number of rotation angles depending on the dimension.

Parameters **dim** (`int`) – spatial dimension

Returns Number of angles.

Return type `int`

`gstools.tools.rotated_main_axes(dim, angles)`

Create list of the main axis defined by the given system rotations.

Parameters

- **dim** (`int`) – spatial dimension
- **angles** (`float` or `list`) – the rotation angles of the target coordinate-system

Returns Main axes of the target coordinate-system.

Return type `numpy.ndarray`

`gstools.tools.rotation_planes(dim)`

Get all 2D sub-planes for rotation.

Parameters `dim (int)` – spatial dimension

Returns All 2D sub-planes for rotation.

Return type `list of tuple of int`

`gstools.tools.set_angles(dim, angles)`

Set the angles for the given dimension.

Parameters

- **dim (int)** – spatial dimension
- **angles (float or list)** – the angles of the SRF

Returns `angles` – the angles fitting to the dimension

Return type `float`

Notes

If too few angles are given, they are filled up with 0.

`gstools.tools.set_anis(dim, anis)`

Set the anisotropy ratios for the given dimension.

Parameters

- **dim (int)** – spatial dimension
- **anis (list of float)** – the anisotropy of length scales along the transversal directions

Returns `anis` – the anisotropy of length scales fitting the dimensions

Return type `list of float`

Notes

If too few anisotropy ratios are given, they are filled up with 1.

`gstools.tools.to_vtk(pos, fields, mesh_type='unstructured')`

Create a VTK/PyVista grid.

Parameters

- **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

Returns This will return a PyVista object for the given field data in its appropriate type. Structured meshes will return a `pyvista.RectilinearGrid` and unstructured meshes will return an `pyvista.UnstructuredGrid` object.

Return type `pyvista.RectilinearGrid` or `pyvista.UnstructuredGrid`

`gstools.tools.to_vtk_structured(pos, fields)`

Create a vtk structured rectilinear grid from a field.

Parameters

- **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 their names as keys.

Returns A PyVista rectilinear grid of the structured field data. Data arrays live on the point data of this PyVista dataset.

Return type `pyvista.RectilinearGrid`

`gstools.tools.to_vtk_unstructured(pos, fields)`

Export a field to vtk structured rectilinear grid file.

Parameters

- **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 their names as keys.

Returns A PyVista unstructured grid of the unstructured field data. Data arrays live on the point data of this PyVista dataset. This is essentially a point cloud with no topology.

Return type `pyvista.UnstructuredGrid`

`gstools.tools.tpl_exp_spec_dens(k, dim, len_scale, hurst, len_low=0.0)`

Spectral density of the TPLExponential covariance model.

Parameters

- **k** (`float`) – Radius of the phase: $k = \|\mathbf{k}\|$
- **dim** (`int`) – Dimension of the model.
- **len_scale** (`float`) – Length scale of the model.
- **hurst** (`float`) – Hurst coefficient of the power law.
- **len_low** (`float`, optional) – The lower length scale truncation of the model. Default: 0.0

Returns spectral density of the TPLExponential model

Return type `float`

`gstools.tools.tpl_gau_spec_dens(k, dim, len_scale, hurst, len_low=0.0)`

Spectral density of the TPLGaussian covariance model.

Parameters

- **k** (`float`) – Radius of the phase: $k = \|\mathbf{k}\|$
- **dim** (`int`) – Dimension of the model.
- **len_scale** (`float`) – Length scale of the model.
- **hurst** (`float`) – Hurst coefficient of the power law.
- **len_low** (`float`, optional) – The lower length scale truncation of the model. Default: 0.0

Returns spectral density of the TPLExponential model

Return type `float`

`gstools.tools.tplstable_cor(r, len_scale, hurst, alpha)`

Calculate the correlation function of the TPLStable model.

Given by the following correlation function:

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

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.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.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.EARTH_RADIUS = 6371.0`

earth radius for WGS84 ellipsoid in km

Type `float`

3.12 gstools.transform

GStools subpackage providing transformations to post-process normal fields.

Field-Transformations

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

(Inverse) Box-Cox transformation to denormalize data.

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.discrete(fld, values, thresholds='arithmetic')`

Discrete transformation.

After this transformation, the field has only $\text{len}(\text{values})$ discrete values.

Parameters

- **fld** (*Field*) – Spatial Random Field class containing a generated field. Field will be transformed inplace.
- **values** (`numpy.ndarray`) – The discrete values the field will take
- **thresholds** (`str` or `numpy.ndarray`, optional) – the thresholds, where the value classes are separated possible values are: * “arithmetic”: the mean of the 2 neighbouring values * “equal”: divide the field into equal parts * an array of explicitly given thresholds
Default: “arithmetic”

`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_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_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_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

`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”

3.13 gstools.normalizer

GStools subpackage providing normalization routines.

Base-Normalizer

<code>Normalizer([data])</code>	Normalizer class.
---------------------------------	-------------------

gstools.normalizer.Normalizer

class `gstools.normalizer.Normalizer`(*data=None*, ***parameter*)

Bases: `object`

Normalizer class.

Parameters

- **data** (*array_like*, *optional*) – Input data to fit the transformation to in order to gain normality. The default is None.
- ****parameter** – Specified parameters given by name. If not given, default parameters will be used.

Attributes

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

Methods

<code>denormalize(data)</code>	Transform to input distribution.
<code>derivative(data)</code>	Factor for normal PDF to gain target PDF.
<code>fit(data[, skip])</code>	Fitting the transformation to data by maximizing Log-Likelihood.
<code>kernel_loglikelihood(data)</code>	Kernel Log-Likelihood for given data with current parameters.
<code>likelihood(data)</code>	Likelihood for given data with current parameters.
<code>loglikelihood(data)</code>	Log-Likelihood for given data with current parameters.
<code>normalize(data)</code>	Transform to normal distribution.

denormalize(*data*)

Transform to input distribution.

Parameters **data** (*array_like*) – Input data (normal distributed).

Returns Denormalized data.

Return type `numpy.ndarray`

derivative(*data*)

Factor for normal PDF to gain target PDF.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Derivative of the normalization transformation function.

Return type `numpy.ndarray`

fit(*data*, *skip=None*, ***kwargs*)

Fitting the transformation to data by maximizing Log-Likelihood.

Parameters

- **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.
- **skip** (*list* of *str* or *None*, optional) – Names of parameters to be skipped in fitting. The default is *None*.
- ****kwargs** – Keyword arguments passed to `scipy.optimize.minimize_scalar` when only one parameter present or `scipy.optimize.minimize`.

Returns Optimal parameters given by names.

Return type *dict*

kernel_loglikelihood(data)

Kernel Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Kernel Log-Likelihood of the given data.

Return type *float*

Notes

This loglikelihood function is neglecting additive constants, that are not needed for optimization.

likelihood(data)

Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Likelihood of the given data.

Return type *float*

loglikelihood(data)

Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Log-Likelihood of the given data.

Return type *float*

normalize(data)

Transform to normal distribution.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Normalized data.

Return type *numpy.ndarray*

default_parameter = {}

Default parameters of the Normalizer.

Type *dict*

denormalize_range = (-inf, inf)

Valid range for output/normal data.

Type *tuple*

property name

The name of the normalizer class.

Type *str*

normalize_range = (-inf, inf)

Valid range for input data.

Type `tuple`

Field-Normalizer

<code>LogNormal([data])</code>	Log-normal fields.
<code>BoxCox([data])</code>	Box-Cox (1964) transformed fields.
<code>BoxCoxShift([data])</code>	Box-Cox (1964) transformed fields including shifting.
<code>YeoJohnson([data])</code>	Yeo-Johnson (2000) transformed fields.
<code>Modulus([data])</code>	Modulus or John-Draper (1980) transformed fields.
<code>Manly([data])</code>	Manly (1971) transformed fields.

gstools.normalizer.LogNormal

class `gstools.normalizer.LogNormal`(*data=None*, ***parameter*)

Bases: `gstools.normalizer.base.Normalizer`

Log-normal fields.

Notes

This parameter-free transformation is given by:

$$y = \log(x)$$

Attributes

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

Methods

<code>denormalize(data)</code>	Transform to input distribution.
<code>derivative(data)</code>	Factor for normal PDF to gain target PDF.
<code>fit(data[, skip])</code>	Fitting the transformation to data by maximizing Log-Likelihood.
<code>kernel_loglikelihood(data)</code>	Kernel Log-Likelihood for given data with current parameters.
<code>likelihood(data)</code>	Likelihood for given data with current parameters.
<code>loglikelihood(data)</code>	Log-Likelihood for given data with current parameters.
<code>normalize(data)</code>	Transform to normal distribution.

denormalize(*data*)

Transform to input distribution.

Parameters *data* (*array_like*) – Input data (normal distributed).

Returns Denormalized data.

Return type `numpy.ndarray`

derivative(*data*)

Factor for normal PDF to gain target PDF.

Parameters *data* (*array_like*) – Input data (not normal distributed).

Returns Derivative of the normalization transformation function.

Return type `numpy.ndarray`

fit(*data*, *skip=None*, ***kwargs*)

Fitting the transformation to data by maximizing Log-Likelihood.

Parameters

- **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.
- **skip** (*list* of *str* or *None*, optional) – Names of parameters to be skipped in fitting. The default is *None*.
- ****kwargs** – Keyword arguments passed to `scipy.optimize.minimize_scalar` when only one parameter present or `scipy.optimize.minimize`.

Returns Optimal paramters given by names.

Return type *dict*

kernel_loglikelihood(*data*)

Kernel Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Kernel Log-Likelihood of the given data.

Return type *float*

Notes

This loglikelihood function is neglecting additive constants, that are not needed for optimization.

likelihood(*data*)

Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Likelihood of the given data.

Return type *float*

loglikelihood(*data*)

Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Log-Likelihood of the given data.

Return type *float*

normalize(*data*)

Transform to normal distribution.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Normalized data.

Return type *numpy.ndarray*

default_parameter = {}

Default parameters of the Normalizer.

Type *dict*

denormalize_range = (-inf, inf)

Valid range for output/normal data.

Type *tuple*

property name

The name of the normalizer class.

Type `str`

normalize_range = (0.0, inf)

Valid range for input data.

gstools.normalizer.BoxCox

class `gstools.normalizer.BoxCox`(*data=None*, ***parameter*)

Bases: `gstools.normalizer.base.Normalizer`

Box-Cox (1964) transformed fields.

Parameters

- **data** (*array_like*, *optional*) – Input data to fit the transformation in order to gain normality. The default is None.
- **lambda** (*float*, *optional*) – Shape parameter. Default: 1

Notes

This transformation is given by [Box1964]:

$$y = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \log(x) & \lambda = 0 \end{cases}$$

References

Attributes

denormalize_range *tuple*: Valid range for output data depending on lambda.

name *str*: The name of the normalizer class.

Methods

<code>denormalize</code> (data)	Transform to input distribution.
<code>derivative</code> (data)	Factor for normal PDF to gain target PDF.
<code>fit</code> (data[, skip])	Fitting the transformation to data by maximizing Log-Likelihood.
<code>kernel_loglikelihood</code> (data)	Kernel Log-Likelihood for given data with current parameters.
<code>likelihood</code> (data)	Likelihood for given data with current parameters.
<code>loglikelihood</code> (data)	Log-Likelihood for given data with current parameters.
<code>normalize</code> (data)	Transform to normal distribution.

`denormalize`(data)

Transform to input distribution.

Parameters **data** (*array_like*) – Input data (normal distributed).

Returns Denormalized data.

Return type `numpy.ndarray`

`derivative`(data)

Factor for normal PDF to gain target PDF.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Derivative of the normalization transformation function.

Return type `numpy.ndarray`

fit(*data*, *skip*=None, ***kwargs*)

Fitting the transformation to data by maximizing Log-Likelihood.

Parameters

- **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.
- **skip** (*list* of *str* or *None*, optional) – Names of parameters to be skipped in fitting. The default is None.
- ****kwargs** – Keyword arguments passed to `scipy.optimize.minimize_scalar` when only one parameter present or `scipy.optimize.minimize`.

Returns Optimal paramters given by names.

Return type *dict*

kernel_loglikelihood(*data*)

Kernel Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Kernel Log-Likelihood of the given data.

Return type *float*

Notes

This loglikelihood function is neglecting additive constants, that are not needed for optimization.

likelihood(*data*)

Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Likelihood of the given data.

Return type *float*

loglikelihood(*data*)

Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Log-Likelihood of the given data.

Return type *float*

normalize(*data*)

Transform to normal distribution.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Normalized data.

Return type *numpy.ndarray*

default_parameter = {'lmbda': 1}

Default parameter of the BoxCox-Normalizer.

Type *dict*

property denormalize_range

Valid range for output data depending on lmbda.

(-1/lmbda, inf) or (-inf, -1/lmbda)

Type *tuple*

property name

The name of the normalizer class.

Type `str`

normalize_range = (0.0, inf)

Valid range for input data.

Type `tuple`

gstools.normalizer.BoxCoxShift

class `gstools.normalizer.BoxCoxShift`(*data=None, **parameter*)

Bases: `gstools.normalizer.base.Normalizer`

Box-Cox (1964) transformed fields including shifting.

Parameters

- **data** (*array_like, optional*) – Input data to fit the transformation in order to gain normality. The default is None.
- **lmbda** (*float, optional*) – Shape parameter. Default: 1
- **shift** (*float, optional*) – Shift parameter. Default: 0

Notes

This transformation is given by [Box1964]:

$$y = \begin{cases} \frac{(x+s)^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \log(x+s) & \lambda = 0 \end{cases}$$

Fitting the shift parameter is rather hard. You should consider skipping “shift” during fitting:

```
>>> data = range(5)
>>> norm = BoxCoxShift(shift=0.5)
>>> norm.fit(data, skip=["shift"])
{'shift': 0.5, 'lmbda': 0.6747515267420799}
```

References**Attributes**

denormalize_range *tuple*: Valid range for output data depending on lmbda.

name *str*: The name of the normalizer class.

normalize_range *tuple*: Valid range for input data depending on shift.

Methods

<code>denormalize</code> (data)	Transform to input distribution.
<code>derivative</code> (data)	Factor for normal PDF to gain target PDF.
<code>fit</code> (data[, skip])	Fitting the transformation to data by maximizing Log-Likelihood.
<code>kernel_loglikelihood</code> (data)	Kernel Log-Likelihood for given data with current parameters.
<code>likelihood</code> (data)	Likelihood for given data with current parameters.
<code>loglikelihood</code> (data)	Log-Likelihood for given data with current parameters.
<code>normalize</code> (data)	Transform to normal distribution.

denormalize(data)

Transform to input distribution.

Parameters **data** (*array_like*) – Input data (normal distributed).

Returns Denormalized data.

Return type `numpy.ndarray`

derivative(*data*)

Factor for normal PDF to gain target PDF.

Parameters *data* (*array_like*) – Input data (not normal distributed).

Returns Derivative of the normalization transformation function.

Return type `numpy.ndarray`

fit(*data*, *skip=None*, ***kwargs*)

Fitting the transformation to data by maximizing Log-Likelihood.

Parameters

- **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.
- **skip** (*list* of *str* or *None*, optional) – Names of parameters to be skipped in fitting. The default is *None*.
- ****kwargs** – Keyword arguments passed to `scipy.optimize.minimize_scalar` when only one parameter present or `scipy.optimize.minimize`.

Returns Optimal parameters given by names.

Return type `dict`

kernel_loglikelihood(*data*)

Kernel Log-Likelihood for given data with current parameters.

Parameters *data* (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Kernel Log-Likelihood of the given data.

Return type `float`

Notes

This loglikelihood function is neglecting additive constants, that are not needed for optimization.

likelihood(*data*)

Likelihood for given data with current parameters.

Parameters *data* (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Likelihood of the given data.

Return type `float`

loglikelihood(*data*)

Log-Likelihood for given data with current parameters.

Parameters *data* (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Log-Likelihood of the given data.

Return type `float`

normalize(*data*)

Transform to normal distribution.

Parameters *data* (*array_like*) – Input data (not normal distributed).

Returns Normalized data.

Return type `numpy.ndarray`


```
default_parameter = {'lambda': 1, 'shift': 0}
```

Default parameters of the BoxCoxShift-Normalizer.

Type `dict`

property **denormalize_range**

Valid range for output data depending on `lambda`.

$(-1/\lambda, \infty)$ or $(-\infty, -1/\lambda)$

Type `tuple`

property **name**

The name of the normalizer class.

Type `str`

property **normalize_range**

Valid range for input data depending on `shift`.

$(-\text{shift}, \infty)$

Type `tuple`

gstools.normalizer.YeoJohnson

class `gstools.normalizer.YeoJohnson`(*data=None*, ***parameter*)

Bases: `gstools.normalizer.base.Normalizer`

Yeo-Johnson (2000) transformed fields.

Parameters

- **data** (*array_like*, *optional*) – Input data to fit the transformation in order to gain normality. The default is None.
- **lmbda** (*float*, *optional*) – Shape parameter. Default: 1

Notes

This transformation is given by [Yeo2000]:

$$y = \begin{cases} \frac{(x+1)^\lambda - 1}{\lambda} & x \geq 0, \lambda \neq 0 \\ \log(x + 1) & x \geq 0, \lambda = 0 \\ -\frac{(|x|+1)^{2-\lambda} - 1}{2-\lambda} & x < 0, \lambda \neq 2 \\ -\log(|x| + 1) & x < 0, \lambda = 2 \end{cases}$$

References

Attributes

name *str*: The name of the normalizer class.

Methods

<code>denormalize</code> (<i>data</i>)	Transform to input distribution.
<code>derivative</code> (<i>data</i>)	Factor for normal PDF to gain target PDF.
<code>fit</code> (<i>data</i> [, <i>skip</i>])	Fitting the transformation to data by maximizing Log-Likelihood.
<code>kernel_loglikelihood</code> (<i>data</i>)	Kernel Log-Likelihood for given data with current parameters.
<code>likelihood</code> (<i>data</i>)	Likelihood for given data with current parameters.
<code>loglikelihood</code> (<i>data</i>)	Log-Likelihood for given data with current parameters.
<code>normalize</code> (<i>data</i>)	Transform to normal distribution.

`denormalize`(*data*)

Transform to input distribution.

Parameters **data** (*array_like*) – Input data (normal distributed).

Returns Denormalized data.

Return type `numpy.ndarray`

`derivative`(*data*)

Factor for normal PDF to gain target PDF.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Derivative of the normalization transformation function.

Return type `numpy.ndarray`

fit(*data*, *skip*=None, ***kwargs*)

Fitting the transformation to data by maximizing Log-Likelihood.

Parameters

- **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.
- **skip** (*list* of *str* or *None*, optional) – Names of parameters to be skipped in fitting. The default is None.
- ****kwargs** – Keyword arguments passed to `scipy.optimize.minimize_scalar` when only one parameter present or `scipy.optimize.minimize`.

Returns Optimal paramters given by names.

Return type *dict*

kernel_loglikelihood(*data*)

Kernel Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Kernel Log-Likelihood of the given data.

Return type *float*

Notes

This loglikelihood function is neglecting additive constants, that are not needed for optimization.

likelihood(*data*)

Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Likelihood of the given data.

Return type *float*

loglikelihood(*data*)

Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Log-Likelihood of the given data.

Return type *float*

normalize(*data*)

Transform to normal distribution.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Normalized data.

Return type *numpy.ndarray*

default_parameter = {'lmbda': 1}

Default parameter of the YeoJohnson-Normalizer.

Type *dict*

denormalize_range = (-inf, inf)

Valid range for output/normal data.

Type *tuple*

property name

The name of the normalizer class.

Type `str`

normalize_range = (-inf, inf)

Valid range for input data.

Type `tuple`

gstools.normalizer.Modulus

class `gstools.normalizer.Modulus`(*data=None, **parameter*)

Bases: `gstools.normalizer.base.Normalizer`

Modulus or John-Draper (1980) transformed fields.

Parameters

- **data** (*array_like, optional*) – Input data to fit the transformation in order to gain normality. The default is None.
- **lambda** (*float, optional*) – Shape parameter. Default: 1

Notes

This transformation is given by [John1980]:

$$y = \begin{cases} \operatorname{sgn}(x) \frac{(|x|+1)^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \operatorname{sgn}(x) \log(|x| + 1) & \lambda = 0 \end{cases}$$

References

Attributes

name *str*: The name of the normalizer class.

Methods

<code>denormalize</code> (<i>data</i>)	Transform to input distribution.
<code>derivative</code> (<i>data</i>)	Factor for normal PDF to gain target PDF.
<code>fit</code> (<i>data</i> [, <i>skip</i>])	Fitting the transformation to data by maximizing Log-Likelihood.
<code>kernel_loglikelihood</code> (<i>data</i>)	Kernel Log-Likelihood for given data with current parameters.
<code>likelihood</code> (<i>data</i>)	Likelihood for given data with current parameters.
<code>loglikelihood</code> (<i>data</i>)	Log-Likelihood for given data with current parameters.
<code>normalize</code> (<i>data</i>)	Transform to normal distribution.

denormalize(*data*)

Transform to input distribution.

Parameters **data** (*array_like*) – Input data (normal distributed).

Returns Denormalized data.

Return type `numpy.ndarray`

derivative(*data*)

Factor for normal PDF to gain target PDF.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Derivative of the normalization transformation function.

Return type `numpy.ndarray`

fit(*data, skip=None, **kwargs*)

Fitting the transformation to data by maximizing Log-Likelihood.

Parameters

- **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.
- **skip** (*list* of *str* or *None*, optional) – Names of parameters to be skipped in fitting. The default is *None*.
- ****kwargs** – Keyword arguments passed to `scipy.optimize.minimize_scalar` when only one parameter present or `scipy.optimize.minimize`.

Returns Optimal paramters given by names.

Return type *dict*

kernel_loglikelihood(data)

Kernel Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Kernel Log-Likelihood of the given data.

Return type *float*

Notes

This loglikelihood function is neglecting additive constants, that are not needed for optimization.

likelihood(data)

Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Likelihood of the given data.

Return type *float*

loglikelihood(data)

Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Log-Likelihood of the given data.

Return type *float*

normalize(data)

Transform to normal distribution.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Normalized data.

Return type *numpy.ndarray*

default_parameter = {'lmbda': 1}

Default parameter of the Modulus-Normalizer.

Type *dict*

denormalize_range = (-inf, inf)

Valid range for output/normal data.

Type *tuple*

property name

The name of the normalizer class.

Type *str*

normalize_range = (-inf, inf)

Valid range for input data.

Type `tuple`

gstools.normalizer.Manly

class `gstools.normalizer.Manly`(*data=None*, ***parameter*)

Bases: `gstools.normalizer.base.Normalizer`

Manly (1971) transformed fields.

Parameters

- **data** (*array_like*, *optional*) – Input data to fit the transformation in order to gain normality. The default is None.
- **lmbda** (*float*, *optional*) – Shape parameter. Default: 1

Notes

This transformation is given by [Manly1976]:

$$y = \begin{cases} \frac{\exp(\lambda x) - 1}{\lambda} & \lambda \neq 0 \\ x & \lambda = 0 \end{cases}$$

References

Attributes

denormalize_range *tuple*: Valid range for output data depending on lmbda.

name *str*: The name of the normalizer class.

Methods

<code>denormalize</code> (data)	Transform to input distribution.
<code>derivative</code> (data)	Factor for normal PDF to gain target PDF.
<code>fit</code> (data[, skip])	Fitting the transformation to data by maximizing Log-Likelihood.
<code>kernel_loglikelihood</code> (data)	Kernel Log-Likelihood for given data with current parameters.
<code>likelihood</code> (data)	Likelihood for given data with current parameters.
<code>loglikelihood</code> (data)	Log-Likelihood for given data with current parameters.
<code>normalize</code> (data)	Transform to normal distribution.

`denormalize`(data)

Transform to input distribution.

Parameters **data** (*array_like*) – Input data (normal distributed).

Returns Denormalized data.

Return type `numpy.ndarray`

`derivative`(data)

Factor for normal PDF to gain target PDF.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Derivative of the normalization transformation function.

Return type `numpy.ndarray`

fit(*data*, *skip*=None, ***kwargs*)

Fitting the transformation to data by maximizing Log-Likelihood.

Parameters

- **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.
- **skip** (*list* of *str* or *None*, optional) – Names of parameters to be skipped in fitting. The default is None.
- ****kwargs** – Keyword arguments passed to `scipy.optimize.minimize_scalar` when only one parameter present or `scipy.optimize.minimize`.

Returns Optimal paramters given by names.

Return type *dict*

kernel_loglikelihood(*data*)

Kernel Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Kernel Log-Likelihood of the given data.

Return type *float*

Notes

This loglikelihood function is neglecting additive constants, that are not needed for optimization.

likelihood(*data*)

Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Likelihood of the given data.

Return type *float*

loglikelihood(*data*)

Log-Likelihood for given data with current parameters.

Parameters **data** (*array_like*) – Input data to fit the transformation to in order to gain normality.

Returns Log-Likelihood of the given data.

Return type *float*

normalize(*data*)

Transform to normal distribution.

Parameters **data** (*array_like*) – Input data (not normal distributed).

Returns Normalized data.

Return type *numpy.ndarray*

default_parameter = {'lmbda': 1}

Default parameter of the Manly-Normalizer.

Type *dict*

property denormalize_range

Valid range for output data depending on lmbda.

(-1/lmbda, inf) or (-inf, -1/lmbda)

Type *tuple*

property name

The name of the normalizer class.

Type `str`

normalize_range = (-inf, inf)

Valid range for input data.

Type `tuple`

Convenience Routines

<code>apply_mean_norm_trend(pos, field[, mean, ...])</code>	Apply mean, de-normalization and trend to given field.
<code>remove_trend_norm_mean(pos, field[, mean, ...])</code>	Remove trend, de-normalization and mean from given field.

gstools.normalizer.apply_mean_norm_trend

`gstools.normalizer.apply_mean_norm_trend(pos, field, mean=None, normalizer=None, trend=None, mesh_type='unstructured', value_type='scalar', check_shape=True, stacked=False)`

Apply mean, de-normalization and trend to given field.

Parameters

- **pos** (*iterable*) – Position tuple, containing main direction and transversal directions.
- **field** (*numpy.ndarray* or *list* of *numpy.ndarray*) – The spatially distributed data. You can pass a list of fields, that will be used simultaneously. Then you need to set `stacked=True`.
- **mean** (*None* or *float* or *callable*, optional) – Mean of the field if wanted. Could also be a callable. The default is *None*.
- **normalizer** (*None* or *Normalizer*, optional) – Normalizer to be applied to the field. The default is *None*.
- **trend** (*None* or *float* or *callable*, optional) – Trend of the denormalized fields. If no normalizer is applied, this behaves equal to ‘mean’. The default is *None*.
- **mesh_type** (*str*, optional) – ‘structured’ / ‘unstructured’ Default: ‘unstructured’
- **value_type** (*str*, optional) – Value type of the field. Either “scalar” or “vector”. The default is “scalar”.
- **check_shape** (*bool*, optional) – Wheather to check pos and field shapes. The default is *True*.
- **stacked** (*bool*, optional) – Wheather the field is stacked or not. The default is *False*.

Returns *field* – The transformed field.

Return type *numpy.ndarray*

gstools.normalizer.remove_trend_norm_mean

`gstools.normalizer.remove_trend_norm_mean(pos, field, mean=None, normalizer=None, trend=None, mesh_type='unstructured', value_type='scalar', check_shape=True, stacked=False, fit_normalizer=False)`

Remove trend, de-normalization and mean from given field.

Parameters

- **pos** (*iterable*) – Position tuple, containing main direction and transversal directions.
- **field** (*numpy.ndarray* or *list* of *numpy.ndarray*) – The spatially distributed data. You can pass a list of fields, that will be used simultaneously. Then you need to set `stacked=True`.
- **mean** (*None* or *float* or *callable*, optional) – Mean of the field if wanted. Could also be a callable. The default is *None*.

- **normalizer** (`None` or `Normalizer`, optional) – Normalizer to be applied to the field. The default is `None`.
- **trend** (`None` or `float` or `callable`, optional) – Trend of the denormalized fields. If no normalizer is applied, this behaves equal to ‘mean’. The default is `None`.
- **mesh_type** (`str`, optional) – ‘structured’ / ‘unstructured’ Default: ‘unstructured’
- **value_type** (`str`, optional) – Value type of the field. Either “scalar” or “vector”. The default is “scalar”.
- **check_shape** (`bool`, optional) – Wheather to check pos and field shapes. The default is `True`.
- **stacked** (`bool`, optional) – Wheather the field is stacked or not. The default is `False`.
- **fit_normalizer** (`bool`, optional) – Wheater to fit the data-normalizer to the given (detrended) field. Default: `False`

Returns

- **field** (`numpy.ndarray`) – The cleaned field.
- **normalizer** (`Normalizer`, optional) – The fitted normalizer for the given data. Only provided if `fit_normalizer` is `True`.

CHAPTER 4

CHANGELOG

All notable changes to **GSTools** will be documented in this file.

4.1 1.3.0 - Pure Pink - 2021-04

Topics

Geographical Coordinates Support (#113)

- added boolean init parameter `latlon` to indicate a geographic model. When given, spatial dimension is fixed to `dim=3`, `anis` and `angles` will be ignored, since anisotropy is not well-defined on a sphere.
- add property `field_dim` to indicate the dimension of the resulting field. Will be 2 if `latlon=True`
- added `yadrenko` variogram, covariance and correlation method, since the geographic models are derived from standard models in 3D by plugging in the chordal distance of two points on a sphere derived from there great-circle distance `zeta`:
 - `vario_yadrenko`: given by `variogram(2 * np.sin(zeta / 2))`
 - `cov_yadrenko`: given by `covariance(2 * np.sin(zeta / 2))`
 - `cor_yadrenko`: given by `correlation(2 * np.sin(zeta / 2))`
- added plotting routines for `yadrenko` methods described above
- the `isometrize` and `anisometrize` methods will convert `latlon` tuples (given in degree) to points on the unit-sphere in 3D and vice versa
- representation of geographical models don't display the `dim`, `anis` and `angles` parameters, but `latlon=True`
- `fit_variogram` will expect an estimated variogram with great-circle distances given in radians
- **Variogram estimation**
 - `latlon` switch implemented in `estimate_vario` routine
 - will return a variogram estimated by the great-circle distance (haversine formula) given in radians
- **Field**
 - added plotting routines for `latlon` fields
 - no vector fields possible on `latlon` fields

- correctly handle pos tuple for latlon fields

Krige Unification (#97)

- Swiss Army Knife for kriging: The `Krige` class now provides everything in one place
- “Kriging the mean” is now possible with the switch `only_mean` in the call routine
- `Simple/Ordinary/Universal/ExtDrift/Detrended` are only shortcuts to `Krige` with limited input parameter list
- We now use the `covariance` function to build up the kriging matrix (instead of variogram)
- An `unbiased` switch was added to enable simple kriging (where the unbiased condition is not given)
- An `exact` switch was added to allow smoother results, if a `nugget` is present in the model
- An `cond_err` parameter was added, where measurement error variances can be given for each conditional point
- pseudo-inverse matrix is now used to solve the kriging system (can be disabled by the new switch `pseudo_inv`), this is equal to solving the system with least-squares and prevents numerical errors
- added options `fit_normalizer` and `fit_variogram` to automatically fit normalizer and variogram to given data

Directional Variograms and Auto-binning (#87, #106, #131)

- new routine name `vario_estimate` instead of `vario_estimate_unstructured` (old kept for legacy code) for simplicity
- new routine name `vario_estimate_axis` instead of `vario_estimate_structured` (old kept for legacy code) for simplicity
- **`vario_estimate`**
 - added simple automatic binning routine to determine bins from given data (one third of box diameter as max bin distance, sturges rule for number of bins)
 - allow to pass multiple fields for joint variogram estimation (e.g. for daily precipitation) on same mesh
 - `no_data` option added to allow missing values
 - **masked fields**
 - * user can now pass a masked array (or a list of masked arrays) to deselect data points.
 - * in addition, a `mask` keyword was added to provide an external mask
 - **directional variograms**
 - * directional variograms can now be estimated
 - * either provide a list of direction vectors or angles for directions (spherical coordinates)
 - * can be controlled by given angle tolerance and (optional) bandwidth
 - * prepared for nD
 - structured fields (pos tuple describes axes) can now be passed to estimate an isotropic or directional variogram
 - distance calculation in cython routines is now independent of dimension
- **`vario_estimate_axis`**
 - estimation along array axis now possible in arbitrary dimensions
 - `no_data` option added to allow missing values (solves #83)

- axis can be given by name ("x", "y", "z") or axis number (0, 1, 2, 3, ...)

Better Variogram fitting (#78, #145)

- fixing sill possible now
- loss is now selectable for smoother handling of outliers
- r2 score can now be returned to get an impression of the goodness of fitting
- weights can be passed
- instead of deselecting parameters, one can also give fix values for each parameter
- default init guess for `len_scale` is now mean of given bin-centers
- default init guess for `var` and `nugget` is now mean of given variogram values

CovModel update (#109, #122, #157)

- add new `rescale` argument and attribute to the `CovModel` class to be able to rescale the `len_scale` (usefull for unit conversion or rescaling `len_scale` to coincide with the `integral_scale` like it's the case with the Gaussian model) See: [#90](#), [GeoStat-Framework/PyKrige#119](#)
- added new `len_rescaled` attribute to the `CovModel` class, which is the rescaled `len_scale`:
`len_rescaled = len_scale / rescale`
- new method `default_rescale` to provide default rescale factor (can be overridden)
- remove `doctest` calls
- docstring updates in `CovModel` and derived models
- updated all models to use the `cor` routine and make use of the `rescale` argument (See: [#90](#))
- TPL models got a separate base class to not repeat code
- added **new models** (See: [#88](#)):
 - **HyperSpherical**: (Replaces the old **Intersection** model) Derived from the intersection of hyperspheres in arbitrary dimensions. Coincides with the linear model in 1D, the circular model in 2D and the classical spherical model in 3D
 - **SuperSpherical**: like the **HyperSpherical**, but the shape parameter derived from dimension can be set by the user. Coincides with the **HyperSpherical** model by default
 - **JBessel**: a hole model valid in all dimensions. The shape parameter controls the dimension it was derived from. For `nu=0.5` this model coincides with the well known wave hole model.
 - **TPLSimple**: a simple truncated power law controlled by a shape parameter `nu`. Coincides with the truncated linear model for `nu=1`
 - **Cubic**: to be compatible with `scikit-gstat` in the future
- all arguments are now stored as float internally ([#157](#))
- string representation of the `CovModel` class is now using a float precision (`CovModel._prec=3`) to truncate longish output
- string representation of the `CovModel` class now only shows `anis` and `angles` if model is anisotropic resp. rotated
- dimension validity check: raise a warning, if given model is not valid in the desired dimension (See: [#86](#))

Normalizer, Trend and Mean (#124)

- new `normalize` submodule containing power-transforms for data to gain normality
- Base-Class: `Normalizer` providing basic functionality including maximum likelihood fitting
- added: `LogNormal`, `BoxCox`, `BoxCoxShift`, `YeoJohnson`, `Modulus` and `Manly`
- `normalizer`, `trend` and `mean` can be passed to SRF, Krige and variogram estimation routines
 - A trend can be a callable function, that represents a trend in input data. For example a linear decrease of temperature with height.
 - The normalizer will be applied after the data was detrended, i.e. the trend was subtracted from the data, in order to gain normality.
 - The mean is now interpreted as the mean of the normalized data. The user could also provide a callable mean, but it is mostly meant to be constant.

Arbitrary dimensions (#112)

- allow arbitrary dimensions in all routines (`CovModel`, `Krige`, `SRF`, `variogram`)
- anisotropy and rotation following a generalization of tait-bryan angles
- `CovModel` provides `isometrize` and `anisometrize` routines to convert points

New Class for Conditioned Random Fields (#130)

- **THIS BREAKS BACKWARD COMPATIBILITY**
- `CondSRF` replaces the conditioning feature of the `SRF` class, which was cumbersome and limited to Ordinary and Simple kriging
- `CondSRF` behaves similar to the `SRF` class, but instead of a covariance model, it takes a kriging class as input. With this kriging class, all conditioning related settings are defined.

Enhancements

- Python 3.9 Support [#107](#)
- add routines to format struct. pos tuple by given dim or shape
- add routine to format struct. pos tuple by given shape (variogram helper)
- remove `field.tools` subpackage
- support `meshio` ≥ 4.0 and add as dependency
- PyVista mesh support [#59](#)
- added `EARTH_RADIUS` as constant providing earths radius in km (can be used to rescale models)
- add routines `latlon2pos` and `pos2latlon` to convert lat-lon coordinates to points on unit-sphere and vice versa
- a lot of new examples and tutorials
- `RandMeth` class got a switch to select the sampling strategy
- plotter for n-D fields added [#141](#)
- antialias for contour plots of 2D fields [#141](#)
- building from source is now configured with `pyproject.toml` to care about build dependencies, see [#154](#)

Changes

- drop support for Python 3.5 [#146](#)
- added a finit limit for shape-parameters in some CovModels [#147](#)
- drop usage of `pos2xyz` and `xyz2pos`
- remove structured option from generators (structured pos need to be converted first)
- explicitly assert `dim=2,3` when generating vector fields
- simplify `pre_pos` routine to save pos tuple and reformat it an unstructured tuple
- simplify field shaping
- simplify plotting routines
- only the "unstructured" keyword is recognized everywhere, everything else is interpreted as "structured" (e.g. "rectilinear")
- use GitHub-Actions instead of TravisCI
- parallel build now controlled by env-var `GSTOOLS_BUILD_PARALLEL=1`, see [#154](#)
- install extra target for `[dev]` dropped, can be reproduced by `pip install gstools[test, doc]`, see [#154](#)

Bugfixes

- typo in keyword argument for `vario_estimate_structured` [#80](#)
- isotropic rotation of SRF was not possible [#100](#)
- `CovModel.opt_arg` now sorted [#103](#)
- `CovModel.fit`: check if weights are given as a string (numpy comparison error) [#111](#)
- several pylint fixes ([#159](#))

4.2 1.2.1 - Volatile Violet - 2020-04-14

Bugfixes

- `ModuleNotFoundError` is not present in py35
- Fixing Cressie-Bug [#76](#)
- Adding analytical formula for integral scales of rational and stable model
- remove prange from `IncomprRandMeth` summators to prevent errors on Win and macOS

4.3 1.2.0 - Volatile Violet - 2020-03-20

Enhancements

- different variogram estimator functions can now be used [#51](#)
- the `TPLGaussian` and `TPLExponential` now have analytical spectra [#67](#)
- added property `is_isotropic` to `CovModel` [#67](#)
- reworked the whole kriging sub-module to provide multiple kriging methods [#67](#)

- Simple
 - Ordinary
 - Universal
 - External Drift Kriging
 - Detrended Kriging
- a new transformation function for discrete fields has been added #70
 - reworked tutorial section in the documentation #63
 - pyvista interface #29

Changes

- Python versions 2.7 and 3.4 are no longer supported #40 #43
- CovModel: in 3D the input of anisotropy is now treated slightly different: #67
 - single given anisotropy value [e] is converted to [1, e] (it was [e, e] before)
 - two given length-scales [l_1, l_2] are converted to [l_1, l_2, l_2] (it was [l_1, l_2, l_1] before)

Bugfixes

- a race condition in the structured variogram estimation has been fixed #51

4.4 1.1.1 - Reverberating Red - 2019-11-08

Enhancements

- added a changelog. See: [commit fbea883](#)

Changes

- deprecation warnings are now printed if Python versions 2.7 or 3.4 are used #40 #41

Bugfixes

- define `spectral_density` instead of `spectrum` in covariance models since Cov-base derives spectrum. See: [commit 00f2747](#)
- better boundaries for CovModel parameters. See: <https://github.com/GeoStat-Framework/GSTools/issues/37>

4.5 1.1.0 - Reverberating Red - 2019-10-01

Enhancements

- by using Cython for all the heavy computations, we could achieve quite some speed ups and reduce the memory consumption significantly #16
- parallel computation in Cython is now supported with the help of OpenMP and the performance increase is nearly linear with increasing cores #16
- new submodule `krige` providing simple (known mean) and ordinary (estimated mean) kriging working analogous to the `srf` class
- interface to `pykrige` to use the `gstools` `CovModel` with the `pykrige` routines (<https://github.com/bsmurphy/PyKrige/issues/124>)
- the `srf` class now provides a `plot` and a `vtk_export` routine
- incompressible flow fields can now be generated #14
- new submodule providing several field transformations like: Zinn&Harvey, log-normal, bimodal, ... #13
- Python 3.4 and 3.7 wheel support #19
- field can now be generated directly on meshes from `meshio` and `ogs5py`, see: [commit f4a3439](#)
- the `srf` and kriging classes now store the last `pos`, `mesh_type` and `field` values to keep them accessible, see: [commit 29f7f1b](#)
- tutorials on all important features of GStools have been written for you guys #20
- a new interface to `pyvista` is provided to export fields to python `vtk` representation, which can be used for plotting, exploring and exporting fields #29

Changes

- the license was changed from GPL to LGPL in order to promote the use of this library #25
- the rotation angles are now interpreted in positive direction (counter clock wise)
- the `force_moments` keyword was removed from the `SRF` call method, it is now in provided as a field transformation #13
- drop support of python implementations of the variogram estimators #18
- the `variogram_normed` method was removed from the `CovModel` class due to redundancy [commit 25b1647](#)
- the position vector of 1D fields does not have to be provided in a list-like object with length 1 [commit a6f5be8](#)

Bugfixes

- several minor bugfixes

4.6 1.0.1 - Bouncy Blue - 2019-01-18

Bugfixes

- fixed Numpy and Cython version during build process

4.7 1.0.0 - Bouncy Blue - 2019-01-16

Enhancements

- added a new covariance class, which allows the easy usage of arbitrary covariance models
- added many predefined covariance models, including truncated power law models
- added [tutorials](#) and examples, showing and explaining the main features of GSTools
- variogram models can be fitted to data
- prebuilt binaries for many Linux distributions, Mac OS and Windows, making the installation, especially of the Cython code, much easier
- the generated fields can now easily be exported to vtk files
- variance scaling is supported for coarser grids
- added pure Python versions of the variogram estimators, in case somebody has problems compiling Cython code
- the [documentation](#) is now a lot cleaner and easier to use
- the code is a lot cleaner and more consistent now
- unit tests are now automatically tested when new code is pushed
- test coverage of code is shown
- GeoStat Framework now has a website, visit us: <https://geostat-framework.github.io/>

Changes

- release is not downwards compatible with release v0.4.0
- SRF creation has been adapted for the CovModel
- a tuple `pos` is now used instead of `x`, `y`, and `z` for the axes
- renamed `estimate_unstructured` and `estimate_structured` to `vario_estimate_unstructured` and `vario_estimate_structured` for less ambiguity

Bugfixes

- several minor bugfixes

4.8 0.4.0 - Glorious Green - 2018-07-17

Bugfixes

- import of cython functions put into a try-block

4.9 0.3.6 - Original Orange - 2018-07-17

First release of GSTools.

- [Webster2007] Webster, R. and Oliver, M. A. "Geostatistics for environmental scientists.", John Wiley & Sons. (2007)
- [Webster2007] Webster, R. and Oliver, M. A. "Geostatistics for environmental scientists.", John Wiley & Sons. (2007)
- [Rasmussen2003] Rasmussen, C. E., "Gaussian processes in machine learning." Summer school on machine learning. Springer, Berlin, Heidelberg, (2003)
- [Wackernagel2003] Wackernagel, H. "Multivariate geostatistics", Springer, Berlin, Heidelberg (2003)
- [Rasmussen2003] Rasmussen, C. E., "Gaussian processes in machine learning." Summer school on machine learning. Springer, Berlin, Heidelberg, (2003)
- [Chiles2009] Chiles, J. P., & Delfiner, P., "Geostatistics: modeling spatial uncertainty" (Vol. 497), John Wiley & Sons. (2009)
- [Webster2007] Webster, R. and Oliver, M. A. "Geostatistics for environmental scientists.", John Wiley & Sons. (2007)
- [Webster2007] Webster, R. and Oliver, M. A. "Geostatistics for environmental scientists.", John Wiley & Sons. (2007)
- [Webster2007] Webster, R. and Oliver, M. A. "Geostatistics for environmental scientists.", John Wiley & Sons. (2007)
- [Matern1960] Matern B., "Spatial Variation", Swedish National Institute for Forestry Research, (1960)
- [Matern1960] Matern B., "Spatial Variation", Swedish National Institute for Forestry Research, (1960)
- [Chiles2009] Chiles, J. P., & Delfiner, P., "Geostatistics: modeling spatial uncertainty" (Vol. 497), John Wiley & Sons. (2009)
- [Federico1997] Di Federico, V. and Neuman, S. P., "Scaling of random fields by means of truncated power variograms and associated spectra", Water Resources Research, 33, 1075–1085. (1997)
- [Federico1997] Di Federico, V. and Neuman, S. P., "Scaling of random fields by means of truncated power variograms and associated spectra", Water Resources Research, 33, 1075–1085. (1997)
- [Wendland1995] Wendland, H., "Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree.", Advances in computational Mathematics 4.1, 389-396. (1995)
- [Kraichnan1970] Kraichnan, R. H., "Diffusion by a random velocity field.", The physics of fluids, 13(1), 22-31., (1970)
- [Hesse2014] Heße, F., Prykhodko, V., Schlüter, S., and Attinger, S., "Generating random fields with a truncated power-law variogram: A comparison of several numerical methods", Environmental Modelling & Software, 55, 32-48., (2014)

- [Attinger03] Attinger, S. 2003, “Generalized coarse graining procedures for flow in porous media”, *Computational Geosciences*, 7(4), 253–273.
- [Webster2007] Webster, R. and Oliver, M. A. “Geostatistics for environmental scientists.”, John Wiley & Sons. (2007)
- [Webster2007] Webster, R. and Oliver, M. A. “Geostatistics for environmental scientists.”, John Wiley & Sons. (2007)
- [Wackernagel2003] Wackernagel, H., “Multivariate geostatistics”, Springer, Berlin, Heidelberg (2003)
- [Box1964] G.E.P. Box and D.R. Cox, “An Analysis of Transformations”, *Journal of the Royal Statistical Society B*, 26, 211-252, (1964)
- [Box1964] G.E.P. Box and D.R. Cox, “An Analysis of Transformations”, *Journal of the Royal Statistical Society B*, 26, 211-252, (1964)
- [Yeo2000] I.K. Yeo and R.A. Johnson, “A new family of power transformations to improve normality or symmetry.” *Biometrika*, 87(4), pp.954-959, (2000).
- [John1980] J. A. John, and N. R. Draper, “An Alternative Family of Transformations.” *Journal of the Royal Statistical Society C*, 29.2, 190-197, (1980)
- [Manly1976] B. F. J. Manly, “Exponential data transformations.”, *Journal of the Royal Statistical Society D*, 25.1, 37-42 (1976).

g

`gstools`, 113
`gstools.covmodel`, 116
`gstools.covmodel.plot`, 327
`gstools.field`, 329
`gstools.field.generator`, 341
`gstools.field.upscaling`, 347
`gstools.krige`, 352
`gstools.normalizer`, 408
`gstools.random`, 395
`gstools.tools`, 398
`gstools.transform`, 405
`gstools.variogram`, 348

Symbols

`__call__()` (*gstools.field.CondSRF* method), 334
`__call__()` (*gstools.field.Field* method), 338
`__call__()` (*gstools.field.SRF* method), 330
`__call__()` (*gstools.field.generator.IncomprRandMeth*
method), 342
`__call__()` (*gstools.field.generator.RandMeth*
method), 344
`__call__()` (*gstools.krige.Detrended* method), 389
`__call__()` (*gstools.krige.ExtDrift* method), 382
`__call__()` (*gstools.krige.Krige* method), 354
`__call__()` (*gstools.krige.Ordinary* method), 368
`__call__()` (*gstools.krige.Simple* method), 361
`__call__()` (*gstools.krige.Universal* method), 376
`__call__()` (*gstools.random.MasterRNG* method),
 395

A

`ang2dir()` (in module *gstools.tools*), 399
`angles` (*gstools.covmodel.Circular* property), 220
`angles` (*gstools.covmodel.CovModel* property), 123
`angles` (*gstools.covmodel.Cubic* property), 196
`angles` (*gstools.covmodel.Exponential* property), 149
`angles` (*gstools.covmodel.Gaussian* property), 136
`angles` (*gstools.covmodel.HyperSpherical* property),
 244
`angles` (*gstools.covmodel.JBessel* property), 269
`angles` (*gstools.covmodel.Linear* property), 208
`angles` (*gstools.covmodel.Matern* property), 161
`angles` (*gstools.covmodel.Rational* property), 185
`angles` (*gstools.covmodel.Spherical* property), 232
`angles` (*gstools.covmodel.Stable* property), 172
`angles` (*gstools.covmodel.SuperSpherical* property),
 257
`angles` (*gstools.covmodel.TPLExponential* property),
 295
`angles` (*gstools.covmodel.TPLGaussian* property),
 281
`angles` (*gstools.covmodel.TPLSimple* property), 322
`angles` (*gstools.covmodel.TPLStable* property), 309
`anis` (*gstools.covmodel.Circular* property), 220
`anis` (*gstools.covmodel.CovModel* property), 123
`anis` (*gstools.covmodel.Cubic* property), 196
`anis` (*gstools.covmodel.Exponential* property), 149

`anis` (*gstools.covmodel.Gaussian* property), 136
`anis` (*gstools.covmodel.HyperSpherical* property), 244
`anis` (*gstools.covmodel.JBessel* property), 269
`anis` (*gstools.covmodel.Linear* property), 208
`anis` (*gstools.covmodel.Matern* property), 161
`anis` (*gstools.covmodel.Rational* property), 185
`anis` (*gstools.covmodel.Spherical* property), 232
`anis` (*gstools.covmodel.Stable* property), 172
`anis` (*gstools.covmodel.SuperSpherical* property), 257
`anis` (*gstools.covmodel.TPLExponential* property),
 295
`anis` (*gstools.covmodel.TPLGaussian* property), 281
`anis` (*gstools.covmodel.TPLSimple* property), 322
`anis` (*gstools.covmodel.TPLStable* property), 309
`anis_bounds` (*gstools.covmodel.Circular* property),
 221
`anis_bounds` (*gstools.covmodel.CovModel* property),
 123
`anis_bounds` (*gstools.covmodel.Cubic* property), 197
`anis_bounds` (*gstools.covmodel.Exponential* prop-
 erty), 149
`anis_bounds` (*gstools.covmodel.Gaussian* property),
 136
`anis_bounds` (*gstools.covmodel.HyperSpherical* prop-
 erty), 245
`anis_bounds` (*gstools.covmodel.JBessel* property),
 269
`anis_bounds` (*gstools.covmodel.Linear* property), 209
`anis_bounds` (*gstools.covmodel.Matern* property),
 161
`anis_bounds` (*gstools.covmodel.Rational* property),
 185
`anis_bounds` (*gstools.covmodel.Spherical* property),
 233
`anis_bounds` (*gstools.covmodel.Stable* property), 173
`anis_bounds` (*gstools.covmodel.SuperSpherical* prop-
 erty), 257
`anis_bounds` (*gstools.covmodel.TPLExponential*
 property), 296
`anis_bounds` (*gstools.covmodel.TPLGaussian* prop-
 erty), 282
`anis_bounds` (*gstools.covmodel.TPLSimple* property),
 322
`anis_bounds` (*gstools.covmodel.TPLStable* property),

- 309
- `anisometrize()` (*gstools.covmodel.Circular* method), 216
- `anisometrize()` (*gstools.covmodel.CovModel* method), 119
- `anisometrize()` (*gstools.covmodel.Cubic* method), 192
- `anisometrize()` (*gstools.covmodel.Exponential* method), 144
- `anisometrize()` (*gstools.covmodel.Gaussian* method), 131
- `anisometrize()` (*gstools.covmodel.HyperSpherical* method), 240
- `anisometrize()` (*gstools.covmodel.JBessel* method), 264
- `anisometrize()` (*gstools.covmodel.Linear* method), 204
- `anisometrize()` (*gstools.covmodel.Matern* method), 156
- `anisometrize()` (*gstools.covmodel.Rational* method), 180
- `anisometrize()` (*gstools.covmodel.Spherical* method), 228
- `anisometrize()` (*gstools.covmodel.Stable* method), 168
- `anisometrize()` (*gstools.covmodel.SuperSpherical* method), 252
- `anisometrize()` (*gstools.covmodel.TPLExponential* method), 291
- `anisometrize()` (*gstools.covmodel.TPLGaussian* method), 277
- `anisometrize()` (*gstools.covmodel.TPLSimple* method), 317
- `anisometrize()` (*gstools.covmodel.TPLStable* method), 305
- `apply_mean_norm_trend()` (in module *gstools.normalizer*), 429
- `arg` (*gstools.covmodel.Circular* property), 221
- `arg` (*gstools.covmodel.CovModel* property), 123
- `arg` (*gstools.covmodel.Cubic* property), 197
- `arg` (*gstools.covmodel.Exponential* property), 149
- `arg` (*gstools.covmodel.Gaussian* property), 136
- `arg` (*gstools.covmodel.HyperSpherical* property), 245
- `arg` (*gstools.covmodel.JBessel* property), 269
- `arg` (*gstools.covmodel.Linear* property), 209
- `arg` (*gstools.covmodel.Matern* property), 161
- `arg` (*gstools.covmodel.Rational* property), 185
- `arg` (*gstools.covmodel.Spherical* property), 233
- `arg` (*gstools.covmodel.Stable* property), 173
- `arg` (*gstools.covmodel.SuperSpherical* property), 257
- `arg` (*gstools.covmodel.TPLExponential* property), 296
- `arg` (*gstools.covmodel.TPLGaussian* property), 282
- `arg` (*gstools.covmodel.TPLSimple* property), 322
- `arg` (*gstools.covmodel.TPLStable* property), 310
- `arg_bounds` (*gstools.covmodel.Circular* property), 221
- `arg_bounds` (*gstools.covmodel.CovModel* property), 124
- `arg_bounds` (*gstools.covmodel.Cubic* property), 197
- `arg_bounds` (*gstools.covmodel.Exponential* property), 149
- `arg_bounds` (*gstools.covmodel.Gaussian* property), 136
- `arg_bounds` (*gstools.covmodel.HyperSpherical* property), 245
- `arg_bounds` (*gstools.covmodel.JBessel* property), 269
- `arg_bounds` (*gstools.covmodel.Linear* property), 209
- `arg_bounds` (*gstools.covmodel.Matern* property), 161
- `arg_bounds` (*gstools.covmodel.Rational* property), 185
- `arg_bounds` (*gstools.covmodel.Spherical* property), 233
- `arg_bounds` (*gstools.covmodel.Stable* property), 173
- `arg_bounds` (*gstools.covmodel.SuperSpherical* property), 257
- `arg_bounds` (*gstools.covmodel.TPLExponential* property), 296
- `arg_bounds` (*gstools.covmodel.TPLGaussian* property), 282
- `arg_bounds` (*gstools.covmodel.TPLSimple* property), 322
- `arg_bounds` (*gstools.covmodel.TPLStable* property), 310
- `arg_list` (*gstools.covmodel.Circular* property), 221
- `arg_list` (*gstools.covmodel.CovModel* property), 124
- `arg_list` (*gstools.covmodel.Cubic* property), 197
- `arg_list` (*gstools.covmodel.Exponential* property), 149
- `arg_list` (*gstools.covmodel.Gaussian* property), 136
- `arg_list` (*gstools.covmodel.HyperSpherical* property), 245
- `arg_list` (*gstools.covmodel.JBessel* property), 269
- `arg_list` (*gstools.covmodel.Linear* property), 209
- `arg_list` (*gstools.covmodel.Matern* property), 161
- `arg_list` (*gstools.covmodel.Rational* property), 185
- `arg_list` (*gstools.covmodel.Spherical* property), 233
- `arg_list` (*gstools.covmodel.Stable* property), 173
- `arg_list` (*gstools.covmodel.SuperSpherical* property), 257
- `arg_list` (*gstools.covmodel.TPLExponential* property), 296
- `arg_list` (*gstools.covmodel.TPLGaussian* property), 282
- `arg_list` (*gstools.covmodel.TPLSimple* property), 322
- `arg_list` (*gstools.covmodel.TPLStable* property), 310
- ## B
- `binary()` (in module *gstools.transform*), 405
- `BoxCox` (class in *gstools.normalizer*), 414
- `boxcox()` (in module *gstools.transform*), 405
- `BoxCoxShift` (class in *gstools.normalizer*), 417
- ## C
- `calc_integral_scale()` (*gstools.covmodel.Circular* method), 216

`calc_integral_scale()`
 (`gstools.covmodel.CovModel` method), 119
`calc_integral_scale()` (`gstools.covmodel.Cubic` method), 192
`calc_integral_scale()`
 (`gstools.covmodel.Exponential` method), 144
`calc_integral_scale()`
 (`gstools.covmodel.Gaussian` method), 131
`calc_integral_scale()`
 (`gstools.covmodel.HyperSpherical` method), 240
`calc_integral_scale()` (`gstools.covmodel.JBessel` method), 264
`calc_integral_scale()` (`gstools.covmodel.Linear` method), 204
`calc_integral_scale()` (`gstools.covmodel.Matern` method), 156
`calc_integral_scale()` (`gstools.covmodel.Rational` method), 180
`calc_integral_scale()`
 (`gstools.covmodel.Spherical` method), 228
`calc_integral_scale()` (`gstools.covmodel.Stable` method), 168
`calc_integral_scale()`
 (`gstools.covmodel.SuperSpherical` method), 252
`calc_integral_scale()`
 (`gstools.covmodel.TPLExponential` method), 291
`calc_integral_scale()`
 (`gstools.covmodel.TPLGaussian` method), 277
`calc_integral_scale()`
 (`gstools.covmodel.TPLSimple` method), 317
`calc_integral_scale()`
 (`gstools.covmodel.TPLStable` method), 305
`check_arg_bounds()` (`gstools.covmodel.Circular` method), 216
`check_arg_bounds()` (`gstools.covmodel.CovModel` method), 119
`check_arg_bounds()` (`gstools.covmodel.Cubic` method), 192
`check_arg_bounds()` (`gstools.covmodel.Exponential` method), 144
`check_arg_bounds()` (`gstools.covmodel.Gaussian` method), 131
`check_arg_bounds()`
 (`gstools.covmodel.HyperSpherical` method), 240
`check_arg_bounds()` (`gstools.covmodel.JBessel` method), 264
`check_arg_bounds()` (`gstools.covmodel.Linear` method), 204
`check_arg_bounds()` (`gstools.covmodel.Matern` method), 156
`check_arg_bounds()` (`gstools.covmodel.Rational` method), 180
`check_arg_bounds()` (`gstools.covmodel.Spherical` method), 228
`check_arg_bounds()` (`gstools.covmodel.Stable` method), 168
`check_arg_bounds()`
 (`gstools.covmodel.SuperSpherical` method), 252
`check_arg_bounds()` (`gstools.covmodel.TPLExponential` method), 291
`check_arg_bounds()` (`gstools.covmodel.TPLGaussian` method), 277
`check_arg_bounds()` (`gstools.covmodel.TPLSimple` method), 317
`check_arg_bounds()` (`gstools.covmodel.TPLStable` method), 305
`check_opt_arg()` (`gstools.covmodel.Circular` method), 216
`check_opt_arg()` (`gstools.covmodel.CovModel` method), 119
`check_opt_arg()` (`gstools.covmodel.Cubic` method), 192
`check_opt_arg()` (`gstools.covmodel.Exponential` method), 144
`check_opt_arg()` (`gstools.covmodel.Gaussian` method), 156
`check_arg_bounds()` (`gstools.covmodel.Rational` method), 180
`check_arg_bounds()` (`gstools.covmodel.Spherical` method), 228
`check_arg_bounds()` (`gstools.covmodel.Stable` method), 168
`check_arg_bounds()`
 (`gstools.covmodel.SuperSpherical` method), 252
`check_arg_bounds()`
 (`gstools.covmodel.TPLExponential` method), 291
`check_arg_bounds()`
 (`gstools.covmodel.TPLGaussian` method), 277
`check_arg_bounds()` (`gstools.covmodel.TPLSimple` method), 317
`check_arg_bounds()` (`gstools.covmodel.TPLStable` method), 305
`check_dim()` (`gstools.covmodel.Circular` method), 216
`check_dim()` (`gstools.covmodel.CovModel` method), 119
`check_dim()` (`gstools.covmodel.Cubic` method), 192
`check_dim()` (`gstools.covmodel.Exponential` method), 144
`check_dim()` (`gstools.covmodel.Gaussian` method), 131
`check_dim()` (`gstools.covmodel.HyperSpherical` method), 240
`check_dim()` (`gstools.covmodel.JBessel` method), 264
`check_dim()` (`gstools.covmodel.Linear` method), 204
`check_dim()` (`gstools.covmodel.Matern` method), 156
`check_dim()` (`gstools.covmodel.Rational` method), 180
`check_dim()` (`gstools.covmodel.Spherical` method), 228
`check_dim()` (`gstools.covmodel.Stable` method), 168
`check_dim()`
 (`gstools.covmodel.SuperSpherical` method), 252
`check_dim()` (`gstools.covmodel.TPLExponential` method), 291
`check_dim()` (`gstools.covmodel.TPLGaussian` method), 277
`check_dim()` (`gstools.covmodel.TPLSimple` method), 317
`check_dim()` (`gstools.covmodel.TPLStable` method), 305

method), 131
`check_opt_arg()` (*gstools.covmodel.HyperSpherical method*), 240
`check_opt_arg()` (*gstools.covmodel.JBessel method*), 264
`check_opt_arg()` (*gstools.covmodel.Linear method*), 204
`check_opt_arg()` (*gstools.covmodel.Matern method*), 156
`check_opt_arg()` (*gstools.covmodel.Rational method*), 180
`check_opt_arg()` (*gstools.covmodel.Spherical method*), 228
`check_opt_arg()` (*gstools.covmodel.Stable method*), 168
`check_opt_arg()` (*gstools.covmodel.SuperSpherical method*), 252
`check_opt_arg()` (*gstools.covmodel.TPExponential method*), 291
`check_opt_arg()` (*gstools.covmodel.TPLGaussian method*), 277
`check_opt_arg()` (*gstools.covmodel.TPLSimple method*), 317
`check_opt_arg()` (*gstools.covmodel.TPLStable method*), 305
`Circular` (*class in gstools.covmodel*), 213
`cond_err` (*gstools.krige.Detrended property*), 392
`cond_err` (*gstools.krige.ExtDrift property*), 386
`cond_err` (*gstools.krige.Krige property*), 357
`cond_err` (*gstools.krige.Ordinary property*), 371
`cond_err` (*gstools.krige.Simple property*), 364
`cond_err` (*gstools.krige.Universal property*), 379
`cond_ext_drift` (*gstools.krige.Detrended property*), 392
`cond_ext_drift` (*gstools.krige.ExtDrift property*), 386
`cond_ext_drift` (*gstools.krige.Krige property*), 357
`cond_ext_drift` (*gstools.krige.Ordinary property*), 371
`cond_ext_drift` (*gstools.krige.Simple property*), 365
`cond_ext_drift` (*gstools.krige.Universal property*), 379
`cond_mean` (*gstools.krige.Detrended property*), 392
`cond_mean` (*gstools.krige.ExtDrift property*), 386
`cond_mean` (*gstools.krige.Krige property*), 357
`cond_mean` (*gstools.krige.Ordinary property*), 372
`cond_mean` (*gstools.krige.Simple property*), 365
`cond_mean` (*gstools.krige.Universal property*), 379
`cond_no` (*gstools.krige.Detrended property*), 393
`cond_no` (*gstools.krige.ExtDrift property*), 386
`cond_no` (*gstools.krige.Krige property*), 357
`cond_no` (*gstools.krige.Ordinary property*), 372
`cond_no` (*gstools.krige.Simple property*), 365
`cond_no` (*gstools.krige.Universal property*), 379
`cond_pos` (*gstools.krige.Detrended property*), 393
`cond_pos` (*gstools.krige.ExtDrift property*), 386
`cond_pos` (*gstools.krige.Krige property*), 357
`cond_pos` (*gstools.krige.Ordinary property*), 372
`cond_pos` (*gstools.krige.Simple property*), 365
`cond_pos` (*gstools.krige.Universal property*), 379
`cond_trend` (*gstools.krige.Detrended property*), 393
`cond_trend` (*gstools.krige.ExtDrift property*), 386
`cond_trend` (*gstools.krige.Krige property*), 357
`cond_trend` (*gstools.krige.Ordinary property*), 372
`cond_trend` (*gstools.krige.Simple property*), 365
`cond_trend` (*gstools.krige.Universal property*), 379
`cond_val` (*gstools.krige.Detrended property*), 393
`cond_val` (*gstools.krige.ExtDrift property*), 386
`cond_val` (*gstools.krige.Krige property*), 357
`cond_val` (*gstools.krige.Ordinary property*), 372
`cond_val` (*gstools.krige.Simple property*), 365
`cond_val` (*gstools.krige.Universal property*), 379
`CondSRF` (*class in gstools.field*), 334
`confidence_scaling()` (*in module gstools.tools*), 399
`cor()` (*gstools.covmodel.Circular method*), 216
`cor()` (*gstools.covmodel.Cubic method*), 192
`cor()` (*gstools.covmodel.Exponential method*), 144
`cor()` (*gstools.covmodel.Gaussian method*), 132
`cor()` (*gstools.covmodel.HyperSpherical method*), 240
`cor()` (*gstools.covmodel.JBessel method*), 264
`cor()` (*gstools.covmodel.Linear method*), 204
`cor()` (*gstools.covmodel.Matern method*), 156
`cor()` (*gstools.covmodel.Rational method*), 180
`cor()` (*gstools.covmodel.Spherical method*), 228
`cor()` (*gstools.covmodel.Stable method*), 168
`cor()` (*gstools.covmodel.SuperSpherical method*), 252
`cor()` (*gstools.covmodel.TPExponential method*), 291
`cor()` (*gstools.covmodel.TPLGaussian method*), 277
`cor()` (*gstools.covmodel.TPLSimple method*), 317
`cor()` (*gstools.covmodel.TPLStable method*), 305
`cor_axis()` (*gstools.covmodel.Circular method*), 216
`cor_axis()` (*gstools.covmodel.CovModel method*), 119
`cor_axis()` (*gstools.covmodel.Cubic method*), 192
`cor_axis()` (*gstools.covmodel.Exponential method*), 144
`cor_axis()` (*gstools.covmodel.Gaussian method*), 132
`cor_axis()` (*gstools.covmodel.HyperSpherical method*), 240
`cor_axis()` (*gstools.covmodel.JBessel method*), 264
`cor_axis()` (*gstools.covmodel.Linear method*), 204
`cor_axis()` (*gstools.covmodel.Matern method*), 156
`cor_axis()` (*gstools.covmodel.Rational method*), 180
`cor_axis()` (*gstools.covmodel.Spherical method*), 228
`cor_axis()` (*gstools.covmodel.Stable method*), 168
`cor_axis()` (*gstools.covmodel.SuperSpherical method*), 252
`cor_axis()` (*gstools.covmodel.TPExponential method*), 291
`cor_axis()` (*gstools.covmodel.TPLGaussian method*), 277
`cor_axis()` (*gstools.covmodel.TPLSimple method*), 317

`cor_axis()` (*gstools.covmodel.TPLStable method*), 305
`cor_spatial()` (*gstools.covmodel.Circular method*), 216
`cor_spatial()` (*gstools.covmodel.CovModel method*), 119
`cor_spatial()` (*gstools.covmodel.Cubic method*), 192
`cor_spatial()` (*gstools.covmodel.Exponential method*), 144
`cor_spatial()` (*gstools.covmodel.Gaussian method*), 132
`cor_spatial()` (*gstools.covmodel.HyperSpherical method*), 240
`cor_spatial()` (*gstools.covmodel.JBessel method*), 264
`cor_spatial()` (*gstools.covmodel.Linear method*), 204
`cor_spatial()` (*gstools.covmodel.Matern method*), 156
`cor_spatial()` (*gstools.covmodel.Rational method*), 180
`cor_spatial()` (*gstools.covmodel.Spherical method*), 228
`cor_spatial()` (*gstools.covmodel.Stable method*), 168
`cor_spatial()` (*gstools.covmodel.SuperSpherical method*), 252
`cor_spatial()` (*gstools.covmodel.TPLExponential method*), 291
`cor_spatial()` (*gstools.covmodel.TPLGaussian method*), 277
`cor_spatial()` (*gstools.covmodel.TPLSimple method*), 317
`cor_spatial()` (*gstools.covmodel.TPLStable method*), 305
`cor_yadrenko()` (*gstools.covmodel.Circular method*), 216
`cor_yadrenko()` (*gstools.covmodel.CovModel method*), 119
`cor_yadrenko()` (*gstools.covmodel.Cubic method*), 192
`cor_yadrenko()` (*gstools.covmodel.Exponential method*), 144
`cor_yadrenko()` (*gstools.covmodel.Gaussian method*), 132
`cor_yadrenko()` (*gstools.covmodel.HyperSpherical method*), 240
`cor_yadrenko()` (*gstools.covmodel.JBessel method*), 264
`cor_yadrenko()` (*gstools.covmodel.Linear method*), 204
`cor_yadrenko()` (*gstools.covmodel.Matern method*), 156
`cor_yadrenko()` (*gstools.covmodel.Rational method*), 180
`cor_yadrenko()` (*gstools.covmodel.Spherical method*), 228
`cor_yadrenko()` (*gstools.covmodel.Stable method*), 168
`cor_yadrenko()` (*gstools.covmodel.SuperSpherical method*), 252
`cor_yadrenko()` (*gstools.covmodel.TPLExponential method*), 291
`cor_yadrenko()` (*gstools.covmodel.TPLGaussian method*), 277
`cor_yadrenko()` (*gstools.covmodel.TPLSimple method*), 317
`cor_yadrenko()` (*gstools.covmodel.TPLStable method*), 305
`correlation()` (*gstools.covmodel.Circular method*), 216
`correlation()` (*gstools.covmodel.Cubic method*), 192
`correlation()` (*gstools.covmodel.Exponential method*), 144
`correlation()` (*gstools.covmodel.Gaussian method*), 132
`correlation()` (*gstools.covmodel.HyperSpherical method*), 240
`correlation()` (*gstools.covmodel.JBessel method*), 264
`correlation()` (*gstools.covmodel.Linear method*), 204
`correlation()` (*gstools.covmodel.Matern method*), 156
`correlation()` (*gstools.covmodel.Rational method*), 180
`correlation()` (*gstools.covmodel.Spherical method*), 228
`correlation()` (*gstools.covmodel.Stable method*), 168
`correlation()` (*gstools.covmodel.SuperSpherical method*), 252
`correlation()` (*gstools.covmodel.TPLExponential method*), 291
`correlation()` (*gstools.covmodel.TPLGaussian method*), 277
`correlation()` (*gstools.covmodel.TPLSimple method*), 318
`correlation()` (*gstools.covmodel.TPLStable method*), 305
`cov_axis()` (*gstools.covmodel.Circular method*), 216
`cov_axis()` (*gstools.covmodel.CovModel method*), 119
`cov_axis()` (*gstools.covmodel.Cubic method*), 192
`cov_axis()` (*gstools.covmodel.Exponential method*), 144
`cov_axis()` (*gstools.covmodel.Gaussian method*), 132
`cov_axis()` (*gstools.covmodel.HyperSpherical method*), 240
`cov_axis()` (*gstools.covmodel.JBessel method*), 264
`cov_axis()` (*gstools.covmodel.Linear method*), 204
`cov_axis()` (*gstools.covmodel.Matern method*), 156
`cov_axis()` (*gstools.covmodel.Rational method*), 180
`cov_axis()` (*gstools.covmodel.Spherical method*), 228

`cov_axis()` (*gstools.covmodel.Stable method*), 168
`cov_axis()` (*gstools.covmodel.SuperSpherical method*), 252
`cov_axis()` (*gstools.covmodel.TPLExponential method*), 291
`cov_axis()` (*gstools.covmodel.TPLGaussian method*), 277
`cov_axis()` (*gstools.covmodel.TPLSimple method*), 318
`cov_axis()` (*gstools.covmodel.TPLStable method*), 305
`cov_nugget()` (*gstools.covmodel.Circular method*), 216
`cov_nugget()` (*gstools.covmodel.CovModel method*), 119
`cov_nugget()` (*gstools.covmodel.Cubic method*), 192
`cov_nugget()` (*gstools.covmodel.Exponential method*), 144
`cov_nugget()` (*gstools.covmodel.Gaussian method*), 132
`cov_nugget()` (*gstools.covmodel.HyperSpherical method*), 240
`cov_nugget()` (*gstools.covmodel.JBessel method*), 264
`cov_nugget()` (*gstools.covmodel.Linear method*), 204
`cov_nugget()` (*gstools.covmodel.Matern method*), 157
`cov_nugget()` (*gstools.covmodel.Rational method*), 181
`cov_nugget()` (*gstools.covmodel.Spherical method*), 228
`cov_nugget()` (*gstools.covmodel.Stable method*), 168
`cov_nugget()` (*gstools.covmodel.SuperSpherical method*), 253
`cov_nugget()` (*gstools.covmodel.TPLExponential method*), 291
`cov_nugget()` (*gstools.covmodel.TPLGaussian method*), 277
`cov_nugget()` (*gstools.covmodel.TPLSimple method*), 318
`cov_nugget()` (*gstools.covmodel.TPLStable method*), 305
`cov_spatial()` (*gstools.covmodel.Circular method*), 217
`cov_spatial()` (*gstools.covmodel.CovModel method*), 119
`cov_spatial()` (*gstools.covmodel.Cubic method*), 193
`cov_spatial()` (*gstools.covmodel.Exponential method*), 144
`cov_spatial()` (*gstools.covmodel.Gaussian method*), 132
`cov_spatial()` (*gstools.covmodel.HyperSpherical method*), 241
`cov_spatial()` (*gstools.covmodel.JBessel method*), 265
`cov_spatial()` (*gstools.covmodel.Linear method*), 205
`cov_spatial()` (*gstools.covmodel.Matern method*), 157
`cov_spatial()` (*gstools.covmodel.Rational method*), 181
`cov_spatial()` (*gstools.covmodel.Spherical method*), 229
`cov_spatial()` (*gstools.covmodel.Stable method*), 168
`cov_spatial()` (*gstools.covmodel.SuperSpherical method*), 253
`cov_spatial()` (*gstools.covmodel.TPLExponential method*), 291
`cov_spatial()` (*gstools.covmodel.TPLGaussian method*), 277
`cov_spatial()` (*gstools.covmodel.TPLSimple method*), 318
`cov_spatial()` (*gstools.covmodel.TPLStable method*), 305
`cov_yadrenko()` (*gstools.covmodel.Circular method*), 217
`cov_yadrenko()` (*gstools.covmodel.CovModel method*), 119
`cov_yadrenko()` (*gstools.covmodel.Cubic method*), 193
`cov_yadrenko()` (*gstools.covmodel.Exponential method*), 145
`cov_yadrenko()` (*gstools.covmodel.Gaussian method*), 132
`cov_yadrenko()` (*gstools.covmodel.HyperSpherical method*), 241
`cov_yadrenko()` (*gstools.covmodel.JBessel method*), 265
`cov_yadrenko()` (*gstools.covmodel.Linear method*), 205
`cov_yadrenko()` (*gstools.covmodel.Matern method*), 157
`cov_yadrenko()` (*gstools.covmodel.Rational method*), 181
`cov_yadrenko()` (*gstools.covmodel.Spherical method*), 229
`cov_yadrenko()` (*gstools.covmodel.Stable method*), 168
`cov_yadrenko()` (*gstools.covmodel.SuperSpherical method*), 253
`cov_yadrenko()` (*gstools.covmodel.TPLExponential method*), 291
`cov_yadrenko()` (*gstools.covmodel.TPLGaussian method*), 277
`cov_yadrenko()` (*gstools.covmodel.TPLSimple method*), 318
`cov_yadrenko()` (*gstools.covmodel.TPLStable method*), 305
`covariance()` (*gstools.covmodel.Circular method*), 217
`covariance()` (*gstools.covmodel.Cubic method*), 193
`covariance()` (*gstools.covmodel.Exponential method*), 145
`covariance()` (*gstools.covmodel.Gaussian method*),

132
 covariance() (*gstools.covmodel.HyperSpherical method*), 241
 covariance() (*gstools.covmodel.JBessel method*), 265
 covariance() (*gstools.covmodel.Linear method*), 205
 covariance() (*gstools.covmodel.Matern method*), 157
 covariance() (*gstools.covmodel.Rational method*), 181
 covariance() (*gstools.covmodel.Spherical method*), 229
 covariance() (*gstools.covmodel.Stable method*), 168
 covariance() (*gstools.covmodel.SuperSpherical method*), 253
 covariance() (*gstools.covmodel.TPLExponential method*), 291
 covariance() (*gstools.covmodel.TPLGaussian method*), 277
 covariance() (*gstools.covmodel.TPLSimple method*), 318
 covariance() (*gstools.covmodel.TPLStable method*), 305
 CovModel (*class in gstools.covmodel*), 116
 Cubic (*class in gstools.covmodel*), 189

D

default_arg_bounds() (*gstools.covmodel.Circular method*), 217
 default_arg_bounds() (*gstools.covmodel.CovModel method*), 120
 default_arg_bounds() (*gstools.covmodel.Cubic method*), 193
 default_arg_bounds() (*gstools.covmodel.Exponential method*), 145
 default_arg_bounds() (*gstools.covmodel.Gaussian method*), 132
 default_arg_bounds() (*gstools.covmodel.HyperSpherical method*), 241
 default_arg_bounds() (*gstools.covmodel.JBessel method*), 265
 default_arg_bounds() (*gstools.covmodel.Linear method*), 205
 default_arg_bounds() (*gstools.covmodel.Matern method*), 157
 default_arg_bounds() (*gstools.covmodel.Rational method*), 181
 default_arg_bounds() (*gstools.covmodel.Spherical method*), 229
 default_arg_bounds() (*gstools.covmodel.Stable method*), 168
 default_arg_bounds() (*gstools.covmodel.SuperSpherical method*), 253
 default_arg_bounds()

(*gstools.covmodel.TPLExponential method*), 291
 default_arg_bounds() (*gstools.covmodel.TPLGaussian method*), 277
 default_arg_bounds() (*gstools.covmodel.TPLSimple method*), 318
 default_arg_bounds() (*gstools.covmodel.TPLStable method*), 305
 default_opt_arg() (*gstools.covmodel.Circular method*), 217
 default_opt_arg() (*gstools.covmodel.CovModel method*), 120
 default_opt_arg() (*gstools.covmodel.Cubic method*), 193
 default_opt_arg() (*gstools.covmodel.Exponential method*), 145
 default_opt_arg() (*gstools.covmodel.Gaussian method*), 132
 default_opt_arg() (*gstools.covmodel.HyperSpherical method*), 241
 default_opt_arg() (*gstools.covmodel.JBessel method*), 265
 default_opt_arg() (*gstools.covmodel.Linear method*), 205
 default_opt_arg() (*gstools.covmodel.Matern method*), 157
 default_opt_arg() (*gstools.covmodel.Rational method*), 181
 default_opt_arg() (*gstools.covmodel.Spherical method*), 229
 default_opt_arg() (*gstools.covmodel.Stable method*), 169
 default_opt_arg() (*gstools.covmodel.SuperSpherical method*), 253
 default_opt_arg() (*gstools.covmodel.TPLExponential method*), 292
 default_opt_arg() (*gstools.covmodel.TPLGaussian method*), 278
 default_opt_arg() (*gstools.covmodel.TPLSimple method*), 318
 default_opt_arg() (*gstools.covmodel.TPLStable method*), 305
 default_opt_arg_bounds() (*gstools.covmodel.Circular method*), 217
 default_opt_arg_bounds() (*gstools.covmodel.CovModel method*), 120
 default_opt_arg_bounds() (*gstools.covmodel.Cubic method*), 193
 default_opt_arg_bounds() (*gstools.covmodel.Exponential method*), 145
 default_opt_arg_bounds() (*gstools.covmodel.Gaussian method*), 132
 default_opt_arg_bounds()

(*gstools.covmodel.HyperSpherical* method), 241

default_opt_arg_bounds() (*gstools.covmodel.JBessel* method), 265

default_opt_arg_bounds() (*gstools.covmodel.Linear* method), 205

default_opt_arg_bounds() (*gstools.covmodel.Matern* method), 157

default_opt_arg_bounds() (*gstools.covmodel.Rational* method), 181

default_opt_arg_bounds() (*gstools.covmodel.Spherical* method), 229

default_opt_arg_bounds() (*gstools.covmodel.Stable* method), 169

default_opt_arg_bounds() (*gstools.covmodel.SuperSpherical* method), 253

default_opt_arg_bounds() (*gstools.covmodel.TPLExponential* method), 292

default_opt_arg_bounds() (*gstools.covmodel.TPLGaussian* method), 278

default_opt_arg_bounds() (*gstools.covmodel.TPLSimple* method), 318

default_opt_arg_bounds() (*gstools.covmodel.TPLStable* method), 306

default_parameter (*gstools.normalizer.BoxCox* attribute), 415

default_parameter (*gstools.normalizer.BoxCoxShift* attribute), 418

default_parameter (*gstools.normalizer.LogNormal* attribute), 412

default_parameter (*gstools.normalizer.Manly* attribute), 427

default_parameter (*gstools.normalizer.Modulus* attribute), 424

default_parameter (*gstools.normalizer.Normalizer* attribute), 409

default_parameter (*gstools.normalizer.YeoJohnson* attribute), 421

default_rescale() (*gstools.covmodel.Circular* method), 217

default_rescale() (*gstools.covmodel.CovModel* method), 120

default_rescale() (*gstools.covmodel.Cubic* method), 193

default_rescale() (*gstools.covmodel.Exponential* method), 145

default_rescale() (*gstools.covmodel.Gaussian* method), 132

default_rescale() (*gstools.covmodel.HyperSpherical* method), 241

default_rescale() (*gstools.covmodel.JBessel* method), 265

default_rescale() (*gstools.covmodel.Linear* method), 205

default_rescale() (*gstools.covmodel.Matern* method), 157

default_rescale() (*gstools.covmodel.Rational* method), 181

default_rescale() (*gstools.covmodel.Spherical* method), 229

default_rescale() (*gstools.covmodel.Stable* method), 169

default_rescale() (*gstools.covmodel.SuperSpherical* method), 253

default_rescale() (*gstools.covmodel.TPLExponential* method), 292

default_rescale() (*gstools.covmodel.TPLGaussian* method), 278

default_rescale() (*gstools.covmodel.TPLSimple* method), 318

default_rescale() (*gstools.covmodel.TPLStable* method), 306

denormalize() (*gstools.normalizer.BoxCox* method), 414

denormalize() (*gstools.normalizer.BoxCoxShift* method), 417

denormalize() (*gstools.normalizer.LogNormal* method), 411

denormalize() (*gstools.normalizer.Manly* method), 426

denormalize() (*gstools.normalizer.Modulus* method), 423

denormalize() (*gstools.normalizer.Normalizer* method), 408

denormalize() (*gstools.normalizer.YeoJohnson* method), 420

denormalize_range (*gstools.normalizer.BoxCox* property), 415

denormalize_range (*gstools.normalizer.BoxCoxShift* property), 419

denormalize_range (*gstools.normalizer.LogNormal* attribute), 412

denormalize_range (*gstools.normalizer.Manly* property), 427

denormalize_range (*gstools.normalizer.Modulus* attribute), 424

denormalize_range (*gstools.normalizer.Normalizer* attribute), 409

denormalize_range (*gstools.normalizer.YeoJohnson* attribute), 421

derivative() (*gstools.normalizer.BoxCox* method), 414

derivative() (*gstools.normalizer.BoxCoxShift* method), 418

derivative() (*gstools.normalizer.LogNormal* method), 411

derivative() (*gstools.normalizer.Manly* method), 426

derivative() (*gstools.normalizer.Modulus* method), 423

derivative() (*gstools.normalizer.Normalizer* method), 408

- method*), 408
 - `derivative()` (*gstools.normalizer.YeoJohnson* *method*), 420
 - `Detrended` (*class in gstools.krige*), 388
 - `dim` (*gstools.covmodel.Circular* *property*), 221
 - `dim` (*gstools.covmodel.CovModel* *property*), 124
 - `dim` (*gstools.covmodel.Cubic* *property*), 197
 - `dim` (*gstools.covmodel.Exponential* *property*), 149
 - `dim` (*gstools.covmodel.Gaussian* *property*), 137
 - `dim` (*gstools.covmodel.HyperSpherical* *property*), 245
 - `dim` (*gstools.covmodel.JBessel* *property*), 269
 - `dim` (*gstools.covmodel.Linear* *property*), 209
 - `dim` (*gstools.covmodel.Matern* *property*), 161
 - `dim` (*gstools.covmodel.Rational* *property*), 185
 - `dim` (*gstools.covmodel.Spherical* *property*), 233
 - `dim` (*gstools.covmodel.Stable* *property*), 173
 - `dim` (*gstools.covmodel.SuperSpherical* *property*), 257
 - `dim` (*gstools.covmodel.TPLeponential* *property*), 296
 - `dim` (*gstools.covmodel.TPLGaussian* *property*), 282
 - `dim` (*gstools.covmodel.TPLSimple* *property*), 322
 - `dim` (*gstools.covmodel.TPLStable* *property*), 310
 - `dim` (*gstools.field.CondSRF* *property*), 337
 - `dim` (*gstools.field.Field* *property*), 340
 - `dim` (*gstools.field.SRF* *property*), 332
 - `dim` (*gstools.krige.Detrended* *property*), 393
 - `dim` (*gstools.krige.ExtDrift* *property*), 386
 - `dim` (*gstools.krige.Krige* *property*), 358
 - `dim` (*gstools.krige.Ordinary* *property*), 372
 - `dim` (*gstools.krige.Simple* *property*), 365
 - `dim` (*gstools.krige.Universal* *property*), 379
 - `discrete()` (*in module gstools.transform*), 405
 - `dist_func` (*gstools.covmodel.Circular* *property*), 221
 - `dist_func` (*gstools.covmodel.CovModel* *property*), 124
 - `dist_func` (*gstools.covmodel.Cubic* *property*), 197
 - `dist_func` (*gstools.covmodel.Exponential* *property*), 149
 - `dist_func` (*gstools.covmodel.Gaussian* *property*), 137
 - `dist_func` (*gstools.covmodel.HyperSpherical* *property*), 245
 - `dist_func` (*gstools.covmodel.JBessel* *property*), 269
 - `dist_func` (*gstools.covmodel.Linear* *property*), 209
 - `dist_func` (*gstools.covmodel.Matern* *property*), 161
 - `dist_func` (*gstools.covmodel.Rational* *property*), 185
 - `dist_func` (*gstools.covmodel.Spherical* *property*), 233
 - `dist_func` (*gstools.covmodel.Stable* *property*), 173
 - `dist_func` (*gstools.covmodel.SuperSpherical* *property*), 257
 - `dist_func` (*gstools.covmodel.TPLeponential* *property*), 296
 - `dist_func` (*gstools.covmodel.TPLGaussian* *property*), 282
 - `dist_func` (*gstools.covmodel.TPLSimple* *property*), 322
 - `dist_func` (*gstools.covmodel.TPLStable* *property*), 310
 - `dist_gen()` (*in module gstools.random*), 397
 - `do_rotation` (*gstools.covmodel.Circular* *property*), 221
 - `do_rotation` (*gstools.covmodel.CovModel* *property*), 124
 - `do_rotation` (*gstools.covmodel.Cubic* *property*), 197
 - `do_rotation` (*gstools.covmodel.Exponential* *property*), 149
 - `do_rotation` (*gstools.covmodel.Gaussian* *property*), 137
 - `do_rotation` (*gstools.covmodel.HyperSpherical* *property*), 245
 - `do_rotation` (*gstools.covmodel.JBessel* *property*), 269
 - `do_rotation` (*gstools.covmodel.Linear* *property*), 209
 - `do_rotation` (*gstools.covmodel.Matern* *property*), 161
 - `do_rotation` (*gstools.covmodel.Rational* *property*), 185
 - `do_rotation` (*gstools.covmodel.Spherical* *property*), 233
 - `do_rotation` (*gstools.covmodel.Stable* *property*), 173
 - `do_rotation` (*gstools.covmodel.SuperSpherical* *property*), 257
 - `do_rotation` (*gstools.covmodel.TPLeponential* *property*), 296
 - `do_rotation` (*gstools.covmodel.TPLGaussian* *property*), 282
 - `do_rotation` (*gstools.covmodel.TPLSimple* *property*), 322
 - `do_rotation` (*gstools.covmodel.TPLStable* *property*), 310
 - `drift_functions` (*gstools.krige.Detrended* *property*), 393
 - `drift_functions` (*gstools.krige.ExtDrift* *property*), 386
 - `drift_functions` (*gstools.krige.Krige* *property*), 358
 - `drift_functions` (*gstools.krige.Ordinary* *property*), 372
 - `drift_functions` (*gstools.krige.Simple* *property*), 365
 - `drift_functions` (*gstools.krige.Universal* *property*), 379
 - `drift_no` (*gstools.krige.Detrended* *property*), 393
 - `drift_no` (*gstools.krige.ExtDrift* *property*), 386
 - `drift_no` (*gstools.krige.Krige* *property*), 358
 - `drift_no` (*gstools.krige.Ordinary* *property*), 372
 - `drift_no` (*gstools.krige.Simple* *property*), 365
 - `drift_no` (*gstools.krige.Universal* *property*), 379
- ## E
- `EARTH_RADIUS` (*in module gstools.tools*), 404
 - `exact` (*gstools.krige.Detrended* *property*), 393
 - `exact` (*gstools.krige.ExtDrift* *property*), 386
 - `exact` (*gstools.krige.Krige* *property*), 358
 - `exact` (*gstools.krige.Ordinary* *property*), 372
 - `exact` (*gstools.krige.Simple* *property*), 365
 - `exact` (*gstools.krige.Universal* *property*), 379
 - `exp_int()` (*in module gstools.tools*), 399

Exponential (class in *gstools.covmodel*), 141
 ext_drift_no (*gstools.krige.Detrended* property), 393
 ext_drift_no (*gstools.krige.ExtDrift* property), 386
 ext_drift_no (*gstools.krige.Krige* property), 358
 ext_drift_no (*gstools.krige.Ordinary* property), 372
 ext_drift_no (*gstools.krige.Simple* property), 365
 ext_drift_no (*gstools.krige.Universal* property), 379
 ExtDrift (class in *gstools.krige*), 381

F

Field (class in *gstools.field*), 338
 field_dim (*gstools.covmodel.Circular* property), 221
 field_dim (*gstools.covmodel.CovModel* property), 124
 field_dim (*gstools.covmodel.Cubic* property), 197
 field_dim (*gstools.covmodel.Exponential* property), 149
 field_dim (*gstools.covmodel.Gaussian* property), 137
 field_dim (*gstools.covmodel.HyperSpherical* property), 245
 field_dim (*gstools.covmodel.JBessel* property), 269
 field_dim (*gstools.covmodel.Linear* property), 209
 field_dim (*gstools.covmodel.Matern* property), 161
 field_dim (*gstools.covmodel.Rational* property), 185
 field_dim (*gstools.covmodel.Spherical* property), 233
 field_dim (*gstools.covmodel.Stable* property), 173
 field_dim (*gstools.covmodel.SuperSpherical* property), 257
 field_dim (*gstools.covmodel.TPLeponential* property), 296
 field_dim (*gstools.covmodel.TPLGaussian* property), 282
 field_dim (*gstools.covmodel.TPLSimple* property), 323
 field_dim (*gstools.covmodel.TPLStable* property), 310
 fit() (*gstools.normalizer.BoxCox* method), 414
 fit() (*gstools.normalizer.BoxCoxShift* method), 418
 fit() (*gstools.normalizer.LogNormal* method), 411
 fit() (*gstools.normalizer.Manly* method), 426
 fit() (*gstools.normalizer.Modulus* method), 423
 fit() (*gstools.normalizer.Normalizer* method), 408
 fit() (*gstools.normalizer.YeoJohnson* method), 420
 fit_variogram() (*gstools.covmodel.Circular* method), 217
 fit_variogram() (*gstools.covmodel.CovModel* method), 120
 fit_variogram() (*gstools.covmodel.Cubic* method), 193
 fit_variogram() (*gstools.covmodel.Exponential* method), 145
 fit_variogram() (*gstools.covmodel.Gaussian* method), 132
 fit_variogram() (*gstools.covmodel.HyperSpherical* method), 241
 fit_variogram() (*gstools.covmodel.JBessel* method), 265

fit_variogram() (*gstools.covmodel.Linear* method), 205
 fit_variogram() (*gstools.covmodel.Matern* method), 157
 fit_variogram() (*gstools.covmodel.Rational* method), 181
 fit_variogram() (*gstools.covmodel.Spherical* method), 229
 fit_variogram() (*gstools.covmodel.Stable* method), 169
 fit_variogram() (*gstools.covmodel.SuperSpherical* method), 253
 fit_variogram() (*gstools.covmodel.TPLeponential* method), 292
 fit_variogram() (*gstools.covmodel.TPLGaussian* method), 278
 fit_variogram() (*gstools.covmodel.TPLSimple* method), 318
 fit_variogram() (*gstools.covmodel.TPLStable* method), 306
 fix_dim() (*gstools.covmodel.Circular* method), 219
 fix_dim() (*gstools.covmodel.CovModel* method), 121
 fix_dim() (*gstools.covmodel.Cubic* method), 195
 fix_dim() (*gstools.covmodel.Exponential* method), 147
 fix_dim() (*gstools.covmodel.Gaussian* method), 134
 fix_dim() (*gstools.covmodel.HyperSpherical* method), 243
 fix_dim() (*gstools.covmodel.JBessel* method), 267
 fix_dim() (*gstools.covmodel.Linear* method), 207
 fix_dim() (*gstools.covmodel.Matern* method), 159
 fix_dim() (*gstools.covmodel.Rational* method), 183
 fix_dim() (*gstools.covmodel.Spherical* method), 231
 fix_dim() (*gstools.covmodel.Stable* method), 171
 fix_dim() (*gstools.covmodel.SuperSpherical* method), 255
 fix_dim() (*gstools.covmodel.TPLeponential* method), 294
 fix_dim() (*gstools.covmodel.TPLGaussian* method), 280
 fix_dim() (*gstools.covmodel.TPLSimple* method), 320
 fix_dim() (*gstools.covmodel.TPLStable* method), 307

G

Gaussian (class in *gstools.covmodel*), 128
 generate_grid() (in module *gstools.tools*), 399
 generate_st_grid() (in module *gstools.tools*), 399
 generator (*gstools.field.CondSRF* property), 337
 generator (*gstools.field.SRF* property), 332
 get_mean() (*gstools.krige.Detrended* method), 390
 get_mean() (*gstools.krige.ExtDrift* method), 383
 get_mean() (*gstools.krige.Krige* method), 355
 get_mean() (*gstools.krige.Ordinary* method), 369
 get_mean() (*gstools.krige.Simple* method), 362
 get_mean() (*gstools.krige.Universal* method), 376
 get_nugget() (*gstools.field.generator.IncomprRandMeth* method), 342

- `get_nugget()` (*gstools.field.generator.RandMeth* method), 345
 - `get_scaling()` (*gstools.field.CondSRF* method), 335
 - `givens_rotation()` (in module *gstools.tools*), 400
 - gstools*
 - module, 113
 - gstools.covmodel*
 - module, 116
 - gstools.covmodel.plot*
 - module, 327
 - gstools.field*
 - module, 329
 - gstools.field.generator*
 - module, 341
 - gstools.field.upscaling*
 - module, 347
 - gstools.krige*
 - module, 352
 - gstools.normalizer*
 - module, 408
 - gstools.random*
 - module, 395
 - gstools.tools*
 - module, 398
 - gstools.transform*
 - module, 405
 - gstools.variogram*
 - module, 348
- ## H
- hankel_kw* (*gstools.covmodel.Circular* property), 221
 - hankel_kw* (*gstools.covmodel.CovModel* property), 124
 - hankel_kw* (*gstools.covmodel.Cubic* property), 197
 - hankel_kw* (*gstools.covmodel.Exponential* property), 149
 - hankel_kw* (*gstools.covmodel.Gaussian* property), 137
 - hankel_kw* (*gstools.covmodel.HyperSpherical* property), 245
 - hankel_kw* (*gstools.covmodel.JBessel* property), 270
 - hankel_kw* (*gstools.covmodel.Linear* property), 209
 - hankel_kw* (*gstools.covmodel.Matern* property), 162
 - hankel_kw* (*gstools.covmodel.Rational* property), 186
 - hankel_kw* (*gstools.covmodel.Spherical* property), 233
 - hankel_kw* (*gstools.covmodel.Stable* property), 173
 - hankel_kw* (*gstools.covmodel.SuperSpherical* property), 258
 - hankel_kw* (*gstools.covmodel.TPExponential* property), 296
 - hankel_kw* (*gstools.covmodel.TPLGaussian* property), 282
 - hankel_kw* (*gstools.covmodel.TPLSimple* property), 323
 - hankel_kw* (*gstools.covmodel.TPLStable* property), 310
 - has_cdf* (*gstools.covmodel.Circular* property), 221
 - has_cdf* (*gstools.covmodel.CovModel* property), 124
 - has_cdf* (*gstools.covmodel.Cubic* property), 197
 - has_cdf* (*gstools.covmodel.Exponential* property), 150
 - has_cdf* (*gstools.covmodel.Gaussian* property), 137
 - has_cdf* (*gstools.covmodel.HyperSpherical* property), 245
 - has_cdf* (*gstools.covmodel.JBessel* property), 270
 - has_cdf* (*gstools.covmodel.Linear* property), 209
 - has_cdf* (*gstools.covmodel.Matern* property), 162
 - has_cdf* (*gstools.covmodel.Rational* property), 186
 - has_cdf* (*gstools.covmodel.Spherical* property), 233
 - has_cdf* (*gstools.covmodel.Stable* property), 173
 - has_cdf* (*gstools.covmodel.SuperSpherical* property), 258
 - has_cdf* (*gstools.covmodel.TPExponential* property), 296
 - has_cdf* (*gstools.covmodel.TPLGaussian* property), 282
 - has_cdf* (*gstools.covmodel.TPLSimple* property), 323
 - has_cdf* (*gstools.covmodel.TPLStable* property), 310
 - has_const_mean* (*gstools.krige.Detrended* property), 393
 - has_const_mean* (*gstools.krige.ExtDrift* property), 386
 - has_const_mean* (*gstools.krige.Krige* property), 358
 - has_const_mean* (*gstools.krige.Ordinary* property), 372
 - has_const_mean* (*gstools.krige.Simple* property), 365
 - has_const_mean* (*gstools.krige.Universal* property), 379
 - has_ppf* (*gstools.covmodel.Circular* property), 221
 - has_ppf* (*gstools.covmodel.CovModel* property), 124
 - has_ppf* (*gstools.covmodel.Cubic* property), 197
 - has_ppf* (*gstools.covmodel.Exponential* property), 150
 - has_ppf* (*gstools.covmodel.Gaussian* property), 137
 - has_ppf* (*gstools.covmodel.HyperSpherical* property), 245
 - has_ppf* (*gstools.covmodel.JBessel* property), 270
 - has_ppf* (*gstools.covmodel.Linear* property), 209
 - has_ppf* (*gstools.covmodel.Matern* property), 162
 - has_ppf* (*gstools.covmodel.Rational* property), 186
 - has_ppf* (*gstools.covmodel.Spherical* property), 233
 - has_ppf* (*gstools.covmodel.Stable* property), 173
 - has_ppf* (*gstools.covmodel.SuperSpherical* property), 258
 - has_ppf* (*gstools.covmodel.TPExponential* property), 296
 - has_ppf* (*gstools.covmodel.TPLGaussian* property), 282
 - has_ppf* (*gstools.covmodel.TPLSimple* property), 323
 - has_ppf* (*gstools.covmodel.TPLStable* property), 310
 - HyperSpherical* (class in *gstools.covmodel*), 237
- ## I
- inc_beta()* (in module *gstools.tools*), 400
 - inc_gamma()* (in module *gstools.tools*), 400
 - IncomprRandMeth* (class in *gstools.field.generator*), 341
 - int_drift_no* (*gstools.krige.Detrended* property), 393

`int_drift_no` (`gstools.krige.ExtDrift` property), 386
`int_drift_no` (`gstools.krige.Krige` property), 358
`int_drift_no` (`gstools.krige.Ordinary` property), 372
`int_drift_no` (`gstools.krige.Simple` property), 365
`int_drift_no` (`gstools.krige.Universal` property), 379
`integral_scale` (`gstools.covmodel.Circular` property), 222
`integral_scale` (`gstools.covmodel.CovModel` property), 124
`integral_scale` (`gstools.covmodel.Cubic` property), 198
`integral_scale` (`gstools.covmodel.Exponential` property), 150
`integral_scale` (`gstools.covmodel.Gaussian` property), 137
`integral_scale` (`gstools.covmodel.HyperSpherical` property), 246
`integral_scale` (`gstools.covmodel.JBessel` property), 270
`integral_scale` (`gstools.covmodel.Linear` property), 210
`integral_scale` (`gstools.covmodel.Matern` property), 162
`integral_scale` (`gstools.covmodel.Rational` property), 186
`integral_scale` (`gstools.covmodel.Spherical` property), 234
`integral_scale` (`gstools.covmodel.Stable` property), 174
`integral_scale` (`gstools.covmodel.SuperSpherical` property), 258
`integral_scale` (`gstools.covmodel.TPExponential` property), 297
`integral_scale` (`gstools.covmodel.TPLGaussian` property), 283
`integral_scale` (`gstools.covmodel.TPLSimple` property), 323
`integral_scale` (`gstools.covmodel.TPLStable` property), 310
`integral_scale_vec` (`gstools.covmodel.Circular` property), 222
`integral_scale_vec` (`gstools.covmodel.CovModel` property), 124
`integral_scale_vec` (`gstools.covmodel.Cubic` property), 198
`integral_scale_vec` (`gstools.covmodel.Exponential` property), 150
`integral_scale_vec` (`gstools.covmodel.Gaussian` property), 137
`integral_scale_vec` (`gstools.covmodel.HyperSpherical` property), 246
`integral_scale_vec` (`gstools.covmodel.JBessel` property), 270
`integral_scale_vec` (`gstools.covmodel.Linear` property), 210
`integral_scale_vec` (`gstools.covmodel.Matern` property), 162
`integral_scale_vec` (`gstools.covmodel.Rational` property), 186
`integral_scale_vec` (`gstools.covmodel.Spherical` property), 234
`integral_scale_vec` (`gstools.covmodel.Stable` property), 174
`integral_scale_vec` (`gstools.covmodel.SuperSpherical` property), 258
`integral_scale_vec` (`gstools.covmodel.TPExponential` property), 297
`integral_scale_vec` (`gstools.covmodel.TPLGaussian` property), 283
`integral_scale_vec` (`gstools.covmodel.TPLSimple` property), 323
`integral_scale_vec` (`gstools.covmodel.TPLStable` property), 310
`is_isotropic` (`gstools.covmodel.Circular` property), 222
`is_isotropic` (`gstools.covmodel.CovModel` property), 125
`is_isotropic` (`gstools.covmodel.Cubic` property), 198
`is_isotropic` (`gstools.covmodel.Exponential` property), 150
`is_isotropic` (`gstools.covmodel.Gaussian` property), 137
`is_isotropic` (`gstools.covmodel.HyperSpherical` property), 246
`is_isotropic` (`gstools.covmodel.JBessel` property), 270
`is_isotropic` (`gstools.covmodel.Linear` property), 210
`is_isotropic` (`gstools.covmodel.Matern` property), 162
`is_isotropic` (`gstools.covmodel.Rational` property), 186
`is_isotropic` (`gstools.covmodel.Spherical` property), 234
`is_isotropic` (`gstools.covmodel.Stable` property), 174
`is_isotropic` (`gstools.covmodel.SuperSpherical` property), 258
`is_isotropic` (`gstools.covmodel.TPExponential` property), 297
`is_isotropic` (`gstools.covmodel.TPLGaussian` property), 283
`is_isotropic` (`gstools.covmodel.TPLSimple` property), 323
`is_isotropic` (`gstools.covmodel.TPLStable` property), 311
`iso_arg` (`gstools.covmodel.Circular` property), 222
`iso_arg` (`gstools.covmodel.CovModel` property), 125
`iso_arg` (`gstools.covmodel.Cubic` property), 198
`iso_arg` (`gstools.covmodel.Exponential` property), 150
`iso_arg` (`gstools.covmodel.Gaussian` property), 137

- `iso_arg` (`gstools.covmodel.HyperSpherical` property), 246
 - `iso_arg` (`gstools.covmodel.JBessel` property), 270
 - `iso_arg` (`gstools.covmodel.Linear` property), 210
 - `iso_arg` (`gstools.covmodel.Matern` property), 162
 - `iso_arg` (`gstools.covmodel.Rational` property), 186
 - `iso_arg` (`gstools.covmodel.Spherical` property), 234
 - `iso_arg` (`gstools.covmodel.Stable` property), 174
 - `iso_arg` (`gstools.covmodel.SuperSpherical` property), 258
 - `iso_arg` (`gstools.covmodel.TPLExponential` property), 297
 - `iso_arg` (`gstools.covmodel.TPLGaussian` property), 283
 - `iso_arg` (`gstools.covmodel.TPLSimple` property), 323
 - `iso_arg` (`gstools.covmodel.TPLStable` property), 311
 - `iso_arg_list` (`gstools.covmodel.Circular` property), 222
 - `iso_arg_list` (`gstools.covmodel.CovModel` property), 125
 - `iso_arg_list` (`gstools.covmodel.Cubic` property), 198
 - `iso_arg_list` (`gstools.covmodel.Exponential` property), 150
 - `iso_arg_list` (`gstools.covmodel.Gaussian` property), 138
 - `iso_arg_list` (`gstools.covmodel.HyperSpherical` property), 246
 - `iso_arg_list` (`gstools.covmodel.JBessel` property), 270
 - `iso_arg_list` (`gstools.covmodel.Linear` property), 210
 - `iso_arg_list` (`gstools.covmodel.Matern` property), 162
 - `iso_arg_list` (`gstools.covmodel.Rational` property), 186
 - `iso_arg_list` (`gstools.covmodel.Spherical` property), 234
 - `iso_arg_list` (`gstools.covmodel.Stable` property), 174
 - `iso_arg_list` (`gstools.covmodel.SuperSpherical` property), 258
 - `iso_arg_list` (`gstools.covmodel.TPLExponential` property), 297
 - `iso_arg_list` (`gstools.covmodel.TPLGaussian` property), 283
 - `iso_arg_list` (`gstools.covmodel.TPLSimple` property), 323
 - `iso_arg_list` (`gstools.covmodel.TPLStable` property), 311
 - `isometrize()` (`gstools.covmodel.Circular` method), 219
 - `isometrize()` (`gstools.covmodel.CovModel` method), 121
 - `isometrize()` (`gstools.covmodel.Cubic` method), 195
 - `isometrize()` (`gstools.covmodel.Exponential` method), 147
 - `isometrize()` (`gstools.covmodel.Gaussian` method), 134
 - `isometrize()` (`gstools.covmodel.HyperSpherical` method), 243
 - `isometrize()` (`gstools.covmodel.JBessel` method), 267
 - `isometrize()` (`gstools.covmodel.Linear` method), 207
 - `isometrize()` (`gstools.covmodel.Matern` method), 159
 - `isometrize()` (`gstools.covmodel.Rational` method), 183
 - `isometrize()` (`gstools.covmodel.Spherical` method), 231
 - `isometrize()` (`gstools.covmodel.Stable` method), 171
 - `isometrize()` (`gstools.covmodel.SuperSpherical` method), 255
 - `isometrize()` (`gstools.covmodel.TPLExponential` method), 294
 - `isometrize()` (`gstools.covmodel.TPLGaussian` method), 280
 - `isometrize()` (`gstools.covmodel.TPLSimple` method), 320
 - `isometrize()` (`gstools.covmodel.TPLStable` method), 307
- ## J
- `JBessel` (class in `gstools.covmodel`), 261
- ## K
- `kernel_loglikelihood()` (`gstools.normalizer.BoxCox` method), 415
 - `kernel_loglikelihood()` (`gstools.normalizer.BoxCoxShift` method), 418
 - `kernel_loglikelihood()` (`gstools.normalizer.LogNormal` method), 412
 - `kernel_loglikelihood()` (`gstools.normalizer.Manly` method), 427
 - `kernel_loglikelihood()` (`gstools.normalizer.Modulus` method), 424
 - `kernel_loglikelihood()` (`gstools.normalizer.Normalizer` method), 409
 - `kernel_loglikelihood()` (`gstools.normalizer.YeoJohnson` method), 421
 - `Krige` (class in `gstools.krige`), 352
 - `krige` (`gstools.field.CondSRF` property), 337
 - `krige_size` (`gstools.krige.Detrended` property), 393
 - `krige_size` (`gstools.krige.ExtDrift` property), 386
 - `krige_size` (`gstools.krige.Krige` property), 358
 - `krige_size` (`gstools.krige.Ordinary` property), 372
 - `krige_size` (`gstools.krige.Simple` property), 365
 - `krige_size` (`gstools.krige.Universal` property), 380
- ## L
- `latlon` (`gstools.covmodel.Circular` property), 222

`latlon` (`gstools.covmodel.CovModel` property), 125
`latlon` (`gstools.covmodel.Cubic` property), 198
`latlon` (`gstools.covmodel.Exponential` property), 150
`latlon` (`gstools.covmodel.Gaussian` property), 138
`latlon` (`gstools.covmodel.HyperSpherical` property), 246
`latlon` (`gstools.covmodel.JBessel` property), 270
`latlon` (`gstools.covmodel.Linear` property), 210
`latlon` (`gstools.covmodel.Matern` property), 162
`latlon` (`gstools.covmodel.Rational` property), 186
`latlon` (`gstools.covmodel.Spherical` property), 234
`latlon` (`gstools.covmodel.Stable` property), 174
`latlon` (`gstools.covmodel.SuperSpherical` property), 258
`latlon` (`gstools.covmodel.TPLExponential` property), 297
`latlon` (`gstools.covmodel.TPLGaussian` property), 283
`latlon` (`gstools.covmodel.TPLSimple` property), 323
`latlon` (`gstools.covmodel.TPLStable` property), 311
`len_low_rescaled` (`gstools.covmodel.TPLExponential` property), 297
`len_low_rescaled` (`gstools.covmodel.TPLGaussian` property), 283
`len_low_rescaled` (`gstools.covmodel.TPLStable` property), 311
`len_rescaled` (`gstools.covmodel.Circular` property), 222
`len_rescaled` (`gstools.covmodel.CovModel` property), 125
`len_rescaled` (`gstools.covmodel.Cubic` property), 198
`len_rescaled` (`gstools.covmodel.Exponential` property), 150
`len_rescaled` (`gstools.covmodel.Gaussian` property), 138
`len_rescaled` (`gstools.covmodel.HyperSpherical` property), 246
`len_rescaled` (`gstools.covmodel.JBessel` property), 270
`len_rescaled` (`gstools.covmodel.Linear` property), 210
`len_rescaled` (`gstools.covmodel.Matern` property), 162
`len_rescaled` (`gstools.covmodel.Rational` property), 186
`len_rescaled` (`gstools.covmodel.Spherical` property), 234
`len_rescaled` (`gstools.covmodel.Stable` property), 174
`len_rescaled` (`gstools.covmodel.SuperSpherical` property), 258
`len_rescaled` (`gstools.covmodel.TPLExponential` property), 297
`len_rescaled` (`gstools.covmodel.TPLGaussian` property), 283
`len_rescaled` (`gstools.covmodel.TPLSimple` property), 323
`len_rescaled` (`gstools.covmodel.TPLStable` property), 311
`len_scale` (`gstools.covmodel.Circular` property), 222
`len_scale` (`gstools.covmodel.CovModel` property), 125
`len_scale` (`gstools.covmodel.Cubic` property), 198
`len_scale` (`gstools.covmodel.Exponential` property), 150
`len_scale` (`gstools.covmodel.Gaussian` property), 138
`len_scale` (`gstools.covmodel.HyperSpherical` property), 246
`len_scale` (`gstools.covmodel.JBessel` property), 270
`len_scale` (`gstools.covmodel.Linear` property), 210
`len_scale` (`gstools.covmodel.Matern` property), 162
`len_scale` (`gstools.covmodel.Rational` property), 186
`len_scale` (`gstools.covmodel.Spherical` property), 234
`len_scale` (`gstools.covmodel.Stable` property), 174
`len_scale` (`gstools.covmodel.SuperSpherical` property), 258
`len_scale` (`gstools.covmodel.TPLExponential` property), 297
`len_scale` (`gstools.covmodel.TPLGaussian` property), 283
`len_scale` (`gstools.covmodel.TPLSimple` property), 324
`len_scale` (`gstools.covmodel.TPLStable` property), 311
`len_scale_bounds` (`gstools.covmodel.Circular` property), 222
`len_scale_bounds` (`gstools.covmodel.CovModel` property), 125
`len_scale_bounds` (`gstools.covmodel.Cubic` property), 198
`len_scale_bounds` (`gstools.covmodel.Exponential` property), 150
`len_scale_bounds` (`gstools.covmodel.Gaussian` property), 138
`len_scale_bounds` (`gstools.covmodel.HyperSpherical` property), 246
`len_scale_bounds` (`gstools.covmodel.JBessel` property), 270
`len_scale_bounds` (`gstools.covmodel.Linear` property), 210
`len_scale_bounds` (`gstools.covmodel.Matern` property), 162
`len_scale_bounds` (`gstools.covmodel.Rational` property), 186
`len_scale_bounds` (`gstools.covmodel.Spherical` property), 234
`len_scale_bounds` (`gstools.covmodel.Stable` property), 174
`len_scale_bounds` (`gstools.covmodel.SuperSpherical` property), 258
`len_scale_bounds` (`gstools.covmodel.TPLExponential` property), 297
`len_scale_bounds` (`gstools.covmodel.TPLGaussian` property), 283
`len_scale_bounds` (`gstools.covmodel.TPLSimple`

- property*), 324
- `len_scale_bounds` (*gstools.covmodel.TPLStable property*), 311
- `len_scale_vec` (*gstools.covmodel.Circular property*), 223
- `len_scale_vec` (*gstools.covmodel.CovModel property*), 125
- `len_scale_vec` (*gstools.covmodel.Cubic property*), 199
- `len_scale_vec` (*gstools.covmodel.Exponential property*), 151
- `len_scale_vec` (*gstools.covmodel.Gaussian property*), 138
- `len_scale_vec` (*gstools.covmodel.HyperSpherical property*), 247
- `len_scale_vec` (*gstools.covmodel.JBessel property*), 271
- `len_scale_vec` (*gstools.covmodel.Linear property*), 211
- `len_scale_vec` (*gstools.covmodel.Matern property*), 163
- `len_scale_vec` (*gstools.covmodel.Rational property*), 187
- `len_scale_vec` (*gstools.covmodel.Spherical property*), 235
- `len_scale_vec` (*gstools.covmodel.Stable property*), 175
- `len_scale_vec` (*gstools.covmodel.SuperSpherical property*), 259
- `len_scale_vec` (*gstools.covmodel.TPLExponential property*), 298
- `len_scale_vec` (*gstools.covmodel.TPLGaussian property*), 284
- `len_scale_vec` (*gstools.covmodel.TPLSimple property*), 324
- `len_scale_vec` (*gstools.covmodel.TPLStable property*), 311
- `len_up` (*gstools.covmodel.TPLExponential property*), 298
- `len_up` (*gstools.covmodel.TPLGaussian property*), 284
- `len_up` (*gstools.covmodel.TPLStable property*), 312
- `len_up_rescaled` (*gstools.covmodel.TPLExponential property*), 298
- `len_up_rescaled` (*gstools.covmodel.TPLGaussian property*), 284
- `len_up_rescaled` (*gstools.covmodel.TPLStable property*), 312
- `likelihood()` (*gstools.normalizer.BoxCox method*), 415
- `likelihood()` (*gstools.normalizer.BoxCoxShift method*), 418
- `likelihood()` (*gstools.normalizer.LogNormal method*), 412
- `likelihood()` (*gstools.normalizer.Manly method*), 427
- `likelihood()` (*gstools.normalizer.Modulus method*), 424
- `likelihood()` (*gstools.normalizer.Normalizer method*), 409
- `likelihood()` (*gstools.normalizer.YeoJohnson method*), 421
- `Linear` (*class in gstools.covmodel*), 201
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Circular method*), 219
- `ln_spectral_rad_pdf()` (*gstools.covmodel.CovModel method*), 122
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Cubic method*), 195
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Exponential method*), 147
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Gaussian method*), 134
- `ln_spectral_rad_pdf()` (*gstools.covmodel.HyperSpherical method*), 243
- `ln_spectral_rad_pdf()` (*gstools.covmodel.JBessel method*), 267
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Linear method*), 207
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Matern method*), 159
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Rational method*), 183
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Spherical method*), 231
- `ln_spectral_rad_pdf()` (*gstools.covmodel.Stable method*), 171
- `ln_spectral_rad_pdf()` (*gstools.covmodel.SuperSpherical method*), 255
- `ln_spectral_rad_pdf()` (*gstools.covmodel.TPLExponential method*), 294
- `ln_spectral_rad_pdf()` (*gstools.covmodel.TPLGaussian method*), 280
- `ln_spectral_rad_pdf()` (*gstools.covmodel.TPLSimple method*), 320
- `ln_spectral_rad_pdf()` (*gstools.covmodel.TPLStable method*), 308
- `loglikelihood()` (*gstools.normalizer.BoxCox method*), 415
- `loglikelihood()` (*gstools.normalizer.BoxCoxShift method*), 418
- `loglikelihood()` (*gstools.normalizer.LogNormal method*), 412
- `loglikelihood()` (*gstools.normalizer.Manly method*), 427
- `loglikelihood()` (*gstools.normalizer.Modulus method*), 424
- `loglikelihood()` (*gstools.normalizer.Normalizer method*), 409

method), 409
loglikelihood() (*gstools.normalizer.YeoJohnson method*), 421
LogNormal (*class in gstools.normalizer*), 411

M

main_axes() (*gstools.covmodel.Circular method*), 219
main_axes() (*gstools.covmodel.CovModel method*), 122
main_axes() (*gstools.covmodel.Cubic method*), 195
main_axes() (*gstools.covmodel.Exponential method*), 147
main_axes() (*gstools.covmodel.Gaussian method*), 134
main_axes() (*gstools.covmodel.HyperSpherical method*), 243
main_axes() (*gstools.covmodel.JBessel method*), 267
main_axes() (*gstools.covmodel.Linear method*), 207
main_axes() (*gstools.covmodel.Matern method*), 159
main_axes() (*gstools.covmodel.Rational method*), 183
main_axes() (*gstools.covmodel.Spherical method*), 231
main_axes() (*gstools.covmodel.Stable method*), 171
main_axes() (*gstools.covmodel.SuperSpherical method*), 255
main_axes() (*gstools.covmodel.TPLExponential method*), 294
main_axes() (*gstools.covmodel.TPLGaussian method*), 280
main_axes() (*gstools.covmodel.TPLSimple method*), 320
main_axes() (*gstools.covmodel.TPLStable method*), 308
Manly (*class in gstools.normalizer*), 426
MasterRNG (*class in gstools.random*), 395
Matern (*class in gstools.covmodel*), 153
matrix_anisometrize() (*in module gstools.tools*), 400
matrix_anisotropify() (*in module gstools.tools*), 400
matrix_derotate() (*in module gstools.tools*), 401
matrix_isometrize() (*in module gstools.tools*), 401
matrix_isotropify() (*in module gstools.tools*), 401
matrix_rotate() (*in module gstools.tools*), 401
mean(*gstools.field.CondSRF property*), 337
mean(*gstools.field.Field property*), 340
mean(*gstools.field.SRF property*), 332
mean(*gstools.krige.Detrended property*), 393
mean(*gstools.krige.ExtDrift property*), 387
mean(*gstools.krige.Krige property*), 358
mean(*gstools.krige.Ordinary property*), 372
mean(*gstools.krige.Simple property*), 365
mean(*gstools.krige.Universal property*), 380
mesh() (*gstools.field.CondSRF method*), 335
mesh() (*gstools.field.Field method*), 338
mesh() (*gstools.field.SRF method*), 330

mesh() (*gstools.krige.Detrended method*), 390
mesh() (*gstools.krige.ExtDrift method*), 383
mesh() (*gstools.krige.Krige method*), 355
mesh() (*gstools.krige.Ordinary method*), 369
mesh() (*gstools.krige.Simple method*), 362
mesh() (*gstools.krige.Universal method*), 376
mode_no (*gstools.field.generator.IncomprRandMeth property*), 343
mode_no (*gstools.field.generator.RandMeth property*), 345
model (*gstools.field.CondSRF property*), 337
model (*gstools.field.Field property*), 340
model (*gstools.field.generator.IncomprRandMeth property*), 343
model (*gstools.field.generator.RandMeth property*), 345
model (*gstools.field.SRF property*), 332
model (*gstools.krige.Detrended property*), 393
model (*gstools.krige.ExtDrift property*), 387
model (*gstools.krige.Krige property*), 358
model (*gstools.krige.Ordinary property*), 373
model (*gstools.krige.Simple property*), 366
model (*gstools.krige.Universal property*), 380
module
 gstools, 113
 gstools.covmodel, 116
 gstools.covmodel.plot, 327
 gstools.field, 329
 gstools.field.generator, 341
 gstools.field.upscaling, 347
 gstools.krige, 352
 gstools.normalizer, 408
 gstools.random, 395
 gstools.tools, 398
 gstools.transform, 405
 gstools.variogram, 348
Modulus (*class in gstools.normalizer*), 423

N

name (*gstools.covmodel.Circular property*), 223
name (*gstools.covmodel.CovModel property*), 126
name (*gstools.covmodel.Cubic property*), 199
name (*gstools.covmodel.Exponential property*), 151
name (*gstools.covmodel.Gaussian property*), 138
name (*gstools.covmodel.HyperSpherical property*), 247
name (*gstools.covmodel.JBessel property*), 271
name (*gstools.covmodel.Linear property*), 211
name (*gstools.covmodel.Matern property*), 163
name (*gstools.covmodel.Rational property*), 187
name (*gstools.covmodel.Spherical property*), 235
name (*gstools.covmodel.Stable property*), 175
name (*gstools.covmodel.SuperSpherical property*), 259
name (*gstools.covmodel.TPLExponential property*), 298
name (*gstools.covmodel.TPLGaussian property*), 284
name (*gstools.covmodel.TPLSimple property*), 324
name (*gstools.covmodel.TPLStable property*), 312
name (*gstools.field.CondSRF property*), 337

- `name` (*gstools.field.Field* property), 340
- `name` (*gstools.field.generator.IncomprRandMeth* property), 343
- `name` (*gstools.field.generator.RandMeth* property), 345
- `name` (*gstools.field.SRF* property), 333
- `name` (*gstools.krige.Detrended* property), 394
- `name` (*gstools.krige.ExtDrift* property), 387
- `name` (*gstools.krige.Krige* property), 358
- `name` (*gstools.krige.Ordinary* property), 373
- `name` (*gstools.krige.Simple* property), 366
- `name` (*gstools.krige.Universal* property), 380
- `name` (*gstools.normalizer.BoxCox* property), 416
- `name` (*gstools.normalizer.BoxCoxShift* property), 419
- `name` (*gstools.normalizer.LogNormal* property), 412
- `name` (*gstools.normalizer.Manly* property), 428
- `name` (*gstools.normalizer.Modulus* property), 424
- `name` (*gstools.normalizer.Normalizer* property), 409
- `name` (*gstools.normalizer.YeoJohnson* property), 421
- `no_of_angles()` (in module *gstools.tools*), 401
- `normal_force_moments()` (in module *gstools.transform*), 406
- `normal_to_arcsin()` (in module *gstools.transform*), 406
- `normal_to_lognormal()` (in module *gstools.transform*), 406
- `normal_to_uniform()` (in module *gstools.transform*), 406
- `normal_to_uquad()` (in module *gstools.transform*), 406
- `normalize()` (*gstools.normalizer.BoxCox* method), 415
- `normalize()` (*gstools.normalizer.BoxCoxShift* method), 418
- `normalize()` (*gstools.normalizer.LogNormal* method), 412
- `normalize()` (*gstools.normalizer.Manly* method), 427
- `normalize()` (*gstools.normalizer.Modulus* method), 424
- `normalize()` (*gstools.normalizer.Normalizer* method), 409
- `normalize()` (*gstools.normalizer.YeoJohnson* method), 421
- `normalize_range` (*gstools.normalizer.BoxCox* attribute), 416
- `normalize_range` (*gstools.normalizer.BoxCoxShift* property), 419
- `normalize_range` (*gstools.normalizer.LogNormal* attribute), 413
- `normalize_range` (*gstools.normalizer.Manly* attribute), 428
- `normalize_range` (*gstools.normalizer.Modulus* attribute), 425
- `normalize_range` (*gstools.normalizer.Normalizer* attribute), 410
- `normalize_range` (*gstools.normalizer.YeoJohnson* attribute), 422
- `Normalizer` (class in *gstools.normalizer*), 408
- `normalizer` (*gstools.field.CondSRF* property), 337
- `normalizer` (*gstools.field.Field* property), 340
- `normalizer` (*gstools.field.SRF* property), 333
- `normalizer` (*gstools.krige.Detrended* property), 394
- `normalizer` (*gstools.krige.ExtDrift* property), 387
- `normalizer` (*gstools.krige.Krige* property), 358
- `normalizer` (*gstools.krige.Ordinary* property), 373
- `normalizer` (*gstools.krige.Simple* property), 366
- `normalizer` (*gstools.krige.Universal* property), 380
- `nugget` (*gstools.covmodel.Circular* property), 223
- `nugget` (*gstools.covmodel.CovModel* property), 126
- `nugget` (*gstools.covmodel.Cubic* property), 199
- `nugget` (*gstools.covmodel.Exponential* property), 151
- `nugget` (*gstools.covmodel.Gaussian* property), 138
- `nugget` (*gstools.covmodel.HyperSpherical* property), 247
- `nugget` (*gstools.covmodel.JBessel* property), 271
- `nugget` (*gstools.covmodel.Linear* property), 211
- `nugget` (*gstools.covmodel.Matern* property), 163
- `nugget` (*gstools.covmodel.Rational* property), 187
- `nugget` (*gstools.covmodel.Spherical* property), 235
- `nugget` (*gstools.covmodel.Stable* property), 175
- `nugget` (*gstools.covmodel.SuperSpherical* property), 259
- `nugget` (*gstools.covmodel.TPLExponential* property), 298
- `nugget` (*gstools.covmodel.TPLGaussian* property), 284
- `nugget` (*gstools.covmodel.TPLSimple* property), 324
- `nugget` (*gstools.covmodel.TPLStable* property), 312
- `nugget_bounds` (*gstools.covmodel.Circular* property), 223
- `nugget_bounds` (*gstools.covmodel.CovModel* property), 126
- `nugget_bounds` (*gstools.covmodel.Cubic* property), 199
- `nugget_bounds` (*gstools.covmodel.Exponential* property), 151
- `nugget_bounds` (*gstools.covmodel.Gaussian* property), 138
- `nugget_bounds` (*gstools.covmodel.HyperSpherical* property), 247
- `nugget_bounds` (*gstools.covmodel.JBessel* property), 271
- `nugget_bounds` (*gstools.covmodel.Linear* property), 211
- `nugget_bounds` (*gstools.covmodel.Matern* property), 163
- `nugget_bounds` (*gstools.covmodel.Rational* property), 187
- `nugget_bounds` (*gstools.covmodel.Spherical* property), 235
- `nugget_bounds` (*gstools.covmodel.Stable* property), 175
- `nugget_bounds` (*gstools.covmodel.SuperSpherical* property), 259
- `nugget_bounds` (*gstools.covmodel.TPLExponential* property), 298
- `nugget_bounds` (*gstools.covmodel.TPLGaussian* property), 284

property), 284
nugget_bounds (*gstools.covmodel.TPLSimple* property), 324
nugget_bounds (*gstools.covmodel.TPLStable* property), 312

O

opt_arg (*gstools.covmodel.Circular* property), 223
opt_arg (*gstools.covmodel.CovModel* property), 126
opt_arg (*gstools.covmodel.Cubic* property), 199
opt_arg (*gstools.covmodel.Exponential* property), 151
opt_arg (*gstools.covmodel.Gaussian* property), 139
opt_arg (*gstools.covmodel.HyperSpherical* property), 247
opt_arg (*gstools.covmodel.JBessel* property), 271
opt_arg (*gstools.covmodel.Linear* property), 211
opt_arg (*gstools.covmodel.Matern* property), 163
opt_arg (*gstools.covmodel.Rational* property), 187
opt_arg (*gstools.covmodel.Spherical* property), 235
opt_arg (*gstools.covmodel.Stable* property), 175
opt_arg (*gstools.covmodel.SuperSpherical* property), 259
opt_arg (*gstools.covmodel.TPLExponential* property), 298
opt_arg (*gstools.covmodel.TPLGaussian* property), 284
opt_arg (*gstools.covmodel.TPLSimple* property), 324
opt_arg (*gstools.covmodel.TPLStable* property), 312
opt_arg_bounds (*gstools.covmodel.Circular* property), 223
opt_arg_bounds (*gstools.covmodel.CovModel* property), 126
opt_arg_bounds (*gstools.covmodel.Cubic* property), 199
opt_arg_bounds (*gstools.covmodel.Exponential* property), 151
opt_arg_bounds (*gstools.covmodel.Gaussian* property), 139
opt_arg_bounds (*gstools.covmodel.HyperSpherical* property), 247
opt_arg_bounds (*gstools.covmodel.JBessel* property), 271
opt_arg_bounds (*gstools.covmodel.Linear* property), 211
opt_arg_bounds (*gstools.covmodel.Matern* property), 163
opt_arg_bounds (*gstools.covmodel.Rational* property), 187
opt_arg_bounds (*gstools.covmodel.Spherical* property), 235
opt_arg_bounds (*gstools.covmodel.Stable* property), 175
opt_arg_bounds (*gstools.covmodel.SuperSpherical* property), 259
opt_arg_bounds (*gstools.covmodel.TPLExponential* property), 299
opt_arg_bounds (*gstools.covmodel.TPLGaussian* property), 285

opt_arg_bounds (*gstools.covmodel.TPLSimple* property), 324
opt_arg_bounds (*gstools.covmodel.TPLStable* property), 312
Ordinary (class in *gstools.krige*), 367

P

percentile_scale() (*gstools.covmodel.Circular* method), 219
percentile_scale() (*gstools.covmodel.CovModel* method), 122
percentile_scale() (*gstools.covmodel.Cubic* method), 195
percentile_scale() (*gstools.covmodel.Exponential* method), 147
percentile_scale() (*gstools.covmodel.Gaussian* method), 134
percentile_scale() (*gstools.covmodel.HyperSpherical* method), 243
percentile_scale() (*gstools.covmodel.JBessel* method), 267
percentile_scale() (*gstools.covmodel.Linear* method), 207
percentile_scale() (*gstools.covmodel.Matern* method), 159
percentile_scale() (*gstools.covmodel.Rational* method), 183
percentile_scale() (*gstools.covmodel.Spherical* method), 231
percentile_scale() (*gstools.covmodel.Stable* method), 171
percentile_scale() (*gstools.covmodel.SuperSpherical* method), 255
percentile_scale() (*gstools.covmodel.TPLExponential* method), 294
percentile_scale() (*gstools.covmodel.TPLGaussian* method), 280
percentile_scale() (*gstools.covmodel.TPLSimple* method), 320
percentile_scale() (*gstools.covmodel.TPLStable* method), 308
plot() (*gstools.covmodel.Circular* method), 219
plot() (*gstools.covmodel.CovModel* method), 122
plot() (*gstools.covmodel.Cubic* method), 195
plot() (*gstools.covmodel.Exponential* method), 147
plot() (*gstools.covmodel.Gaussian* method), 134
plot() (*gstools.covmodel.HyperSpherical* method), 243
plot() (*gstools.covmodel.JBessel* method), 267
plot() (*gstools.covmodel.Linear* method), 207
plot() (*gstools.covmodel.Matern* method), 159
plot() (*gstools.covmodel.Rational* method), 183
plot() (*gstools.covmodel.Spherical* method), 231
plot() (*gstools.covmodel.Stable* method), 171

- `plot()` (*gstools.covmodel.SuperSpherical method*), 255
- `plot()` (*gstools.covmodel.TPLExponential method*), 294
- `plot()` (*gstools.covmodel.TPLGaussian method*), 280
- `plot()` (*gstools.covmodel.TPLSimple method*), 320
- `plot()` (*gstools.covmodel.TPLStable method*), 308
- `plot()` (*gstools.field.CondSRF method*), 335
- `plot()` (*gstools.field.Field method*), 339
- `plot()` (*gstools.field.SRF method*), 331
- `plot()` (*gstools.krige.Detrended method*), 391
- `plot()` (*gstools.krige.ExtDrift method*), 384
- `plot()` (*gstools.krige.Krige method*), 355
- `plot()` (*gstools.krige.Ordinary method*), 370
- `plot()` (*gstools.krige.Simple method*), 363
- `plot()` (*gstools.krige.Universal method*), 377
- `plot_cor_axis()` (*in module gstools.covmodel.plot*), 327
- `plot_cor_spatial()` (*in module gstools.covmodel.plot*), 327
- `plot_cor_yadrenko()` (*in module gstools.covmodel.plot*), 327
- `plot_correlation()` (*in module gstools.covmodel.plot*), 327
- `plot_cov_axis()` (*in module gstools.covmodel.plot*), 327
- `plot_cov_spatial()` (*in module gstools.covmodel.plot*), 327
- `plot_cov_yadrenko()` (*in module gstools.covmodel.plot*), 327
- `plot_covariance()` (*in module gstools.covmodel.plot*), 327
- `plot_spectral_density()` (*in module gstools.covmodel.plot*), 327
- `plot_spectral_rad_pdf()` (*in module gstools.covmodel.plot*), 328
- `plot_spectrum()` (*in module gstools.covmodel.plot*), 328
- `plot_vario_axis()` (*in module gstools.covmodel.plot*), 328
- `plot_vario_spatial()` (*in module gstools.covmodel.plot*), 328
- `plot_vario_yadrenko()` (*in module gstools.covmodel.plot*), 328
- `plot_variogram()` (*in module gstools.covmodel.plot*), 328
- `post_field()` (*gstools.field.CondSRF method*), 335
- `post_field()` (*gstools.field.Field method*), 339
- `post_field()` (*gstools.field.SRF method*), 331
- `post_field()` (*gstools.krige.Detrended method*), 391
- `post_field()` (*gstools.krige.ExtDrift method*), 384
- `post_field()` (*gstools.krige.Krige method*), 356
- `post_field()` (*gstools.krige.Ordinary method*), 370
- `post_field()` (*gstools.krige.Simple method*), 363
- `post_field()` (*gstools.krige.Universal method*), 377
- `pre_pos()` (*gstools.field.CondSRF method*), 336
- `pre_pos()` (*gstools.field.Field method*), 339
- `pre_pos()` (*gstools.field.SRF method*), 331
- `pre_pos()` (*gstools.krige.Detrended method*), 391
- `pre_pos()` (*gstools.krige.ExtDrift method*), 384
- `pre_pos()` (*gstools.krige.Krige method*), 356
- `pre_pos()` (*gstools.krige.Ordinary method*), 370
- `pre_pos()` (*gstools.krige.Simple method*), 363
- `pre_pos()` (*gstools.krige.Universal method*), 377
- `pseudo_inv` (*gstools.krige.Detrended property*), 394
- `pseudo_inv` (*gstools.krige.ExtDrift property*), 387
- `pseudo_inv` (*gstools.krige.Krige property*), 358
- `pseudo_inv` (*gstools.krige.Ordinary property*), 373
- `pseudo_inv` (*gstools.krige.Simple property*), 366
- `pseudo_inv` (*gstools.krige.Universal property*), 380
- `pseudo_inv_type` (*gstools.krige.Detrended property*), 394
- `pseudo_inv_type` (*gstools.krige.ExtDrift property*), 387
- `pseudo_inv_type` (*gstools.krige.Krige property*), 358
- `pseudo_inv_type` (*gstools.krige.Ordinary property*), 373
- `pseudo_inv_type` (*gstools.krige.Simple property*), 366
- `pseudo_inv_type` (*gstools.krige.Universal property*), 380
- `pykrige_angle` (*gstools.covmodel.Circular property*), 223
- `pykrige_angle` (*gstools.covmodel.CovModel property*), 126
- `pykrige_angle` (*gstools.covmodel.Cubic property*), 199
- `pykrige_angle` (*gstools.covmodel.Exponential property*), 151
- `pykrige_angle` (*gstools.covmodel.Gaussian property*), 139
- `pykrige_angle` (*gstools.covmodel.HyperSpherical property*), 247
- `pykrige_angle` (*gstools.covmodel.JBessel property*), 272
- `pykrige_angle` (*gstools.covmodel.Linear property*), 211
- `pykrige_angle` (*gstools.covmodel.Matern property*), 164
- `pykrige_angle` (*gstools.covmodel.Rational property*), 188
- `pykrige_angle` (*gstools.covmodel.Spherical property*), 235
- `pykrige_angle` (*gstools.covmodel.Stable property*), 175
- `pykrige_angle` (*gstools.covmodel.SuperSpherical property*), 260
- `pykrige_angle` (*gstools.covmodel.TPLExponential property*), 299
- `pykrige_angle` (*gstools.covmodel.TPLGaussian property*), 285
- `pykrige_angle` (*gstools.covmodel.TPLSimple property*), 325
- `pykrige_angle` (*gstools.covmodel.TPLStable property*), 313
- `pykrige_angle_x` (*gstools.covmodel.Circular prop-*

erty), 223
`pykrige_angle_x` (`gstools.covmodel.CovModel` *property*), 126
`pykrige_angle_x` (`gstools.covmodel.Cubic` *property*), 199
`pykrige_angle_x` (`gstools.covmodel.Exponential` *property*), 151
`pykrige_angle_x` (`gstools.covmodel.Gaussian` *property*), 139
`pykrige_angle_x` (`gstools.covmodel.HyperSpherical` *property*), 247
`pykrige_angle_x` (`gstools.covmodel.JBessel` *property*), 272
`pykrige_angle_x` (`gstools.covmodel.Linear` *property*), 211
`pykrige_angle_x` (`gstools.covmodel.Matern` *property*), 164
`pykrige_angle_x` (`gstools.covmodel.Rational` *property*), 188
`pykrige_angle_x` (`gstools.covmodel.Spherical` *property*), 235
`pykrige_angle_x` (`gstools.covmodel.Stable` *property*), 175
`pykrige_angle_x` (`gstools.covmodel.SuperSpherical` *property*), 260
`pykrige_angle_x` (`gstools.covmodel.TPExponential` *property*), 299
`pykrige_angle_x` (`gstools.covmodel.TPLGaussian` *property*), 285
`pykrige_angle_x` (`gstools.covmodel.TPLSimple` *property*), 325
`pykrige_angle_x` (`gstools.covmodel.TPLStable` *property*), 313
`pykrige_angle_y` (`gstools.covmodel.Circular` *property*), 223
`pykrige_angle_y` (`gstools.covmodel.CovModel` *property*), 126
`pykrige_angle_y` (`gstools.covmodel.Cubic` *property*), 199
`pykrige_angle_y` (`gstools.covmodel.Exponential` *property*), 152
`pykrige_angle_y` (`gstools.covmodel.Gaussian` *property*), 139
`pykrige_angle_y` (`gstools.covmodel.HyperSpherical` *property*), 247
`pykrige_angle_y` (`gstools.covmodel.JBessel` *property*), 272
`pykrige_angle_y` (`gstools.covmodel.Linear` *property*), 211
`pykrige_angle_y` (`gstools.covmodel.Matern` *property*), 164
`pykrige_angle_y` (`gstools.covmodel.Rational` *property*), 188
`pykrige_angle_y` (`gstools.covmodel.Spherical` *property*), 235
`pykrige_angle_y` (`gstools.covmodel.Stable` *property*), 175
`pykrige_angle_y` (`gstools.covmodel.SuperSpherical` *property*), 260
`pykrige_angle_y` (`gstools.covmodel.TPExponential` *property*), 299
`pykrige_angle_y` (`gstools.covmodel.TPLGaussian` *property*), 285
`pykrige_angle_y` (`gstools.covmodel.TPLSimple` *property*), 325
`pykrige_angle_y` (`gstools.covmodel.TPLStable` *property*), 313
`pykrige_angle_z` (`gstools.covmodel.Circular` *property*), 223
`pykrige_angle_z` (`gstools.covmodel.CovModel` *property*), 126
`pykrige_angle_z` (`gstools.covmodel.Cubic` *property*), 199
`pykrige_angle_z` (`gstools.covmodel.Exponential` *property*), 152
`pykrige_angle_z` (`gstools.covmodel.Gaussian` *property*), 139
`pykrige_angle_z` (`gstools.covmodel.HyperSpherical` *property*), 247
`pykrige_angle_z` (`gstools.covmodel.JBessel` *property*), 272
`pykrige_angle_z` (`gstools.covmodel.Linear` *property*), 211
`pykrige_angle_z` (`gstools.covmodel.Matern` *property*), 164
`pykrige_angle_z` (`gstools.covmodel.Rational` *property*), 188
`pykrige_angle_z` (`gstools.covmodel.Spherical` *property*), 235
`pykrige_angle_z` (`gstools.covmodel.Stable` *property*), 175
`pykrige_angle_z` (`gstools.covmodel.SuperSpherical` *property*), 260
`pykrige_angle_z` (`gstools.covmodel.TPExponential` *property*), 299
`pykrige_angle_z` (`gstools.covmodel.TPLGaussian` *property*), 285
`pykrige_angle_z` (`gstools.covmodel.TPLSimple` *property*), 325
`pykrige_angle_z` (`gstools.covmodel.TPLStable` *property*), 313
`pykrige_anis` (`gstools.covmodel.Circular` *property*), 223
`pykrige_anis` (`gstools.covmodel.CovModel` *property*), 126
`pykrige_anis` (`gstools.covmodel.Cubic` *property*), 199
`pykrige_anis` (`gstools.covmodel.Exponential` *property*), 152
`pykrige_anis` (`gstools.covmodel.Gaussian` *property*), 139
`pykrige_anis` (`gstools.covmodel.HyperSpherical` *property*), 247
`pykrige_anis` (`gstools.covmodel.JBessel` *property*), 272
`pykrige_anis` (`gstools.covmodel.Linear` *property*),

- 211
 pykrige_anis (*gstools.covmodel.Matern* property), 164
 pykrige_anis (*gstools.covmodel.Rational* property), 188
 pykrige_anis (*gstools.covmodel.Spherical* property), 235
 pykrige_anis (*gstools.covmodel.Stable* property), 175
 pykrige_anis (*gstools.covmodel.SuperSpherical* property), 260
 pykrige_anis (*gstools.covmodel.TPLExponential* property), 299
 pykrige_anis (*gstools.covmodel.TPLGaussian* property), 285
 pykrige_anis (*gstools.covmodel.TPLSimple* property), 325
 pykrige_anis (*gstools.covmodel.TPLStable* property), 313
 pykrige_anis_y (*gstools.covmodel.Circular* property), 224
 pykrige_anis_y (*gstools.covmodel.CovModel* property), 126
 pykrige_anis_y (*gstools.covmodel.Cubic* property), 200
 pykrige_anis_y (*gstools.covmodel.Exponential* property), 152
 pykrige_anis_y (*gstools.covmodel.Gaussian* property), 139
 pykrige_anis_y (*gstools.covmodel.HyperSpherical* property), 248
 pykrige_anis_y (*gstools.covmodel.JBessel* property), 272
 pykrige_anis_y (*gstools.covmodel.Linear* property), 212
 pykrige_anis_y (*gstools.covmodel.Matern* property), 164
 pykrige_anis_y (*gstools.covmodel.Rational* property), 188
 pykrige_anis_y (*gstools.covmodel.Spherical* property), 236
 pykrige_anis_y (*gstools.covmodel.Stable* property), 176
 pykrige_anis_y (*gstools.covmodel.SuperSpherical* property), 260
 pykrige_anis_y (*gstools.covmodel.TPLExponential* property), 299
 pykrige_anis_y (*gstools.covmodel.TPLGaussian* property), 285
 pykrige_anis_y (*gstools.covmodel.TPLSimple* property), 325
 pykrige_anis_y (*gstools.covmodel.TPLStable* property), 313
 pykrige_anis_z (*gstools.covmodel.Circular* property), 224
 pykrige_anis_z (*gstools.covmodel.CovModel* property), 126
 pykrige_anis_z (*gstools.covmodel.Cubic* property), 200
 200
 pykrige_anis_z (*gstools.covmodel.Exponential* property), 152
 pykrige_anis_z (*gstools.covmodel.Gaussian* property), 139
 pykrige_anis_z (*gstools.covmodel.HyperSpherical* property), 248
 pykrige_anis_z (*gstools.covmodel.JBessel* property), 272
 pykrige_anis_z (*gstools.covmodel.Linear* property), 212
 pykrige_anis_z (*gstools.covmodel.Matern* property), 164
 pykrige_anis_z (*gstools.covmodel.Rational* property), 188
 pykrige_anis_z (*gstools.covmodel.Spherical* property), 236
 pykrige_anis_z (*gstools.covmodel.Stable* property), 176
 pykrige_anis_z (*gstools.covmodel.SuperSpherical* property), 260
 pykrige_anis_z (*gstools.covmodel.TPLExponential* property), 299
 pykrige_anis_z (*gstools.covmodel.TPLGaussian* property), 285
 pykrige_anis_z (*gstools.covmodel.TPLSimple* property), 325
 pykrige_anis_z (*gstools.covmodel.TPLStable* property), 313
 pykrige_kwargs (*gstools.covmodel.Circular* property), 224
 pykrige_kwargs (*gstools.covmodel.CovModel* property), 126
 pykrige_kwargs (*gstools.covmodel.Cubic* property), 200
 pykrige_kwargs (*gstools.covmodel.Exponential* property), 152
 pykrige_kwargs (*gstools.covmodel.Gaussian* property), 139
 pykrige_kwargs (*gstools.covmodel.HyperSpherical* property), 248
 pykrige_kwargs (*gstools.covmodel.JBessel* property), 272
 pykrige_kwargs (*gstools.covmodel.Linear* property), 212
 pykrige_kwargs (*gstools.covmodel.Matern* property), 164
 pykrige_kwargs (*gstools.covmodel.Rational* property), 188
 pykrige_kwargs (*gstools.covmodel.Spherical* property), 236
 pykrige_kwargs (*gstools.covmodel.Stable* property), 176
 pykrige_kwargs (*gstools.covmodel.SuperSpherical* property), 260
 pykrige_kwargs (*gstools.covmodel.TPLExponential* property), 299
 pykrige_kwargs (*gstools.covmodel.TPLGaussian* property), 285

property), 285
 pykrige_kwarg (gstools.covmodel.TPLSimple property), 325
 pykrige_kwarg (gstools.covmodel.TPLStable property), 313
 pykrige_vario() (gstools.covmodel.Circular method), 219
 pykrige_vario() (gstools.covmodel.CovModel method), 122
 pykrige_vario() (gstools.covmodel.Cubic method), 195
 pykrige_vario() (gstools.covmodel.Exponential method), 147
 pykrige_vario() (gstools.covmodel.Gaussian method), 135
 pykrige_vario() (gstools.covmodel.HyperSpherical method), 243
 pykrige_vario() (gstools.covmodel.JBessel method), 268
 pykrige_vario() (gstools.covmodel.Linear method), 207
 pykrige_vario() (gstools.covmodel.Matern method), 160
 pykrige_vario() (gstools.covmodel.Rational method), 184
 pykrige_vario() (gstools.covmodel.Spherical method), 231
 pykrige_vario() (gstools.covmodel.Stable method), 171
 pykrige_vario() (gstools.covmodel.SuperSpherical method), 256
 pykrige_vario() (gstools.covmodel.TPLExponential method), 294
 pykrige_vario() (gstools.covmodel.TPLGaussian method), 280
 pykrige_vario() (gstools.covmodel.TPLSimple method), 321
 pykrige_vario() (gstools.covmodel.TPLStable method), 308

R

RandMeth (class in gstools.field.generator), 343
 random (gstools.random.RNG property), 397
 Rational (class in gstools.covmodel), 177
 remove_trend_norm_mean() (in module gstools.normalizer), 429
 rescale (gstools.covmodel.Circular property), 224
 rescale (gstools.covmodel.CovModel property), 126
 rescale (gstools.covmodel.Cubic property), 200
 rescale (gstools.covmodel.Exponential property), 152
 rescale (gstools.covmodel.Gaussian property), 139
 rescale (gstools.covmodel.HyperSpherical property), 248
 rescale (gstools.covmodel.JBessel property), 272
 rescale (gstools.covmodel.Linear property), 212
 rescale (gstools.covmodel.Matern property), 164
 rescale (gstools.covmodel.Rational property), 188
 rescale (gstools.covmodel.Spherical property), 236

rescale (gstools.covmodel.Stable property), 176
 rescale (gstools.covmodel.SuperSpherical property), 260
 rescale (gstools.covmodel.TPLExponential property), 299
 rescale (gstools.covmodel.TPLGaussian property), 285
 rescale (gstools.covmodel.TPLSimple property), 325
 rescale (gstools.covmodel.TPLStable property), 313
 reset_seed() (gstools.field.generator.IncomprRandMeth method), 342
 reset_seed() (gstools.field.generator.RandMeth method), 345
 RNG (class in gstools.random), 395
 rotated_main_axes() (in module gstools.tools), 401
 rotation_planes() (in module gstools.tools), 401

S

sample_dist() (gstools.random.RNG method), 396
 sample_ln_pdf() (gstools.random.RNG method), 396
 sample_sphere() (gstools.random.RNG method), 396
 sampling (gstools.field.generator.IncomprRandMeth property), 343
 sampling (gstools.field.generator.RandMeth property), 345
 seed (gstools.field.generator.IncomprRandMeth property), 343
 seed (gstools.field.generator.RandMeth property), 345
 seed (gstools.random.MasterRNG property), 395
 seed (gstools.random.RNG property), 397
 set_angles() (in module gstools.tools), 402
 set_anis() (in module gstools.tools), 402
 set_arg_bounds() (gstools.covmodel.Circular method), 220
 set_arg_bounds() (gstools.covmodel.CovModel method), 122
 set_arg_bounds() (gstools.covmodel.Cubic method), 196
 set_arg_bounds() (gstools.covmodel.Exponential method), 147
 set_arg_bounds() (gstools.covmodel.Gaussian method), 135
 set_arg_bounds() (gstools.covmodel.HyperSpherical method), 244
 set_arg_bounds() (gstools.covmodel.JBessel method), 268
 set_arg_bounds() (gstools.covmodel.Linear method), 208
 set_arg_bounds() (gstools.covmodel.Matern method), 160
 set_arg_bounds() (gstools.covmodel.Rational method), 184
 set_arg_bounds() (gstools.covmodel.Spherical method), 232
 set_arg_bounds() (gstools.covmodel.Stable method), 172

- `set_arg_bounds()` (`gstools.covmodel.SuperSpherical` method), 256
`set_arg_bounds()` (`gstools.covmodel.TPLExponential` method), 295
`set_arg_bounds()` (`gstools.covmodel.TPLGaussian` method), 281
`set_arg_bounds()` (`gstools.covmodel.TPLSimple` method), 321
`set_arg_bounds()` (`gstools.covmodel.TPLStable` method), 308
`set_condition()` (`gstools.krige.Detrended` method), 391
`set_condition()` (`gstools.krige.ExtDrift` method), 384
`set_condition()` (`gstools.krige.Krige` method), 356
`set_condition()` (`gstools.krige.Ordinary` method), 370
`set_condition()` (`gstools.krige.Simple` method), 363
`set_condition()` (`gstools.krige.Universal` method), 377
`set_drift_functions()` (`gstools.krige.Detrended` method), 392
`set_drift_functions()` (`gstools.krige.ExtDrift` method), 385
`set_drift_functions()` (`gstools.krige.Krige` method), 356
`set_drift_functions()` (`gstools.krige.Ordinary` method), 371
`set_drift_functions()` (`gstools.krige.Simple` method), 364
`set_drift_functions()` (`gstools.krige.Universal` method), 378
`set_generator()` (`gstools.field.CondSRF` method), 336
`set_generator()` (`gstools.field.SRF` method), 332
`sill` (`gstools.covmodel.Circular` property), 224
`sill` (`gstools.covmodel.CovModel` property), 126
`sill` (`gstools.covmodel.Cubic` property), 200
`sill` (`gstools.covmodel.Exponential` property), 152
`sill` (`gstools.covmodel.Gaussian` property), 139
`sill` (`gstools.covmodel.HyperSpherical` property), 248
`sill` (`gstools.covmodel.JBessel` property), 272
`sill` (`gstools.covmodel.Linear` property), 212
`sill` (`gstools.covmodel.Matern` property), 164
`sill` (`gstools.covmodel.Rational` property), 188
`sill` (`gstools.covmodel.Spherical` property), 236
`sill` (`gstools.covmodel.Stable` property), 176
`sill` (`gstools.covmodel.SuperSpherical` property), 260
`sill` (`gstools.covmodel.TPLExponential` property), 299
`sill` (`gstools.covmodel.TPLGaussian` property), 285
`sill` (`gstools.covmodel.TPLSimple` property), 325
`sill` (`gstools.covmodel.TPLStable` property), 313
`Simple` (class in `gstools.krige`), 360
`spectral_density()` (`gstools.covmodel.Circular` method), 220
`spectral_density()` (`gstools.covmodel.CovModel` method), 122
`spectral_density()` (`gstools.covmodel.Cubic` method), 196
`spectral_density()` (`gstools.covmodel.Exponential` method), 148
`spectral_density()` (`gstools.covmodel.Gaussian` method), 135
`spectral_density()` (`gstools.covmodel.HyperSpherical` method), 244
`spectral_density()` (`gstools.covmodel.JBessel` method), 268
`spectral_density()` (`gstools.covmodel.Linear` method), 208
`spectral_density()` (`gstools.covmodel.Matern` method), 160
`spectral_density()` (`gstools.covmodel.Rational` method), 184
`spectral_density()` (`gstools.covmodel.Spherical` method), 232
`spectral_density()` (`gstools.covmodel.Stable` method), 172
`spectral_density()` (`gstools.covmodel.SuperSpherical` method), 256
`spectral_density()` (`gstools.covmodel.TPLExponential` method), 295
`spectral_density()` (`gstools.covmodel.TPLGaussian` method), 281
`spectral_density()` (`gstools.covmodel.TPLSimple` method), 321
`spectral_density()` (`gstools.covmodel.TPLStable` method), 308
`spectral_rad_cdf()` (`gstools.covmodel.Exponential` method), 148
`spectral_rad_cdf()` (`gstools.covmodel.Gaussian` method), 135
`spectral_rad_pdf()` (`gstools.covmodel.Circular` method), 220
`spectral_rad_pdf()` (`gstools.covmodel.CovModel` method), 123
`spectral_rad_pdf()` (`gstools.covmodel.Cubic` method), 196
`spectral_rad_pdf()` (`gstools.covmodel.Exponential` method), 148
`spectral_rad_pdf()` (`gstools.covmodel.Gaussian` method), 135
`spectral_rad_pdf()` (`gstools.covmodel.HyperSpherical` method), 244
`spectral_rad_pdf()` (`gstools.covmodel.JBessel` method), 268
`spectral_rad_pdf()` (`gstools.covmodel.Linear` method), 208
`spectral_rad_pdf()` (`gstools.covmodel.Matern` method), 160
`spectral_rad_pdf()` (`gstools.covmodel.Rational` method), 184

- method*), 184
 - `spectral_rad_pdf()` (*gstools.covmodel.Spherical method*), 232
 - `spectral_rad_pdf()` (*gstools.covmodel.Stable method*), 172
 - `spectral_rad_pdf()` (*gstools.covmodel.SuperSpherical method*), 256
 - `spectral_rad_pdf()` (*gstools.covmodel.TPLExponential method*), 295
 - `spectral_rad_pdf()` (*gstools.covmodel.TPLGaussian method*), 281
 - `spectral_rad_pdf()` (*gstools.covmodel.TPLSimple method*), 321
 - `spectral_rad_pdf()` (*gstools.covmodel.TPLStable method*), 309
 - `spectral_rad_ppf()` (*gstools.covmodel.Exponential method*), 148
 - `spectral_rad_ppf()` (*gstools.covmodel.Gaussian method*), 135
 - `spectrum()` (*gstools.covmodel.Circular method*), 220
 - `spectrum()` (*gstools.covmodel.CovModel method*), 123
 - `spectrum()` (*gstools.covmodel.Cubic method*), 196
 - `spectrum()` (*gstools.covmodel.Exponential method*), 148
 - `spectrum()` (*gstools.covmodel.Gaussian method*), 135
 - `spectrum()` (*gstools.covmodel.HyperSpherical method*), 244
 - `spectrum()` (*gstools.covmodel.JBessel method*), 268
 - `spectrum()` (*gstools.covmodel.Linear method*), 208
 - `spectrum()` (*gstools.covmodel.Matern method*), 160
 - `spectrum()` (*gstools.covmodel.Rational method*), 184
 - `spectrum()` (*gstools.covmodel.Spherical method*), 232
 - `spectrum()` (*gstools.covmodel.Stable method*), 172
 - `spectrum()` (*gstools.covmodel.SuperSpherical method*), 256
 - `spectrum()` (*gstools.covmodel.TPLExponential method*), 295
 - `spectrum()` (*gstools.covmodel.TPLGaussian method*), 281
 - `spectrum()` (*gstools.covmodel.TPLSimple method*), 321
 - `spectrum()` (*gstools.covmodel.TPLStable method*), 309
 - Spherical* (class in *gstools.covmodel*), 225
 - SRF* (class in *gstools.field*), 329
 - Stable* (class in *gstools.covmodel*), 165
 - `standard_bins()` (in module *gstools.variogram*), 351
 - `structured()` (*gstools.field.CondSRF method*), 336
 - `structured()` (*gstools.field.Field method*), 340
 - `structured()` (*gstools.field.SRF method*), 332
 - `structured()` (*gstools.krige.Detrended method*), 392
 - `structured()` (*gstools.krige.ExtDrift method*), 385
 - `structured()` (*gstools.krige.Krige method*), 357
 - `structured()` (*gstools.krige.Ordinary method*), 371
 - `structured()` (*gstools.krige.Simple method*), 364
 - `structured()` (*gstools.krige.Universal method*), 378
 - `structured()` (*gstools.krige.Simple method*), 364
 - `structured()` (*gstools.krige.Universal method*), 378
 - `SuperSpherical` (class in *gstools.covmodel*), 249
- ## T
- `to_pyvista()` (*gstools.field.CondSRF method*), 336
 - `to_pyvista()` (*gstools.field.Field method*), 340
 - `to_pyvista()` (*gstools.field.SRF method*), 332
 - `to_pyvista()` (*gstools.krige.Detrended method*), 392
 - `to_pyvista()` (*gstools.krige.ExtDrift method*), 385
 - `to_pyvista()` (*gstools.krige.Krige method*), 357
 - `to_pyvista()` (*gstools.krige.Ordinary method*), 371
 - `to_pyvista()` (*gstools.krige.Simple method*), 364
 - `to_pyvista()` (*gstools.krige.Universal method*), 378
 - `to_vtk()` (in module *gstools.tools*), 402
 - `to_vtk_structured()` (in module *gstools.tools*), 402
 - `to_vtk_unstructured()` (in module *gstools.tools*), 403
 - `tpl_exp_spec_dens()` (in module *gstools.tools*), 403
 - `tpl_gau_spec_dens()` (in module *gstools.tools*), 403
 - TPLExponential* (class in *gstools.covmodel*), 287
 - TPLGaussian* (class in *gstools.covmodel*), 273
 - TPLSimple* (class in *gstools.covmodel*), 314
 - TPLStable* (class in *gstools.covmodel*), 301
 - `tplstable_cor()` (in module *gstools.tools*), 403
 - trend* (*gstools.field.CondSRF property*), 337
 - trend* (*gstools.field.Field property*), 340
 - trend* (*gstools.field.SRF property*), 333
 - trend* (*gstools.krige.Detrended property*), 394
 - trend* (*gstools.krige.ExtDrift property*), 387
 - trend* (*gstools.krige.Krige property*), 359
 - trend* (*gstools.krige.Ordinary property*), 373
 - trend* (*gstools.krige.Simple property*), 366
 - trend* (*gstools.krige.Universal property*), 380
- ## U
- unbiased* (*gstools.krige.Detrended property*), 394
 - unbiased* (*gstools.krige.ExtDrift property*), 387
 - unbiased* (*gstools.krige.Krige property*), 359
 - unbiased* (*gstools.krige.Ordinary property*), 373
 - unbiased* (*gstools.krige.Simple property*), 366
 - unbiased* (*gstools.krige.Universal property*), 380
 - Universal* (class in *gstools.krige*), 374
 - `unstructured()` (*gstools.field.CondSRF method*), 336
 - `unstructured()` (*gstools.field.Field method*), 340
 - `unstructured()` (*gstools.field.SRF method*), 332
 - `unstructured()` (*gstools.krige.Detrended method*), 392
 - `unstructured()` (*gstools.krige.ExtDrift method*), 385
 - `unstructured()` (*gstools.krige.Krige method*), 357
 - `unstructured()` (*gstools.krige.Ordinary method*), 371
 - `unstructured()` (*gstools.krige.Simple method*), 364
 - `unstructured()` (*gstools.krige.Universal method*), 378
 - `update()` (*gstools.field.generator.IncomprRandMeth method*), 342

`update()` (*gstools.field.generator.RandMeth* method), 345
`upscaling` (*gstools.field.SRF* property), 333
`upscaling_func()` (*gstools.field.SRF* method), 332

V

`value_type` (*gstools.field.CondSRF* property), 337
`value_type` (*gstools.field.Field* property), 340
`value_type` (*gstools.field.generator.IncomprRandMeth* property), 343
`value_type` (*gstools.field.generator.RandMeth* property), 345
`value_type` (*gstools.field.SRF* property), 333
`value_type` (*gstools.krige.Detrended* property), 394
`value_type` (*gstools.krige.ExtDrift* property), 387
`value_type` (*gstools.krige.Krige* property), 359
`value_type` (*gstools.krige.Ordinary* property), 373
`value_type` (*gstools.krige.Simple* property), 366
`value_type` (*gstools.krige.Universal* property), 380
`var` (*gstools.covmodel.Circular* property), 224
`var` (*gstools.covmodel.CovModel* property), 127
`var` (*gstools.covmodel.Cubic* property), 200
`var` (*gstools.covmodel.Exponential* property), 152
`var` (*gstools.covmodel.Gaussian* property), 139
`var` (*gstools.covmodel.HyperSpherical* property), 248
`var` (*gstools.covmodel.JBessel* property), 272
`var` (*gstools.covmodel.Linear* property), 212
`var` (*gstools.covmodel.Matern* property), 164
`var` (*gstools.covmodel.Rational* property), 188
`var` (*gstools.covmodel.Spherical* property), 236
`var` (*gstools.covmodel.Stable* property), 176
`var` (*gstools.covmodel.SuperSpherical* property), 260
`var` (*gstools.covmodel.TPLeExponential* property), 299
`var` (*gstools.covmodel.TPLGaussian* property), 285
`var` (*gstools.covmodel.TPLSimple* property), 325
`var` (*gstools.covmodel.TPLStable* property), 313
`var_bounds` (*gstools.covmodel.Circular* property), 224
`var_bounds` (*gstools.covmodel.CovModel* property), 127
`var_bounds` (*gstools.covmodel.Cubic* property), 200
`var_bounds` (*gstools.covmodel.Exponential* property), 152
`var_bounds` (*gstools.covmodel.Gaussian* property), 139
`var_bounds` (*gstools.covmodel.HyperSpherical* property), 248
`var_bounds` (*gstools.covmodel.JBessel* property), 272
`var_bounds` (*gstools.covmodel.Linear* property), 212
`var_bounds` (*gstools.covmodel.Matern* property), 164
`var_bounds` (*gstools.covmodel.Rational* property), 188
`var_bounds` (*gstools.covmodel.Spherical* property), 236
`var_bounds` (*gstools.covmodel.Stable* property), 176
`var_bounds` (*gstools.covmodel.SuperSpherical* property), 260
`var_bounds` (*gstools.covmodel.TPLeExponential* property), 299
`var_bounds` (*gstools.covmodel.TPLGaussian* property), 285
`var_bounds` (*gstools.covmodel.TPLSimple* property), 325
`var_bounds` (*gstools.covmodel.TPLStable* property), 313
`var_coarse_graining()` (in module *gstools.field.upscaling*), 347
`var_factor()` (*gstools.covmodel.Circular* method), 220
`var_factor()` (*gstools.covmodel.CovModel* method), 123
`var_factor()` (*gstools.covmodel.Cubic* method), 196
`var_factor()` (*gstools.covmodel.Exponential* method), 148
`var_factor()` (*gstools.covmodel.Gaussian* method), 136
`var_factor()` (*gstools.covmodel.HyperSpherical* method), 244
`var_factor()` (*gstools.covmodel.JBessel* method), 268
`var_factor()` (*gstools.covmodel.Linear* method), 208
`var_factor()` (*gstools.covmodel.Matern* method), 160
`var_factor()` (*gstools.covmodel.Rational* method), 184
`var_factor()` (*gstools.covmodel.Spherical* method), 232
`var_factor()` (*gstools.covmodel.Stable* method), 172
`var_factor()` (*gstools.covmodel.SuperSpherical* method), 256
`var_factor()` (*gstools.covmodel.TPLeExponential* method), 295
`var_factor()` (*gstools.covmodel.TPLGaussian* method), 281
`var_factor()` (*gstools.covmodel.TPLSimple* method), 321
`var_factor()` (*gstools.covmodel.TPLStable* method), 309
`var_no_scaling()` (in module *gstools.field.upscaling*), 347
`var_raw` (*gstools.covmodel.Circular* property), 224
`var_raw` (*gstools.covmodel.CovModel* property), 127
`var_raw` (*gstools.covmodel.Cubic* property), 200
`var_raw` (*gstools.covmodel.Exponential* property), 152
`var_raw` (*gstools.covmodel.Gaussian* property), 140
`var_raw` (*gstools.covmodel.HyperSpherical* property), 248
`var_raw` (*gstools.covmodel.JBessel* property), 272
`var_raw` (*gstools.covmodel.Linear* property), 212
`var_raw` (*gstools.covmodel.Matern* property), 164
`var_raw` (*gstools.covmodel.Rational* property), 188
`var_raw` (*gstools.covmodel.Spherical* property), 236
`var_raw` (*gstools.covmodel.Stable* property), 176
`var_raw` (*gstools.covmodel.SuperSpherical* property), 260

`var_raw` (*gstools.covmodel.TPLExponential* property), 300

`var_raw` (*gstools.covmodel.TPLGaussian* property), 286

`var_raw` (*gstools.covmodel.TPLSimple* property), 325

`var_raw` (*gstools.covmodel.TPLStable* property), 313

`vario_axis`() (*gstools.covmodel.Circular* method), 220

`vario_axis`() (*gstools.covmodel.CovModel* method), 123

`vario_axis`() (*gstools.covmodel.Cubic* method), 196

`vario_axis`() (*gstools.covmodel.Exponential* method), 148

`vario_axis`() (*gstools.covmodel.Gaussian* method), 136

`vario_axis`() (*gstools.covmodel.HyperSpherical* method), 244

`vario_axis`() (*gstools.covmodel.JBessel* method), 268

`vario_axis`() (*gstools.covmodel.Linear* method), 208

`vario_axis`() (*gstools.covmodel.Matern* method), 160

`vario_axis`() (*gstools.covmodel.Rational* method), 184

`vario_axis`() (*gstools.covmodel.Spherical* method), 232

`vario_axis`() (*gstools.covmodel.Stable* method), 172

`vario_axis`() (*gstools.covmodel.SuperSpherical* method), 256

`vario_axis`() (*gstools.covmodel.TPLExponential* method), 295

`vario_axis`() (*gstools.covmodel.TPLGaussian* method), 281

`vario_axis`() (*gstools.covmodel.TPLSimple* method), 321

`vario_axis`() (*gstools.covmodel.TPLStable* method), 309

`vario_estimate`() (in module *gstools.variogram*), 348

`vario_estimate_axis`() (in module *gstools.variogram*), 350

`vario_nugget`() (*gstools.covmodel.Circular* method), 220

`vario_nugget`() (*gstools.covmodel.CovModel* method), 123

`vario_nugget`() (*gstools.covmodel.Cubic* method), 196

`vario_nugget`() (*gstools.covmodel.Exponential* method), 148

`vario_nugget`() (*gstools.covmodel.Gaussian* method), 136

`vario_nugget`() (*gstools.covmodel.HyperSpherical* method), 244

`vario_nugget`() (*gstools.covmodel.JBessel* method), 268

`vario_nugget`() (*gstools.covmodel.Linear* method), 208

`vario_nugget`() (*gstools.covmodel.Matern* method), 160

`vario_nugget`() (*gstools.covmodel.Rational* method), 184

`vario_nugget`() (*gstools.covmodel.Spherical* method), 232

`vario_nugget`() (*gstools.covmodel.Stable* method), 172

`vario_nugget`() (*gstools.covmodel.SuperSpherical* method), 256

`vario_nugget`() (*gstools.covmodel.TPLExponential* method), 295

`vario_nugget`() (*gstools.covmodel.TPLGaussian* method), 281

`vario_nugget`() (*gstools.covmodel.TPLSimple* method), 321

`vario_nugget`() (*gstools.covmodel.TPLStable* method), 309

`vario_yadrenko`() (*gstools.covmodel.Circular* method), 220

`vario_yadrenko`() (*gstools.covmodel.CovModel* method), 123

`vario_yadrenko`() (*gstools.covmodel.Cubic* method), 196

`vario_yadrenko`() (*gstools.covmodel.Exponential* method), 148

`vario_yadrenko`() (*gstools.covmodel.Gaussian* method), 136

`vario_yadrenko`() (*gstools.covmodel.HyperSpherical* method), 244

`vario_yadrenko`() (*gstools.covmodel.JBessel* method), 268

`vario_yadrenko`() (*gstools.covmodel.Linear* method), 208

`vario_yadrenko`() (*gstools.covmodel.Matern* method), 160

`vario_yadrenko`() (*gstools.covmodel.Rational* method), 184

`vario_yadrenko`() (*gstools.covmodel.Spherical* method), 232

`vario_yadrenko`() (*gstools.covmodel.Stable* method), 172

`vario_yadrenko`() (*gstools.covmodel.SuperSpherical* method), 256

`vario_yadrenko`() (*gstools.covmodel.TPLExponential* method), 295

`vario_yadrenko`() (*gstools.covmodel.TPLGaussian* method), 281

`vario_yadrenko`() (*gstools.covmodel.TPLSimple* method), 321

`vario_yadrenko`() (*gstools.covmodel.TPLStable* method), 309

- method), 148
- `vario_yadrenko()` (`gstools.covmodel.Gaussian` method), 136
- `vario_yadrenko()` (`gstools.covmodel.HyperSpherical` method), 244
- `vario_yadrenko()` (`gstools.covmodel.JBessel` method), 268
- `vario_yadrenko()` (`gstools.covmodel.Linear` method), 208
- `vario_yadrenko()` (`gstools.covmodel.Matern` method), 160
- `vario_yadrenko()` (`gstools.covmodel.Rational` method), 184
- `vario_yadrenko()` (`gstools.covmodel.Spherical` method), 232
- `vario_yadrenko()` (`gstools.covmodel.Stable` method), 172
- `vario_yadrenko()` (`gstools.covmodel.SuperSpherical` method), 256
- `vario_yadrenko()` (`gstools.covmodel.TPLExponential` method), 295
- `vario_yadrenko()` (`gstools.covmodel.TPLGaussian` method), 281
- `vario_yadrenko()` (`gstools.covmodel.TPLSimple` method), 322
- `vario_yadrenko()` (`gstools.covmodel.TPLStable` method), 309
- `variogram()` (`gstools.covmodel.Circular` method), 220
- `variogram()` (`gstools.covmodel.Cubic` method), 196
- `variogram()` (`gstools.covmodel.Exponential` method), 148
- `variogram()` (`gstools.covmodel.Gaussian` method), 136
- `variogram()` (`gstools.covmodel.HyperSpherical` method), 244
- `variogram()` (`gstools.covmodel.JBessel` method), 269
- `variogram()` (`gstools.covmodel.Linear` method), 208
- `variogram()` (`gstools.covmodel.Matern` method), 161
- `variogram()` (`gstools.covmodel.Rational` method), 185
- `variogram()` (`gstools.covmodel.Spherical` method), 232
- `variogram()` (`gstools.covmodel.Stable` method), 172
- `variogram()` (`gstools.covmodel.SuperSpherical` method), 257
- `variogram()` (`gstools.covmodel.TPLExponential` method), 295
- `variogram()` (`gstools.covmodel.TPLGaussian` method), 281
- `variogram()` (`gstools.covmodel.TPLSimple` method), 322
- `variogram()` (`gstools.covmodel.TPLStable` method), 309
- `verbose` (`gstools.field.generator.IncomprRandMeth` property), 343
- `verbose` (`gstools.field.generator.RandMeth` property), 346
- `vtk_export()` (`gstools.field.CondSRF` method), 336
- `vtk_export()` (`gstools.field.Field` method), 340
- `vtk_export()` (`gstools.field.SRF` method), 332
- `vtk_export()` (`gstools.krige.Detrended` method), 392
- `vtk_export()` (`gstools.krige.ExtDrift` method), 385
- `vtk_export()` (`gstools.krige.Krige` method), 357
- `vtk_export()` (`gstools.krige.Ordinary` method), 371
- `vtk_export()` (`gstools.krige.Simple` method), 364
- `vtk_export()` (`gstools.krige.Universal` method), 378
- `vtk_export()` (in module `gstools.tools`), 404
- `vtk_export_structured()` (in module `gstools.tools`), 404
- `vtk_export_unstructured()` (in module `gstools.tools`), 404
- ## Y
- `YeoJohnson` (class in `gstools.normalizer`), 420
- ## Z
- `zinnharvey()` (in module `gstools.transform`), 407