GeoStatTools Documentation

Release 1.3.5

Sebastian Müller, Lennart Schüler

Jan 20, 2022
## 1 GSTools Quickstart

1.1 Purpose ................................................................. 1
1.2 Installation ............................................................. 2
   conda ................................................................. 2
   pip ................................................................. 2
1.3 Citation ..................................................................... 3
1.4 Tutorials and Examples ............................................... 3
1.5 Spatial Random Field Generation ................................. 3
   Examples ............................................................... 4
1.6 Estimating and fitting variograms ............................... 6
   Examples ............................................................... 6
1.7 Kriging and Conditioned Random Fields ................. 7
   Example .............................................................. 7
1.8 User defined covariance models ............................... 8
   Example .............................................................. 8
1.9 Incompressible Vector Field Generation .................. 9
   Example .............................................................. 9
1.10 VTK/PyVista Export ............................................... 10
1.11 Requirements ........................................................ 10
   Optional .............................................................. 11
   Contact ............................................................... 11
1.12 License ............................................................... 11

## 2 GSTools Tutorials

2.1 Random Field Generation ........................................ 13
   Examples ............................................................... 13
2.2 The Covariance Model ............................................. 26
   Provided Covariance Models .................................. 27
   Examples ............................................................... 27
2.3 Variogram Estimation .............................................. 40
   Examples ............................................................... 40
2.4 Random Vector Field Generation ............................ 53
   Examples ............................................................... 54
2.5 Kriging ................................................................. 57
   Provided Kriging Methods ..................................... 58
   Examples ............................................................... 58
2.6 Conditioned Fields ................................................ 72
   Examples ............................................................... 72
2.7 Field transformations ............................................. 76
   Examples ............................................................... 76
2.8 Geographic Coordinates ......................................... 85
3 GSTools API

3.1 Purpose ..................................................... 117
3.2 Subpackages ............................................... 117
3.3 Classes ...................................................... 117
   Krige ......................................................... 117
   Spatial Random Field ........................................ 118
   Covariance Base-Class ....................................... 118
   Covariance Models ........................................... 118
3.4 Functions .................................................. 118
   VTK-Export ................................................ 118
   Geometric .................................................. 119
   Variogram Estimation ....................................... 119
3.5 Misc .......................................................... 119
3.6 gstools.covmodel ........................................... 120
   Subpackages ................................................ 120
   Covariance Base-Class ....................................... 120
   Covariance Models ........................................... 132
3.7 gstools.field ............................................... 333
   Subpackages ................................................ 333
   Spatial Random Field ........................................ 333
   Field Base Class ........................................... 346
   gstools.field.generator .................................... 352
   gstools.field.upscaling ................................... 358
3.8 gstools.variogram .......................................... 359
   Variogram estimation ....................................... 359
   Binning ..................................................... 362
3.9 gstools.krige ............................................... 363
   Kriging Classes ............................................. 363
3.10 gstools.random ............................................. 418
    Random Number Generator ................................ 418
    Seed Generator ............................................ 420
    Distribution factory ...................................... 421
3.11 gstools.tools .............................................. 422
    Export .................................................... 422
    Special functions ......................................... 424
    Geometric ................................................ 426
    Misc ...................................................... 431
3.12 gstools.transform ........................................ 432
    Wrapper ................................................... 432
    Field Transformations ..................................... 432
    Array Transformations .................................... 437
3.13 gstools.normalizer ....................................... 441
    Base-Normalizer .......................................... 441
    Field-Normalizer ......................................... 444
    Convenience Routines ..................................... 462

4 Changelog .................................................... 465
1.1 Purpose

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
- (automated) 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
1.2 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@main
```

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

**Speeding up GSTools by parallelization**

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@main
```

**Using experimental GSTools-Core for even more speed**

You can install the optional dependency GSTools-Core, which is a re-implementation of the main algorithms used in GSTools. The new package uses the language Rust and it should be faster (in some cases by orders of magnitude), safer, and it will potentially completely replace the current standard implementation in Cython. Once the package GSTools-Core is available on your machine, it will be used by default. In case you want to switch back to the Cython implementation, you can set `gstools.config.USE_RUST=False` in your code. This also works at runtime. You can install the optional dependency e.g. by

```
pip install gstools[rust]
```

or by manually installing the package

```
pip install gstools-core
```

GSTools-Core will automatically use all your cores in parallel, without having to use OpenMP or a local C compiler. In case you want to restrict the number of threads used, you can set the environment variable `RAYON_NUM_THREADS` to the desired amount.
1.3 Citation

If you are using GSTools in your publication please cite our paper:


You can cite the Zenodo code publication of GSTools by:


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

1.4 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

1.5 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.

```python
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.

```python
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)
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)
```
As an example, let's export a three-dimensional field to a VTK file, which can be visualized with ParaView or PyVista in Python:

```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()
```

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:
1.6 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.

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

# generate a synthetic field with an exponential model
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)
```
1.7 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:

```python
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]

# 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)

# same output positions for all ensemble members
grid_pos = np.linspace(0.0, 15.0, 151)
cond_srf.set_pos(grid_pos)

# seeded ensemble generation
seed = gs.random.MasterRNG(20170519)
for i in range(100):
    field = cond_srf(seed=seed(), store=f'field_{i}')
    plt.plot(grid_pos, field, color='k', alpha=0.1)
plt.scatter(cond_pos, cond_val, color='k')
plt.show()
```
1.8 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 $h = r/l$

```python
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.
1.9 Incompressible Vector Field Generation

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

Example

```python
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

![Field 2D structured: (2, 100, 100)](image)
1.10  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 \texttt{vtk\_export()} or you could create a VTK/PyVista dataset for use in Python with \texttt{to\_pyvista()} method:

```python
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 \texttt{field.vtr} or creates a PyVista mesh in memory for immediate 3D plotting in Python.

1.11  Requirements

- Numpy $\geq$ 1.14.5
- SciPy $\geq$ 1.1.0
- hankel $\geq$ 1.0.0
- emcee $\geq$ 3.0.0
- pyevtk $\geq$ 1.1.1
- meshio $\geq$ 4.0.0
Optional

- GSTools-Core >= 0.2.0
- matplotlib
- pyvista

Contact

You can contact us via info@geostat-framework.org.

1.12 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 \textit{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 \textit{SRF} class for the actual generation of the spatial random field. But \textit{SRF} also needs a covariance model and we will simply take the \textit{Gaussian} model.

```python
import gstools as gs
x = y = range(100)
```

Now we create the covariance model with the parameters $\sigma^2$ and $\lambda$ and hand it over to \textit{SRF}. By specifying a seed, we make sure to create reproducible results:

```python
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 0.707 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.

We will set the position tuple \(pos\) before generation to reuse it afterwards.

```python
import matplotlib.pyplot as pt
import numpy as np
import gstools as gs

x = y = np.arange(100)
model = gs.Gaussian(dim=2, var=1, len_scale=10)
srf = gs.SRF(model)
srf.set_pos([x, y], "structured")
```

This time, we did not provide a seed to \(S RF\), as the seeds will be used during the actual computation of the fields. We will create four ensemble members, for better visualisation, save them in to \( srf \) class and in a first step, we will be using the loop counter as the seeds.
ens_no = 4
for i in range(ens_no):
    srf(seed=i, store=f"field{i}")

Now let's have a look at the results. We can access the fields by name or index:

```python
fig, ax = pt.subplots(2, 2, sharex=True, sharey=True)
ax = ax.flatten()
for i in range(ens_no):
    ax[i].imshow(srf[i].T, origin="lower")
pt.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

```python
from gstools.random import MasterRNG
seed = MasterRNG(20170519)
for i in range(ens_no):
    srf(seed=seed(), store=f"better_field{i}")
```

Total running time of the script: ( 0 minutes 3.790 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 $\theta = \pi/8$

```python
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

```python
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 0.996 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.

```python
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()
a.set_aspect("equal")
```

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

**Total running time of the script:** ( 0 minutes 1.075 seconds)
Merging two Fields

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

```python
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:

```python
# 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)

# (continues on next page)
```
field2 = srf((x2, y2))
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 be increasing the resolution of the grids by a factor of 10.

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

**Total running time of the script:** (0 minutes 2.312 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`.

```python
import matplotlib.pyplot as plt
import matplotlib.tri as tri
```

(continues on next page)
import meshio
import meshzoo
import numpy as np
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`.

```python
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!

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

Out:

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

Total running time of the script: (0 minutes 1.518 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.

```python
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).

```python
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.

```python
data = 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.

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

Now we have access to PyVista’s abundancy 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.

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

The result should look like this:
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.

```python
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.

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

Out:

```python
Exponential(dim=4, var=0.98, len_scale=5.0, nugget=1.55e-10)
```

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:

```python
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.

```python
srf.plot()
```
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 $\gamma$, 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
- $\ell$ is the main correlation length
- $s$ is a scaling factor for unit conversion or normalization
- $\sigma^2$ 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**\([\text{dim, var, len\_scale, nugget, ...}]\): The Gaussian covariance model.
- **Exponential**\([\text{dim, var, len\_scale, nugget, ...}]\): The Exponential covariance model.
- **Matern**\([\text{dim, var, len\_scale, nugget, anis, ...}]\): The Matérn covariance model.
- **Stable**\([\text{dim, var, len\_scale, nugget, anis, ...}]\): The stable covariance model.
- **Rational**\([\text{dim, var, len\_scale, nugget, ...}]\): The rational quadratic covariance model.
- **Cubic**\([\text{dim, var, len\_scale, nugget, anis, ...}]\): The Cubic covariance model.
- **Linear**\([\text{dim, var, len\_scale, nugget, anis, ...}]\): The bounded linear covariance model.
- **Circular**\([\text{dim, var, len\_scale, nugget, ...}]\): The circular covariance model.
- **Spherical**\([\text{dim, var, len\_scale, nugget, ...}]\): The Spherical covariance model.
- **HyperSpherical**\([\text{dim, var, len\_scale, ...}]\): The Hyper-Spherical covariance model.
- **SuperSpherical**\([\text{dim, var, len\_scale, ...}]\): The Super-Spherical covariance model.
- **JBessel**\([\text{dim, var, len\_scale, nugget, anis, ...}]\): The J-Bessel hole model.
- **TPLSimple**\([\text{dim, var, len\_scale, nugget, ...}]\): The simply truncated power law model.

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

- **TPLGaussian**\([\text{dim, var, len\_scale, nugget, ...}]\): Truncated-Power-Law with Gaussian modes.
- **TPLExponential**\([\text{dim, var, len\_scale, ...}]\): Truncated-Power-Law with Exponential modes.
- **TPLStable**\([\text{dim, var, len\_scale, nugget, ...}]\): 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:

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

(continues on next page)
Here the parameter \( h \) stands for the normalized range \( r / \text{len}_\text{scale} \). Now we can instantiate this model:

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

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

```python
model.plot()
```

This is almost identical to the already provided Gaussian model. There, a scaling factor is implemented so the \( \text{len}_\text{scale} \) coincides with the integral scale:

```python
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:

```python
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 \textit{above} the nugget, which is sometimes called \textit{partial sill}.

• A covariance model can also have additional parameters.

**Total running time of the script:** (0 minutes 0.261 seconds)

**Basic Methods**

The covariance model class \texttt{CovModel} of GSTools provides a set of handy methods. One of the following functions defines the main characterization of the variogram:

• \texttt{CovModel.variogram}: The variogram of the model given by

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

• \texttt{CovModel.covariance}: The (auto-)covariance of the model given by

\[
C(r) = \sigma^2 \cdot \rho(r)
\]

• \texttt{CovModel.correlation}: The (auto-)correlation (or normalized covariance) of the model given by

\[
\rho(r)
\]

• \texttt{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 0.135 seconds)

**Anisotropy and Rotation**

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

```python
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. Alternatively you can provide a list of length-scales:

```python
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)
```
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:

```python
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 0.844 seconds)

**Spectral methods**

The spectrum of a covariance model is given by:

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

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

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

Where \(k = ||k||\).
Depending on the spectrum, the spectral-density is defined by:

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

You can access these methods by:

```python
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 0.130 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:

```python
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:

```python
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.

```python
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

```python
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 0.154 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 \textbf{stable} covariance model and we can define it by

```python
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 \texttt{CovModel.default_opt_arg} to provide a standard value for the optional argument \texttt{alpha}. We can access it in the correlation function by \texttt{self.alpha}.

Now we can instantiate this model by either setting \texttt{alpha} implicitly with the default value or explicitly:

```python
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 0.138 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.

```python
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': 1.0245739832546161, 'len_scale': 5.081592175463755, 'nugget': 0.0, 'alpha': 0.906704090624874}

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: \([-\infty, \infty]\)
- variance: \([0.0, \infty]\)
- len scale: \([0.0, \infty]\)
- nugget: \([0.0, \infty]\)

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

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

- `CovModel.default_opt_arg_bounds()` to provide standard bounds for your additional parameters.
- `CovModel.default_arg_bounds()` to provide default boundaries for arguments.
- `CovModel.set_arg_bounds([check_args])` to set bounds for the parameters of the model.
- `CovModel.check_arg_bounds()` to 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:

- `CovModel.var_bounds` for bounds for the variance.
- `CovModel.len_scale_bounds` for bounds for the length scale.
- `CovModel.nugget_bounds` for bounds for the nugget.
- `CovModel.opt_arg_bounds` for bounds for the optional arguments.

continues on next page
Table 4 – continued from previous page

| CovModel.arg_bounds | Bounds for all parameters. |

Total running time of the script: (0 minutes 0.156 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

```python
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)
pdf(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 0.507 seconds

Finding the best fitting variogram model

```python
import numpy as np
from matplotlib import pyplot as plt
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 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.
Iterate over all models, fit their variogram and calculate the r2 score.

```python
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 = {}

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

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)

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

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)

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

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

# 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,
                                           (continues on next page)
```

**Total running time of the script:** (0 minutes 1.129 seconds)
Afterwards we can use the estimated variogram to fit a model to it:

```python
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.

```python
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")
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 7.405 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.

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

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.

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

```python
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:
```

(continues on next page)
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.

```python
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.

```python
srf.plot()
```

2.3. Variogram Estimation
Total running time of the script: 0 minutes 7.764 seconds

Fit Variogram with automatic binning

```python
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.

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

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

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

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

Plot the fitting result.

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

Total running time of the script: (0 minutes 0.379 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.

```python
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],
        [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.

```python
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.

```python
# 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 `Krige` class, by setting `fit_variogram=True`. Then the whole procedure shortens:

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

# plot the result
krire.plot()
# show the fitting results
print(krire.model)
```
This example shows, that setting up variogram estimation and kriging routines is straight forward with GSTools!

**Total running time of the script:** ( 0 minutes 0.632 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 $e_1$. 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 $\nabla \cdot U = 0$. This can be done by using the randomization method we already know, but adding a projector to every mode being summed:

$$U(x) = \bar{U}e_1 - \sqrt{\frac{\sigma^2}{N}} \sum_{i=1}^{N} p(k_i) [Z_{1,i} \cos (\langle k_i, x \rangle) + \sin (\langle k_i, x \rangle)]$$

with the projector

$$p(k_i) = e_1 - \frac{k_i k_1}{k^2}.$$

By calculating $\nabla \cdot U = 0$, 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:

```python
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

```python
# 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 2.592 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.

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

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

create a uniform grid with PyVista

dims, spacing, origin = (40, 30, 10), (1, 1, 1), (-10, 0, 0)
mesh = pv.UniformGrid(dims=dims, spacing=spacing, origin=origin)
```
create an incompressible random 3d velocity field on the given mesh with added mean velocity in x-direction

```python
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

```python
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:

```
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.

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

The result should look like this:
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 $z(x_1) \ldots z(x_n)$ 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 $W = (w_1, \ldots, w_n)$ 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, which is evaluated at the desired points with the given functions, the external drift has to 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 (substracting 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)
Consequently, 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:

```python
import gstools as gs
# conditions
cond_pos = [...]  # Positions of the known data points
cond_val = [...]  # Values at these points
model = gs.Gaussian(dim=1, var=0.5, len_scale=2)  # Define the Gaussian covariance model
krig = gs.krige.Simple(model, cond_pos=cond_pos, cond_val=cond_val, mean=1)  # Initialize the kriging class
```

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`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Krige</code></td>
<td>A Swiss Army knife for kriging.</td>
</tr>
<tr>
<td><code>Simple</code></td>
<td>Simple kriging.</td>
</tr>
<tr>
<td><code>Ordinary</code></td>
<td>Ordinary kriging.</td>
</tr>
<tr>
<td><code>Universal</code></td>
<td>Universal kriging.</td>
</tr>
<tr>
<td><code>ExtDrift</code></td>
<td>External drift kriging (EDK).</td>
</tr>
<tr>
<td><code>Detrended</code></td>
<td>Detrended simple kriging.</td>
</tr>
</tbody>
</table>

### 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.

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

# 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]

# resulting grid
gridx = np.linspace(0.0, 15.0, 151)

# spatial random field class
model = Gaussian(dim=1, var=0.5, len_scale=2)

krig = krige.Simple(model, mean=1, cond_pos=cond_pos, cond_val=cond_val)
krig(gridx)

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

Total running time of the script: 0 minutes 0.143 seconds

2.5. Kriging
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:

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

Thereby $c(x_i, x_j)$ is the covariance of the given observations and $\mu$ 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`.

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

# 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]

# 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)
kri = krig(gridx)

ax = kri.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
ax.legend()
```
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:

```python
import numpy as nprom matplotlib import pyplot as pltfrom pykrige.ok import OrdinaryKriging
import gstools as gss # conditioning datacond_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]g # grid definition for output fieldgridx = 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:

```python
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.

```python
OK2 = gs.krige.Ordinary(model, [cond_x, cond_y], cond_val, exact=True)
OK2.structured([gridx, gridy])
ax = OK2.plot()
ax.set_aspect("equal")
```
import matplotlib.pyplot as plt
import numpy as np
from gstools import Gaussian, krige

# 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]

# 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)

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

(continues on next 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 0.145 seconds)
External Drift Kriging

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

# synthetic conditions 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)

tax = krig.plot()
tax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
tax.plot(gridx, grid_drift, label="drift")
tax.legend()
```

Total running time of the script: (0 minutes 0.152 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.

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

# synthetic conditions 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 = krige.Universal(model, cond_pos, cond_val, "linear")
krig(gridx)
ax = krig.plot()
ax.scatter(cond_pos, cond_val, color="k", zorder=10, label="Conditions")
```

(continues on next page)
GeoStatTools Documentation, Release 1.3.5

(continued from previous page)

```python
ax.plot(gridx, gridx * 0.1 + 1, "-", label="linear drift")
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 0.169 seconds)

**Detrended Kriging**

```python
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 conditions 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)
```

(continues on next page)
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.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 0.165 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 conditions 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
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 = krig.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 0.164 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.

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

# conditions
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, 0.0]

# 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.

```python
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")
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()

Field 1D: (151,)

Total running time of the script: ( 0 minutes 0.142 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.

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

# conditions
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 = 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: 0.140 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:

```python
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.

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

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)
cond_srf.set_pos(gridx)

To generate the ensemble we will use a seed-generator. We can specify individual names for each field by the keyword `store`:

```python
seed = gs.random.MasterRNG(20170519)
for i in range(100):
    cond_srf(seed=seed(), store=f"f{i}")
    label = "Conditioned ensemble" if i == 0 else None
    plt.plot(gridx, cond_srf[f"f{i}"], color="k", alpha=0.1, label=label)

fields = [cond_srf[f"f{i}"] for i in range(100)]
plt.plot(gridx, np.mean(fields, axis=0), linestyle=":", label="Ensemble mean")
plt.plot(gridx, cond_srf.krige.field, linestyle="dashed", label="kriged field")
```

(continues on next page)
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 1.252 seconds)
Creating an Ensemble of conditioned 2D Fields

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

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

# conditioning data (x, y, value)
cond_pos = [[0.3, 1.9, 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)
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)
cond_srf.set_pos([x, y], "structured")

to generate the ensemble we will use a seed-generator. By specifying store=["fld[i]", False, False], only the conditioned field is stored with the specified name. The raw random field and the raw kriging field is not stored. This way, we can access each conditioned field by index cond_srf[i]:

```python
seed = gs.random.MasterRNG(20170519)
ens_no = 4
for i in range(ens_no):
    cond_srf(seed=seed(), store=["fld" + str(i), False, False])
```

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

```python
fig, ax = plt.subplots(ens_no + 1, ens_no + 1, figsize=(8, 8))
# plotting kwargs for scatter and image
vmax = np.max(cond_srf.all_fields)
sc_kw = dict(c=cond_val, edgecolors="k", vmin=0, vmax=vmax)
im_kw = dict(extent=2 * [0, 5], origin="lower", vmin=0, vmax=vmax)
for i in range(ens_no):
    # conditioned fields and conditions
    ax[i + 1, 0].imshow(cond_srf[i].T, **im_kw)
    ax[i + 1, 0].scatter(*cond_pos, **sc_kw)
    ax[i + 1, 0].set_ylabel(f"Field {i}", fontsize=10)
    ax[0, i + 1].imshow(cond_srf[i].T, **im_kw)
    ax[0, i + 1].scatter(*cond_pos, **sc_kw)
    ax[0, i + 1].set_title(f"Field {i}", fontsize=10)
# absolute differences
for j in range(ens_no):
    ax[i + 1, j + 1].imshow(np.abs(cond_srf[i] - cond_srf[j]).T, **im_kw)
# beautify plots
ax[0, 0].axis("off")
for ax in ax.flatten():
    ax.set_xticklabels([]), ax.set_yticklabels([])
fig.subplots_adjust(wspace=0, hspace=0)
fig.show()
```
To check if the generated fields are correct, we can have a look at their names:

```python
print(cond_srf.field_names)
```

Out:

```
['fld0', 'fld1', 'fld2', 'fld3']
```

**Total running time of the script:** ( 0 minutes 1.125 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:

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

All the transformations take a field class, that holds a generated field, as input and will manipulate this field inplace or store it with a given name.

Simply apply a transformation to a field class:

```python
import gstools as gs
...
srf = gs.SRF(model)
srf(...)gs.transform.normal_to_lognormal(srf)
```

Or use the provided wrapper:

```python
import gstools as gs
...
srf = gs.SRF(model)
srf(...)srf.transform("lognormal")
```

Examples

log-normal fields

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

See `transform.normal_to_lognormal`
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])
srf.transform("normal_to_lognormal")  # also "lognormal" works
srf.plot()

Total running time of the script: (0 minutes 0.683 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.

See transform.binary
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])
srf.transform("binary")
srf.plot()

Total running time of the script: ( 0 minutes 0.721 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.

See transform.discrete

import numpy as np
import gstools as gs

# 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)
srf.structured([x, y])
Create 5 equidistantly spaced values, thresholds are the arithmetic means

```python
values1 = np.linspace(np.min(srf.field), np.max(srf.field), 5)
srf.transform("discrete", store="f1", values=values1)
srf.plot("f1")
```

Calculate thresholds for equal shares but apply different values to the separated classes

```python
values2 = [0, -1, 2, -3, 4]
srf.transform("discrete", store="f2", values=values2, thresholds="equal")
srf.plot("f2")
```
Create user defined thresholds and apply different values to the separated classes

```python
def srf.transform("discrete", store="f3", values=values3, thresholds=thresholds)
srf.plot("f3")
```
Total running time of the script: 0 minutes 2.967 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.

See transform.zinnharvey
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])
srf.transform("zinnharvey", conn="high")
srf.plot()

Total running time of the script: 0 minutes 0.741 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.

See: `transform.normal_to_arcsin` and `transform.normal_to_uquad`
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])
srf.transform("normal_to_arcsin")  # also "arcsin" works
srf.plot()

Total running time of the script: (0 minutes 0.752 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.

We can select the desired field by its name and we can define an output name to store the field.

If you don’t specify field and store everything happens inplace.

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)
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.

All stored fields can be accessed and plotted by name:

```python
print("Max binary value:", srf.f_binary.max())
srf.plot(field="f_zinnharvey")
```
2.8 Geographic Coordinates

GSTools provides support for geographic coordinates given by:

- latitude \( \text{lat} \): specifies the north–south position of a point on the Earth’s surface
- longitude \( \text{lon} \): 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 \( \text{latlon=True} \) in your desired model (see \texttt{CovModel})

```python
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 \textit{Yadrenko} model on a sphere (see here). The \texttt{len_scale} is given in radians to scale the arc-length. In order to have a more meaningful length scale, one can use the \texttt{rescale} argument:

```python
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 $\zeta$ 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 $2 \cdot \sin(\zeta/2)$ 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.

```python
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!

```python
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)
```
GeoStatTools Documentation, Release 1.3.5

Out:

\texttt{Gaussian(latlon=True, var=1.02, len\_scale=8.3e+02, nugget=0.0, rescale=6.37e+03)}

\textbf{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.

\textbf{Total running time of the script:} (0 minutes 9.130 seconds)

\textbf{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 out.

\begin{verbatim}
import matplotlib.pyplot as plt
import numpy as np
import gstools as gs

def get_borders_germany():
    """Download simple german shape file with cartopy."""

    (continues on next page)
\end{verbatim}
import geopandas as gp  # 0.8.1
from cartopy.io import shapereader as shp_read  # version 0.18.0

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.

2.8. Geographic Coordinates 89
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.

```python
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)
```

```
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:

```python
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.

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

uk.set_pos((g_lat, g_lon), mesh_type="structured")
uk(return_var=False, store="temp_field")
uk(only_mean=True, store="mean_field")
```

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

```python
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, uk["temp_field"], levels, cmap="coolwarm")
co2 = ax[2].contourf(g_lon, g_lat, uk["mean_field"], 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 from DWD (2020-06-09 12:00)")
ax[1].set_title("Interpolated temperature with North-South drift")
ax[2].set_title("Estimated mean drift from 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:

```python
fig, ax = plt.subplots()
ax.plot(g_lat, uk["temp_field"][:, 50], label="Interpolated temperature")
ax.plot(g_lat, uk["mean_field"][:, 50], label="North-South mean drift")
ax.set_xlabel("Lat in deg")
ax.set_ylabel("T in [°C]")
```

(continues on next page)
Interpretation of the results is now up to you! ;-)  

**Total running time of the script:** ( 0 minutes 7.699 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 $st\_anis$:

```python
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_rf = gs.SRF(st_model)
st_field = st_rf(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.

```python
import copy
import matplotlib.pyplot as plt
import numpy as np
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:

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

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

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

```python
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import numpy as np
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)
# 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/stable/lib/python3.7/
˓→site-packages/gstools/transform/array.py:137: 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.

```python
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 $P$ / 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 42.355 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 parameteric 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 substracted 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogNormal([data])</td>
<td>Log-normal fields.</td>
</tr>
<tr>
<td>BoxCox([data])</td>
<td>Box-Cox (1964) transformed fields.</td>
</tr>
<tr>
<td>BoxCoxShift([data])</td>
<td>Box-Cox (1964) transformed fields including shifting.</td>
</tr>
<tr>
<td>YeoJohnson([data])</td>
<td>Yeo-Johnson (2000) transformed fields.</td>
</tr>
<tr>
<td>Modulus([data])</td>
<td>Modulus or John-Draper (1980) transformed fields.</td>
</tr>
<tr>
<td>Manly([data])</td>
<td>Manly (1971) transformed fields.</td>
</tr>
</tbody>
</table>

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

# 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]

# 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()
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.

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

# 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.

```python
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`.

```python
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:

```python
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.
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.

```python
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 0.392 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.

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

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 lambda and plot their transformation behavior within the interval [-5, 5].

For the shape parameter lambda, we create a list of 8 values ranging from -1 to 2.5.
```python
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!

```python
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, lmbda in enumerate(lmbdas):
        ax.flat[i].plot(x,
            norm(lmbda=lmbda).normalize(x),
            label=r'$\lambda=' + str(lmbda) + '$',
            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_ymargin(-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()
```
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 0.337 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- $\ell_{\text{low}}$ and an upper length-scale $\ell_{\text{up}}$.

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:

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

$$\sigma^2(\lambda) = C \cdot \lambda^{2H}$$

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

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

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

$$\sigma_{\ell_{\text{up}}}^2 = C \cdot \frac{\ell_{\text{up}}^{2H}}{2H}$$

Field 2D structured: (100, 100)
```python
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 13.521 seconds)

### Exporting Fields

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

These can be viewed for example with Paraview.

```python
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 0.444 seconds)
Check Random Sampling

import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import gtools 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])

(continues on next page)
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 were 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,
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.

```python
import os
import matplotlib.pyplot as plt
import numpy as np
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 urllib.request
    import zipfile
    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 shutil
    import pyvista as pv
    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])
    # asign 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
```

(continues on next page)
Downloading and Preprocessing

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

```python
# 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

```python
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

```python
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.

```python
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.

```python
# 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.

```python
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.

```python
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.

```python
# 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

```python
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

```python
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.

```python
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",
)
```

(continues on next page)
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):
        ", fit_model)
print("semivariogram model (in x-dir.):
        ", fit_model_x)
print("semivariogram model (in y-dir.):
        ", 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 straightforward now, to generate new Herten-like realisations.

```python
# 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 3.652 seconds)

### Standalone Field class

The Field class of GSTools can be used to plot arbitrary data in nD.

In the following example, we will produce 10000 random points in 4D with random values and plot them.

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

rng = np.random.RandomState(19970221)
x0 = rng.rand(10000) * 100.0
x1 = rng.rand(10000) * 100.0
x2 = rng.rand(10000) * 100.0
```
x3 = rng.rand(10000) * 100.0
values = rng.rand(10000) * 100.0

Only thing needed to instantiate the Field is the dimension.
Afterwards we can call the instance like all other Fields (SRF, Krig or CondSRF), but with an additional field.

```python
plotter = gs.field.Field(dim=4)
plotter(pos=(x0, x1, x2, x3), field=values)
plotter.plot()
```

Total running time of the script: ( 0 minutes 0.614 seconds)
CHAPTER 3

GSTOOLS API

3.1 Purpose

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

3.2 Subpackages

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

3.3 Classes

Kriging

Swiss-Army-Knife for Kriging. For short cut classes see: gtools.krige

\[Krige(model, cond_pos, cond_val[, ...])\] A Swiss Army knife for kriging.
Spatial Random Field

Classes for (conditioned) random field generation

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

Covariance Base-Class

Class to construct user defined covariance models

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

Covariance Models

Standard Covariance Models

| Gaussian(dim, var, len_scale, nugget, ...) | The Gaussian covariance model. |
| Exponential(dim, var, len_scale, nugget, ...) | The Exponential covariance model. |
| Matern(dim, var, len_scale, nugget, anis, ...) | The Matérn covariance model. |
| 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. |

Truncated Power Law Covariance Models

| TPLGaussian(dim, var, len_scale, nugget, ...) | Truncated-Power-Law with Gaussian modes. |
| TPLExponential(dim, var, len_scale, ...) | Truncated-Power-Law with Exponential modes. |
| TPLStable(dim, var, len_scale, nugget, ...) | Truncated-Power-Law with Stable modes. |
| TPLSimple(dim, var, len_scale, nugget, ...) | The simply truncated power law model. |

3.4 Functions

VTK-Export

Routines to export fields to the vtk format

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

Some convenient functions for geometric operations

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

Variogram Estimation

Estimate the variogram of a given field with these routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vario_estimate(pos, field[, bin_edges, ...])</code></td>
<td>Estimates the empirical variogram.</td>
</tr>
<tr>
<td><code>vario_estimate_axis(field[, direction, ...])</code></td>
<td>Estimates the variogram along array axis.</td>
</tr>
<tr>
<td><code>standard_bins([pos, dim, latlon, mesh_type, ...])</code></td>
<td>Get standard binning.</td>
</tr>
</tbody>
</table>

3.5 Misc

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EARTH_RADIUS</code></td>
<td>earth radius for WGS84 ellipsoid in km</td>
</tr>
</tbody>
</table>
3.6 gstools.covmodel

GSTools subpackage providing a set of handy covariance models.

Subpackages

| plot | GSTools subpackage providing plotting routines for the covariance models. |

Covariance Base-Class

Class to construct user defined covariance models

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

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 \( \alpha \) is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, \( \text{dim} \) will be set to 3 and anisotropy will be disabled. \( \text{rescale} \) can be set to e.g. earth’s radius, to have a meaningful \( \text{len}_\text{scale} \) parameter. Default: False

• **var_raw** (float or None, optional) – raw variance of the model which will be multiplied with \( \text{CovModel}.\text{var}_\text{factor} \) to result in the actual variance. If given, \( \text{var} \) will be ignored. (This is just for models that override \( \text{CovModel}.\text{var}_\text{factor} \)) Default: None

• **hankel_kw** (dict or None, optional) – Modify the init-arguments of \( \text{hankel}.\text{SymmetricFourierTransform} \) 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: \( \text{hankel}.\text{SymmetricFourierTransform} \) 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 float: 2D rotation angle for pykrige.
pykrige_angle_x float: 3D rotation angle around x for pykrige.
pykrige_angle_y float: 3D rotation angle around y for pykrige.
pykrige_angle_z float: 3D rotation angle around z for pykrige.
pykrige_anis float: 2D anisotropy ratio for pykrige.
pykrige_anis_y float: 3D anisotropy ratio in y direction for pykrige.
pykrige_anis_z float: 3D anisotropy ratio in z direction for pykrige.
pykrige_kwargs dict: 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

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

continues on next page
Table 13 – continued from previous page

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

**anisometrize(pos)**

- Bring a position tuple into the anisotropic coordinate-system.

**calc_integral_scale()**

- Calculate the integral scale of the isotropy 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)
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' : \(\rho(z) = 2 \times ((1 + z)^{0.5} \times 0.5 - 1)\). The smooth approximation of \(l1\) (absolute value) loss. Usually a good choice for robust least squares.
  - 'huber' : \(\rho(z) = z\) if \(z < 1\) else \(2z^{0.5} - 1\). 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) – Dictionairy with the fitted parameter values
- **pcov** (numpy.ndarray) – The estimated covariance of \(popt\) from scipi.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.

```python
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: \( k = ||k|| \)

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

**Parameters**
- \( r \) (float) – Radius of the phase: \( r = ||r|| \)

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

This is given by:
\[
S(k) = \left(\frac{1}{2\pi}\right)^n \int C(r)e^{ik\cdot r}d^n r
\]

Internally, this is calculated by the hankel transformation:
\[
S(k) = \left(\frac{1}{2\pi}\right)^n \frac{(2\pi)^{n/2}}{k^{n/2-1}} \int_0^{\infty} r^{n/2}C(r)J_{n/2-1}(kr)dr
\]

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

**Parameters**
- \( k \) (float) – Radius of the phase: \( k = ||k|| \)

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
GeoStatTools Documentation, Release 1.3.5

property **arg**
   Names of all arguments.
   Type list of str

property **arg_bounds**
   Bounds for all parameters.
   Type dict

   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").

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.
   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 setable.
   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]`

**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]`

**Type** numpy.ndarray
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>property name</td>
<td>The name of the CovModel class.</td>
<td>str</td>
</tr>
<tr>
<td>property nugget</td>
<td>The nugget of the model.</td>
<td>float</td>
</tr>
<tr>
<td>property nugget_bounds</td>
<td>Bounds for the nugget.</td>
<td>list</td>
</tr>
<tr>
<td>Notes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Is a list of 2 or 3 values: [a, b] or [a, b, &lt;type&gt;] where &lt;type&gt; is one of &quot;oo&quot;, &quot;cc&quot;, &quot;oc&quot; or &quot;co&quot; to define if the bounds are open (&quot;o&quot;) or closed (&quot;c&quot;).</td>
<td></td>
</tr>
<tr>
<td>property opt_arg</td>
<td>Names of the optional arguments.</td>
<td>list of str</td>
</tr>
<tr>
<td>property opt_arg_bounds</td>
<td>Bounds for the optional arguments.</td>
<td>dict</td>
</tr>
<tr>
<td>Notes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Keys are the opt-arg names and values are lists of 2 or 3 values: [a, b] or [a, b, &lt;type&gt;] where &lt;type&gt; is one of &quot;oo&quot;, &quot;cc&quot;, &quot;oc&quot; or &quot;co&quot; to define if the bounds are open (&quot;o&quot;) or closed (&quot;c&quot;).</td>
<td></td>
</tr>
<tr>
<td>property pykrige_angle</td>
<td>2D rotation angle for pykrige.</td>
<td></td>
</tr>
<tr>
<td>property pykrige_angle_x</td>
<td>3D rotation angle around x for pykrige.</td>
<td></td>
</tr>
<tr>
<td>property pykrige_angle_y</td>
<td>3D rotation angle around y for pykrige.</td>
<td></td>
</tr>
<tr>
<td>property pykrige_angle_z</td>
<td>3D rotation angle around z for pykrige.</td>
<td></td>
</tr>
<tr>
<td>property pykrige_anis</td>
<td>2D anisotropy ratio for pykrige.</td>
<td></td>
</tr>
<tr>
<td>property pykrige_anis_y</td>
<td>3D anisotropy ratio in y direction for pykrige.</td>
<td></td>
</tr>
<tr>
<td>property pykrige_anis_z</td>
<td>3D anisotropy ratio in z direction for pykrige.</td>
<td></td>
</tr>
<tr>
<td>property pykrige_kwargs</td>
<td>Keyword arguments for pykrige routines.</td>
<td></td>
</tr>
<tr>
<td>property rescale</td>
<td>Rescale factor for the length scale of the model.</td>
<td>float</td>
</tr>
</tbody>
</table>
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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian(dim, var, len_scale, nugget, ...)</td>
<td>The Gaussian covariance model.</td>
</tr>
<tr>
<td>Exponential(dim, var, len_scale, nugget, ...)</td>
<td>The Exponential covariance model.</td>
</tr>
<tr>
<td>Matern(dim, var, len_scale, nugget, anis, ...)</td>
<td>The Matérn covariance model.</td>
</tr>
<tr>
<td>Stable(dim, var, len_scale, nugget, anis, ...)</td>
<td>The stable covariance model.</td>
</tr>
<tr>
<td>Rational(dim, var, len_scale, nugget, ...)</td>
<td>The rational quadratic covariance model.</td>
</tr>
<tr>
<td>Cubic(dim, var, len_scale, nugget, anis, ...)</td>
<td>The Cubic covariance model.</td>
</tr>
<tr>
<td>Circular(dim, var, len_scale, nugget, ...)</td>
<td>The bounded linear covariance model.</td>
</tr>
<tr>
<td>Spherical(dim, var, len_scale, nugget, ...)</td>
<td>The Spherical covariance model.</td>
</tr>
<tr>
<td>HyperSpherical(dim, var, len_scale, ...)</td>
<td>The Hyper-Spherical covariance model.</td>
</tr>
<tr>
<td>SuperSpherical(dim, var, len_scale, ...)</td>
<td>The Super-Spherical covariance model.</td>
</tr>
</tbody>
</table>

**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 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 \( \alpha \) 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 kwars.
- **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

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

continues on next page
Table 15 – continued from previous page

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

**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\( (r, \text{axis}=0) \)
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\( (r, \text{axis}=0) \)
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\( (x\_data, y\_data, \text{anis}=\text{True}, \text{sill}=\text{None}, \text{init\_guess}=\text{default'}, \text{weights}=\text{None}, \text{method}=\text{trf'}, \text{loss}=\text{soft\_l1'}, \text{max\_eval}=\text{None}, \text{return\_r2}=\text{False}, \text{curve\_fit\_kwargs}=\text{None}, *\text{para\_select}) \)
Fitting the variogram-model to an empirical variogram.

Parameters

• \text{x\_data} (numpy.ndarray) – The bin-centers of the empirical variogram.

• \text{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.

• \text{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 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 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 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.
  
  

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 <= 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’
GeoStatTools Documentation, Release 1.3.5

- **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's 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: \( k = ||k|| \)

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:
\[ S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r)e^{ik\cdot r}d^nr \]
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: \( k = \|k\| \)

- **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.

- **angles**
  Rotation angles (in rad) of the model.

  Type: `numpy.ndarray`

- **anis**
  The anisotropy factors of the model.

  Type: `numpy.ndarray`

- **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`

- **arg**
  Names of all arguments.

  Type: `list of str`

- **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`

- **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 setable.

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]`

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.
   Type list

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").

property len_scale_vec
The length scales in each direction.
   Type numpy.ndarray

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]

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.
   Type list

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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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
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, \( \text{anis} \) 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 \( \text{len\_scale} \)s 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, \( \text{len\_scale} \) 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 \( \alpha \) is the great-circle distance, which is equal...
to the spatial distance of two points in 3D. As a consequence, \( \text{dim} \) will be set to 3 and anisotropy will be disabled. \( \text{rescale} \) can be set to e.g. earth’s radius, to have a meaningful \( \text{len}_\text{scale} \) parameter. Default: False

- **\text{var}_\text{raw}** (float or None, optional) – raw variance of the model which will be multiplied with \( \text{CovModel.var}_\text{factor} \) to result in the actual variance. If given, \( \text{var} \) will be ignored. (This is just for models that override \( \text{CovModel.var}_\text{factor} \)) Default: None

- **\text{hankel}_\text{kw}** (dict or None, optional) – Modify the init-arguments of \( \text{hankel.SymmetricFourierTransform} \) 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

- **\*\*\text{opt}_\text{arg}** – Optional arguments are covered by these keyword arguments. If present, they are described in the section *Other Parameters*.

### Attributes

- **\text{angles}** numpy.ndarray: Rotation angles (in rad) of the model.
- **\text{anis}** numpy.ndarray: The anisotropy factors of the model.
- **\text{anis}_\text{bounds}** list: Bounds for the nugget.
- **\text{arg}** list of str: Names of all arguments.
- **\text{arg}_\text{bounds}** dict: Bounds for all parameters.
- **\text{arg}_\text{list}** list of float: Values of all arguments.
- **\text{dim}** int: The dimension of the model.
- **\text{dist}_\text{func}** tuple of callable: pdf, cdf and ppf.
- **\text{do}_\text{rotation}** bool: State if a rotation is performed.
- **\text{field}_\text{dim}** int: The field dimension of the model.
- **\text{hankel}_\text{kw}** dict: \( \text{hankel.SymmetricFourierTransform} \) kwargs.
- **\text{has}_\text{cdf}** bool: State if a cdf is defined by the user.
- **\text{has}_\text{ppf}** bool: State if a ppf is defined by the user.
- **\text{integral}_\text{scale}** float: The main integral scale of the model.
- **\text{integral}_\text{scale}_\text{vec}** numpy.ndarray: The integral scales in each direction.
- **\text{is}_\text{isotropic}** bool: State if a model is isotropic.
- **\text{iso}_\text{arg}** list of str: Names of isotropic arguments.
- **\text{iso}_\text{arg}_\text{list}** list of float: Values of isotropic arguments.
- **\text{latlon}** bool: Whether the model depends on geographical coords.
- **\text{len}_\text{rescaled}** float: The rescaled main length scale of the model.
- **\text{len}_\text{scale}** float: The main length scale of the model.
- **\text{len}_\text{scale}_\text{bounds}** list: Bounds for the length scale.
- **\text{len}_\text{scale}_\text{vec}** numpy.ndarray: The length scales in each direction.
- **\text{name}** str: The name of the \( \text{CovModel} \) class.
- **\text{nugget}** float: The nugget of the model.
- **\text{nugget}_\text{bounds}** list: Bounds for the nugget.
- **\text{opt}_\text{arg}** list of str: Names of the optional arguments.
- **\text{opt}_\text{arg}_\text{bounds}** dict: Bounds for the optional arguments.
- **\text{pykrige}_\text{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

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

continues on next page
Table 16 – continued from previous page

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

- **anisometrize(pos)**
  - Bring a position tuple into the anisotropic coordinate-system.

- **calc_integral_scale()**
  - Calculate the integral scale of the isotropic 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)
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 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:
  - 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.
  - sill will be deslected 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 scipy’s curve_fit. The following keyword values are allowed:
  - ’linear’ (default): \( \rho(z) = z \). Gives a standard least-squares problem.
  - ’soft_l1’: \( \rho(z) = 2 \cdot ((1 + z)^{0.5} - 1) \). 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 \( 2z^{0.5} - 1 \). 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’s 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 \( \text{popt} \) from scipy.optimize.curve_fit. To compute one standard deviation errors on the parameters use \( \text{perr} = \text{np.sqrt(np.diag(pcov))} \).

- **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: \(k = \|k\|\)

**spectral_rad_cdf**\(r\)

Exponential radial spectral cdf.

**spectral_rad_pdf**\(r\)

Radial spectral density of the model.

**spectral_rad_ppf**\(a\)

Exponential radial spectral ppf.

**Notes**

Not defined for 3D.

**spectrum**\(k\)

Spectrum of the covariance model.

This is given by:

\[ S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r)e^{ikr}dr \]

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: \(k = \|k\|\)

**var_factor**()

Factor for the variance.

**vario_axis**\(r, \text{axis}=0\)

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.
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 setable.
Type float

property integral_scale_vec
The integral scales in each direction.

Notes
This is calculated by:
- \( \text{integral_scale_vec}[0] = \text{integral_scale} \)
- \( \text{integral_scale_vec}[1] = \text{integral_scale} \times \text{anis}[0] \)
- \( \text{integral_scale_vec}[2] = \text{integral_scale} \times \text{anis}[1] \)

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 \([a, b, \text{<type>}]\) where \text{<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:

- \(\text{len_scale_vec}[0] = \text{len_scale}\)
- \(\text{len_scale_vec}[1] = \text{len_scale} \times \text{anis}[0]\)
- \(\text{len_scale_vec}[2] = \text{len_scale} \times \text{anis}[1]\)

**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 \([a, b, \text{<type>}]\) where \text{<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 \([a, b, \text{<type>}]\) where \text{<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
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)

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 $s = 1$. $\Gamma$ is the gamma function and $K_\nu$ is the modified Bessel function of the second kind.

$\nu$ is a shape parameter and should be $\geq 0.2$.

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

$$\rho(r) = \exp \left(-\left(s \cdot \frac{r^2}{2\ell^2}\right)\right)$$

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
GeoStatTools Documentation, Release 1.3.5

- **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 $\alpha$ 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 lenght scale.
- **len_scale_vec** *numpy.ndarray*: The length scales in each direction.
- **name** *str*: The name of the CovModel class.
nugget float: The nugget of the model.
nugget_bounds list: Bounds for the nugget.
opt_arg list of str: Names of the optional arguments.
opt_arg_bounds dict: Bounds for the optional arguments.
pykrige_angle 2D rotation angle for pykrige.
pykrige_angle_x 3D rotation angle around x for pykrige.
pykrige_angle_y 3D rotation angle around y for pykrige.
pykrige_angle_z 3D rotation angle around z for pykrige.
pykrige_anis 2D anisotropy ratio for pykrige.
pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.
pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.
pykrige_kwargs Keyword arguments for pykrige routines.
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anisometrize(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td>calc_integral_scale()</td>
<td>Calculate the integral scale of the isotrope model.</td>
</tr>
<tr>
<td>check_arg_bounds()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td>check_dim(dim)</td>
<td>Check the given dimension.</td>
</tr>
<tr>
<td>check_opt_arg()</td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td>cor(h)</td>
<td>Matérn normalized correlation function.</td>
</tr>
<tr>
<td>cor_axis(r[, axis])</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td>cor_spatial(pos)</td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cor_yadrenko(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>correlation(r)</td>
<td>Correlation function of the model.</td>
</tr>
<tr>
<td>cov_axis(r[, axis])</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td>cov_nugget(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>cov_spatial(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cov_yadrenko(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>covariance(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td>default_arg_bounds()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td>default_opt_arg()</td>
<td>Defaults for the optional arguments.</td>
</tr>
<tr>
<td>default_opt_arg_bounds()</td>
<td>Defaults for boundaries of the optional arguments.</td>
</tr>
<tr>
<td>default_rescale()</td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td>fit_variogram(x_data, y_data[, anis, sill, ...])</td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
</tbody>
</table>

continues on next page
Table 17 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fix_dim()</code></td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td><code>isometrize(pos)</code></td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
<tr>
<td><code>ln_spectral_rad_pdf(r)</code></td>
<td>Log radial spectral density of the model.</td>
</tr>
<tr>
<td><code>main_axes()</code></td>
<td>Axes of the rotated coordinate-system.</td>
</tr>
<tr>
<td><code>percentile_scale([per])</code></td>
<td>Calculate the percentile scale of the isotrope model.</td>
</tr>
<tr>
<td><code>plot(func)</code></td>
<td>Plot a function of a the CovModel.</td>
</tr>
<tr>
<td><code>pykrige_vario([args, r])</code></td>
<td>Isotropic variogram of the model for pykrige.</td>
</tr>
<tr>
<td><code>set_arg_bounds([check_args])</code></td>
<td>Set bounds for the parameters of the model.</td>
</tr>
<tr>
<td><code>spectral_density(k)</code></td>
<td>Spectral density of the covariance model.</td>
</tr>
<tr>
<td><code>spectral_rad_pdf(r)</code></td>
<td>Radial spectral density of the model.</td>
</tr>
<tr>
<td><code>spectrum(k)</code></td>
<td>Spectrum of the covariance model.</td>
</tr>
<tr>
<td><code>var_factor()</code></td>
<td>Factor for the variance.</td>
</tr>
<tr>
<td><code>vario_axis(r[, axis])</code></td>
<td>Variogram along axis of anisotropy.</td>
</tr>
<tr>
<td><code>vario_nugget(r)</code></td>
<td>Isotropic variogram of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><code>vario.spatial(pos)</code></td>
<td>Spatial variogram respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>vario_yadrenko(zeta)</code></td>
<td>Yadrenko variogram for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>variogram(r)</code></td>
<td>Isotropic variogram of the model.</td>
</tr>
</tbody>
</table>

- `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.
GeoStatTools Documentation, Release 1.3.5

\texttt{cov_axis}(r, axis=0)

Covariance along axis of anisotropy.

\texttt{cov_nugget}(r)

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

\texttt{cov_spatial}(pos)

Spatial covariance respecting anisotropy and rotation.

\texttt{cov_yadrenko}(zeta)

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

\texttt{covariance}(r)

Covariance of the model.

\texttt{default_arg_bounds}()

Provide default boundaries for arguments.

Given as a dictionary.

\texttt{default_opt_arg}()

Defaults for the optional arguments.

- \{"nu": 1.0\}

\textbf{Returns} Defaults for optional arguments

\textbf{Return type} \texttt{dict}

\texttt{default_opt_arg_bounds}()

Defaults for boundaries of the optional arguments.

- \{"nu": [0.2, 30.0, "cc"]\}

\textbf{Returns} Boundaries for optional arguments

\textbf{Return type} \texttt{dict}

\texttt{default_rescale}()

Provide default rescaling factor.

\texttt{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.

\textbf{Parameters}

- \texttt{x_data} (\texttt{numpy.ndarray}) – The bin-centers of the empirical variogram.

- \texttt{y_data} (\texttt{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.

- \texttt{anis} (\texttt{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

- \texttt{sill} (\texttt{float} or \texttt{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:
  - \texttt{sill} = \texttt{var} + \texttt{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 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 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 <= 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 \( \text{perr} = \text{np.sqrt(np.diag(pcov))} \).

• 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.

```python
fix_dim()
```
Set a fix dimension for the model.

```python
isometrize(pos)
```
Make a position tuple ready for isotropic operations.

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

```python
main_axes()
```
Axes of the rotated coordinate-system.

```python
percentile_scale(per=0.9)
```
Calculate the percentile scale of the isotropic model.

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

```python
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: \( k = \| k \| \)

`spectral_rad_pdf(r)`

Radial spectral density of the model.

`spectrum(k)`

Spectrum of the covariance model.

This is given by:

\[ S(k) = \left( \frac{1}{2\pi} \right)^{n} \int C(r) e^{ikr} dr \]

Internally, this is calculated by the hankel transformation:

\[ S(k) = \left( \frac{1}{2\pi} \right)^{n} \frac{(2\pi)^{n/2}}{k^{n/2-1}} \int_{0}^{\infty} r^{n/2} C(r) J_{n/2-1}(kr) dr \]

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

**Parameters**

- **k** *(float)* – Radius of the phase: \( k = \| k \| \)

`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 setable.
    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]
    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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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:

- \(\text{len_scale_vec}[0] = \text{len_scale}\)
- \(\text{len_scale_vec}[1] = \text{len_scale} \times \text{anis}[0]\)
- \(\text{len_scale_vec}[2] = \text{len_scale} \times \text{anis}[1]\)

**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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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.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]:

\[
\rho(r) = \exp\left(-\left(\frac{s \cdot r}{\ell}\right)^\alpha\right)
\]

Where the standard rescale factor is \( s = 1 \). \( \alpha \) 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, \( \text{anis} \) 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 \( \text{len} \text{.scales} \) 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, \( \text{len} \text{.scale} \) 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 $\alpha$ is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, $\text{dim}$ will be set to 3 and anisotropy will be disabled. $\text{rescale}$ can be set to e.g. earth’s radius, to have a meaningful $\text{len\_scale}$ parameter. Default: False

- **var\_raw** (float or None, optional) – raw variance of the model which will be multiplied with $\text{CovModel.var\_factor}$ to result in the actual variance. If given, var will be ignored. (This is just for models that override $\text{CovModel.var\_factor}$). Default: None

- **hankel\_kw** (dict or None, optional) – Modify the init-arguments of $\text{hankel.SymmetricFourierTransform}$ 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: $\text{hankel.SymmetricFourierTransform}$ 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

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

continues on next page
Table 18 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plot(func)</code></td>
<td>Plot a function of a the CovModel.</td>
</tr>
<tr>
<td><code>pykrige_vario(args, r)</code></td>
<td>Isotropic variogram of the model for pykrige.</td>
</tr>
<tr>
<td><code>set_arg_bounds(check_args)</code></td>
<td>Set bounds for the parameters of the model.</td>
</tr>
<tr>
<td><code>spectral_density(k)</code></td>
<td>Spectral density of the covariance model.</td>
</tr>
<tr>
<td><code>spectral_rad_pdf(r)</code></td>
<td>Radial spectral density of the model.</td>
</tr>
<tr>
<td><code>spectrum(k)</code></td>
<td>Spectrum of the covariance model.</td>
</tr>
<tr>
<td><code>var_factor()</code></td>
<td>Factor for the variance.</td>
</tr>
<tr>
<td><code>vacio_axis(r[, axis])</code></td>
<td>Variogram along axis of anisotropy.</td>
</tr>
<tr>
<td><code>vacio_nugget(r)</code></td>
<td>Isotropic variogram of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><code>vacio_spatial(pos)</code></td>
<td>Spatial variogram respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>vacio_yadrenko(zeta)</code></td>
<td>Yadrenko variogram for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>variogram(r)</code></td>
<td>Isotropic variogram of the model.</td>
</tr>
</tbody>
</table>

**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)**

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)
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 scipy’s curve_fit. The following keyword values are allowed:
  - ’linear’ (default): \( \rho(z) = z \). Gives a standard least-squares problem.
  - ’soft_l1’: \( \rho(z) = 2 \times ((1 + z)^{0.5} - 1) \). 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 \( 2z^{0.5} - 1 \). 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_kwarg** *(dict, optional)* – Other keyword arguments passed to scipy’s 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 \( \text{perr} = \sqrt{\text{np.diag(pcov)}} \).
• **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: $k = ||k||$

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

**spectrum**$(k)$
Spectrum of the covariance model.
This is given by:

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

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: $k = ||k||$

**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.
property anis_bounds

    Bounds for the nugget.

    Notes

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

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 \([a, b, \text{<type>}]\) where \text{<type>} is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

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 setable.
    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]
    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]`

**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
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 \( s = 1 \). \( \alpha \) is a shape parameter and should be \( > 0.5 \).

For \( \alpha \to \infty \) 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, \( \text{anis} \) 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 \( \text{len}\_\text{scale} \)s 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, \( \text{len}\_\text{scale} \) 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 $\alpha$ is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, $\text{dim}$ 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 lenght scale.
- `len_scale_vec` *numpy.ndarray*: The length scales in each direction.
- `name` *str*: The name of the CovModel class.
**nugget** float: The nugget of the model.

**nugget_bounds** list: Bounds for the nugget.

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

**opt_arg_bounds** dict: Bounds for the optional arguments.

**pykrige_angle** 2D rotation angle for pykrige.

**pykrige_angle_x** 3D rotation angle around x for pykrige.

**pykrige_angle_y** 3D rotation angle around y for pykrige.

**pykrige_angle_z** 3D rotation angle around z for pykrige.

**pykrige_anis** 2D anisotropy ratio for pykrige.

**pykrige_anis_y** 3D anisotropy ratio in y direction for pykrige.

**pykrige_anis_z** 3D anisotropy ratio in z direction for pykrige.

**pykrige_kwargs** Keyword arguments for pykrige routines.

**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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>anisometrize</strong>(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td><strong>calc_integral_scale</strong>()</td>
<td>Calculate the integral scale of the isotropic model.</td>
</tr>
<tr>
<td><strong>check_arg_bounds</strong>()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td><strong>check_dim</strong>(dim)</td>
<td>Check the given dimension.</td>
</tr>
<tr>
<td><strong>check_opt_arg</strong>()</td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td><strong>cor</strong>(h)</td>
<td>Rational normalized correlation function.</td>
</tr>
<tr>
<td><strong>cor_axis</strong>(r[, axis])</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td><strong>cor_spatial</strong>(pos)</td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><strong>cor_yadrenko</strong>(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><strong>correlation</strong>(r)</td>
<td>Correlation function of the model.</td>
</tr>
<tr>
<td><strong>cov_axis</strong>(r[, axis])</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td><strong>cov_nugget</strong>(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><strong>cov_spatial</strong>(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><strong>cov_yadrenko</strong>(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><strong>covariance</strong>(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td><strong>default_arg_bounds</strong>()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td><strong>default_opt_arg</strong>()</td>
<td>Defaults for the optional arguments.</td>
</tr>
<tr>
<td><strong>default_opt_arg_bounds</strong>()</td>
<td>Defaults for boundaries of the optional arguments.</td>
</tr>
<tr>
<td><strong>default_rescale</strong>()</td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td><strong>fit_variogram</strong>(x_data, y_data[, anis, sill, ...])</td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fix_dim()</code></td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td><code>isometrize(pos)</code></td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
<tr>
<td><code>ln_spectral_rad_pdf(r)</code></td>
<td>Log radial spectral density of the model.</td>
</tr>
<tr>
<td><code>main_axes()</code></td>
<td>Axes of the rotated coordinate-system.</td>
</tr>
<tr>
<td><code>percentile_scale([per])</code></td>
<td>Calculate the percentile scale of the isotropic model.</td>
</tr>
<tr>
<td><code>plot([func])</code></td>
<td>Plot a function of the CovModel.</td>
</tr>
<tr>
<td><code>pykrige_vario([args, r])</code></td>
<td>Isotropic variogram of the model for pykrige.</td>
</tr>
<tr>
<td><code>set_arg_bounds([check_args])</code></td>
<td>Set bounds for the parameters of the model.</td>
</tr>
<tr>
<td><code>spectral_density(k)</code></td>
<td>Spectral density of the covariance model.</td>
</tr>
<tr>
<td><code>spectral_rad_pdf(r)</code></td>
<td>Radial spectral density of the model.</td>
</tr>
<tr>
<td><code>spectrum(k)</code></td>
<td>Spectrum of the covariance model.</td>
</tr>
<tr>
<td><code>var_factor()</code></td>
<td>Factor for the variance.</td>
</tr>
<tr>
<td><code>vario_axis(r[, axis])</code></td>
<td>Variogram along axis of anisotropy.</td>
</tr>
<tr>
<td><code>vario_nugget(r)</code></td>
<td>Isotropic variogram of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><code>vario_spatial(pos)</code></td>
<td>Spatial variogram respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>vario_yadrenko(zeta)</code></td>
<td>Yadrenko variogram for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>variogram(r)</code></td>
<td>Isotropic variogram of the model.</td>
</tr>
</tbody>
</table>

- **anisometrize(pos)**
  Bring a position tuple into the anisotropic coordinate-system.

- **calc_integral_scale()**
  Calculate the integral scale of the isotropic 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 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:
  
  - 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 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 $1 / (x_{\text{data}} + 1)$
  - list: weights given per bin
  - callable: function applied to $x_{\text{data}}$
  If callable, it must take a 1-d ndarray. Then $weights = f(x_{\text{data}})$. 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's curve_fit. The following keyword values are allowed:
  - 'linear' (default): $\rho(z) = z$. Gives a standard least-squares problem.
  - 'soft_l1': $\rho(z) = 2 * (\sqrt{1 + z} - 1)$. 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's 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](https://docs.python.org/3/library/stdtypes.html#dict)) – Dictionary with the fitted parameter values
- **pcov** ([numpy.ndarray](https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html)) – 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](https://docs.python.org/3/library/functions.html#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.

```python
fix_dim()
```
Set a fix dimension for the model.

```python
isometrize(pos)
```
Make a position tuple ready for isotropic operations.

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

```python
main_axes()
```
Axes of the rotated coordinate-system.

```python
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

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

Parameters

- **func** ([str](https://docs.python.org/3/library/functions.html#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: \( k = ||k|| \)

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

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

This is given by:

\[ S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r)e^{ik\cdot r}dr \]

Internally, this is calculated by the hankel transformation:

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

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

Parameters **k**(float) – Radius of the phase: \( k = ||k|| \)

**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 setable.

    Type `float`

**property integral_scale_vec**

The integral scales in each direction.

    Notes

This is calculated by:

- \( \text{integral_scale_vec}[0] = \text{integral_scale} \)
- \( \text{integral_scale_vec}[1] = \text{integral_scale} \times \text{anis}[0] \)
- \( \text{integral_scale_vec}[2] = \text{integral_scale} \times \text{anis}[1] \)

    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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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:

- \(\text{len_scale_vec}[0] = \text{len_scale}\)
- \(\text{len_scale_vec}[1] = \text{len_scale} \times \text{anis}[0]\)
- \(\text{len_scale_vec}[2] = \text{len_scale} \times \text{anis}[1]\)

---

**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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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.Cubic**

```python
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]:

\[
\rho(r) = \begin{cases} 
1 - 7 \left(\frac{s}{\ell}\right)^2 + \frac{35}{4} \left(\frac{s}{\ell}\right)^3 - \frac{7}{2} \left(\frac{s}{\ell}\right)^5 + \frac{3}{4} \left(\frac{s}{\ell}\right)^7 & r < \frac{\ell}{s} \\
0 & r \geq \frac{\ell}{s}
\end{cases}
\]

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 $\alpha$ 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": 10000, "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.

pykridge_angle 2D rotation angle for pykridge.
pykridge_angle_x 3D rotation angle around x for pykridge.
pykridge_angle_y 3D rotation angle around y for pykridge.
pykridge_angle_z 3D rotation angle around z for pykridge.

pykridge_anis 2D anisotropy ratio for pykridge.
pykridge_anis_y 3D anisotropy ratio in y direction for pykridge.
pykridge_anis_z 3D anisotropy ratio in z direction for pykridge.

pykridge_kwargs Keyword arguments for pykridge 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

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.
cor(h) Spherical normalized correlation function.
cor_axis(r[, axis]) 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]) 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.
default_opt_arg() Provide default optional arguments by the user.
default_opt_arg_bounds() Provide default boundaries for optional arguments.
default_rescale() Provide default rescaling factor.
fit_variogram(x_data, y_data[, anis, sill, ...]) Fitting the variogram-model to an empirical variogram.
fix_dim() Set a fix dimension for the model.
isometrize(pos) Make a position tuple ready for isotropic operations.

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

**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)**
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 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 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:
  
  - 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 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 scipy’s curve_fit. The following keyword values are allowed:
  - ’linear’ (default): \( \rho(z) = z \). Gives a standard least-squares problem.
  - ’soft_l1’: \( \rho(z) = 2 \times ((1 + z)^{0.5} - 1) \). 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 \( 2z^{0.5} - 1 \). 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_kwarg** (dict, optional) – Other keyword arguments passed to scipy’s 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 \( \text{perr} = \text{np.sqrt(np.diag(pcov))} \).
• **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.

```python
fix_dim()
```

Set a fix dimension for the model.

```python
isometrize(pos)
```

Make a position tuple ready for isotropic operations.

```python
ln_spectral_rad_pdf(r)
```

Log radial spectral density of the model.

```python
main_axes()
```

Axes of the rotated coordinate-system.

```python
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

```python
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: \( k = ||k|| \)

spectral_rad_pdf(r)

Radial spectral density of the model.

spectrum(k)

Spectrum of the covariance model.

This is given by:

\[ S(k) = \left(\frac{1}{2\pi}\right)^n \int C(r)e^{ikr}d^nr \]

Internally, this is calculated by the hankel transformation:

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

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

Parameters **k** (float) – Radius of the phase: \( k = ||k|| \)

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.

varioagrampen(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 setable.
    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]

    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:

- \( \text{len_scale_vec}[0] = \text{len_scale} \)
- \( \text{len_scale_vec}[1] = \text{len_scale} \times \text{anis}[0] \)
- \( \text{len_scale_vec}[2] = \text{len_scale} \times \text{anis}[1] \)

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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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
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 \( \ell \).

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

- \texttt{dim} (int, optional) – dimension of the model. Default: 3
- \texttt{var} (float, optional) – variance of the model (the nugget is not included in “this” variance) Default: 1.0
- \texttt{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, \texttt{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
- \texttt{nugget} (float, optional) – nugget of the model. Default: 0.0
- \texttt{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 \texttt{len\_scales} are given. Default: 1.0
- \texttt{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
- \texttt{integral\_scale} (float or list or None, optional) – If given, \texttt{len\_scale} will be ignored and recalculated, so that the integral scale of the model matches the given one. Default: None
- \texttt{rescale} (float or None, optional) – Optional rescaling factor to divide the length scale with. This could be used for unit convertion 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 \( \alpha \) is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, \( \text{dim} \) will be set to 3 and anisotropy will be disabled. \( \text{rescale} \) can be set to e.g. earth’s radius, to have a meaningful \( \text{len\_scale} \) parameter. Default: False

**var\_raw** (float or None, optional) – raw variance of the model which will be multiplied with \( \text{CovModel.var\_factor} \) to result in the actual variance. If given, \( \text{var} \) will be ignored. (This is just for models that override \( \text{CovModel.var\_factor} \)) Default: None

**hankel\_kw** (dict or None, optional) – Modify the init-arguments of \( \text{hankel.SymmetricFourierTransform} \) 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: \( \text{hankel.SymmetricFourierTransform} \) 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anisometrize(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td>calc_integral_scale()</td>
<td>Calculate the integral scale of the isotrope model.</td>
</tr>
<tr>
<td>check_arg_bounds()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td>check_dim(dim)</td>
<td>Linear model is only valid in 1D.</td>
</tr>
<tr>
<td>check_opt_arg()</td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td>cor(h)</td>
<td>Linear normalized correlation function.</td>
</tr>
<tr>
<td>cor_axis[r[, axis]]</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td>cor_spatial(pos)</td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cor_yadrenko(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>correlation(r)</td>
<td>Correlation function of the model.</td>
</tr>
<tr>
<td>cov_axis[r[, axis]]</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td>cov_nugget(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>cov_spatial(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cov_yadrenko(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>covariance(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td>default_arg_bounds()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td>default_opt_arg()</td>
<td>Provide default optional arguments by the user.</td>
</tr>
<tr>
<td>default_opt_arg_bounds()</td>
<td>Provide default boundaries for optional arguments.</td>
</tr>
<tr>
<td>default_rescale()</td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td>fit_variogram(x_data, y_data[, anis, sill, ...])</td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
<tr>
<td>fix_dim()</td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td>isometrize(pos)</td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
</tbody>
</table>
Table 21 – continued from previous page

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

**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_kwars=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 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 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 ndaray. 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's 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': \(\rho(z) = z \text{ if } z <= 1 \text{ else } 2*\text{z}**0.5 - 1\). 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 ndaray \(z=f**2\) 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's 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)* – Dictionay 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 = \|k\|

spectral_rad_pdf(r)
Radial spectral density of the model.

spectrum(k)
Spectrum of the covariance model.
This is given by:

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

Internally, this is calculated by the hankel transformation:

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

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

Parameters k (float) – Radius of the phase: $k = \|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.
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 setable.
    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]
    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]`

*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.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, **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 \( \ell \).

**Notes**

This model is given by the following correlation function [Webster2007]:

\[
\rho(r) = \begin{cases} 
\frac{2}{\pi} \cdot \left( \cos^{-1} \left( \frac{s \cdot \tau}{\ell} \right) - s \cdot \tau \cdot \sqrt{1 - (s \cdot \tau)^2} \right) & r < \frac{\ell}{s} \\
0 & r \geq \frac{\ell}{s}
\end{cases}
\]

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

---

3.6. gstools.covmodel
• **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 $\alpha$ 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>anisometrize</strong>(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td><strong>calc_integral_scale</strong>()</td>
<td>Calculate the integral scale of the isotrope model.</td>
</tr>
<tr>
<td><strong>check_arg_bounds</strong>()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td><strong>check_dim</strong>(dim)</td>
<td>Circular model is only valid in 1D and 2D.</td>
</tr>
<tr>
<td><strong>check_opt_arg</strong>()</td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td><strong>cor</strong>(h)</td>
<td>Circular normalized correlation function.</td>
</tr>
<tr>
<td><strong>cor_axis</strong>(r[, axis])</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td><strong>cor_spatial</strong>(pos)</td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><strong>cor_yadrenko</strong>(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><strong>correlation</strong>(r)</td>
<td>Correlation function of the model.</td>
</tr>
<tr>
<td><strong>cov_axis</strong>(r[, axis])</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td><strong>cov_nugget</strong>(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><strong>cov_spatial</strong>(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><strong>cov_yadrenko</strong>(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><strong>covariance</strong>(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td><strong>default_arg_bounds</strong>()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td><strong>default_opt_arg</strong>()</td>
<td>Provide default optional arguments by the user.</td>
</tr>
<tr>
<td><strong>default_opt_arg_bounds</strong>()</td>
<td>Provide default boundaries for optional arguments.</td>
</tr>
<tr>
<td><strong>default_rescale</strong>()</td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td><strong>fit_variogram</strong>(x_data, y_data[, anis, sill, ...])</td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
<tr>
<td><strong>fix_dim</strong>()</td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td><strong>isometrize</strong>(pos)</td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
</tbody>
</table>

continues on next page
Table 22 – continued from previous page

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

**anisometrize(pos)**

Bring a position tuple into the anisotropic coordinate-system.

**calc_integral_scale()**

Calculate the integral scale of the isotropic 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\( (x\_data, y\_data, \text{anis=True, sill=None, init\_guess='default', weights=None,} \text{method='trf', loss='soft\_l1', max\_eval=None, return\_r2=False,} \text{curve\_fit\_kwargs=None, **para\_select}) \)
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 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:
  
  - 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 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': \(\rho(z) = 2 \cdot ((1 + z)^{0.5} - 1)\). 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 \(2z^{0.5} - 1\). 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_kwarg**(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 \(\text{perr} = \text{np}.\sqrt{\text{np}.\text{diag}(\text{pcov})}\).
• **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: \( k = ||k|| \)

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

**spectrum**(`k`)
Spectrum of the covariance model.
This is given by:
\[
S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r) e^{i k \cdot r} d^n r
\]
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: \( k = ||k|| \)

**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.

**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 setable.

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]`

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:

- \( \text{len_scale_vec}[0] = \text{len_scale} \)
- \( \text{len_scale_vec}[1] = \text{len_scale} \times \text{anis}[0] \)
- \( \text{len_scale_vec}[2] = \text{len_scale} \times \text{anis}[1] \)

**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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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 \([a, b, \text{<type}>]\) where \(<\text{type}>\) 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
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 \( \ell \).

Notes

This model is given by the following correlation function [Webster2007]:

\[
\rho(r) = \begin{cases} 
1 - \frac{3}{2} \cdot \frac{s \cdot r}{\ell} + \frac{1}{2} \cdot \left( \frac{s \cdot r}{\ell} \right)^3 & r < \frac{\ell}{s} \\
0 & r \geq \frac{\ell}{s}
\end{cases}
\]

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, \( \text{anis} \) 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 \( \text{len}\_\text{scale} \) 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, \( \text{len}\_\text{scale} \) 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 $\alpha$ 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anisometrize(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td>calc_integral_scale()</td>
<td>Calculate the integral scale of the isotrope model.</td>
</tr>
<tr>
<td>check_arg_bounds()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td>check_dim(dim)</td>
<td>Spherical model is only valid in 1D, 2D and 3D.</td>
</tr>
<tr>
<td>check_opt_arg()</td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td>cor_axis(r[, axis])</td>
<td>Spherical normalized correlation function.</td>
</tr>
<tr>
<td>cor_spatial(pos)</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td>cor_yadrenko(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>correlation(r)</td>
<td>Correlation function of the model.</td>
</tr>
<tr>
<td>cov_axis(r[, axis])</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td>cov_nugget(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>cov_spatial(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cov_yadrenko(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>covariance(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td>default_arg_bounds()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td>default_opt_arg()</td>
<td>Provide default optional arguments by the user.</td>
</tr>
<tr>
<td>default_opt_arg_bounds()</td>
<td>Provide default boundaries for optional arguments.</td>
</tr>
<tr>
<td>default_rescale()</td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td>fit_variogram(x_data, y_data[, anis, sill, ...])</td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
<tr>
<td>fix_dim()</td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td>isometrize(pos)</td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
</tbody>
</table>

continues on next page
Table 23 – continued from previous page

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

**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\( (x\_data, y\_data, anis=True, sill=None, init\_guess='default', weights=None, method='trf', loss='soft_l1', max\_eval=None, return\_r2=False, curve\_fit\_kwags=None, \*\*\*para\_select) \)
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 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 scipy’s curve_fit. The following keyword values are allowed:
  - 'linear' (default): \( \rho(z) = z \). Gives a standard least-squares problem.
  - 'soft_l1': \( \rho(z) = 2 \cdot ((1 + z)^{0.5} - 1) \). 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 \( 2z^{0.5} - 1 \). 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: \( 100 \times n \).

• 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’s 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 \( \text{perr} = \text{np.sqrt(np.diag(pcov))} \).
- **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.

```python
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(pos=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: $k = ||k||$

spectral_rad_pdf(r)
Radial spectral density of the model.

spectrum(k)
Spectrum of the covariance model.
This is given by:
$$S(k) = \left(\frac{1}{2\pi}\right)^n \int C(r)e^{ik\cdot r}d^n r$$
Internally, this is calculated by the hankel transformation:
$$S(k) = \left(\frac{1}{2\pi}\right)^n \cdot \left(\frac{2\pi}{k^{n/2-1}}\right)^{n/2} \int_0^\infty r^{n/2}C(r)J_{n/2-1}(kr)dr$$
Where $C(r)$ is the covariance function of the model.

Parameters k (float) – Radius of the phase: $k = ||k||$

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 setable.
    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]

    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]`

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.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 \( \ell \).

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]:

\[
\rho(r) = \begin{cases} 
1 - s \cdot \frac{r}{\ell}, & \frac{r}{\ell} \leq 1 \\
0, & \frac{r}{\ell} > 1
\end{cases}
\]

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 \( \alpha \) is the great-circle distance, which is equal to the spatial distance of two points in 3D. As a consequence, \( \text{dim} \) will be set to 3 and anisotropy will be disabled. \( \text{rescale} \) can be set to e.g. earth’s radius, to have a meaningful \( \text{len\_scale} \) parameter. Default: False

• **var\_raw** (float or None, optional) – raw variance of the model which will be multiplied with \( \text{CovModel.var\_factor} \) to result in the actual variance. If given, \( \text{var} \) will be ignored. (This is just for models that override \( \text{CovModel.var\_factor} \)) Default: None

• **hankel\_kw** (dict or None, optional) – Modify the init-arguments of \( \text{hankel.SymmetricFourierTransform} \) 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: \( \text{hankel.SymmetricFourierTransform} \) 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**

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

continues on next page
Table 24 – continued from previous page

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

**anisometrize(pos)**
Bring a position tuple into the anisotropic coordinate-system.

**calc_integral_scale()**
Calculate the integral scale of the isotropic 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:
  - 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": false
}
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** *(str, optional) – Algorithm to perform minimization. Only *trf* or *dogbox*.
  - ‘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’s curve_fit. The following keyword values are allowed:
  - ‘linear’ (default) : \( \rho(z) = z \). Gives a standard least-squares problem.
  - ‘soft_l1’ : \( \rho(z) = 2 \times ((1 + z)^{0.5} - 1) \). 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 \( 2z^{0.5} - 1 \). 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’s 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 \( \text{popt} \) from scipy.optimize.curve_fit. To compute one standard deviation errors on the parameters use \( \text{perr} = \text{np.sqrt(np.diag(pcov))} \)
• **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:
\[ \hat{S}(k) = \frac{S(k)}{\sigma^2} \]
Where \( S(k) \) is the spectrum of the covariance model.

Parameters

- **k** (float) – Radius of the phase: \( k = ||k|| \)

spectral_rad_pdf(r)
Radial spectral density of the model.

spectrum(k)
Spectrum of the covariance model.
This is given by:
\[ S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r)e^{ikr}dr \]
Internally, this is calculated by the hankel transformation:
\[ S(k) = \left( \frac{1}{2\pi} \right)^n \cdot \left( \frac{2\pi}{k^{n/2}} \right)^{n/2-1} \int_0^\infty r^{n/2}C(r)J_{n/2-1}(kr)dr \]
Where \( C(r) \) is the covariance function of the model.

Parameters

- **k** (float) – Radius of the phase: \( k = ||k|| \)

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.
**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").

**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").

**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**

**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 setable.
  
  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]

  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
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len_scale_vec</code></td>
<td>The length scales in each direction.</td>
<td>numpy.ndarray</td>
</tr>
<tr>
<td><code>name</code></td>
<td>The name of the CovModel class.</td>
<td>str</td>
</tr>
<tr>
<td><code>nugget</code></td>
<td>The nugget of the model.</td>
<td>float</td>
</tr>
<tr>
<td><code>nugget_bounds</code></td>
<td>Bounds for the nugget.</td>
<td>list</td>
</tr>
<tr>
<td><code>opt_arg</code></td>
<td>Names of the optional arguments.</td>
<td>list of str</td>
</tr>
<tr>
<td><code>opt_arg_bounds</code></td>
<td>Bounds for the optional arguments.</td>
<td>dict</td>
</tr>
<tr>
<td><code>pykrige_angle</code></td>
<td>2D rotation angle for pykrige.</td>
<td></td>
</tr>
<tr>
<td><code>pykrige_angle_x</code></td>
<td>3D rotation angle around x for pykrige.</td>
<td></td>
</tr>
<tr>
<td><code>pykrige_angle_y</code></td>
<td>3D rotation angle around y for pykrige.</td>
<td></td>
</tr>
<tr>
<td><code>pykrige_angle_z</code></td>
<td>3D rotation angle around z for pykrige.</td>
<td></td>
</tr>
</tbody>
</table>
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.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 $\ell$. 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]:

$$\rho(r) = \begin{cases} 
1 - s \cdot \frac{r}{\ell} \cdot \frac{\frac{1}{2} - \nu, \frac{3}{2} (s \cdot \frac{r}{\ell})^2}{\frac{1}{2} - \nu, 1} & r < \frac{\ell}{s} \\
0 & r \geq \frac{\ell}{s}
\end{cases}$$

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: $[(\text{dim}\cdot 1)/2, 50]$ Default: $(\text{dim}\cdot 1)/2$
- **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
\[ r = 2 \sin(\alpha/2) \]

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 lenght scale.
- `len_scale_vec` `numpy.ndarray`: The length scales in each direction.
- `name` `str`: The name of the CovModel class.
**nugget float**: The nugget of the model.

**nugget_bounds list**: Bounds for the nugget.

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

**opt_arg Bounds dict**: Bounds for the optional arguments.

**pykrige_angle**: 2D rotation angle for pykrige.

**pykrige_angle_x**: 3D rotation angle around x for pykrige.

**pykrige_angle_y**: 3D rotation angle around y for pykrige.

**pykrige_angle_z**: 3D rotation angle around z for pykrige.

**pykrige_anis**: 2D anisotropy ratio for pykrige.

**pykrige_anis_y**: 3D anisotropy ratio in y direction for pykrige.

**pykrige_anis_z**: 3D anisotropy ratio in z direction for pykrige.

**pykrige_kwargs**: Keyword arguments for pykrige routines.

**rescale**: 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anisometrize(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td>calc_integral_scale()</td>
<td>Calculate the integral scale of the isotropic model.</td>
</tr>
<tr>
<td>check_arg_bounds()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td>check_dim(dim)</td>
<td>Check the given dimension.</td>
</tr>
<tr>
<td>check_opt_arg()</td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td>cor(h)</td>
<td>Super-Spherical normalized correlation function.</td>
</tr>
<tr>
<td>cor_axis(r[, axis])</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td>cor_spatial(pos)</td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cor_yadrenko(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>correlation(r)</td>
<td>Correlation function of the model.</td>
</tr>
<tr>
<td>cov_axis(r[, axis])</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td>cov_nugget(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>cov_spatial(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cov_yadrenko(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>covariance(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td>default_arg_bounds()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td>default_opt_arg()</td>
<td>Defaults for the optional arguments.</td>
</tr>
<tr>
<td>default_opt_arg_bounds()</td>
<td>Defaults for boundaries of the optional arguments.</td>
</tr>
<tr>
<td>default_rescale()</td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td>fit_variogram(x_data, y_data[, anis, sill, ...])</td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
<tr>
<td>Method Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>fix_dim()</code></td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td><code>isometrize(pos)</code></td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
<tr>
<td><code>ln_spectral_rad_pdf(r)</code></td>
<td>Log radial spectral density of the model.</td>
</tr>
<tr>
<td><code>main_axes()</code></td>
<td>Axes of the rotated coordinate-system.</td>
</tr>
<tr>
<td><code>percentile_scale([per])</code></td>
<td>Calculate the percentile scale of the isotropic model.</td>
</tr>
<tr>
<td><code>plot([func])</code></td>
<td>Plot a function of a the CovModel.</td>
</tr>
<tr>
<td><code>pykrige_vario([args, r])</code></td>
<td>Isotropic variogram of the model for pykrige.</td>
</tr>
<tr>
<td><code>set_arg_bounds([check_args])</code></td>
<td>Set bounds for the parameters of the model.</td>
</tr>
<tr>
<td><code>spectral_density(k)</code></td>
<td>Spectral density of the covariance model.</td>
</tr>
<tr>
<td><code>spectral_rad_pdf(r)</code></td>
<td>Radial spectral density of the model.</td>
</tr>
<tr>
<td><code>spectrum(k)</code></td>
<td>Spectrum of the covariance model.</td>
</tr>
<tr>
<td><code>var_factor()</code></td>
<td>Factor for the variance.</td>
</tr>
<tr>
<td><code>vario_axis(r[, axis])</code></td>
<td>Variogram along axis of anisotropy.</td>
</tr>
<tr>
<td><code>vario_nugget(r)</code></td>
<td>Isotropic variogram of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><code>vario_spatial(pos)</code></td>
<td>Spatial variogram respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>vario_yadrenko(zeta)</code></td>
<td>Yadrenko variogram for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>variogram(r)</code></td>
<td>Isotropic variogram of the model.</td>
</tr>
</tbody>
</table>

- `anisometrize(pos)`
  - Bring a position tuple into the anisotropic coordinate-system.

- `calc_integral_scale()`
  - Calculate the integral scale of the isotropic 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** \( (r, \text{axis}=0) \)
Covariance along axis of anisotropy.

**cov_nugget** \( (r) \)
Isotropic covariance of the model respecting the nugget at \( r=0 \).

**cov_spatial** \( (\text{pos}) \)
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": \((\text{dim}-1)/2\)\}

  **Returns**
  Defaults for optional arguments

  **Return type**
  dict

**default_opt_arg_bounds**
Defaults for boundaries of the optional arguments.
- \{"nu": \[\((\text{dim}-1)/2, 50.0\)\]\}

  **Returns**
  Boundaries for optional arguments

  **Return type**
  dict

**default_rescale**
Provide default rescaling factor.

**fit_variogram** \( (\text{x_data}, \text{y_data}, \text{anis}=\text{True}, \text{sill}=\text{None}, \text{init_guess}=\text{'default'}, \text{weights}=\text{None}, \text{method}=\text{'trf'}, \text{loss}=\text{'soft_l1'}, \text{max_eval}=\text{None}, \text{return_r2}=\text{False}, \text{curve_fit_kwargs}=\text{None}, **\text{para_select})\)
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 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:
  - 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 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 <= 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.

```python
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: \( k = ||k|| \)

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

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

This is given by:

\[ S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r)e^{ik\cdot r}dr \]

Internally, this is calculated by the hankel transformation:

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

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

Parameters k (float) – Radius of the phase: \( k = ||k|| \)

**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.
**vário_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 setable.
    Type float

property integral_scale_vec
    The integral scales in each direction.
    Type numpy.ndarray

Notes
This is calculated by:

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

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
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]:

\[ \rho(r) = \Gamma(\nu + 1) \frac{J_\nu(s \cdot \frac{r}{\ell})}{(s \cdot \frac{r}{2\pi})^\nu} \]

Where the standard rescale factor is \( s = 1 \). \( \Gamma \) is the gamma function and \( J_\nu \) 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 unstable for \( \nu \to \frac{d}{2} - 1 \).

For \( \nu = \frac{1}{2} \) (valid in d=1,2,3) we get the so-called ‘Wave’ model:

\[ \rho(r) = \frac{\sin(s \cdot \frac{r}{\ell})}{s \cdot \frac{r}{\ell}} \]

References

Parameters

- **nu** (float, optional) – Shape parameter. Standard range: \([\text{dim}/2 - 1, 50]\) 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 \([e_y, e_z]\).
  - \( e_y = \frac{l_y}{l_x} \)
  - \( e_z = \frac{l_z}{l_x} \)
  If only one value is given in 3D, \( e_y \) will be set to 1. This value will be ignored, if multiple \( \text{len} \_\text{scale}s \) 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 \( \alpha \) 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anisometrize(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td>calc_integral_scale()</td>
<td>Calculate the integral scale of the isotrope model.</td>
</tr>
<tr>
<td>check_arg_bounds()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td>check_dim(dim)</td>
<td>Check the given dimension.</td>
</tr>
<tr>
<td>check_opt_arg()</td>
<td>Check the optional arguments.</td>
</tr>
<tr>
<td>cor(r)</td>
<td>J-Bessel correlation.</td>
</tr>
<tr>
<td>cor_axis(r[, axis])</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td>cor_spatial(pos)</td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cor_yadrenko(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>correlation(r)</td>
<td>Correlation function of the model.</td>
</tr>
<tr>
<td>cov_axis(r[, axis])</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td>cov_nugget(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>cov_spatial(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cov_yadrenko(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>covariance(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td>default_arg_bounds()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td>default_opt_arg()</td>
<td>Defaults for the optional arguments.</td>
</tr>
</tbody>
</table>

continues on next page
Table 26 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>default_opt_arg_bounds()</code></td>
<td>Defaults for boundaries of the optional arguments.</td>
</tr>
<tr>
<td><code>default_rescale()</code></td>
<td>Provides default rescaling factor.</td>
</tr>
<tr>
<td><code>fit_variogram(x_data, y_data[, anis, sill, ...])</code></td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
<tr>
<td><code>fix_dim()</code></td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td><code>isometrize(pos)</code></td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
<tr>
<td><code>ln_spectral_rad_pdf(r)</code></td>
<td>Log radial spectral density of the model.</td>
</tr>
<tr>
<td><code>main_axes()</code></td>
<td>Axes of the rotated coordinate-system.</td>
</tr>
<tr>
<td><code>percentile_scale([per])</code></td>
<td>Calculate the percentile scale of the isotrope model.</td>
</tr>
<tr>
<td><code>plot([func])</code></td>
<td>Plot a function of the CovModel.</td>
</tr>
<tr>
<td><code>pykrige_vario([args, r])</code></td>
<td>Isotropic variogram of the model for pykrige.</td>
</tr>
<tr>
<td><code>set_arg_bounds([check_args])</code></td>
<td>Set bounds for the parameters of the model.</td>
</tr>
<tr>
<td><code>spectral_density(k)</code></td>
<td>Spectral density of the covariance model.</td>
</tr>
<tr>
<td><code>spectral_rad_pdf(r)</code></td>
<td>Radial spectral density of the model.</td>
</tr>
<tr>
<td><code>spectrum(k)</code></td>
<td>Spectrum of the covariance model.</td>
</tr>
<tr>
<td><code>var_factor()</code></td>
<td>Factor for the variance.</td>
</tr>
<tr>
<td><code>vario_axis(r[, axis])</code></td>
<td>Variogram along axis of anisotropy.</td>
</tr>
<tr>
<td><code>vario_nugget(r)</code></td>
<td>Isotropic variogram of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><code>vario_spatial(pos)</code></td>
<td>Spatial variogram respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>vario_yadrenko(zeta)</code></td>
<td>Yadrenko variogram for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>variogram(r)</code></td>
<td>Isotropic variogram of the model.</td>
</tr>
</tbody>
</table>

- `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_kwargss=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 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 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=True, 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 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 scipy’s curve_fit. The following keyword values are allowed:
  - 'linear' (default): \( \rho(z) = z \). Gives a standard least-squares problem.
  - 'soft_l1': \( \rho(z) = 2 \cdot ((1 + z)^{0.5} - 1) \). 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 \( 2z^{0.5} - 1 \). 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’s 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.

```python
fix_dim()
```

Set a fix dimension for the model.

```python
isometrize(pos)
```

Make a position tuple ready for isotropic operations.

```python
ln_spectral_rad_pdf(r)
```

Log radial spectral density of the model.

```python
main_axes()
```

Axes of the rotated coordinate-system.

```python
percentile_scale(per=0.9)
```

Calculate the percentile scale of the isotropic model.

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

```python
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: $k = \|k\|$  

### `spectral_rad_pdf(r)`
Radial spectral density of the model.

### `spectrum(k)`
Spectrum of the covariance model.

This is given by:

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

Internally, this is calculated by the hankel transformation:

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

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

**Parameters**

- **k** (`float`) – Radius of the phase: $k = \|k\|$  

### `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 kwars.

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 setable.

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]

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:
- \( \text{len_scale_vec}[0] = \text{len_scale} \)
- \( \text{len_scale_vec}[1] = \text{len_scale} \times \text{anis}[0] \)
- \( \text{len_scale_vec}[2] = \text{len_scale} \times \text{anis}[1] \)

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
Truncated Power Law Covariance Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPLGaussian(dim, var, len_scale, nugget, ...)</td>
<td>Truncated-Power-Law with Gaussian modes.</td>
</tr>
<tr>
<td>TPLExponential(dim, var, len_scale, ...)</td>
<td>Truncated-Power-Law with Exponential modes.</td>
</tr>
<tr>
<td>TPLStable(dim, var, len_scale, nugget, ...)</td>
<td>Truncated-Power-Law with Stable modes.</td>
</tr>
<tr>
<td>TPLSimple(dim, var, len_scale, nugget, ...)</td>
<td>The simply truncated power law model.</td>
</tr>
</tbody>
</table>

**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:

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

\[
\sigma^2_{\ell_{\text{low}}, \ell_{\text{up}}} = \frac{C \cdot (\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H})}{2H}
\]

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:

- `len_up = len_low + len_scale`

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

- \(E_s(x)\) is the exponential integral.
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(α/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.
<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angles numpy.ndarray</td>
<td>Rotation angles (in rad) of the model.</td>
</tr>
<tr>
<td>anis numpy.ndarray</td>
<td>The anisotropy factors of the model.</td>
</tr>
<tr>
<td>anis_bounds list</td>
<td>Bounds for the nugget.</td>
</tr>
<tr>
<td>arg list of str</td>
<td>Names of all arguments.</td>
</tr>
<tr>
<td>arg_bounds dict</td>
<td>Bounds for all parameters.</td>
</tr>
<tr>
<td>arg_list list of float</td>
<td>Values of all arguments.</td>
</tr>
<tr>
<td>dim int</td>
<td>The dimension of the model.</td>
</tr>
<tr>
<td>dist_func tuple of callable</td>
<td>pdf, cdf and ppf.</td>
</tr>
<tr>
<td>do_rotation bool</td>
<td>State if a rotation is performed.</td>
</tr>
<tr>
<td>field_dim int</td>
<td>The field dimension of the model.</td>
</tr>
<tr>
<td>hankel_kw dict</td>
<td><code>hankel.SymmetricFourierTransform</code> kwargs.</td>
</tr>
<tr>
<td>has_cdf bool</td>
<td>State if a cdf is defined by the user.</td>
</tr>
<tr>
<td>has_ppf bool</td>
<td>State if a ppf is defined by the user.</td>
</tr>
<tr>
<td>integral_scale float</td>
<td>The main integral scale of the model.</td>
</tr>
<tr>
<td>integral_scale_vec numpy.ndarray</td>
<td>The integral scales in each direction.</td>
</tr>
<tr>
<td>is_isotropic bool</td>
<td>State if a model is isotropic.</td>
</tr>
<tr>
<td>iso_arg list of str</td>
<td>Names of isotropic arguments.</td>
</tr>
<tr>
<td>iso_arg_list list of float</td>
<td>Values of isotropic arguments.</td>
</tr>
<tr>
<td>latlon bool</td>
<td>Whether the model depends on geographical coords.</td>
</tr>
<tr>
<td>len_low_rescaled float</td>
<td>Lower length scale truncation rescaled.</td>
</tr>
<tr>
<td>len_rescaled float</td>
<td>The rescaled main length scale of the model.</td>
</tr>
<tr>
<td>len_scale float</td>
<td>The main length scale of the model.</td>
</tr>
<tr>
<td>len_scale_bounds list</td>
<td>Bounds for the length scale.</td>
</tr>
<tr>
<td>len_scale_vec numpy.ndarray</td>
<td>The length scales in each direction.</td>
</tr>
<tr>
<td>len_up float</td>
<td>Upper length scale truncation of the model.</td>
</tr>
<tr>
<td>len_up_rescaled float</td>
<td>Upper length scale truncation rescaled.</td>
</tr>
<tr>
<td>name str</td>
<td>The name of the CovModel class.</td>
</tr>
<tr>
<td>nugget float</td>
<td>The nugget of the model.</td>
</tr>
<tr>
<td>nugget_bounds list</td>
<td>Bounds for the nugget.</td>
</tr>
<tr>
<td>opt_arg list of str</td>
<td>Names of the optional arguments.</td>
</tr>
<tr>
<td>opt_arg_bounds dict</td>
<td>Bounds for the optional arguments.</td>
</tr>
<tr>
<td>pykrige_angle 2D rotation angle for pykrige.</td>
<td></td>
</tr>
<tr>
<td>pykrige_angle_x 3D rotation angle around x for pykrige.</td>
<td></td>
</tr>
<tr>
<td>pykrige_angle_y 3D rotation angle around y for pykrige.</td>
<td></td>
</tr>
<tr>
<td>pykrige_angle_z 3D rotation angle around z for pykrige.</td>
<td></td>
</tr>
<tr>
<td>pykrige_anis 2D anisotropy ratio for pykrige.</td>
<td></td>
</tr>
<tr>
<td>pykrige_anis_y 3D anisotropy ratio in y direction for pykrige.</td>
<td></td>
</tr>
<tr>
<td>pykrige_anis_z 3D anisotropy ratio in z direction for pykrige.</td>
<td></td>
</tr>
</tbody>
</table>
**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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anisometrize(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td>calc_integral_scale()</td>
<td>Calculate the integral scale of the isotrope model.</td>
</tr>
<tr>
<td>check_arg_bounds()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td>check_dim(dim)</td>
<td>Check the given dimension.</td>
</tr>
<tr>
<td>check_opt_arg()</td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td>cor(h)</td>
<td>TPL with Gaussian modes - normalized correlation function.</td>
</tr>
<tr>
<td>cor_axis(r[, axis])</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td>cor_spatial(pos)</td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cor_yadrenko(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>correlation(r)</td>
<td>TPL with Gaussian modes - correlation function.</td>
</tr>
<tr>
<td>cov_axis(r[, axis])</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td>cov_nugget(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>cov_spatial(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cov_yadrenko(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>covariance(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td>default_arg_bounds()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td>default_opt_arg()</td>
<td>Defaults for the optional arguments.</td>
</tr>
<tr>
<td>default_opt_arg_bounds()</td>
<td>Defaults for boundaries of the optional arguments.</td>
</tr>
<tr>
<td>default_rescale()</td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td>fit_variogram(x_data, y_data[, anis, sill, ...])</td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
<tr>
<td>fix_dim()</td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td>isometrize(pos)</td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
<tr>
<td>ln_spectral_rad_pdf(r)</td>
<td>Log radial spectral density of the model.</td>
</tr>
<tr>
<td>main_axes()</td>
<td>Axes of the rotated coordinate-system.</td>
</tr>
<tr>
<td>percentile_scale([per])</td>
<td>Calculate the percentile scale of the isotrope model.</td>
</tr>
<tr>
<td>plot([func])</td>
<td>Plot a function of a the CovModel.</td>
</tr>
<tr>
<td>pykrige_vario(args, r)</td>
<td>Isotropic variogram of the model for pykrige.</td>
</tr>
<tr>
<td>set_arg_bounds([check_args])</td>
<td>Set bounds for the parameters of the model.</td>
</tr>
<tr>
<td>spectral_density(k)</td>
<td>Spectral density of the covariance model.</td>
</tr>
<tr>
<td>spectral_rad_pdf(r)</td>
<td>Radial spectral density of the model.</td>
</tr>
<tr>
<td>spectrum(k)</td>
<td>Spectrum of the covariance model.</td>
</tr>
<tr>
<td>var_factor()</td>
<td>Factor for C (intensity of variation) to result in variance.</td>
</tr>
<tr>
<td>vario_axis(r[, axis])</td>
<td>Variogram along axis of anisotropy.</td>
</tr>
</tbody>
</table>

continues on next page
Table 28 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vario_nugget(r)</td>
<td>Isotropic variogram of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>vario_spatial(pos)</td>
<td>Spatial variogram respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>vario_yadrenko(zeta)</td>
<td>Yadrenko variogram for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>variogram(r)</td>
<td>Isotropic variogram of the model.</td>
</tr>
</tbody>
</table>

**anisometrize(pos)**
Bring a position tuple into the anisotropic coordinate-system.

**calc_integral_scale()**
Calculate the integral scale of the isotropic 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)

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 scipy’s curve_fit.
  The following keyword values are allowed:
  - 'linear' (default): \( \rho(z) = z \). Gives a standard least-squares problem.
  - 'soft_l1': \( \rho(z) = 2 \ast ((1 + z)**0.5 - 1) \). The smooth approximation of \( \ell_1 \) (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’s 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 \( \text{perr} = \text{np.sqrt(np.diag(pcov))} \).
• **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: \( k = ||k|| \)

**spectral_rad_pdf**(r)

Radial spectral density of the model.

**spectrum**(k)

Spectrum of the covariance model.

This is given by:

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

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: \( k = ||k|| \)

**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 setable.
    
    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]

    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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type \texttt{list}

**property len_scale_vec**
The length scales in each direction.

Notes
This is calculated by:
\begin{align*}
\cdot \ len\_scale\_vec[0] &= len\_scale \\
\cdot \ len\_scale\_vec[1] &= len\_scale*\text{anis}[0] \\
\cdot \ len\_scale\_vec[2] &= len\_scale*\text{anis}[1]
\end{align*}

Type \texttt{numpy.ndarray}

**property len_up**
Upper length scale truncation of the model.
\begin{align*}
\cdot \ len\_up &= len\_low + len\_scale
\end{align*}

Type \texttt{float}

**property len_up_rescaled**
Upper length scale truncation rescaled.
\begin{align*}
\cdot \ len\_up\_rescaled &= (len\_low + len\_scale) / \text{rescale}
\end{align*}

Type \texttt{float}

**property name**
The name of the CovModel class.

Type \texttt{str}

**property nugget**
The nugget of the model.

Type \texttt{float}

**property nugget_bounds**
Bounds for the nugget.

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

Type \texttt{list}

**property opt_arg**
Names of the optional arguments.

Type \texttt{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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) 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
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_args)

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:

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

\[
\sigma^2(\lambda) = C \cdot \lambda^{2H}
\]

This results in:

\[
\gamma_{\ell_{\text{low}}, \ell_{\text{up}}} (r) = \sigma_{\ell_{\text{low}}, \ell_{\text{up}}}^2 \cdot \left(1 - 2H \cdot \frac{\ell_{\text{up}}^2 \cdot E_{1+2H} \left[\frac{r}{\ell_{\text{up}}}\right] - \ell_{\text{up}}^2 \cdot E_{1+2H} \left[\frac{r}{\ell_{\text{low}}}\right]}{\ell_{\text{up}}^2 - \ell_{\text{low}}^2}\right)
\]

\[
\sigma_{\ell_{\text{low}}, \ell_{\text{up}}}^2 = \frac{C \cdot (\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H})}{2H}
\]

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
- \( 0 < H < \frac{1}{2} \): 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:

\[
- \ell_{\text{up}} = \ell_{\text{len}} + \text{len_scale}
\]

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

- **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(α/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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>anisometrize(pos)</code></td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td><code>calc_integral_scale()</code></td>
<td>Calculate the integral scale of the isotrope model.</td>
</tr>
<tr>
<td><code>check_arg_bounds()</code></td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td><code>check_dim(dim)</code></td>
<td>Check the given dimension.</td>
</tr>
<tr>
<td><code>check_opt_arg()</code></td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td><code>cor(h)</code></td>
<td>TPL with Exponential modes - normalized correlation function.</td>
</tr>
<tr>
<td><code>cor_axis(r[, axis])</code></td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td><code>cor_spatial(pos)</code></td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>cor_yadrenko(zeta)</code></td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>correlation(r)</code></td>
<td>TPL with Exponential modes - correlation function.</td>
</tr>
<tr>
<td><code>cov_axis(r[, axis])</code></td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td><code>cov_nugget(r)</code></td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><code>cov_spatial(pos)</code></td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>cov_yadrenko(zeta)</code></td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>covariance(r)</code></td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td><code>default_arg_bounds()</code></td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td><code>default_opt_arg()</code></td>
<td>Defaults for the optional arguments.</td>
</tr>
<tr>
<td><code>default_opt_arg_bounds()</code></td>
<td>Defaults for boundaries of the optional arguments.</td>
</tr>
<tr>
<td><code>default_rescale()</code></td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td><code>fit_variogram(x_data, y_data[, anis, sill, ...])</code></td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
<tr>
<td><code>fix_dim()</code></td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td><code>isometrize(pos)</code></td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
<tr>
<td><code>ln_spectral_rad_pdf(r)</code></td>
<td>Log radial spectral density of the model.</td>
</tr>
<tr>
<td><code>main_axes()</code></td>
<td>Axes of the rotated coordinate-system.</td>
</tr>
<tr>
<td><code>percentile_scale([per])</code></td>
<td>Calculate the percentile scale of the isotrope model.</td>
</tr>
<tr>
<td><code>plot(func)</code></td>
<td>Plot a function of a the CovModel.</td>
</tr>
<tr>
<td><code>pykrige_variogram([args, r])</code></td>
<td>Isotropic variogram of the model for pykrige.</td>
</tr>
<tr>
<td><code>set_arg_bounds([check_args])</code></td>
<td>Set bounds for the parameters of the model.</td>
</tr>
<tr>
<td><code>spectral_density(k)</code></td>
<td>Spectral density of the covariance model.</td>
</tr>
<tr>
<td><code>spectral_rad_pdf(r)</code></td>
<td>Radial spectral density of the model.</td>
</tr>
<tr>
<td><code>spectrum(k)</code></td>
<td>Spectrum of the covariance model.</td>
</tr>
<tr>
<td><code>var_factor()</code></td>
<td>Factor for C (intensity of variation) to result in variance.</td>
</tr>
</tbody>
</table>

continues on next page
Table 29 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vario_axis(r[, axis])</td>
<td>Variogram along axis of anisotropy.</td>
</tr>
<tr>
<td>vario_nugget(r)</td>
<td>Isotropic variogram of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>vario_spatial(pos)</td>
<td>Spatial variogram respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>vario_yadrenko(zeta)</td>
<td>Yadrenko variogram for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>variogram(r)</td>
<td>Isotropic variogram of the model.</td>
</tr>
</tbody>
</table>

**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)
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 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:
  – sill = var + 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 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 scipy's `curve_fit`

  The following keyword values are allowed:
  - 'linear' (default): \( \rho(z) = z \). Gives a standard least-squares problem.
  - 'soft_l1': \( \rho(z) = 2 \times ((1 + z)^{0.5} - 1) \). The smooth approximation of \( L1 \) (absolute value) loss. Usually a good choice for robust least squares.
  - 'huber': \( \rho(z) = z \text{ if } z \leq 1 \) else \( 2^z - 1 \). 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: \( 100 \times n \).

- **return_r2**(bool, optional) – Whether to return the r2 score of the estimation. Default: False

- **curve_fit_kwarg**s(**dict**, optional) – Other keyword arguments passed to scipy's `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 \( \text{perr} = \sqrt{\text{np.diag(pcov)}} \).
• **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 isotope 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: \( k = \|k\| \)

**spectral_rad_pdf**(r)

Radial spectral density of the model.

**spectrum**(k)

Spectrum of the covariance model.

This is given by:

\[ S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r)e^{ikr}d^nr \]

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: \( k = \|k\| \)

**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.

**property anis**

The anisotropy factors of the model.
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").

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").

property `arg_list`

Values of all arguments.

**Type** list of float

property `dim`

The dimension of the model.

**Type** int

property `dist_func`

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 setable.
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]

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 \([a, b, \text{<type>}]\) where \text{<type>} is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

Type \text{list}

**property len_scale_vec**
The length scales in each direction.

Notes
This is calculated by:

- \(\text{len_scale_vec}[0] = \text{len_scale}\)
- \(\text{len_scale_vec}[1] = \text{len_scale} \times \text{anis}[0]\)
- \(\text{len_scale_vec}[2] = \text{len_scale} \times \text{anis}[1]\)

Type \text{numpy.ndarray}

**property len_up**
Upper length scale truncation of the model.

- \(\text{len_up} = \text{len_low} + \text{len_scale}\)

Type \text{float}

**property len_up_rescaled**
Upper length scale truncation rescaled.

- \(\text{len_up_rescaled} = (\text{len_low} + \text{len_scale}) / \text{rescale}\)

Type \text{float}

**property name**
The name of the CovModel class.

Type \text{str}

**property nugget**
The nugget of the model.

Type \text{float}

**property nugget_bounds**
Bounds for the nugget.

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

Type \text{list}

**property opt_arg**
Names of the optional arguments.

Type \text{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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").

---

**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 \([a, b, \text{<type>}]\) where \(<\text{type}>\) is one of "oo", "cc", "oc" or "co" to define if the bounds are open ("o") or closed ("c").
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)

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:

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

$$\sigma^2(\lambda) = C \cdot \lambda^{2H}$$

This results in:

$$\gamma_{\ell_{\text{low}},\ell_{\text{up}}}(r) = \sigma^2_{\ell_{\text{low}},\ell_{\text{up}}} \cdot \left(1 - \frac{2H}{\alpha} \cdot \frac{\ell_{\text{up}}^{2H} \cdot E_1 \left(\frac{r}{\ell_{\text{up}}}\right)^{\alpha}}{\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H}} \cdot \frac{\ell_{\text{low}}^{2H} \cdot E_1 \left(\frac{r}{\ell_{\text{low}}}\right)^{\alpha}}{\ell_{\text{up}}^{2H} - \ell_{\text{low}}^{2H}}\right)$$

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

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:

- 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
GeoStatTools Documentation, Release 1.3.5

- **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 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 \( \alpha \) 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.
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.
isol_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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anisometrize(pos)</td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td>calc_integral_scale()</td>
<td>Calculate the integral scale of the isotrope model.</td>
</tr>
<tr>
<td>check_arg_bounds()</td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td>check_dim(dim)</td>
<td>Check the given dimension.</td>
</tr>
<tr>
<td>check_opt_arg()</td>
<td>Check the optional arguments.</td>
</tr>
<tr>
<td>cor(h)</td>
<td>TPL with Stable modes - normalized correlation function.</td>
</tr>
<tr>
<td>cor_axis([r[, axis]])</td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td>cor_spatial(pos)</td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cor_yadrenko(zeta)</td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>correlation(r)</td>
<td>TPL with Stable modes - correlation function.</td>
</tr>
<tr>
<td>cov_axis([r[, axis]])</td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td>cov_nugget(r)</td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td>cov_spatial(pos)</td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>cov_yadrenko(zeta)</td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>covariance(r)</td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td>default_arg_bounds()</td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td>default_opt_arg()</td>
<td>Defaults for the optional arguments.</td>
</tr>
<tr>
<td>default_opt_arg_bounds()</td>
<td>Defaults for boundaries of the optional arguments.</td>
</tr>
<tr>
<td>default_rescale()</td>
<td>Provide default rescaling factor.</td>
</tr>
<tr>
<td>fit_variogram(x_data, y_data[, anis, sill, ...])</td>
<td>Fitting the variogram-model to an empirical variogram.</td>
</tr>
<tr>
<td>fix_dim()</td>
<td>Set a fix dimension for the model.</td>
</tr>
<tr>
<td>isometrize(pos)</td>
<td>Make a position tuple ready for isotropic operations.</td>
</tr>
<tr>
<td>ln_spectral_rad_pdf(r)</td>
<td>Log radial spectral density of the model.</td>
</tr>
<tr>
<td>main_axes()</td>
<td>Axes of the rotated coordinate-system.</td>
</tr>
<tr>
<td>percentile_scale([per])</td>
<td>Calculate the percentile scale of the isotrope model.</td>
</tr>
<tr>
<td>plot([func])</td>
<td>Plot a function of a the CovModel.</td>
</tr>
<tr>
<td>pykrige_vario([args, r])</td>
<td>Isotropic variogram of the model for pykrige.</td>
</tr>
<tr>
<td>set_arg_bounds([check_args])</td>
<td>Set bounds for the parameters of the model.</td>
</tr>
<tr>
<td>spectral_density(k)</td>
<td>Spectral density of the covariance model.</td>
</tr>
<tr>
<td>spectral_rad_pdf(r)</td>
<td>Radial spectral density of the model.</td>
</tr>
<tr>
<td>spectrum(k)</td>
<td>Spectrum of the covariance model.</td>
</tr>
<tr>
<td>var_factor()</td>
<td>Factor for C (intensity of variation) to result in variance.</td>
</tr>
<tr>
<td>vario_axis([r[, axis]])</td>
<td>Variogram along axis of anisotropy.</td>
</tr>
<tr>
<td>vario_nugget(r)</td>
<td>Isotropic variogram of the model respecting the nugget at r=0.</td>
</tr>
</tbody>
</table>

continues on next page
### Table 30 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vario_spatial(pos)</td>
<td>Spatial variogram respecting anisotropy and rotation.</td>
</tr>
<tr>
<td>vario_yadrenko(zeta)</td>
<td>Yadrenko variogram for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td>variogram(r)</td>
<td>Isotropic variogram of the model.</td>
</tr>
</tbody>
</table>

#### anisometrize(pos)
Bring a position tuple into the anisotropic coordinate-system.

#### calc_integral_scale()
Calculate the integral scale of the isotropic 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
def 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

def 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=True, 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 scipy's curve_fit. The following keyword values are allowed:
  - ‘linear’ (default): \( \rho(z) = z \). Gives a standard least-squares problem.
  - ‘soft_l1’: \( \rho(z) = 2 \times ((1 + z)^{0.5} - 1) \). The smooth approximation of \( \ell_1 \) (absolute value) loss. Usually a good choice for robust least squares.
  - ‘huber’: \( \rho(z) = z \) if \( z \leq 1 \) else \( 2z^{0.5} - 1 \). 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: \( 100 \times n \).

• **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's 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 \( \text{perr} = \text{np.sqrt(np.diag(pcov))} \).

• **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 \( \text{CovModel.set_arg_bounds} \).

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: $k = ||k||$

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

**spectrum**($k$)
Spectrum of the covariance model.
This is given by:

\[ S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r)e^{ik \cdot r}d^n r \]

Internally, this is calculated by the hankel transformation:

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

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

**Parameters**
- $k$ (float) – Radius of the phase: $k = ||k||$

**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 setable.

**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]

   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]`

   **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:

- `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`
```python
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]:

\[
\rho(r) = \begin{cases} 
(1 - s \cdot \frac{r}{\xi})^\nu & r < \frac{\xi}{s} \\
0 & r \geq \frac{\xi}{s}
\end{cases}
\]

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:

\[
\rho(r) = \begin{cases} 
1 - s \cdot \frac{r}{\xi} & r < \frac{\xi}{s} \\
0 & r \geq \frac{\xi}{s}
\end{cases}
\]

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, \(\text{anis}\) 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 \(\text{len_scale}\)s 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

Chapter 3. GSTools API
```
• **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 $\alpha$ 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>anisometrize(pos)</code></td>
<td>Bring a position tuple into the anisotropic coordinate-system.</td>
</tr>
<tr>
<td><code>calc_integral_scale()</code></td>
<td>Calculate the integral scale of the isotropic model.</td>
</tr>
<tr>
<td><code>check_arg_bounds()</code></td>
<td>Check arguments to be within their given bounds.</td>
</tr>
<tr>
<td><code>check_dim(dim)</code></td>
<td>Check the given dimension.</td>
</tr>
<tr>
<td><code>check_opt_arg()</code></td>
<td>Run checks for the optional arguments.</td>
</tr>
<tr>
<td><code>cor(r)</code></td>
<td>TPL Simple - normalized correlation function.</td>
</tr>
<tr>
<td><code>cor_axis(r[, axis])</code></td>
<td>Correlation along axis of anisotropy.</td>
</tr>
<tr>
<td><code>cor_spatial(pos)</code></td>
<td>Spatial correlation respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>cor_yadrenko(zeta)</code></td>
<td>Yadrenko correlation for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>cov_axis(r[, axis])</code></td>
<td>Covariance along axis of anisotropy.</td>
</tr>
<tr>
<td><code>cov_nugget(r)</code></td>
<td>Isotropic covariance of the model respecting the nugget at r=0.</td>
</tr>
<tr>
<td><code>cov_spatial(pos)</code></td>
<td>Spatial covariance respecting anisotropy and rotation.</td>
</tr>
<tr>
<td><code>cov_yadrenko(zeta)</code></td>
<td>Yadrenko covariance for great-circle distance from latlon-pos.</td>
</tr>
<tr>
<td><code>covariance(r)</code></td>
<td>Covariance of the model.</td>
</tr>
<tr>
<td><code>default_arg_bounds()</code></td>
<td>Provide default boundaries for arguments.</td>
</tr>
<tr>
<td><code>default_opt_arg()</code></td>
<td>Defaults for the optional arguments.</td>
</tr>
</tbody>
</table>
Table 31 – continued from previous page

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

**anisometrize(pos)**

Bring a position tuple into the anisotropic coordinate-system.

**calc_integral_scale()**

Calculate the integral scale of the isotropic 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 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:
sill = var + 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 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 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** *(str, callable, 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’s curve_fit. The following keyword values are allowed:
  - '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 <= 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's 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 isotropic 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: \( k = \|k\| \)

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

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

This is given by:

\[ S(k) = \left( \frac{1}{2\pi} \right)^n \int C(r)e^{ikr}dr \]

Internally, this is calculated by the hankel transformation:

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

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

Parameters **k** (float) – Radius of the phase: \( k = \|k\| \)

**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 setable.

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]

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]`

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

**Type** \texttt{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** \texttt{float}

**property sill**

The sill of the variogram.

**Notes**

This is calculated by:

\[ \text{sill} = \text{variance} + \text{nugget} \]

**Type** \texttt{float}

**property var**

The variance of the model.

**Type** \texttt{float}

**property var_bounds**

Bounds for the variance.

**Notes**

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

**Type** \texttt{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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plot_variogram(model[, x_min, x_max, fig, ax])</code></td>
<td>Plot variogram of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_covariance(model[, x_min, x_max, fig, ax])</code></td>
<td>Plot covariance of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_correlation(model[, x_min, x_max, fig, ax])</code></td>
<td>Plot correlation function of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_cov_yadrenko(model[, x_min, x_max, ...])</code></td>
<td>Plot Yadrenko variogram of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_cor_yadrenko(model[, x_min, x_max, fig, ax])</code></td>
<td>Plot Yadrenko covariance of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_cor_axis(model[, axis, x_min, x_max, ...])</code></td>
<td>Plot Yadrenko correlation function of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_cov_axis(model[, axis, x_min, x_max, ...])</code></td>
<td>Plot variogram of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_cor_axis(model[, axis, x_min, x_max, ...])</code></td>
<td>Plot variogram of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_vario_axis(model[, axis, x_min, x_max, ...])</code></td>
<td>Plot variogram of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_cov_spatial(model[, x_min, x_max, fig, ax])</code></td>
<td>Plot spatial covariance of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_cor_spatial(model[, x_min, x_max, fig, ax])</code></td>
<td>Plot spatial correlation of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_spectrum(model[, x_min, x_max, fig, ax])</code></td>
<td>Plot spectrum of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_spectral_density(model[, x_min, x_max, ...])</code></td>
<td>Plot spectral density of a given CovModel.</td>
</tr>
<tr>
<td><code>plot_spectral_rad_pdf(model[, x_min, x_max, ...])</code></td>
<td>Plot radial spectral pdf of a given CovModel.</td>
</tr>
</tbody>
</table>

```python
gstools.covmodel.plot.plot_cor_axis(model, axis=0, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
gstools.covmodel.plot.plot_cor_spatial(model, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
gstools.covmodel.plot.plot_cor_yadrenko(model, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
gstools.covmodel.plot.plot_correlation(model, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
gstools.covmodel.plot.plot_cov_axis(model, axis=0, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
gstools.covmodel.plot.plot_cov_spatial(model, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
gstools.covmodel.plot.plot_cov_yadrenko(model, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
gstools.covmodel.plot.plot_covariance(model, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
gstools.covmodel.plot.plot_spectral_density(model, x_min=0.0, x_max=None, fig=None, ax=None, **kwargs)
```
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 spectrum 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

- **generator**: GStools subpackage providing generators for spatial random fields.
- **upscaling**: GStools subpackage providing upscaling routines for the spatial random field.

Spatial Random Field

---

<table>
<thead>
<tr>
<th>SRF(model[, mean, normalizer, trend, ...])</th>
<th>A class to generate spatial random fields (SRF).</th>
</tr>
</thead>
<tbody>
<tr>
<td>CondSRF(krigel[, generator])</td>
<td>A class to generate conditioned spatial random fields (SRF).</td>
</tr>
</tbody>
</table>

---

gstools.field.SRF

**class gstools.field.SRF**

```python
class gstools.field.SRF(model, mean=0.0, normalizer=None, trend=None, upscaling='no_scaling', generator='RandMeth', **generator_kwargs)
```

Bases: `gstools.field.base.Field`

A class to generate spatial random fields (SRF).

**Parameters**

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

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

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

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_fields</td>
<td>list: All fields as stacked list.</td>
</tr>
<tr>
<td>dim</td>
<td>int: Dimension of the field.</td>
</tr>
<tr>
<td>field_names</td>
<td>list: Names of present fields.</td>
</tr>
<tr>
<td>field_shape</td>
<td>tuple: The shape of the field.</td>
</tr>
<tr>
<td>generator</td>
<td>callable: The generator of the field.</td>
</tr>
<tr>
<td>latlon</td>
<td>bool: Whether the field depends on geographical coords.</td>
</tr>
<tr>
<td>mean</td>
<td>float or callable: The mean of the field.</td>
</tr>
<tr>
<td>mesh_type</td>
<td>str: The mesh type of the field.</td>
</tr>
<tr>
<td>model</td>
<td>CovModel: The covariance model of the field.</td>
</tr>
<tr>
<td>name</td>
<td>str: The name of the class.</td>
</tr>
<tr>
<td>normalizer</td>
<td>Normalizer: Normalizer of the field.</td>
</tr>
<tr>
<td>pos</td>
<td>tuple: The position tuple of the field.</td>
</tr>
<tr>
<td>trend</td>
<td>float or callable: The trend of the field.</td>
</tr>
<tr>
<td>upscaling</td>
<td>str: Name of the upscaling method.</td>
</tr>
<tr>
<td>value_type</td>
<td>str: Type of the field values (scalar, vector).</td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong></td>
<td>([pos, seed, point_volumes, ...]) Generate the spatial random field.</td>
</tr>
<tr>
<td>delete_fields</td>
<td>([select]) Delete selected fields.</td>
</tr>
<tr>
<td>get_store_config</td>
<td>(store[, default, fld_cnt]) Get storage configuration from given selection.</td>
</tr>
<tr>
<td>mesh</td>
<td>(mesh[, points, direction, name]) Generate a field on a given meshio, ogs5py or PyVista mesh.</td>
</tr>
<tr>
<td>plot</td>
<td>([field, fig, ax]) Plot the spatial random field.</td>
</tr>
<tr>
<td>post_field</td>
<td>(field[, name, process, save]) Postprocessing field values.</td>
</tr>
<tr>
<td>pre_pos</td>
<td>([pos, mesh_type, info]) Preprocessing positions and mesh_type.</td>
</tr>
<tr>
<td>set_generator</td>
<td>(generator, **generator_kwargs) Set the generator for the field.</td>
</tr>
<tr>
<td>set_pos</td>
<td>([pos[, mesh_type, info]]) Set positions and mesh_type.</td>
</tr>
<tr>
<td>structured</td>
<td>(*args, **kwargs) Generate a field on a structured mesh.</td>
</tr>
<tr>
<td>to_pyvista</td>
<td>([field_select, fieldname]) Create a VTK/PyVista grid of the stored field.</td>
</tr>
<tr>
<td>transform</td>
<td>(method[, field, store, process]) Apply field transformation.</td>
</tr>
<tr>
<td>unstructured</td>
<td>(*args, **kwargs) Generate a field on an unstructured mesh.</td>
</tr>
<tr>
<td>upscaling_func</td>
<td>(*args, **kwargs) Upscaling method applied to the field variance.</td>
</tr>
<tr>
<td>vtk_export</td>
<td>([filename[, field_select, fieldname]]) Export the stored field to vtk.</td>
</tr>
</tbody>
</table>

```
__call__((pos=None, seed=nan, point_volumes=0.0, mesh_type='unstructured', post_process=True, store=True))
```

Generate the spatial random field.

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

### Parameters

- **pos** (list, optional) – the position tuple, containing main direction and transversal directions
GeoStatTools Documentation, Release 1.3.5

- **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’
- **post_process** *(bool, optional)* – Whether to apply mean, normalizer and trend to the field. Default: True
- **store** *(str or bool, optional)* – Whether to store field (True/False) with default name or with specified name. The default is True for default name “field”.

**Returns**  
field – the SRF

**Return type**  
numpy.ndarray

**delete_fields** *(select=None)*  
Delete selected fields.

**get_store_config** *(store, default=None, fld_cnt=None)*  
Get storage configuration from given selection.

**Parameters**

- **store** *(str or bool or list, optional)* – Whether to store fields (True/False) with default names or with specified names. The default is True for default names.
- **default** *(str or list, optional)* – Default field names. The default is “field”.
- **fld_cnt** *(None or int, optional)* – Number of fields when using lists. The default is None.

**Returns**

- **name** *(str or list)* – Name(s) of field.
- **save** *(bool or list)* – Whether to save field(s).

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

**Parameters**

- **mesh** *(meshio.Mesh or ogs5py.MSH or PyVista mesh)* – The given 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 __call__.

**Notes**

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

**See:**

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

3.7. gstools.field 335
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=None, mesh_type='unstructured', info=False)
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”
- **info** (bool, optional) – Whether to return information

Returns

- **iso_pos** ((d, n), **numpy.ndarray**) – Isometrized position tuple.
- **shape** (tuple) – Shape of the resulting field.
- **info** (dict, optional) – Information about settings.

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

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.

```py
set_pos(pos, mesh_type='unstructured', info=False)
```

Set positions and mesh_type.

**Parameters**

- **pos (iterable)** – the position tuple, containing main direction and transversal directions
- **mesh_type (str, optional)** – ‘structured’ / ‘unstructured’ Default: “unstructured”
- **info (bool, optional)** – Whether to return information

**Returns**

- **info** – Information about settings.

**Return type**

- **dict**, optional

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

```py
structured(*args, **kwargs)
```

Generate a field on a structured mesh.

See :func:`__call__`

```py
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”

```py
transform(method, field='field', store=True, process=False, **kwargs)
```

Apply field transformation.

**Parameters**

- **method (str)** – Method to use. See :func:`gstools.transform` for available transformations.
- **field (str, optional)** – Name of field to be transformed. The default is “field”.
- **store (str or bool, optional)** – Whether to store field inplace (True/False) or under a given name. The default is True.
- **process (bool, optional)** – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- ****kwargs** – Keyword arguments forwarded to selected method.

**Raises**

- **ValueError** – When method is unknown.

**Returns**

- Transformed field.

**Return type**

- **numpy.ndarray**

```py
unstructured(*args, **kwargs)
```

Generate a field on an unstructured mesh.

See :func:`__call__`

```py
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 all_fields**

All fields as stacked list.

**Type** list

**default_field_names** = ['field']

Default field names.

**Type** list

**property dim**

Dimension of the field.

**Type** int

**property field_names**

Names of present fields.

**Type** list

**property field_shape**

The shape of the field.

**Type** tuple

**property generator**

The generator of the field.

Default: *RandMeth*

**Type** callable

**property latlon**

Whether the field depends on geographical coords.

**Type** bool

**property mean**

The mean of the field.

**Type** float or callable

**property mesh_type**

The mesh type of the field.

**Type** str

**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 pos
 The position tuple of the field.
 Type tuple

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
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

- **all_fields** list: All fields as stacked list.
- **dim** int: Dimension of the field.
- **field_names** list: Names of present fields.
- **field_shape** tuple: The shape of the field.
- **generator** callable: The generator of the field.
- **krige** *Krige*: The underlying kriging class.
- **latlon** bool: Whether the field depends on geographical coords.
- **mean** float or callable: The mean of the field.
- **mesh_type** str: The mesh type of the field.
- **model** *CovModel*: The covariance model of the field.
- **name** str: The name of the class.
- **normalizer** *Normalizer*: Normalizer of the field.
- **pos** tuple: The position tuple of the field.
- **trend** float or callable: The trend of the field.
- **value_type** str: Type of the field values (scalar, vector).

Methods

- __call__([pos, seed, mesh_type, ...]) Generate the conditioned spatial random field.
- delete_fields([select]) Delete selected fields.
- get_scaling(krige_var, shape) Get scaling coefficients for the random field.
- get_store_config(store[, default, fld_cnt]) Get storage configuration from given selection.
- mesh(mesh[, points, direction, name]) Generate a field on a given meshio, ogs5py or PyVista mesh.
- plot([field, fig, ax]) Plot the spatial random field.
- post_field(field[, name, process, save]) Postprocessing field values.
- pre_pos([pos, mesh_type, info]) Preprocessing positions and mesh_type.
- set_generator(generator, **generator_kwargs) Set the generator for the field.
- set_pos(pos[, mesh_type, info]) Set positions and mesh_type.
- structured(*args, **kwargs) Generate a field on a structured mesh.
Table 36 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_pyvista([field_select, fieldname])</code></td>
<td>Create a VTK/PyVista grid of the stored field.</td>
</tr>
<tr>
<td><code>transform(method[, field, store, process])</code></td>
<td>Apply field transformation.</td>
</tr>
<tr>
<td><code>unstructured(*args, **kwargs)</code></td>
<td>Generate a field on an unstructured mesh.</td>
</tr>
<tr>
<td><code>vtk_export(filename[, field_select, fieldname])</code></td>
<td>Export the stored field to vtk.</td>
</tr>
</tbody>
</table>

__call__

```
__call__(pos=None, seed=nan, mesh_type='unstructured', post_process=True, store=True, krige_store=True, **kwargs)
```

Generate the conditioned spatial random field.

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

**Parameters**

- `pos` *(list, optional)* – 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’
- `post_process` *(bool, optional)* – Whether to apply mean, normalizer and trend to the field. Default: `True`
- `store` *(str or bool or list, optional)* – Whether to store fields (True/False) with default names or with specified names. The default is `True` for default names ["field", "raw_field", "raw_krige"].
- `krige_store` *(str or bool or list, optional)* – Whether to store kriging fields (True/False) with default name or with specified names. The default is `True` for default names ["field", "krige_var"].
- `**kwargs` – keyword arguments that are forwarded to the kriging routine in use.

**Returns** `field` – the conditioned SRF

**Return type** `numpy.ndarray`

`delete_fields(select=None)`

Delete selected fields.

`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.

`get_store_config(store, default=None, fld_cnt=None)`

Get storage configuration from given selection.

**Parameters**

- `store` *(str or bool or list, optional)* – Whether to store fields (True/False) with default names or with specified names. The default is `True` for default names.
- `default` *(str or list, optional)* – Default field names. The default is “field”.
- `fld_cnt` *(None or int, optional)* – Number of fields when using lists. The default is None.

**Returns**
Generate a field on a given meshio, ogs5py or PyVista mesh.

**Parameters**

- `mesh (meshio.Mesh or ogs5py.MSH or PyVista mesh)` – The given 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 too few names are given, digits will be appended. Default: “field”
- `**kwargs` – Keyword arguments forwarded to `__call__`.

**Notes**

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

See:

- meshio: https://github.com/nschloe/meshio
- ogs5py: https://github.com/GeoStat-Framework/ogs5py
- PyVista: https://github.com/pyvista/pyvista

**Plot**

Plot the spatial random field.

**Parameters**

- `field (str, optional)` – Field that should be plotted. Default: “field”
- `fig (Figure or None, ax=None, **kwargs)` – 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.

**Postprocess**

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=None, mesh_type='unstructured', info=False)

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”

• info (bool, optional) – Whether to return information

Returns

• iso_pos ((d, n), numpy.ndarray) – Isometrized position tuple.

• shape (tuple) – Shape of the resulting field.

• info (dict, optional) – Information about settings.

Warning: When setting a new position tuple that differs from the present one, all stored fields will be deleted.

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.

set_pos(pos, mesh_type='unstructured', info=False)

Set positions and mesh_type.

Parameters

• pos (iterable) – the position tuple, containing main direction and transversal directions

• mesh_type (str, optional) – 'structured' / 'unstructured' Default: “unstructured”

• info (bool, optional) – Whether to return information

Returns info – Information about settings.

Return type  
dict, optional

Warning: When setting a new position tuple that differs from the present one, all stored fields will be deleted.

structured(*args, **kwargs)

Generate a field on a structured mesh.

See __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”

```python
transform(method, field='field', store=True, process=False, **kwargs)
```

Apply field transformation.

**Parameters**

• **method** *(str)* – Method to use. See `gstools.transform` for available transformations.

• **field** *(str, optional)* – Name of field to be transformed. The default is “field”.

• **store** *(str or bool, optional)* – Whether to store field inplace (True/False) or under a given name. The default is True.

• **process** *(bool, optional)* – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.

• **kwargs** – Keyword arguments forwarded to selected method.

**Raises** `ValueError` – When method is unknown.

**Returns** Transformed field.

**Return type** `numpy.ndarray`

```python
unstructured(*args, **kwargs)
```

Generate a field on an unstructured mesh.

See `__call__`

```python
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 all_fields**

All fields as stacked list.

**Type** `list`

```python
default_field_names = ['field', 'raw_field', 'raw_krige']
```

Default field names.

**Type** `list`

**property dim**

Dimension of the field.

**Type** `int`

**property field_names**

Names of present fields.

**Type** `list`

**property field_shape**

The shape of the field.

**Type** `tuple`

**property generator**

The generator of the field.

**Type** `callable`
property kriging
    The underlying kriging class.
    Type Krige

property latlon
    Whether the field depends on geographical coords.
    Type bool

property mean
    The mean of the field.
    Type float or callable

property mesh_type
    The mesh type of the field.
    Type str

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 pos
    The position tuple of the field.
    Type tuple

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

Field([model, value_type, mean, normalizer, ...]) A base class for random fields, kriging fields, etc.

gstools.field.Field
class gstools.field.Field(model=None, value_type='scalar', mean=None, normalizer=None, 
trend=None, dim=None)

Bases: object

A base class for random fields, kriging fields, etc.

Parameters

• model (CovModel, optional) – 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.

• dim (None or int, optional) – Dimension of the field if no model is given.

Attributes

all_fields list: All fields as stacked list.

dim int: Dimension of the field.

field_names list: Names of present fields.

field_shape tuple: The shape of the field.

latlon bool: Whether the field depends on geographical coords.

mean float or callable: The mean of the field.

mesh_type str: The mesh type of the field.

model CovModel: The covariance model of the field.

name str: The name of the class.

normalizer Normalizer: Normalizer of the field.

pos tuple: The position tuple of the field.

trend float or callable: The trend of the field.

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>([pos, field, mesh_type, ...])</td>
<td>Generate the field.</td>
</tr>
<tr>
<td>delete_fields([select])</td>
<td>Delete selected fields.</td>
</tr>
<tr>
<td>get_store_config(store[, default, fld_cnt])</td>
<td>Get storage configuration from given selection.</td>
</tr>
<tr>
<td>mesh(mesh[, points, direction, name])</td>
<td>Generate a field on a given meshio, ogs5py or PyVista mesh.</td>
</tr>
<tr>
<td>plot(field[, fig, ax])</td>
<td>Plot the spatial random field.</td>
</tr>
<tr>
<td>post_field(field[, name, process, save])</td>
<td>Postprocessing field values.</td>
</tr>
<tr>
<td>pre_pos([pos, mesh_type, info])</td>
<td>Preprocessing positions and mesh_type.</td>
</tr>
<tr>
<td>set_pos(pos[, mesh_type, info])</td>
<td>Set positions and mesh_type.</td>
</tr>
<tr>
<td>structured(*args, **kwargs)</td>
<td>Generate a field on a structured mesh.</td>
</tr>
<tr>
<td>to_pyvista(field_select, fieldname)</td>
<td>Create a VTK/PyVista grid of the stored field.</td>
</tr>
<tr>
<td>transform(method[, field, store, process])</td>
<td>Apply field transformation.</td>
</tr>
<tr>
<td>unstructured(*args, **kwargs)</td>
<td>Generate a field on an unstructured mesh.</td>
</tr>
<tr>
<td>vtk_export(filename[, field_select, fieldname])</td>
<td>Export the stored field to vtk.</td>
</tr>
</tbody>
</table>

### __call__

Generate the field.

**Parameters**

- **pos** (list, optional) – the position tuple, containing main direction and transversal directions
- **field** (numpy.ndarray or None, optional) – the field values. Will be all zeros if None is given.
- **mesh_type** (str, optional) – ‘structured’ / ‘unstructured’. Default: ‘unstructured’
- **post_process** (bool, optional) – Whether to apply mean, normalizer and trend to the field. Default: True
- **store** (str or bool, optional) – Whether to store field (True/False) with default name or with specified name. The default is True for default name “field”.

**Returns**

- **field** – the field values.

**Return type**

numpy.ndarray

### delete_fields(select=None)

Delete selected fields.

### get_store_config(store, default=None, fld_cnt=None)

Get storage configuration from given selection.

**Parameters**

- **store** (str or bool or list, optional) – Whether to store fields (True/False) with default names or with specified names. The default is True for default names.
- **default** (str or list, optional) – Default field names. The default is “field”.
- **fld_cnt** (None or int, optional) – Number of fields when using lists. The default is None.

**Returns**

- **name** (str or list) – Name(s) of field.
- **save** (bool or list) – Whether to save field(s).

### mesh(mesh, points='centroids', direction='all', name='field', **kwargs)

Generate a field on a given meshio, ogs5py or PyVista mesh.

**Parameters**

3.7. gstools.field 347
• **mesh** *(meshio.Mesh or ogs5py.MSH or PyVista mesh)* – The given 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 **__call__**.

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

**See:**
- meshio: https://github.com/nschloe/meshio
- ogs5py: https://github.com/GeoStat-Framework/ogs5py
- PyVista: https://github.com/pyvista/pyvista

**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=None, mesh_type='unstructured', info=False)
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”
• **info** (*bool*, optional) – Whether to return information

**Returns**

• **iso_pos** ((d, n), *numpy.ndarray*) – Isometrized position tuple.
• **shape** (*tuple*) – Shape of the resulting field.
• **info** (*dict*, optional) – Information about settings.

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

`set_pos(pos, mesh_type='unstructured', info=False)`
Set positions and mesh_type.

**Parameters**

• **pos** (*iterable*) – the position tuple, containing main direction and transversal directions
• **mesh_type** (*str*, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”
• **info** (*bool*, optional) – Whether to return information

**Returns** **info** – Information about settings.

**Return type** *dict*, optional

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

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

See __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”

`transform(method, field='field', store=True, process=False, **kwargs)`
Apply field transformation.

**Parameters**

• **method** (*str*) – Method to use. See *gstools.transform* for available transformations.
• **field** (*str*, optional) – Name of field to be transformed. The default is “field”.
• **store** (*str* or *bool*, optional) – Whether to store field inplace (True/False) or under a given name. The default is True.
• **process** (*bool*, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
• **kwargs** – Keyword arguments forwarded to selected method.

**Raises** *ValueError* – When method is unknown.
Returns
Transformed field.

Return type numpy.ndarray

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

See __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 all_fields
All fields as stacked list.

Type list

default_field_names = ['field']
Default field names.

Type list

property dim
Dimension of the field.

Type int

property field_names
Names of present fields.

Type list

property field_shape
The shape of the field.

Type tuple

property latlon
Whether the field depends on geographical coords.

Type bool

property mean
The mean of the field.

Type float or callable

property mesh_type
The mesh type of the field.

Type str

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 pos
    The position tuple of the field.
    Type tuple

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

- **RandMeth** (model[, mode_no, seed, verbose, ...])
  Randomization method for calculating isotropic random fields.

- **IncomprRandMeth** (model[, mean_velocity, ...])
  RandMeth for incompressible random vector fields.

```python
class gstools.field.generator.IncomprRandMeth(model[, mean_velocity, ...])

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]:

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

where:

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

- **__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])**
  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, seed])**
  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]:

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

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

- **__call__**(pos[, add_nugget]) Calculate the random modes for the randomization method.
- **get_nugget**(shape) Generate normal distributed values for the nugget simulation.
- **reset_seed**(seed) Recalculate the random amplitudes and wave numbers with the given seed.
- **update**(model, seed) 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.upscaling

GStools subpackage providing upscaling routines for the spatial random field.

The following functions are provided

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var_coarse_graining(model[, point_volumes])</code></td>
<td>Coarse Graining procedure to upscale the variance for uniform flow.</td>
</tr>
<tr>
<td><code>var_no_scaling(model, *args, **kwargs)</code></td>
<td>Dummy function to bypass scaling.</td>
</tr>
</tbody>
</table>

**gstools.field.upscaling.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 $\lambda$ 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:

$$
\lambda = V^{\frac{1}{3}}
$$

$$
\sigma^2(\lambda) = \sigma^2 \cdot \left( \frac{\ell^2}{\ell^2 + \left(\frac{2\lambda}{3}\right)^2} \right)^{\frac{2}{3}}
$$

$$
\ell(\lambda) = \left( \ell^2 + \left(\frac{\lambda}{2}\right)^2 \right)^{\frac{1}{2}}
$$

Therby $\lambda$ 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.

**gstools.field.upscaling.var_no_scaling(model, *args, **kwargs)**

Dummy function to bypass scaling.

**Parameters**

- `model (CovModel)` – Covariance Model used for the field.

**Returns**

- `var` – The model variance.

**Return type**

- `float`
3.8 gstools.variogram

GStools subpackage providing tools for estimating and fitting variograms.

Variogram estimation

```python
vario_estimate(pos, field[, bin_edges, ...]) Estimates the empirical variogram.
vario_estimate_axis(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:

\[ \gamma(r_k) = \frac{1}{2N(r_k)} \sum_{i=1}^{N(r_k)} (z(x_i) - z(x_i'))^2, \]

with \( r_k \leq \|x_i - x_i'\| < r_{k+1} \) being the bins.

Or if the estimator “cressie” was chosen:

\[ \gamma(r_k) = \frac{1}{2} \left( \frac{1}{N(r_k)} \sum_{i=1}^{N(r_k)} |z(x_i) - z(x_i')|^{0.5} \right)^4 \frac{0.457 + 0.494/N(r_k) + 0.045/N^2(r_k)}{0.457 + 0.494/N(r_k) + 0.045/N^2(r_k)}, \]

with \( r_k \leq \|x_i - x_i'\| < r_{k+1} \) 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 bining 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 \( \varphi \) (ccw from +x in xy plane) for 3d supply two angles which are azimuth \( \varphi \) (ccw from +x in xy plane) and inclination \( \theta \) (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: \( np.pi/8 = 22.5^\circ \)

• **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: \( f(x, [y, z, \ldots]) \) If no normalizer is applied, this behaves equal to ‘mean’. The default is None.

• **fit_normalizer** *(bool, optional)* – Wheather 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:

$$\gamma(r_k) = \frac{1}{2N(r_k)} \sum_{i=1}^{N(r_k)} (z(x_i) - z(x'_i))^2,$$

with $r_k \leq \|x_i - x'_i\| < r_{k+1}$ being the bins.

Or if the estimator “cressie” was chosen:

$$\gamma(r_k) = \frac{1}{2} \left( \frac{1}{N(r_k)} \sum_{i=1}^{N(r_k)} |z(x_i) - z(x'_i)|^{0.5} \right)^4 \frac{1}{0.457 + 0.494/N(r_k) + 0.045/N^2(r_k)},$$

with $r_k \leq \|x_i - x'_i\| < r_{k+1}$ 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 outliers

Default: “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

```python
standard_bins([pos, dim, latlon, mesh_type, ...]) Get standard binning.
```

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 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
- `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.

Krige Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Krige(model, cond_pos, cond_val[, ...])</td>
<td>A Swiss Army knife for kriging.</td>
</tr>
<tr>
<td>Simple(model, cond_pos, cond_val[, mean, ...])</td>
<td>Simple kriging.</td>
</tr>
<tr>
<td>Ordinary(model, cond_pos, cond_val[, ...])</td>
<td>Ordinary kriging.</td>
</tr>
<tr>
<td>Universal(model, cond_pos, cond_val[, ...])</td>
<td>Universal kriging.</td>
</tr>
<tr>
<td>ExtDrift(model, cond_pos, cond_val, ext_drift)</td>
<td>External drift kriging (EDK).</td>
</tr>
<tr>
<td>Detrended(model, cond_pos, cond_val, trend)</td>
<td>Detrended simple kriging.</td>
</tr>
</tbody>
</table>

class gstools.krige.Krige(model, cond_pos, cond_val[, ...])

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 (nan values will be ignored)
- **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 `ltsq`
  - ”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**

- `all_fields` list: All fields as stacked list.
- `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.
**field_names** list: Names of present fields.

**field_shape** tuple: The shape of the field.

**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.

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

**mean** float or callable: The mean of the field.

**mesh_type** str: The mesh type of the field.

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

**name** str: The name of the class.

**normalizer** Normalizer: Normalizer of the field.

**pos** tuple: The position tuple 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>((pos, mesh_type, ext_drift, ...))</td>
<td>Generate the kriging field.</td>
</tr>
<tr>
<td>delete_fields(select)</td>
<td>Delete selected fields.</td>
</tr>
<tr>
<td>get_mean([post_process])</td>
<td>Calculate the estimated mean of the detrended field.</td>
</tr>
<tr>
<td>get_store_config(store[, default, fld_cnt])</td>
<td>Get storage configuration from given selection.</td>
</tr>
<tr>
<td>mesh(mesh[, points, direction, name])</td>
<td>Generate a field on a given meshio, ogs5py or PyVista mesh.</td>
</tr>
<tr>
<td>plot([field, fig, ax])</td>
<td>Plot the spatial random field.</td>
</tr>
<tr>
<td>post_field(field[, name, process, save])</td>
<td>Postprocessing field values.</td>
</tr>
<tr>
<td>pre_pos([pos, mesh_type, info])</td>
<td>Preprocessing positions and mesh_type.</td>
</tr>
<tr>
<td>set_condition([cond_pos, cond_val, ...])</td>
<td>Set the conditions for kriging.</td>
</tr>
<tr>
<td>set_drift_functions([drift_functions])</td>
<td>Set the drift functions for universal kriging.</td>
</tr>
<tr>
<td>set_pos(pos[, mesh_type, info])</td>
<td>Set positions and mesh_type.</td>
</tr>
<tr>
<td>structured(*args, **kwargs)</td>
<td>Generate a field on a structured mesh.</td>
</tr>
<tr>
<td>to_pyvista([field_select, fieldname])</td>
<td>Create a VTK/PyVista grid of the stored field.</td>
</tr>
<tr>
<td>transform(method[, field, store, process])</td>
<td>Apply field transformation.</td>
</tr>
<tr>
<td>unstructured(*args, **kwargs)</td>
<td>Generate a field on an unstructured mesh.</td>
</tr>
<tr>
<td>vtk_export(filename[, field_select, fieldname])</td>
<td>Export the stored field to vtk.</td>
</tr>
</tbody>
</table>

__call__(pos=None, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True, store=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, optional) – 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
- **store** (str or bool or list, optional) – Whether to store kriging fields (True/False) with default name or with specified names. The default is True for default names [“field”, “krige_var”] or “mean_field” if only_mean=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)

**delete_fields**(select=None)

Delete selected fields.

**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.

**get_store_config**(store, default=None, fld_cnt=None)

Get storage configuration from given selection.

Parameters

- **store** (str or bool or list, optional) – Whether to store fields (True/False) with default names or with specified names. The default is True for default names.
- **default** (str or list, optional) – Default field names. The default is “field”.
- **fld_cnt** (None or int, optional) – Number of fields when using lists. The default is None.

Returns

- **name** (str or list) – Name(s) of field.
- **save** (bool or list) – Whether to save field(s).

**mesh**(mesh, points='centroids', direction='all', name='field', **kwargs)

Generate a field on a given meshio, ogs5py or PyVista mesh.
Parameters

- **mesh** *(meshio.Mesh or ogs5py.MSH or PyVista mesh)* – The given 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 `__call__`

**Notes**

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

**See:**

- meshio: [https://github.com/nschloe/meshio](https://github.com/nschloe/meshio)
- PyVista: [https://github.com/pyvista/pyvista](https://github.com/pyvista/pyvista)

**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=None, mesh_type='unstructured', info=False)*

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”

• **info (bool, optional)** – Whether to return information

**Returns**

• **iso_pos ((d, n), numpy.ndarray)** – Isometrized position tuple.

• **shape (tuple)** – Shape of the resulting field.

• **info (dict, optional)** – Information about settings.

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

**set_condition**

```
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 (nan values will be ignored). 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**

```
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.

**set_pos**

```
set_pos(pos, mesh_type=’unstructured’, info=False)
```

Set positions and mesh_type.
Parameters

- **pos (iterable)** – the position tuple, containing main direction and transversal directions
- **mesh_type (str, optional)** – 'structured' / 'unstructured' Default: “unstructured”
- **info (bool, optional)** – Whether to return information

Returns **info** – Information about settings.

Return type **dict**, optional

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

```
structured(*args, **kwargs)
```

Generate a field on a structured mesh.

See **__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”

```
transform(method=None, field='field', store=True, process=False, **kwargs)
```

Apply field transformation.

Parameters

- **method (str)** – Method to use. See **gstools.transform** for available transformations.
- **field (str, optional)** – Name of field to be transformed. The default is “field”.
- **store (str or bool, optional)** – Whether to store field inplace (True/False) or under a given name. The default is True.
- **process (bool, optional)** – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- ****kwargs – Keyword arguments forwarded to selected method.

Raises **ValueError** – When method is unknown.

Returns Transformed field.

Return type **numpy.ndarray**

```
unstructured(*args, **kwargs)
```

Generate a field on an unstructured mesh.

See **__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 all_fields**

All fields as stacked list.

_Type list_

**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_

`default_field_names = ['field', 'krige_var', 'mean_field']`

Default field names.

_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 field_names
    Names of present fields.
    Type list

property field_shape
    The shape of the field.
    Type tuple

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 krig_size
    Size of the kriging system.
    Type int

property latlon
    Whether the field depends on geographical coords.
    Type bool

property mean
    The mean of the field.
    Type float or callable

property mesh_type
    The mesh type of the field.
    Type str

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 pos
    The position tuple of the field.
    Type tuple

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.
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 (nan values will be ignored)
• 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 trend 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

  all_fields list: All fields as stacked list.
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.

Methods

_call_((pos, mesh_type, ext_drift, ...)) Generate the kriging field.
delete_fields([select]) Delete selected fields.
get_mean([post_process]) Calculate the estimated mean of the detrended field.
get_store_config(store[, default, fld_cnt]) Get storage configuration from given selection.
mesh(mesh[, points, direction, name]) Generate a field on a given meshio, ogs5py or PyVista mesh.
plot([field, fig, ax]) Plot the spatial random field.
post_field(field[, name, process, save]) Postprocessing field values.

continues on next page
GeoStatTools Documentation, Release 1.3.5

Table 47 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre_pos((pos, mesh_type, info))</td>
<td>Preprocessing positions and mesh_type.</td>
</tr>
<tr>
<td>set_condition([cond_pos, cond_val, ...])</td>
<td>Set the conditions for kriging.</td>
</tr>
<tr>
<td>set_drift_functions([drift_functions])</td>
<td>Set the drift functions for universal kriging.</td>
</tr>
<tr>
<td>set_pos(pos[, mesh_type, info])</td>
<td>Set positions and mesh_type.</td>
</tr>
<tr>
<td>structured(*args, **kwargs)</td>
<td>Generate a field on a structured mesh.</td>
</tr>
<tr>
<td>to_pyvista([field_select, fieldname])</td>
<td>Create a VTK/PyVista grid of the stored field.</td>
</tr>
<tr>
<td>transform(method[, field, store, process])</td>
<td>Apply field transformation.</td>
</tr>
<tr>
<td>unstructured(*args, **kwargs)</td>
<td>Generate a field on an unstructured mesh.</td>
</tr>
<tr>
<td>vtk_export([filename[, field_select, fieldname]])</td>
<td>Export the stored field to vtk.</td>
</tr>
</tbody>
</table>

**__call__**(pos=None, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True, store=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, optional) – 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
- **store** (str or bool or list, optional) – Whether to store kriging fields (True/False) with default name or with specified names. The default is True for default names ["field", "krige_var"] or "mean_field" if only_mean=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)

**delete_fields**(select=None)

Delete selected fields.

**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`
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.

**get_store_config**

Get storage configuration from given selection.

**Parameters**

- **store** (str or bool or list, optional) – Whether to store fields (True/False) with default names or with specified names. The default is True for default names.
- **default** (str or list, optional) – Default field names. The default is “field”.
- **fld_cnt** (None or int, optional) – Number of fields when using lists. The default is None.

**Returns**

- **name** (str or list) – Name(s) of field.
- **save** (bool or list) – Whether to save field(s).

**mesh**

Generate a field on a given meshio, ogs5py or PyVista mesh.

**Parameters**

- **mesh** (meshio.Mesh or ogs5py.MSH or PyVista mesh) – The given 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 __call__.

**Notes**

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

See:

- meshio: https://github.com/nschloe/meshio
- ogs5py: https://github.com/GeoStat-Framework/ogs5py
- PyVista: https://github.com/pyvista/pyvista

**plot**

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
**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=None, mesh_type=’unstructured’, info=False*)

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”
- **info** (*bool*, optional) – Whether to return information.

**Returns**

- **iso_pos** ((d, n), *numpy.ndarray*) – Isometrized position tuple.
- **shape** (*tuple*) – Shape of the resulting field.
- **info** (*dict*, optional) – Information about settings.

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

**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 (nan values will be ignored). 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.

**set_pos**(pos, mesh_type='unstructured', info=False)

Set positions and mesh_type.

**Parameters**

- **pos** (iterable) – the position tuple, containing main direction and transversal directions

- **mesh_type** (str, optional) – 'structured' / 'unstructured' Default: “unstructured”

- **info** (bool, optional) – Whether to return information

**Returns** info – Information about settings.

**Return type** dict, optional

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

**structured**(*args, **kwargs)

Generate a field on a structured mesh.

See __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”

**transform**(method, field='field', store=True, process=False, **kwargs)

Apply field transformation.

**Parameters**

- **method** (str) – Method to use. See `gstools.transform` for available transformations.

- **field** (str, optional) – Name of field to be transformed. The default is “field”.

- **store** (str or bool, optional) – Whether to store field inplace (True/False) or under a given name. The default is True.
• **process** *(bool, optional)* – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.

• **kwargs** – Keyword arguments forwarded to selected method.

Raises **ValueError** – When method is unknown.

Returns Transformed field.

Return type **numpy.ndarray**

**unstructured**(*args, **kwargs*)

Generate a field on an unstructured mesh.

See **__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 all_fields**

All fields as stacked list.

Type list

**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

**default_field_names** = ['field', 'krige_var', 'mean_field']

Default field names.

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 field_names
    Names of present fields.
    Type list

property field_shape
    The shape of the field.
    Type tuple

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 krig_size
    Size of the kriging system.
    Type int

property latlon
    Whether the field depends on geographical coords.
    Type bool

property mean
    The mean of the field.
    Type float or callable

property mesh_type
    The mesh type of the field.
    Type str

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 pos**
The position tuple of the field.

  Type `tuple`

**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`
class gstools.krige.Oldinary:

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 (nan values will be ignored)
- **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

- **all_fields** *list* - All fields as stacked list.
- **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.
field_names list: Names of present fields.
field_shape tuple: The shape of the field.
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.
latlon bool: Whether the field depends on geographical coords.
mean float or callable: The mean of the field.
mesh_type str: The mesh type of the field.
model CovModel: The covariance model of the field.
name str: The name of the class.
normalizer Normalizer: Normalizer of the field.
pos tuple: The position tuple 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.
delete_fields(select) Delete selected fields.
get_mean([post_process]) Calculate the estimated mean of the detrended field.
get_store_config(store[, default, fld_cnt]) Get storage configuration from given selection.
mesh(mesh[, points, direction, name]) Generate a field on a given meshio, ogs5py or PyVista mesh.
plot([field, fig, ax]) Plot the spatial random field.
post_field(field[, name, process, save]) Postprocessing field values.
pre_pos([pos, mesh_type, info]) 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.

continues on next page
Table 48 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_pos(pos[, mesh_type, info])</code></td>
<td>Set positions and mesh_type.</td>
</tr>
<tr>
<td><code>structured(*args, **kwargs)</code></td>
<td>Generate a field on a structured mesh.</td>
</tr>
<tr>
<td><code>to_pyvista(field_select, fieldname)</code></td>
<td>Create a VTK/PyVista grid of the stored field.</td>
</tr>
<tr>
<td><code>transform(method[, field, store, process])</code></td>
<td>Apply field transformation.</td>
</tr>
<tr>
<td><code>unstructured(*args, **kwargs)</code></td>
<td>Generate a field on an unstructured mesh.</td>
</tr>
<tr>
<td><code>vtk_export(filename[, field_select, fieldname])</code></td>
<td>Export the stored field to vtk.</td>
</tr>
</tbody>
</table>

__call__ (pos=None, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True, store=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, optional) – 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
- `store` (str or bool or list, optional) – Whether to store kriging fields (True/False) with default name or with specified names. The default is True for default names ["field", "krige_var"] or "mean_field" if only_mean=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)

**delete_fields** (select=None)

Delete selected fields.

**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.
get_store_config(store, default=None, fld_cnt=None)
Get storage configuration from given selection.

Parameters

• **store** (str or bool or list, optional) – Whether to store fields (True/False) with default names or with specified names. The default is True for default names.

• **default** (str or list, optional) – Default field names. The default is “field”.

• **fld_cnt** (None or int, optional) – Number of fields when using lists. The default is None.

Returns

• **name** (str or list) – Name(s) of field.

• **save** (bool or list) – Whether to save field(s).

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

Parameters

• **mesh** (meshio.Mesh or ogs5py.MSH or PyVista mesh) – The given 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 __call__.

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

See:

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

• ogs5py: https://github.com/GeoStat-Framework/ogs5py

• PyVista: https://github.com/pyvista/pyvista

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=None, mesh_type='unstructured', info=False*)

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”
- **info** *(bool, optional)* – Whether to return information

Returns

- **iso_pos** *((d, n), numpy.ndarray)* – Isometrized position tuple.
- **shape** *(tuple)* – Shape of the resulting field.
- **info** *(dict, optional)* – Information about settings.

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

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 (nan values will be ignored). 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.*

set_pos*(pos, mesh_type='unstructured', info=False)*

Set positions and mesh_type.

**Parameters**

- **pos** *(iterable) – the position tuple, containing main direction and transversal directions*
- **mesh_type** *(str, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”*
- **info** *(bool, optional) – Whether to return information*

**Returns** *info – Information about settings.*

Return type *dict, optional*

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

structured(*args, **kwargs)*

Generate a field on a structured mesh.

See __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”*

transform(*method, field='field', store=True, process=False, **kwargs)*

Apply field transformation.

**Parameters**

- **method** *(str) – Method to use. See gstools.transform for available transformations.*
- **field** *(str, optional) – Name of field to be transformed. The default is “field”.*
- **store** *(str or bool, optional) – Whether to store field in-place (True/False) or under a given name. The default is True.*
- **process** *(bool, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.*
- ****kwargs – Keyword arguments forwarded to selected method.
Raises `ValueError` – When method is unknown.

Returns Transformed field.

Return type `numpy.ndarray`

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

See `__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 `all_fields`
All fields as stacked list.

Type `list`

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`

`default_field_names = ['field', 'krige_var', 'mean_field']`
Default field names.

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 field_names
    Names of present fields.
        Type list

property field_shape
    The shape of the field.
        Type tuple

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 kriging_size
    Size of the kriging system.
        Type int

property latlon
    Whether the field depends on geographical coords.
        Type bool

property mean
    The mean of the field.
        Type float or callable

property mesh_type
    The mesh type of the field.
        Type str

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 pos**
The position tuple of the field.

Type `tuple`

**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`
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 (nan values will be ignored)
- **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 trend 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)* – Whether 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**

- **all_fields** list: All fields as stacked list.
- **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.
- **field_names** list: Names of present fields.
- **field_shape** tuple: The shape of the field.
- **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.
- **latlon** bool: Whether the field depends on geographical coords.
- **mean** float or callable: The mean of the field.
- **mesh_type** str: The mesh type of the field.
- **model** *CovModel*: The covariance model of the field.
- **name** str: The name of the class.
- **normalizer** *Normalizer*: Normalizer of the field.
- **pos** tuple: The position tuple 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.
delete_fields(select) Delete selected fields.
get_mean(post_process) Calculate the estimated mean of the detrended field.
get_store_config(store[, default, fld_cnt]) Get storage configuration from given selection.
mesh(mesh[, points, direction, name]) Generate a field on a given meshio, ogs5py or PyVista mesh.
plot([field, fig, ax]) Plot the spatial random field.
post_field(field[, name, process, save]) Postprocessing field values.
pre_pos([pos, mesh_type, info]) 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.
set_pos(pos[, mesh_type, info]) Set positions and mesh_type.
structured(*args, **kwargs) Generate a field on a structured mesh.
to_pyvista([field_select, fieldname]) Create a VTK/PyVista grid of the stored field.
transform(method[, field, store, process]) Apply field transformation.
unstructured(*args, **kwargs) Generate a field on an unstructured mesh.
vtk_export(filename[, field_select, fieldname]) Export the stored field to vtk.

__call__(pos=None, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True, store=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, optional) – 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
- store (str or bool or list, optional) – Whether to store kriging fields (True/False) with default name or with specified names. The default is True for default names [“field”, “krige_var”] or “mean_field” if only_mean=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)

delete_fields(select=None) Delete selected fields.
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.

get_store_config(store, default=None, fld_cnt=None)
Get storage configuration from given selection.

Parameters
• store (str or bool or list, optional) – Whether to store fields (True/False) with default names or with specified names. The default is True for default names.
• default (str or list, optional) – Default field names. The default is “field”.
• fld_cnt (None or int, optional) – Number of fields when using lists. The default is None.

Returns
• name (str or list) – Name(s) of field.
• save (bool or list) – Whether to save field(s).

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

Parameters
• mesh (meshio.Mesh or ogs5py.MSH or PyVista mesh) – The given 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 __call__.

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

See:
• meshio: https://github.com/nschloe/meshio
• ogs5py: https://github.com/GeoStat-Framework/ogs5py
• PyVista: https://github.com/pyvista/pyvista
**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=None, mesh_type='unstructured', info=False)*

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”
- **info** *(bool, optional)* – Whether to return information

**Returns**

- **iso_pos** *((d, n), numpy.ndarray)* – Isometrized position tuple.
- **shape** *(tuple)* – Shape of the resulting field.
- **info** *(dict, optional)* – Information about settings.

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

**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 (nan values will be ignored). 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.

**set_pos**(*pos, mesh_type='unstructured', info=False*)

Set positions and mesh_type.

**Parameters**

• **pos** (*iterable*) – the position tuple, containing main direction and transversal directions

• **mesh_type** (*str*, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”

• **info** (*bool*, optional) – Whether to return information

**Returns** *info* – Information about settings.

**Return type** *dict*, optional

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

**structured**(**args, **kwargs*)

Generate a field on a structured mesh.

See **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”
transform(method, field='field', store=True, process=False, **kwargs)

Apply field transformation.

Parameters

- **method** (str) – Method to use. See gstools.transform for available transformations.
- **field** (str, optional) – Name of field to be transformed. The default is “field”.
- **store** (str or bool, optional) – Whether to store field inplace (True/False) or under a given name. The default is True.
- **process** (bool, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- **kwargs** – Keyword arguments forwarded to selected method.

Raises ValueError – When method is unknown.

Returns Transformed field.

Return type numpy.ndarray

unstructured(*args, **kwargs)

Generate a field on an unstructured mesh.

See __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 all_fields

All fields as stacked list.

Type list

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

default_field_names = ['field', 'krige_var', 'mean_field']
Default field names.
    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 field_names
Names of present fields.
    Type list

property field_shape
The shape of the field.
    Type tuple

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 latlon
Whether the field depends on geographical coords.
    Type bool

property mean
The mean of the field.
property mesh_type
    The mesh type of the field.
    
    Type str

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 pos
    The position tuple of the field.
    
    Type tuple

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 (nan values will be ignored)
- **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)** – Whether to fit the data-normalizer to the given conditioning data. Default: False
- **fit_variogram (bool, optional)** – Whether 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

- **all_fields list**: All fields as stacked list.
- **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.
- **field_names list**: Names of present fields.
- **field_shape tuple**: The shape of the field.
- **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.
- **latlon bool**: Whether the field depends on geographical coords.
- **mean float or callable**: The mean of the field.
- **mesh_type str**: The mesh type of the field.
- **model CovModel**: The covariance model of the field.
- **name str**: The name of the class.
- **normalizer Normalizer**: Normalizer of the field.
- **pos tuple**: The position tuple 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=None, mesh_type=’unstructured’, ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True, store=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, optional) – 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
- **store** (str or bool or list, optional) – Whether to store kriging fields (True/False) with default name or with specified names. The default is True for default names [“field”, “krige_var”] or “mean_field” if only_mean=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)

delete_fields(select=None)
Delete selected fields.
**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.

**get_store_config**(*store, default=None, fld_cnt=None*)

Get storage configuration from given selection.

**Parameters**

- **store** *(str or bool or list, optional)* – Whether to store fields (True/False) with default names or with specified names. The default is *True* for default names.

- **default** *(str or list, optional)* – Default field names. The default is “field”.

- **fld_cnt** *(None or int, optional)* – Number of fields when using lists. The default is *None*.

**Returns**

- **name** *(str or list)* – Name(s) of field.

- **save** *(bool or list)* – Whether to save field(s).

**mesh**(*mesh, points='centroids', direction='all', name='field', **kwargs*)

Generate a field on a given meshio, ogs5py or PyVista mesh.

**Parameters**

- **mesh** *(meshio.Mesh or ogs5py.MSH or PyVista mesh)* – The given 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 __call__.

**Notes**

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

**See:**

- meshio: [https://github.com/nschloe/meshio](https://github.com/nschloe/meshio)
- PyVista: [https://github.com/pyvista/pyvista](https://github.com/pyvista/pyvista)
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=None, mesh_type='unstructured', info=False)
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”*
• **info** *(bool, optional) – Whether to return information*

Returns

• **iso_pos** *((d, n), numpy.ndarray) – Isometrized position tuple.*
• **shape** *(tuple) – Shape of the resulting field.*
• **info** *(dict, optional) – Information about settings.*

Warning: When setting a new position tuple that differs from the present one, all stored fields will be deleted.

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 (nan values will be ignored). 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) – Whether to fit the data-normalizer to the given conditioning data. Default: False

• **fit_variogram** (bool, optional) – Whether 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

```python
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** Value Error – If the given drift functions are not callable.

```python
set_pos(pos, mesh_type='unstructured', info=False)
```
Set positions and mesh_type.

**Parameters**
- **pos** ([`iterable`]) – the position tuple, containing main direction and transversal directions
- **mesh_type** ([`str`], optional) – ’structured’ / ’unstructured’ Default: “unstructured”
- **info** (bool, optional) – Whether to return information

**Returns** info – Information about settings.

**Return type** dict, optional

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

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

See `__call__`

```python
to_pyvista(field_select='field', filename='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”
- **filename** ([`str`], optional) – Name of the field in the VTK file. Default: “field”
transform(method, field='field', store=True, process=False, **kwargs)
Apply field transformation.

Parameters
- **method** (str) – Method to use. See `gstools.transform` for available transformations.
- **field** (str, optional) – Name of field to be transformed. The default is “field”.
- **store** (str or bool, optional) – Whether to store field inplace (True/False) or under a given name. The default is True.
- **process** (bool, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- ****kwargs – Keyword arguments forwarded to selected method.

Raises `ValueError` – When method is unknown.

Returns Transformed field.
Return type `numpy.ndarray`

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

See `__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 all_fields
All fields as stacked list.
Type list

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

default_field_names = ['field', 'krige_var', 'mean_field']
    Default field names.
    
    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 field_names
    Names of present fields.
    
    Type list

property field_shape
    The shape of the field.
    
    Type tuple

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 latlon
    Whether the field depends on geographical coords.
    
    Type bool

property mean
    The mean of the field.
property `mesh_type`
The mesh type of the field.

Type `str`

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 `pos`
The position tuple of the field.

Type `tuple`

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**
```python
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 (nan values will be ignored)
- `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 <= 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**

- `all_fields` list: All fields as stacked list.
- `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.
field_names list: Names of present fields.
field_shape tuple: The shape of the field.
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.
latlon bool: Whether the field depends on geographical coords.
mean float or callable: The mean of the field.
mesh_type str: The mesh type of the field.
model CovModel: The covariance model of the field.
name str: The name of the class.
normalizer Normalizer: Normalizer of the field.
pos tuple: The position tuple 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.
delete_fields([select]) Delete selected fields.
get_mean([post_process]) Calculate the estimated mean of the detrended field.
get_store_config(store[, default, fld_cnt]) Get storage configuration from given selection.
mesh([mesh, points, direction, name]) Generate a field on a given meshio, ogs5py or PyVista mesh.
plot([field, fig, ax]) Plot the spatial random field.
post_field([field, name, process, save]) Postprocessing field values.
pre_pos([pos, mesh_type, info]) 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.
set_pos([pos[, mesh_type, info]]) Set positions and mesh_type.
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 51 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform(method[, field, store, process])</td>
<td>Apply field transformation.</td>
</tr>
<tr>
<td>unstructured(*args, **kwargs)</td>
<td>Generate a field on an unstructured mesh.</td>
</tr>
<tr>
<td>vtk_export(filename[, field_select, fieldname])</td>
<td>Export the stored field to vtk.</td>
</tr>
</tbody>
</table>

__call__(pos=None, mesh_type='unstructured', ext_drift=None, chunk_size=None, only_mean=False, return_var=True, post_process=True, store=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, optional)* – 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
- `store` *(str or bool or list, optional)* – Whether to store kriging fields (True/False) with default name or with specified names. The default is True for default names [“field”, “krige_var”] or “mean_field” if only_mean=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)

__delete_fields__(select=None)

Delete selected fields.

__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.

__get_store_config__(store, default=None, fld_cnt=None)

Get storage configuration from given selection.

**Parameters**

3.9. gstools.krige
• **store** *(str or bool or list, optional)* – Whether to store fields (True/False) with default names or with specified names. The default is True for default names.

• **default** *(str or list, optional)* – Default field names. The default is “field”.

• **fld_cnt** *(None or int, optional)* – Number of fields when using lists. The default is None.

Returns

• **name** *(str or list)* – Name(s) of field.

• **save** *(bool or list)* – Whether to save field(s).

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

Parameters

• **mesh** *(meshio.Mesh or ogs5py.MSH or PyVista mesh)* – The given 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 __call__.

Notes

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

See:

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

• ogs5py: https://github.com/GeoStat-Framework/ogs5py

• PyVista: https://github.com/pyvista/pyvista

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=None, mesh_type='unstructured', info=False)*

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”

• **info** *(bool, optional)* – Whether to return information

Returns

• **iso_pos** *((d, n), numpy.ndarray)* – Isometrized position tuple.

• **shape** *(tuple)* – Shape of the resulting field.

• **info** *(dict, optional)* – Information about settings.

**Warning:** When setting a new position tuple that differs from the present one, all stored fields will be deleted.

**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 (nan values will be ignored). 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

3.9. **gstools.krige** 413
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.

set_pos(pos, mesh_type='unstructured', info=False)
Set positions and mesh_type.

Parameters
• pos (iterable) – the position tuple, containing main direction and transversal directions
• mesh_type (str, optional) – ‘structured’ / ‘unstructured’ Default: “unstructured”
• info (bool, optional) – Whether to return information

Returns info – Information about settings.

Return type dict, optional

Warning: When setting a new position tuple that differs from the present one, all stored fields will be deleted.

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

See __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”

transform(method, field='field', store=True, process=False, **kwargs)
Apply field transformation.

Parameters
• method (str) – Method to use. See gstools.transform for available transformations.
• field (str, optional) – Name of field to be transformed. The default is “field”.
• store (str or bool, optional) – Whether to store field inplace (True/False) or under a given name. The default is True.
• process (bool, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
• **kwargs – Keyword arguments forwarded to selected method.

Raises ValueError – When method is unknown.

Returns Transformed field.

Return type numpy.ndarray
unstructured(*args, **kwargs)
    Generate a field on an unstructured mesh.
    See __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 all_fields
    All fields as stacked list.
    Type list

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

default_field_names = ['field', 'krige_var', 'mean_field']
    Default field names.
    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 field_names
    Names of present fields.
    Type list

property field_shape
    The shape of the field.
    Type tuple

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 krig_size
    Size of the kriging system.
    Type int

property latlon
    Whether the field depends on geographical coords.
    Type bool

property mean
    The mean of the field.
    Type float or callable

property mesh_type
    The mesh type of the field.
    Type str

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 pos
    The position tuple of the field.
Type  tuple

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

\[ \text{RNG}([\text{seed}]) \]
A random number generator for different distributions and multiple streams.

gstools.random.RNG

class gstools.random.RNG(\text{seed}=None)

Bases: \text{object}

A random number generator for different distributions and multiple streams.

Parameters

- \text{seed} (int or \text{None}, optional) – The seed of the master RNG, if \text{None}, a random seed is used. Default: \text{None}

Attributes

- \text{random} \text{numpy.random.RandomState}: Randomstate.
- \text{seed} \text{int}: Seed of the master RNG.

Methods

- \text{sample_dist}([\text{pdf, cdf, ppf, size}])\text{ Sample from a distribution given by pdf, cdf and/or ppf.}
- \text{sample_ln_pdf}(\text{ln_pdf}, \text{size}, \text{sample_around}, ...)]\text{ Sample from a distribution given by ln(pdf).}
- \text{sample_sphere}(\text{dim}, \text{size})\text{ Uniform sampling on a d-dimensional sphere.}

\text{sample_dist}(\text{pdf}=None, \text{cdf}=None, \text{ppf}=None, \text{size}=None, **\text{kwargs})

Sample from a distribution given by pdf, cdf and/or ppf.

Parameters

- \text{pdf} (callable or \text{None}, optional) – Probability density function of the given distribution, that takes a single argument Default: \text{None}
- \text{cdf} (callable or \text{None}, optional) – Cumulative distribution function of the given distribution, that takes a single argument Default: \text{None}
- \text{ppf} (callable or \text{None}, optional) – Percent point function of the given distribution, that takes a single argument Default: \text{None}
- \text{size} (int or \text{None}, optional) – sample size. Default: \text{None}
- **\text{kwargs} – Keyword-arguments that are forwarded to \text{scipy.stats.rv_continuous.}

Returns \text{samples} – the samples from the given distribution

Return type \text{float} or \text{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
Seed Generator

\textit{MasterRNG}(seed) \hspace{1cm} \text{Master random number generator for generating seeds.}

\texttt{gstools.random.MasterRNG}

class \texttt{gstools.random.MasterRNG}(seed)

\hspace{1cm} Bases: object

\hspace{1cm} Master random number generator for generating seeds.

\hspace{1cm} Parameters \texttt{seed} (int or \texttt{None}, optional) – The seed of the master RNG, if \texttt{None}, a random seed is used. Default: \texttt{None}

\hspace{1cm} Attributes

\hspace{1cm} \texttt{seed} int: Seed of the master RNG.

Methods

\texttt{__call__()} \hspace{1cm} Return a random seed.

\texttt{__call__()} \hspace{1cm} Return a random seed.

\texttt{property seed} \hspace{1cm} Seed of the master RNG.

\hspace{1cm} The setter property not only saves the new seed, but also creates a new master RNG function with the new seed.

\hspace{1cm} Type int
Distribution factory

\[ \texttt{dist} \texttt{\_} \texttt{gen}([\texttt{pdf} \texttt{\_} \texttt{in}, \texttt{cdf} \texttt{\_} \texttt{in}, \texttt{ppf} \texttt{\_} \texttt{in}]) \] Distribution Factory.

\texttt{gstools.random.dist\_gen}

\texttt{gstools.random.dist\_gen}(pdf\_in=None, cdf\_in=None, ppf\_in=None, **kwargs)
Distribution Factory.

Parameters

- \texttt{pdf\_in} (callable or None, optional) – Probability distribution function of the given distribution, that takes a single argument Default: None
- \texttt{cdf\_in} (callable or None, optional) – Cumulative distribution function of the given distribution, that takes a single argument Default: None
- \texttt{ppf\_in} (callable or None, optional) – Percent point function of the given distribution, that takes a single argument Default: None
- \texttt{**kwargs} – Keyword-arguments forwarded to \texttt{scipy.stats.rv\_continuous}.

Returns dist – The constructed distribution.

Return type \texttt{scipy.stats.rv\_continuous}

Notes

At least pdf or cdf needs to be given.
3.11 gstools.tools

GStools subpackage providing miscellaneous tools.

Export

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vtk_export(filename, pos, fields[, mesh_type])</code></td>
<td>Export a field to vtk.</td>
</tr>
<tr>
<td><code>vtk_export_structured(filename, pos, fields)</code></td>
<td>Export a field to vtk structured rectilinear grid file.</td>
</tr>
<tr>
<td><code>vtk_export_unstructured(filename, pos, fields)</code></td>
<td>Export a field to vtk unstructured grid file.</td>
</tr>
<tr>
<td><code>to_vtk(pos, fields[, mesh_type])</code></td>
<td>Create a VTK/PyVista grid.</td>
</tr>
<tr>
<td><code>to_vtk_structured(pos, fields)</code></td>
<td>Create a vtk structured rectilinear grid from a field.</td>
</tr>
<tr>
<td><code>to_vtk_unstructured(pos, fields)</code></td>
<td>Export a field to vtk structured rectilinear grid file.</td>
</tr>
</tbody>
</table>

**gstools.tools.vtk_export**

`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**

`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**

`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.to_vtk

**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

**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 theirs 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

**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 theirs 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*
## Special functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>confidence_scaling([per])</code></td>
<td>Scaling of standard deviation to get the desired confidence interval.</td>
</tr>
<tr>
<td><code>inc_gamma(s, x)</code></td>
<td>Calculate the (upper) incomplete gamma function.</td>
</tr>
<tr>
<td><code>inc_gamma_low(s, x)</code></td>
<td>Calculate the lower incomplete gamma function.</td>
</tr>
<tr>
<td><code>exp_int(s, x)</code></td>
<td>Calculate the exponential integral $E_s(x)$.</td>
</tr>
<tr>
<td><code>inc_beta(a, b, x)</code></td>
<td>Calculate the incomplete Beta function.</td>
</tr>
<tr>
<td><code>tplstable_cor(r, len_scale, hurst, alpha)</code></td>
<td>Calculate the correlation function of the TPLStable model.</td>
</tr>
<tr>
<td><code>tpl_exp_spec_dens(k, dim, len_scale, hurst)</code></td>
<td>Spectal density of the TPLExponential covariance model.</td>
</tr>
<tr>
<td><code>tpl_gau_spec_dens(k, dim, len_scale, hurst)</code></td>
<td>Spectal density of the TPLGaussian covariance model.</td>
</tr>
</tbody>
</table>

### gstools.tools.confidence_scaling

`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.inc_gamma

`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.inc_gamma_low

`gstools.tools.inc_gamma_low(s, x)`

Calculate the lower incomplete gamma function.

Given by: $\gamma(s, x) = \int_0^x t^{s-1} e^{-t} \, dt$

**Parameters**

- **s** (float) – exponent in the integral
- **x** (numpy.ndarray) – input values
gstools.tools.exp_int

gstools.tools.exp_int(s, x)
Calculate the exponential integral $E_s(x)$.
Given by: $E_s(x) = \int_1^{\infty} e^{-xt} \frac{dt}{t^s}$

Parameters
- s (float) – exponent in the integral (should be > -100)
- x (numpy.ndarray) – input values

gstools.tools.inc_beta

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.tplstable_cor

gstools.tools.tplstable_cor(r, len_scale, hurst, alpha)
Calculate the correlation function of the TPLStable model.
Given by the following correlation function:
$$\rho(r) = 2H \alpha \cdot E_1 \left( \frac{\alpha}{\ell} \right)$$

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.tpl_exp_spec_dens

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: $k = \|k\|$  
- 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
**gstools.tools.tpl_gau_spec_dens**

Spectral density of the TPLGaussian covariance model.

**Parameters**

- **k (float)** – Radius of the phase: $k = \|k\|$
- **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

---

**Geometric**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rotated_main_axes</strong>(dim, angles)</td>
<td>Create list of the main axis defined by the given system rotations.</td>
</tr>
<tr>
<td><strong>set_angles</strong>(dim, angles)</td>
<td>Set the angles for the given dimension.</td>
</tr>
<tr>
<td><strong>set_anis</strong>(dim, anis)</td>
<td>Set the anisotropy ratios for the given dimension.</td>
</tr>
<tr>
<td><strong>no_of_angles</strong>(dim)</td>
<td>Calculate number of rotation angles depending on the dimension.</td>
</tr>
<tr>
<td><strong>rotation_planes</strong>(dim)</td>
<td>Get all 2D sub-planes for rotation.</td>
</tr>
<tr>
<td><strong>givens_rotation</strong>(dim, plane, angle)</td>
<td>Givens rotation matrix in arbitrary dimensions.</td>
</tr>
<tr>
<td><strong>matrix_rotate</strong>(dim, angles)</td>
<td>Create a matrix to rotate points to the target coordinate-system.</td>
</tr>
<tr>
<td><strong>matrix_derotate</strong>(dim, angles)</td>
<td>Create a matrix to derotate points to the initial coordinate-system.</td>
</tr>
<tr>
<td><strong>matrix_isotropify</strong>(dim, anis)</td>
<td>Create a stretching matrix to make things isotrope.</td>
</tr>
<tr>
<td><strong>matrix_anisotropify</strong>(dim, anis)</td>
<td>Create a stretching matrix to make things anisotrope.</td>
</tr>
<tr>
<td><strong>matrix_isometrize</strong>(dim, angles, anis)</td>
<td>Create a matrix to derotate points and make them isotrope.</td>
</tr>
<tr>
<td><strong>matrix_anisometrize</strong>(dim, angles, anis)</td>
<td>Create a matrix to rotate points and make them anisotrope.</td>
</tr>
<tr>
<td><strong>ang2dir</strong>(angles[, dtype, dim])</td>
<td>Convert n-D spherical coordinates to Euclidean direction vectors.</td>
</tr>
<tr>
<td><strong>generate_grid</strong>(pos)</td>
<td>Generate grid from a structured position tuple.</td>
</tr>
<tr>
<td><strong>generate_st_grid</strong>(pos, time[, mesh_type])</td>
<td>Generate spatio-temporal grid from a position tuple and time array.</td>
</tr>
</tbody>
</table>
**gstools.tools.rotated_main_axes**

**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.set_angles**

**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**

**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.no_of_angles

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.rotation_planes

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.givens_rotation

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.matrix_rotate

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.matrix_derotate

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_isotropify

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_anisotropify

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_isometrize

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_anisometrize

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.ang2dir

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.generate_grid

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

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.

Misc

| EARTH_RADIUS | earth radius for WGS84 ellipsoid in km |
3.12 gstools.transform

GStools subpackage providing transformations to post-process normal fields.

Wrapper

```
apply(fld, method[, field, store, process])  Apply field transformation.
```

**gstools.transform.apply**

```
gstools.transform.apply(fld, method, field='field', store=True, process=False, **kwargs)
```

Apply field transformation.

**Parameters**

- `fld (Field)` – Field class containing a generated field.
- `method (str)` – Method to use. See `gstools.transform` for available transformations.
- `field (str, optional)` – Name of field to be transformed. The default is “field”.
- `store (str or bool, optional)` – Whether to store field inplace (True/False) or with a specified name. The default is True.
- `process (bool, optional)` – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- `**kwargs` – Keyword arguments forwarded to selected method.

**Raises** `ValueError` – When method is unknown.

**Returns** Transformed field.

**Return type** `numpy.ndarray`

### Field Transformations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>binary</strong></td>
<td>Binary transformation.</td>
</tr>
<tr>
<td><strong>discrete</strong></td>
<td>Discrete transformation.</td>
</tr>
<tr>
<td><strong>boxcox</strong></td>
<td>(Inverse) Box-Cox transformation to denormalize data.</td>
</tr>
<tr>
<td><strong>zinnharvey</strong></td>
<td>Zinn and Harvey transformation to connect low or high values.</td>
</tr>
<tr>
<td><strong>normal_force_moments</strong></td>
<td>Force moments of a normal distributed field.</td>
</tr>
<tr>
<td><strong>normal_to_lognormal</strong></td>
<td>Transform normal distribution to log-normal distribution.</td>
</tr>
<tr>
<td><strong>normal_to_uniform</strong></td>
<td>Transform normal distribution to uniform distribution on [0, 1].</td>
</tr>
<tr>
<td><strong>normal_to_arcsin</strong></td>
<td>Transform normal distribution to the bimodal arcsin distribution.</td>
</tr>
<tr>
<td><strong>normal_to_uquad</strong></td>
<td>Transform normal distribution to U-quadratic distribution.</td>
</tr>
<tr>
<td><strong>apply_function</strong></td>
<td>Apply function as field transformation.</td>
</tr>
</tbody>
</table>
**gstools.transform.binary**

```python
gstools.transform.binary(fld, divide=None, upper=None, lower=None, field='field', store=True, process=False, keep_mean=True)
```

Binary transformation.

After this transformation, the field only has two values.

**Parameters**

- **fld** *(Field)* – Field class containing a generated field.
- **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)`
- **field** *(str, optional)* – Name of field to be transformed. The default is “field”.
- **store** *(str or bool, optional)* – Whether to store field inplace (True/False) or under a given name. The default is True.
- **process** *(bool, optional)* – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- **keep_mean** *(bool, optional)* – Whether to keep the mean of the field if process=True. The default is True.

**Returns** Transformed field.

**Return type** `numpy.ndarray`

**gstools.transform.discrete**

```python
gstools.transform.discrete(fld, values, thresholds='arithmetic', field='field', store=True, process=False, keep_mean=True)
```

Discrete transformation.

After this transformation, the field has only `len(values)` discrete values.

**Parameters**

- **fld** *(Field)* – Field class containing a generated field.
- **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”
- **field** *(str, optional)* – Name of field to be transformed. The default is “field”.
- **store** *(str or bool, optional)* – Whether to store field inplace (True/False) or under a given name. The default is True.
- **process** *(bool, optional)* – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- **keep_mean** *(bool, optional)* – Whether to keep the mean of the field if process=True. The default is True.

**Returns** Transformed field.

**Return type** `numpy.ndarray`
gstools.transform.boxcox

```
gstools.transform.boxcox(fld, lmbda=1, shift=0, field='field', store=True, process=False, keep_mean=True)
```

(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)` – Field class containing a generated field.
- `lmbda (float, optional)` – The lambda parameter of the Box-Cox transformation. For \( \lambda = 0 \) 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
- `field (str, optional)` – Name of field to be transformed. The default is “field”.
- `store (str or bool, optional)` – Whether to store field inplace (True/False) or under a given name. The default is True.
- `process (bool, optional)` – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- `keep_mean (bool, optional)` – Whether to keep the mean of the field if process=True. The default is True.

**Returns** Transformed field.

**Return type** `numpy.ndarray`

---

gstools.transform.zinnharvey

```
gstools.transform.zinnharvey(fld, conn='low', field='field', store=True, process=False, keep_mean=True)
```

Zinn and Harvey transformation to connect low or high values.

After this transformation, the field is still normal distributed.

**Parameters**

- `fld (Field)` – Field class containing a generated field.
- `conn (str, optional)` – Desired connectivity. Either “low” or “high”. Default: “high”
- `field (str, optional)` – Name of field to be transformed. The default is “field”.
- `store (str or bool, optional)` – Whether to store field inplace (True/False) or under a given name. The default is True.
- `process (bool, optional)` – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- `keep_mean (bool, optional)` – Whether to keep the mean of the field if process=True. The default is True.

**Returns** Transformed field.

**Return type** `numpy.ndarray`
gstools.transform.normal_force_moments

 gstools.transform.normal_force_moments(fld, field='field', store=True, process=False, keep_mean=True)

 Force moments of a normal distributed field.
 After this transformation, the field is still normal distributed.

 Parameters

 • fld (Field) – Field class containing a generated field.
 • field (str, optional) – Name of field to be transformed. The default is “field”.
 • store (str or bool, optional) – Whether to store field inplace (True/False) or under a given name. The default is True.
 • process (bool, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
 • keep_mean (bool, optional) – Whether to keep the mean of the field if process=True. The default is True.

 Returns Transformed field.
 Return type numpy.ndarray

gstools.transform.normal_to_lognormal

 gstools.transform.normal_to_lognormal(fld, field='field', store=True, process=False, keep_mean=True)

 Transform normal distribution to log-normal distribution.
 After this transformation, the field is log-normal distributed.

 Parameters

 • fld (Field) – Field class containing a generated field.
 • field (str, optional) – Name of field to be transformed. The default is “field”.
 • store (str or bool, optional) – Whether to store field inplace (True/False) or under a given name. The default is True.
 • process (bool, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
 • keep_mean (bool, optional) – Whether to keep the mean of the field if process=True. The default is True.

 Returns Transformed field.
 Return type numpy.ndarray

gstools.transform.normal_to_uniform

 gstools.transform.normal_to_uniform(fld, field='field', store=True, process=False, keep_mean=True)

 Transform normal distribution to uniform distribution on [0, 1].
 After this transformation, the field is uniformly distributed on [0, 1].

 Parameters

 • fld (Field) – Field class containing a generated field.
 • keep_mean (bool, optional) – Whether to keep the mean of the field if process=True. The default is True.
gstools.transform.normal_to_arcsin

```python
gstools.transform.normal_to_arcsin(fld, a=None, b=None, field='field', store=True, process=False, keep_mean=True)
```

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*) – Field class containing a generated field.
- `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
- `field` (*str*, optional) – Name of field to be transformed. The default is “field”.
- `store` (*str* or *bool*, optional) – Whether to store field in-place (True/False) or under a given name. The default is True.
- `process` (*bool*, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- `keep_mean` (*bool*, optional) – Whether to keep the mean of the field if process=True. The default is True.

**Returns**

Transformed field.

**Return type**

`numpy.ndarray`

---

gstools.transform.normal_to_uquad

```python
gstools.transform.normal_to_uquad(fld, a=None, b=None, field='field', store=True, process=False, keep_mean=True)
```

Transform normal distribution to U-quadratic distribution.


After this transformation, the field is U-quadratic-distributed on [a, b].

**Parameters**

- `fld` (*Field*) – Field class containing a generated field.
- `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
- `field` (*str*, optional) – Name of field to be transformed. The default is “field”.
- `store` (*str* or *bool*, optional) – Whether to store field in-place (True/False) or under a given name. The default is True.
- `process` (*bool*, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- `keep_mean` (*bool*, optional) – Whether to keep the mean of the field if process=True. The default is True.

**Returns**

Transformed field.

**Return type**

`numpy.ndarray`
gstools.transform.apply_function

Apply function as field transformation.

Parameters

- **fld** (*Field*) – Field class containing a generated field.
- **function** (*callable*) – Function to use.
- **field** (*str*, optional) – Name of field to be transformed. The default is “field”.
- **store** (*str* or *bool*, optional) – Whether to store field inplace (True/False) or under a given name. The default is True.
- **process** (*bool*, optional) – Whether to process in/out fields with trend, normalizer and mean of given Field instance. The default is False.
- **keep_mean** (*bool*, optional) – Whether to keep the mean of the field if process=True. The default is True.
- ****kwargs – Keyword arguments forwarded to given function.

Returns

Transformed field.

Return type: *numpy.ndarray*

Array Transformations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_discrete(field, values[, thresholds, ...])</td>
<td>Discrete transformation.</td>
</tr>
<tr>
<td>array_boxcox(field[, lmbda, shift])</td>
<td>(Inverse) Box-Cox transformation to denormalize data.</td>
</tr>
<tr>
<td>array_zinnharvey(field[, conn, mean, var])</td>
<td>Zinn and Harvey transformation to connect low or high values.</td>
</tr>
<tr>
<td>array_force_moments(field[, mean, var])</td>
<td>Force moments of a normal distributed field.</td>
</tr>
<tr>
<td>array_to_lognormal(field)</td>
<td>Transform normal distribution to log-normal distribution.</td>
</tr>
<tr>
<td>array_to_uniform(field[, mean, var])</td>
<td>Transform normal distribution to uniform distribution on [0, 1].</td>
</tr>
<tr>
<td>array_to_arcsin(field[, mean, var, a, b])</td>
<td>Transform normal distribution to arcsin distribution.</td>
</tr>
<tr>
<td>array_to_uquad(field[, mean, var, a, b])</td>
<td>Transform normal distribution to U-quadratic distribution.</td>
</tr>
</tbody>
</table>

gstools.transform.array_discrete

Discrete transformation.

After this transformation, the field has only *len(values)* discrete values.

Parameters

- **field** (*numpy.ndarray*) – Normal distributed values.
- **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”: devidethe field into equal parts * an array of explicitly given thresholds Default: “arithmetic”

- **mean** (float or :any:`None`) – Mean of the field for “equal” thresholds. Default: np.mean(field)
- **var** (float or :any:`None`) – Variance of the field for “equal” thresholds. Default: np.var(field)

Returns Transformed field.

Return type `numpy.ndarray`

gstools.transform.array_boxcox
gstools.transform.array_boxcox(field, 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

- **field** (`numpy.ndarray`) – Normal distributed values.
- **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.array_zinnharvey
gstools.transform.array_zinnharvey(field, conn='high', mean=None, var=None)

Zinn and Harvey transformation to connect low or high values.

Parameters

- **field** (`numpy.ndarray`) – Normal distributed values.
- **conn** (str, optional) – Desired connectivity. Either “low” or “high”. Default: “high”
- **mean** (float or `None`, optional) – Mean of the given field. If None is given, the mean will be calculated. Default: `None`
- **var** (float or `None`, optional) – Variance of the given field. If None is given, the variance will be calculated. Default: `None`

Returns Transformed field.

Return type `numpy.ndarray`

gstools.transform.array_force_moments
gstools.transform.array_force_moments(field, mean=0, var=1)

Force moments of a normal distributed field.

Parameters

- **field** (`numpy.ndarray`) – Normal distributed values.
- **mean** (float, optional) – Desired mean of the field. Default: 0
- **var** (float or `None`, optional) – Desired variance of the field. Default: 1

Returns Transformed field.
Return type `numpy.ndarray`

`gstools.transform.array_to_lognormal`

`gstools.transform.array_to_lognormal(field)`  
Transform normal distribution to log-normal distribution.  

Parameters  
- `field` (`numpy.ndarray`) – Normal distributed values.  

Returns  
Transformed field.  

Return type `numpy.ndarray`

`gstools.transform.array_to_uniform`

`gstools.transform.array_to_uniform(field, mean=None, var=None)`  
Transform normal distribution to uniform distribution on [0, 1].  

Parameters  
- `field` (`numpy.ndarray`) – Normal distributed values.  
- `mean` (`float` or `None`, optional) – Mean of the given field. If None is given, the mean will be calculated. Default: `None`  
- `var` (`float` or `None`, optional) – Variance of the given field. If None is given, the variance will be calculated. Default: `None`  

Returns  
Transformed field.  

Return type `numpy.ndarray`

`gstools.transform.array_to_arcsin`

`gstools.transform.array_to_arcsin(field, mean=None, var=None, a=None, b=None)`  
Transform normal distribution to arcsin distribution.  

See: https://en.wikipedia.org/wiki/Arcsine_distribution  

Parameters  
- `field` (`numpy.ndarray`) – Normal distributed values.  
- `mean` (`float` or `None`, optional) – Mean of the given field. If None is given, the mean will be calculated. Default: `None`  
- `var` (`float` or `None`, optional) – Variance of the given field. If None is given, the variance will be calculated. Default: `None`  
- `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  

Returns  
Transformed field.  

Return type `numpy.ndarray`
gstools.transform.array_to_uquad

gstools.transform.array_to_uquad(field, mean=None, var=None, a=None, b=None)

Transform normal distribution to U-quadratic distribution.


**Parameters**

- **field** (numpy.ndarray) – Normal distributed values.
- **mean** (float or None, optional) – Mean of the given field. If None is given, the mean will be calculated. Default: None
- **var** (float or None, optional) – Variance of the given field. If None is given, the variance will be calculated. Default: None
- **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

**Returns**  Transformed field.

**Return type**  numpy.ndarray
3.13 gstools.normalizer

GStools subpackage providing normalization routines.

Base-Normalizer

```
Normalizer([data]) 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
    denormalize(data) Transform to input distribution.
    derivative(data) Factor for normal PDF to gain target PDF.
    fit(data[, skip]) Fitting the transformation to data by maximizing Log-Likelihood.
    kernel_loglikelihood(data) Kernel Log-Likelihood for given data with current parameters.
    likelihood(data) Likelihood for given data with current parameters.
    loglikelihood(data) Log-Likelihood for given data with current parameters.
    normalize(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 (not normal distributed).

Returns Log-Likelihood for given data with current parameters.

Return type numpy.ndarray
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 = (-\infty, \infty)

Valid range for input data.

Type tuple
Field-Normalizer

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LogNormal([data])</code></td>
<td>Log-normal fields.</td>
</tr>
<tr>
<td><code>BoxCox([data])</code></td>
<td>Box-Cox (1964) transformed fields.</td>
</tr>
<tr>
<td><code>BoxCoxShift([data])</code></td>
<td>Box-Cox (1964) transformed fields including shifting.</td>
</tr>
<tr>
<td><code>YeoJohnson([data])</code></td>
<td>Yeo-Johnson (2000) transformed fields.</td>
</tr>
<tr>
<td><code>Modulus([data])</code></td>
<td>Modulus or John-Draper (1980) transformed fields.</td>
</tr>
<tr>
<td><code>Manly([data])</code></td>
<td>Manly (1971) transformed fields.</td>
</tr>
</tbody>
</table>

**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**

- `denormalize(data)`
  - Transform to input distribution.

- `derivative(data)`
  - Factor for normal PDF to gain target PDF.

- `fit(data[, skip])`
  - Fitting the transformation to data by maximizing Log-Likelihood.

- `kernel_loglikelihood(data)`
  - Kernel Log-Likelihood for given data with current parameters.

- `likelihood(data)`
  - Likelihood for given data with current parameters.

- `loglikelihood(data)`
  - Log-Likelihood for given data with current parameters.

- `normalize(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** = {}  
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.
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.
- **lmbda** (float, optional) – Shape parameter. Default: 1

Notes

This transformation is given by [Box1964]:

\[
y = \begin{cases} 
    x^{\frac{\lambda - 1}{\lambda}} & \lambda \neq 0 \\
    \log(x) & \lambda = 0 
\end{cases}
\]

References

Attributes

- **denormalize_range** tuple: Valid range for output data depending on lmbda.
- **name** str: The name of the normalizer class.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>denormalize</strong>(data)</td>
<td>Transform to input distribution.</td>
</tr>
<tr>
<td><strong>derivative</strong>(data)</td>
<td>Factor for normal PDF to gain target PDF.</td>
</tr>
<tr>
<td><strong>fit</strong>(data[, skip])</td>
<td>Fitting the transformation to data by maximizing Log-Likelihood</td>
</tr>
<tr>
<td><strong>kernel_loglikelihood</strong>(data)</td>
<td>Kernel Log-Likelihood for given data with current parameters.</td>
</tr>
<tr>
<td><strong>likelihood</strong>(data)</td>
<td>Likelihood for given data with current parameters.</td>
</tr>
<tr>
<td><strong>loglikelihood</strong>(data)</td>
<td>Log-Likelihood for given data with current parameters.</td>
</tr>
<tr>
<td><strong>normalize</strong>(data)</td>
<td>Transform to normal distribution.</td>
</tr>
</tbody>
</table>

**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 skiped 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 = {'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:

```python
>>> 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**

- **denormalize**(data) – Transform to input distribution.
- **derivative**(data) – Factor for normal PDF to gain target PDF.
- **fit**(data[, skip]) – Fitting the transformation to data by maximizing Log-Likelihood.
- **kernel_loglikelihood**(data) – Kernel Log-Likelihood for given data with current parameters.
- **likelihood**(data) – Likelihood for given data with current parameters.
- **loglikelihood**(data) – Log-Likelihood for given data with current parameters.
- **normalize**(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 = {'lmbda': 1, 'shift': 0}

    Default parameters of the BoxCoxShift-Normalizer.

    Type dict

property denormalize_range

    Valid range for output data depending on lmbda.

    \((-1/lmbda, \infty)\) or \((-\infty, -1/lmbda)\)

    Type tuple

property name

    The name of the normalizer class.

    Type str

property normalize_range

    Valid range for input data depending on shift.

    \((-shift, \infty)\)

    Type tuple
class gstools.normalizer.YeoJohnson(data=None, **parameter)

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.
- **lambda** (float, optional) – Shape parameter. Default: 1

Notes

This transformation is given by [Yeo2000]:

\[
y = \begin{cases} 
\frac{(x+1)^{\lambda}-1}{\lambda} & x \geq 0, \lambda \neq 0 \\
\log(x+1) & x \geq 0, \lambda = 0 \\
\frac{(|x|+1)^{\lambda}-1}{2\lambda} & x < 0, \lambda \neq 2 \\
-\log(|x|+1) & x < 0, \lambda = 2 
\end{cases}
\]

References

Attributes

- **name** str: The name of the normalizer class.

Methods

- **denormalize** (data) Transform to input distribution.
- **derivative** (data) Factor for normal PDF to gain target PDF.
- **fit** (data[, skip]) Fitting the transformation to data by maximizing Log-Likelihood.
- **kernel_loglikelihood** (data) Kernel Log-Likelihood for given data with current parameters.
- **loglikelihood** (data) Log-Likelihood for given data with current parameters.
- **normalize** (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 = {'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]:

\[
\begin{align*}
    y &= \begin{cases} 
        \frac{\text{sgn}(x)|x|^{\lambda-1}}{\lambda} & \lambda \neq 0 \\
        \text{sgn}(x) \log(|x| + 1) & \lambda = 0 
    \end{cases}
\end{align*}
\]

References

Attributes

- **name** str: The name of the normalizer class.

Methods

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 (normal distributed).

Returns  
Fitted transformation.

Return type  
None
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 = {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 = (-\infty, \infty)
Valid range for input data.

Type tuple
**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]:

\[
    y = \begin{cases} 
        \frac{\exp(\lambda x) - 1}{\lambda} & \lambda \neq 0 \\
        x & \lambda = 0 
    \end{cases}
\]

**Attributes**

- **denormalize_range** `tuple`: Valid range for output data depending on lmbda.
- **name** `str`: The name of the normalizer class.

**Methods**

- `denormalize(data)` `Transform to input distribution.`
- `derivative(data)` `Factor for normal PDF to gain target PDF.`
- `fit(data[, skip])` `Fitting the transformation to data by maximizing Log-Likelihood.`
- `kernel_loglikelihood(data)` `Kernel Log-Likelihood for given data with current parameters.`
- `likelihood(data)` `Likelihood for given data with current parameters.`
- `loglikelihood(data)` `Log-Likelihood for given data with current parameters.`
- `normalize(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** = {'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

```python
gstools.normalizer.apply_mean_norm_trend

gstools.normalizer.apply_mean_norm_trend(pos, field[, mean, ...,])
Apply mean, de-normalization and trend to given field.

gstools.normalizer.remove_trend_norm_mean

gstools.normalizer.remove_trend_norm_mean(pos, field[, mean, ...,])
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.

Returns `field` – The transformed field.

Return type `numpy.ndarray`

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)* – Wheather 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.
All notable changes to GSTools will be documented in this file.

### 4.1 1.3.5 - Pure Pink - 2022-01

**Changes**

- remove caps for dependencies #229
- build linux wheels with manylinux2014 for all versions (CIBW v2.3.1) #227

**Bugfixes**

- Field.mesh was not compatible with meshio v5.1+ #227

### 4.2 1.3.4 - Pure Pink - 2021-11

**Enhancements**

- add GSTools-Core as optional dependency #215
- provide wheels for Python 3.10 #211
- provide macOS wheels for Apple Silicon #211

**Changes**

- remove unnecessary dim argument in Cython code #216
4.3 1.3.3 - Pure Pink - 2021-08

Enhancements

See: #197

- `gstools.transform`:
  - add keywords `field`, `store`, `process` and `keep_mean` to all transformations to control storage and respect normalizer
  - added `apply_function` transformation
  - added `apply` as wrapper for all transformations
  - added `transform` method to all `Field` (sub)classes as interface to `transform.apply`
  - added checks for normal fields to work smoothly with recently added normalizer submodule

- `Field`:
  - allow naming fields when generating and control storage with `store` keyword
  - all subclasses now have the `post_process` keyword (apply mean, normalizer, trend)
  - added subscription to access fields by name (`Field["field"]`)
  - added `set_pos` method to set position tuple
  - allow reusing present pos tuple
  - added `pos`, `mesh_type`, `field_names`, `field_shape`, `all_fields` properties

- `CondSRF`:
  - memory optimization by forwarding `pos` from underlying `krige` instance
  - only recalculate kriging field if `pos` tuple changed (optimized ensemble generation)

- performance improvement by using `np.asarray` instead of `np.array` where possible
- updated examples to use new features
- added incomplete lower gamma function `inc_gamma_low` (for TPLGaussian spectral density)
- filter `nan` values from `cond_val` array in all kriging routines #201

Bugfixes

- `inc_gamma` was defined wrong for integer `s < 0`

4.4 1.3.2 - Pure Pink - 2021-07

Bugfixes

- `vario_estimate` was altering the input field under certain circumstances #180
- `emcee` v3.1 now requires `nsteps` in `run_mcmc()` to be integer (called in `RNG.sample_ln_pdf`) #184
4.5 1.3.1 - Pure Pink - 2021-06

Enhancements

• Standalone use of Field class #166
• add social badges in README #169, #170

Bugfixes

• use oldest-supported-numpy to build cython extensions #165

4.6 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 $\text{variogram}(2 \times \sin(\zeta / 2))$
  - cov_yadrenko: given by $\text{covariance}(2 \times \sin(\zeta / 2))$
  - cor_yadrenko: given by $\text{correlation}(2 \times \sin(\zeta / 2))$
• 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
- A 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 in 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 earth's 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)

4.7 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

4.8 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 krige 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

4.9 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
4.10 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 kriging 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 29f7e1b
• 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 a6f5b9e

Bugfixes

• several minor bugfixes
4.11 1.0.1 - Bouncy Blue - 2019-01-18

Bugfixes

• fixed Numpy and Cython version during build process

4.12 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
4.13 0.4.0 - Glorious Green - 2018-07-17

Bugfixes

• import of cython functions put into a try-block

4.14 0.3.6 - Original Orange - 2018-07-17

First release of GSTools.


g

gstools.117

gstools.covmodel.120

gstools.covmodel.plot.331

gstools.field.333

gstools.field.generator.352

gstools.field.upscaling.358

gstools.krige.363

gstools.normalizer.441

gstools.random.418

gstools.tools.422

gstools.transform.432

gstools.variogram.359
Symbols

__call__ (gstools.field.CondSRF method), 341
__call__ (gstools.field.Field method), 347
__call__ (gstools.field.SRF method), 334
__call__ (gstools.field.generator.IncomprRandMeth method), 353
__call__ (gstools.field.generator.RandMeth method), 355
__call__ (gstools.krige.Detrended method), 411
__call__ (gstools.krige.ExtDrift method), 402
__call__ (gstools.krige.Gaussian method), 375
__call__ (gstools.krige.Ordinary method), 384
__call__ (gstools.krige.Simple method), 375
__call__ (gstools.krige.Universal method), 393
__call__ (gstools.random.MasterRNG method), 420

call (in module gstools.tools), 430

g2a (gstools.covmodel.Circular property), 224

g2a (gstools.covmodel.Cubic property), 201

g2a (gstools.covmodel.Exponential property), 153

g2a (gstools.covmodel.Gaussian property), 140

g2a (gstools.covmodel.HyperSpherical property), 248

g2a (gstools.covmodel.JBessel property), 273

g2a (gstools.covmodel.Linear property), 212

g2a (gstools.covmodel.Matern property), 165

g2a (gstools.covmodel.Rational property), 189

g2a (gstools.covmodel.Spherical property), 236

g2a (gstools.covmodel.Stable property), 176

g2a (gstools.covmodel.SuperSpherical property), 261

angles (gstools.covmodel.TPLExponential property), 299
angles (gstools.covmodel.TPLGaussian property), 285
angles (gstools.covmodel.TPLSimple property), 326
angles (gstools.covmodel.TPLStable property), 313
anis (gstools.covmodel.Circular property), 224
anis (gstools.covmodel.CovModel property), 127
anis (gstools.covmodel.Cubic property), 200
anis (gstools.covmodel.Exponential property), 153
anis (gstools.covmodel.Gaussian property), 140
anis (gstools.covmodel.HyperSpherical property), 248
anis (gstools.covmodel.JBessel property), 273
anis (gstools.covmodel.Linear property), 212
anis (gstools.covmodel.Matern property), 165
anis (gstools.covmodel.Rational property), 189
anis (gstools.covmodel.Spherical property), 236
anis (gstools.covmodel.Stable property), 176
anis (gstools.covmodel.SuperSpherical property), 261
anis (gstools.covmodel.SuperSpherical property), 299
anis (gstools.covmodel.TPLGaussian property), 285
anis (gstools.covmodel.TPLSimple property), 326
anis (gstools.covmodel.TPLStable property), 313

A

all_fields (gstools.field.CondSRF property), 344
all_fields (gstools.field.Field property), 350
all_fields (gstools.field.SRF property), 338
all_fields (gstools.krige.Detrended method), 415
all_fields (gstools.krige.ExtDrift method), 406
all_fields (gstools.krige.Krige method), 370
all_fields (gstools.krige.Ordinary method), 388
all_fields (gstools.krige.Simple method), 379
all_fields (gstools.krige.Universal method), 397

angles (gstools.covmodel.HyperSpherical property), 248
angles (gstools.covmodel.JBessel property), 273
angles (gstools.covmodel.Linear property), 212
angles (gstools.covmodel.Matern property), 165
angles (gstools.covmodel.Rational property), 189
angles (gstools.covmodel.Spherical property), 236
angles (gstools.covmodel.Stable property), 176
angles (gstools.covmodel.SuperSpherical property), 261
anis_bounds (gstools.covmodel.Stable property), 177
anis_bounds (gstools.covmodel.SuperSpherical property), 261
anis_bounds (gstools.covmodel.TPLExponential property), 300
anis_bounds (gstools.covmodel.TPLGaussian property), 286
anis_bounds (gstools.covmodel.TPLSimple property), 326
anis_bounds (gstools.covmodel.TPLStable property), 313
anisometrize() (gstools.covmodel.Circular method), 220
anisometrize() (gstools.covmodel.CovModel method), 123
anisometrize() (gstools.covmodel.Cubic method), 196
anisometrize() (gstools.covmodel.Exponential method), 148
anisometrize() (gstools.covmodel.Gaussian method), 135
anisometrize() (gstools.covmodel.HyperSpherical method), 244
anisometrize() (gstools.covmodel.JBessel method), 268
anisometrize() (gstools.covmodel.Linear method), 208
anisometrize() (gstools.covmodel.Matern method), 160
anisometrize() (gstools.covmodel.Rational method), 184
anisometrize() (gstools.covmodel.Spherical method), 232
anisometrize() (gstools.covmodel.Stable method), 172
anisometrize() (gstools.covmodel.SuperSpherical method), 256
anisometrize() (gstools.covmodel.TPLExponential method), 295
anisometrize() (gstools.covmodel.TPLGaussian method), 281
anisometrize() (gstools.covmodel.TPLSimple method), 321
anisometrize() (gstools.covmodel.TPLStable method), 309
apply() (in module gstools.transform), 432
apply_function() (in module gstools.transform), 437
apply_mean_norm_trend() (in module gstools.normalizer), 462
arg (gstools.covmodel.Circular property), 225
arg (gstools.covmodel.CovModel property), 127
arg (gstools.covmodel.Cubic property), 201
arg (gstools.covmodel.Exponential property), 153
arg (gstools.covmodel.Gaussian property), 140
arg (gstools.covmodel.HyperSpherical property), 249
arg (gstools.covmodel.JBessel property), 273
arg (gstools.covmodel.Linear property), 213
arg (gstools.covmodel.Matern property), 165
arg (gstools.covmodel.Rational property), 189
arg (gstools.covmodel.Spherical property), 237
arg (gstools.covmodel.Stable property), 177
arg (gstools.covmodel.SuperSpherical property), 261
arg (gstools.covmodel.TPLExponential property), 300
arg (gstools.covmodel.TPLGaussian property), 286
arg (gstools.covmodel.TPLSimple property), 326
arg (gstools.covmodel.TPLStable property), 314
arg_bounds (gstools.covmodel.Circular property), 225
arg_bounds (gstools.covmodel.CovModel property), 128
arg_bounds (gstools.covmodel.Cubic property), 201
arg_bounds (gstools.covmodel.Exponential property), 153
arg_bounds (gstools.covmodel.Gaussian property), 140
arg_bounds (gstools.covmodel.HyperSpherical property), 249
arg_bounds (gstools.covmodel.JBessel property), 273
arg_bounds (gstools.covmodel.Linear property), 165
arg_bounds (gstools.covmodel.Matern property), 189
arg_bounds (gstools.covmodel.Rational property), 237
arg_bounds (gstools.covmodel.Spherical property), 261
arg_bounds (gstools.covmodel.SuperSpherical property), 300
arg_bounds (gstools.covmodel.TPLExponential property), 326
arg_bounds (gstools.covmodel.TPLGaussian property), 314

arg_list (gstools.covmodel.TPLSimple property), 326
arg_list (gstools.covmodel.TPLStable property), 314
array_boxcox() (in module gstools.transform), 438
array_discrete() (in module gstools.transform), 437
array_force_moments() (in module gstools.transform), 438
array_to_arcsin() (in module gstools.transform), 438
array_to_lognormal() (in module gstools.transform), 437
array_to_uniform() (in module gstools.transform), 438
array_zinnharvey() (in module gstools.transform), 438
B
binary() (in module gstools.transform), 433
BoxCox (class in gstools.normalizer), 447
boxcox() (in module gstools.transform), 434
BoxCoxShift (class in gstools.normalizer), 450
C
calc_integral_scale() (gstools.covmodel.Circular method), 220
calc_integral_scale() (gstools.covmodel.CovModel method), 123
calc_integral_scale() (gstools.covmodel.Cubic method), 196
calc_integral_scale() (gstools.covmodel.Exponential method), 148
calc_integral_scale() (gstools.covmodel.Gaussian method), 135
calc_integral_scale() (gstools.covmodel.HyperSpherical method), 244
calc_integral_scale() (gstools.covmodel.JBessel method), 268
calc_integral_scale() (gstools.covmodel.Linear method), 208
calc_integral_scale() (gstools.covmodel.Matern method), 160
calc_integral_scale() (gstools.covmodel.Rational method), 184
calc_integral_scale() (gstools.covmodel.Spherical method), 232
calc_integral_scale() (gstools.covmodel.Stable method), 172
calc_integral_scale() (gstools.covmodel.SuperSpherical method), 256
calc_integral_scale() (gstools.covmodel.TPLExponential method), 295
calc_integral_scale() (gstools.covmodel.TPLGaussian method), 281
calc_integral_scale() (gstools.covmodel.TPLSimple method), 321
calc_integral_scale() (gstools.covmodel.TPLStable method), 309
check_arg_bounds() (gstools.covmodel.Circular method), 220
check_arg_bounds() (gstools.covmodel.CovModel method), 123
check_arg_bounds() (gstools.covmodel.Cubic method), 196
check_arg_bounds() (gstools.covmodel.Exponential method), 148
check_arg_bounds() (gstools.covmodel.Gaussian method), 135
check_arg_bounds() (gstools.covmodel.HyperSpherical method), 244
check_arg_bounds() (gstools.covmodel.JBessel method), 268
check_arg_bounds() (gstools.covmodel.Linear method), 208
check_arg_bounds() (gstools.covmodel.Matern method), 160
check_arg_bounds() (gstools.covmodel.Rational method), 184
check_arg_bounds() (gstools.covmodel.Spherical method), 232
check_arg_bounds() (gstools.covmodel.Stable method), 172
check_arg_bounds() (gstools.covmodel.SuperSpherical method), 256
check_arg_bounds() (gstools.covmodel.TPLExponential method), 295
check_arg_bounds() (gstools.covmodel.TPLGaussian method), 281
check_arg_bounds() (gstools.covmodel.TPLSimple method), 321
check_arg_bounds() (gstools.covmodel.TPLStable method), 309
check_dim() (gstools.covmodel.Circular method), 220
check_dim() (gstools.covmodel.CovModel method), 123
check_dim() (gstools.covmodel.Cubic method), 196
check_dim() (gstools.covmodel.Exponential method), 148
check_dim() (gstools.covmodel.Gaussian method), 135
check_dim() (gstools.covmodel.HyperSpherical method), 244
check_dim()...
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlation()</td>
<td>correlation()</td>
</tr>
<tr>
<td>cov_axis()</td>
<td>cov_axis()</td>
</tr>
<tr>
<td>cov_nugget()</td>
<td>cov_nugget()</td>
</tr>
<tr>
<td>cov_spatial()</td>
<td>cov_spatial()</td>
</tr>
<tr>
<td>cov_yadrenko()</td>
<td>cov_yadrenko()</td>
</tr>
</tbody>
</table>
cov_yadrenko() (gstools.covmodel.JBessel method), 269
cov_yadrenko() (gstools.covmodel.Linear method), 209
cov_yadrenko() (gstools.covmodel.Matern method), 161
cov_yadrenko() (gstools.covmodel.Rational method), 185
cov_yadrenko() (gstools.covmodel.Spherical method), 233
cov_yadrenko() (gstools.covmodel.Stable method), 172
cov_yadrenko() (gstools.covmodel.SuperSpherical method), 257
cov_yadrenko() (gstools.covmodel.TPLExponential method), 281
cov_yadrenko() (gstools.covmodel.TPLGaussian method), 322
cov_yadrenko() (gstools.covmodel.TPLSimple method), 309
cov_yadrenko() (gstools.covmodel.TPLStable method), 257
covariance() (gstools.covmodel.Circular method), 221
covariance() (gstools.covmodel.Cubic method), 197
covariance() (gstools.covmodel.Exponential method), 149
covariance() (gstools.covmodel.Gaussian method), 136
covariance() (gstools.covmodel.HyperSpherical method), 245
covariance() (gstools.covmodel.JBessel method), 269
covariance() (gstools.covmodel.Linear method), 209
covariance() (gstools.covmodel.Matern method), 161
covariance() (gstools.covmodel.Rational method), 185
covariance() (gstools.covmodel.Spherical method), 233
covariance() (gstools.covmodel.Stable method), 172
covariance() (gstools.covmodel.SuperSpherical method), 257
covariance() (gstools.covmodel.TPLExponential method), 295
covariance() (gstools.covmodel.TPLGaussian method), 281
covariance() (gstools.covmodel.TPLSimple method), 322
covariance() (gstools.covmodel.TPLStable method), 309
covariance() (gstools.covmodel.Stable method), 172
covariance() (gstools.covmodel.SuperSpherical method), 257
covariance() (gstools.covmodel.TPLExponential method), 295
default_arg_bounds() (gstools.covmodel.Circular method), 221
default_arg_bounds() (gstools.covmodel.CovModel method), 124
default_arg_bounds() (gstools.covmodel.Cubic method), 197
default_arg_bounds() (gstools.covmodel.Exponential method), 149
default_arg_bounds() (gstools.covmodel.JBessel method), 269
default_arg_bounds() (gstools.covmodel.Linear method), 209
default_arg_bounds() (gstools.covmodel.Matern method), 161
default_arg_bounds() (gstools.covmodel.Rational method), 185
default_arg_bounds() (gstools.covmodel.Spherical method), 233
default_arg_bounds() (gstools.covmodel.Stable method), 172
default_arg_bounds() (gstools.covmodel.SuperSpherical method), 257
default_arg_bounds() (gstools.covmodel.TPLExponential method), 295
default_arg_bounds() (gstools.covmodel.TPLGaussian method), 281
default_arg_bounds() (gstools.covmodel.TPLSimple method), 322
default_arg_bounds() (gstools.covmodel.TPLStable method), 309
default_field_names (gstools.field.CondSRF attribute), 344
default_field_names (gstools.field.Field attribute), 350
default_field_names (gstools.field.SRF attribute), 338
default_field_names (gstools.krige.Detrended attribute), 415
default_field_names (gstools.krige.ExtDrift attribute), 407
default_field_names (gstools.krige.Krige attribute), 370
default_field_names (gstools.krige.Ordinary attribute), 388
default_field_names (gstools.krige.Simple attribute), 379
default_field_names (gstools.krige.Universal attribute), 398

Index 487
default_opt_arg() (gstools.covmodel.Circular method), 221
default_opt_arg() (gstools.covmodel.CovModel method), 124
default_opt_arg() (gstools.covmodel.Cubic method), 197
default_opt_arg() (gstools.covmodel.Exponential method), 149
default_opt_arg() (gstools.covmodel.Gaussian method), 136
default_opt_arg() (gstools.covmodel.HyperSpherical method), 245
default_opt_arg() (gstools.covmodel.JBessel method), 269
default_opt_arg() (gstools.covmodel.Linear method), 209
default_opt_arg() (gstools.covmodel.Matern method), 161
default_opt_arg() (gstools.covmodel.Rational method), 185
default_opt_arg() (gstools.covmodel.Spherical method), 233
default_opt_arg() (gstools.covmodel.Stable method), 173
default_opt_arg() (gstools.covmodel.SuperSpherical method), 257
default_opt_arg() (gstools.covmodel.TPLExponential method), 296
default_opt_arg() (gstools.covmodel.TPLGaussian method), 282
default_opt_arg() (gstools.covmodel.TPLSimple method), 322
default_opt_arg() (gstools.covmodel.TPLStable method), 310

default_opt_arg_bounds() (gstools.covmodel.Circular method), 221
default_opt_arg_bounds() (gstools.covmodel.CovModel method), 124
default_opt_arg_bounds() (gstools.covmodel.Cubic method), 197
default_opt_arg_bounds() (gstools.covmodel.Exponential method), 149
default_opt_arg_bounds() (gstools.covmodel.Gaussian method), 136
default_opt_arg_bounds() (gstools.covmodel.HyperSpherical method), 245
default_opt_arg_bounds() (gstools.covmodel.JBessel method), 269
default_opt_arg_bounds() (gstools.covmodel.Linear method), 209
default_opt_arg_bounds() (gstools.covmodel.Matern method), 161
default_opt_arg_bounds() (gstools.covmodel.Rational method), 185
default_opt_arg_bounds() (gstools.covmodel.Spherical method), 233
default_opt_arg_bounds() (gstools.covmodel.Stable method), 173
default_opt_arg_bounds() (gstools.covmodel.SuperSpherical method), 257

default_parameter (gstools.normalizer.BoxCox attribute), 448
default_parameter (gstools.normalizer.BoxCoxShift attribute), 451
default_parameter (gstools.normalizer.LogNormal attribute), 445
default_parameter (gstools.normalizer.Manly attribute), 460
default_parameter (gstools.normalizer.Modulus attribute), 457
default_parameter (gstools.normalizer.Normalizer attribute), 442
default_parameter (gstools.normalizer.YeoJohnson attribute), 454

default_rescale() (gstools.covmodel.Circular method), 221
default_rescale() (gstools.covmodel.CovModel method), 124
default_rescale() (gstools.covmodel.Cubic method), 197
default_rescale() (gstools.covmodel.Exponential method), 149
default_rescale() (gstools.covmodel.Gaussian method), 136
default_rescale() (gstools.covmodel.HyperSpherical method), 245
default_rescale() (gstools.covmodel.JBessel method), 269
default_rescale() (gstools.covmodel.Linear method), 209
default_rescale() (gstools.covmodel.Matern method), 161
default_rescale() (gstools.covmodel.Rational method), 185
default_rescale() (gstools.covmodel.Spherical method), 233
default_rescale() (gstools.covmodel.Stable method), 173
default_rescale() (gstools.covmodel.SuperSpherical method), 257
default_rescale() (gstools.covmodel.TPLExponential method), 296
default_rescale() (gstools.covmodel.TPLGaussian method), 282
default_rescale() (gstools.covmodel.TPLSimple method), 322
default_rescale() (gstools.covmodelTPLStable method), 310
delete_fields() (gstools.field.CondSRF method), 341
delete_fields() (gstools.field.Field method), 347
delete_fields() (gstools.field.SRF method), 335
delete_fields() (gstools.krige.Detrended method), 411
delete_fields() (gstools.krige.Ordinary method), 384
delete_fields() (gstools.krige.Simple method), 375
delete_fields() (gstools.krige.Universal method), 393
denormalize() (gstools.normalizer.BoxCox method), 447
denormalize() (gstools.normalizer.BoxCoxShift method), 450
denormalize() (gstools.normalizer.LogNormal method), 444
denormalize() (gstools.normalizer.Manly method), 459
denormalize() (gstools.normalizer.Modulus method), 456
denormalize() (gstools.normalizer.Normalizer method), 441
denormalize() (gstools.normalizer.YeoJohnson method), 453
detrended (class in gstools.krige), 409
dim (gstools.covmodel.Circular property), 225
dim (gstools.covmodel.CovModel property), 128
dim (gstools.covmodel.Cubic property), 201
dim (gstools.covmodel.Exponential property), 153
dim (gstools.covmodel.Gaussian property), 141
dim (gstools.covmodel.HyperSpherical property), 249
dim (gstools.covmodel.JBessel property), 273
dim (gstools.covmodel.Linear property), 213
dim (gstools.covmodel.Matern property), 165
dim (gstools.covmodel.Rational property), 189
dim (gstools.covmodel.Spherical property), 237
dim (gstools.covmodel.Stable property), 177
dim (gstools.covmodel.Superspherical property), 261
dim (gstools.covmodel.TPLExponential property), 300
dim (gstools.covmodel.TPLGaussian property), 286
dim (gstools.covmodel.TPLSimple property), 326
dim (gstools.covmodel.TPLStable property), 314
dim (gstools.field.CondSRF property), 344
dim (gstools.field.Field property), 350
dim (gstools.field.SRF property), 338
dim (gstools.krige.Detrended property), 415
dim (gstools.krige.Ordinary property), 388
dim (gstools.krige.Simple property), 380
dim (gstools.krige.Universal property), 398
discrete() (in module gstools.transform), 433
dist_func (gstools.covmodel.Circular property), 225
dist_func (gstools.covmodel.CovModel property), 128
dist_func (gstools.covmodel.Cubic property), 201
dist_func (gstools.covmodel.Exponential property), 153
dist_func (gstools.covmodel.Gaussian property), 141
dist_func (gstools.covmodel.HyperSpherical property), 249
dist_func (gstools.covmodel.JBessel property), 273
dist_func (gstools.covmodel.Linear property), 213
dist_func (gstools.covmodel.Matern property), 165
dist_func (gstools.covmodel.Rational property), 189
dist_func (gstools.covmodel.Spherical property), 237
dist_func (gstools.covmodel.Stable property), 177
dist_func (gstools.covmodel.Superspherical property), 261
dist_func (gstools.covmodel.TPLExponential property), 300
dist_func (gstools.covmodel.TPLGaussian property), 286
dist_func (gstools.covmodel.TPLSimple property), 326
Gaussian (class in gstools.covmodel), 132
generate_grid() (in module gstools.tools), 430
generate_st_grid() (in module gstools.tools), 430
generator (gstools.field.CondSRF property), 344
generator (gstools.field.SRF property), 338
get_mean() (gstools.krige.Detrended method), 411
get_mean() (gstools.krige.ExtDrift method), 402
get_mean() (gstools.krige.Krige method), 366
get_mean() (gstools.krige.Ordinary method), 384
get_nugget() (gstools.field.generator.IncomprRandMeth method), 535
get_nugget() (gstools.field.generator.RandMeth method), 356
get_scaling() (gstools.field.CondSRF method), 341
get_store_config() (gstools.field.CondSRF method), 341
get_store_config() (gstools.field.Field method), 347
get_store_config() (gstools.krige.Detrended method), 411
get_store_config() (gstools.krige.ExtDrift method), 403
get_store_config() (gstools.krige.Krige method), 366
get_store_config() (gstools.krige.Ordinary method), 384
get_store_config() (gstools.krige.Simple method), 376
givens_rotation() (in module gstools.tools), 428
gstools
  module, 117
gstools.covmodel
  module, 120
  plot
  module, 331
gstools.field
  module, 333
  generator
  module, 352
gstools.field.upscale
  module, 358
gstools.krige
  module, 363
gstools.normalizer
  module, 441
gstools.random
module, 418

gstools.tools
 module, 422
gstools.transform
 module, 432
gstools.variogram
 module, 359

H

hankel_kw (gstools.covmodel.Circular property), 225
hankel_kw (gstools.covmodel.CovModel property), 128
hankel_kw (gstools.covmodel.Cubic property), 201
hankel_kw (gstools.covmodel.Exponential property), 153
hankel_kw (gstools.covmodel.Gaussian property), 141
hankel_kw (gstools.covmodel.HyperSpherical property), 249
hankel_kw (gstools.covmodel.JBessel property), 274
hankel_kw (gstools.covmodel.Linear property), 213
hankel_kw (gstools.covmodel.Matern property), 166
hankel_kw (gstools.covmodel.Rational property), 190
hankel_kw (gstools.covmodel.Spherical property), 237
hankel_kw (gstools.covmodel.Stable property), 177
hankel_kw (gstools.covmodel.SuperSpherical property), 262
hankel_kw (gstools.covmodel.TPLExponential property), 300
hankel_kw (gstools.covmodel.TPLGaussian property), 286
hankel_kw (gstools.covmodel.TPLSimple property), 327
hankel_kw (gstools.covmodel.TPLStable property), 262
has_cdf (gstools.covmodel.Circular property), 225
has_cdf (gstools.covmodel.CovModel property), 128
has_cdf (gstools.covmodel.Cubic property), 201
has_cdf (gstools.covmodel.Exponential property), 154
has_cdf (gstools.covmodel.Gaussian property), 141
has_cdf (gstools.covmodel.HyperSpherical property), 249
has_cdf (gstools.covmodel.JBessel property), 274
has_cdf (gstools.covmodel.Linear property), 213
has_cdf (gstools.covmodel.Matern property), 166
has_cdf (gstools.covmodel.Rational property), 190
has_cdf (gstools.covmodel.Spherical property), 237
has_cdf (gstools.covmodel.Stable property), 177
has_cdf (gstools.covmodel.SuperSpherical property), 262

has_const_mean (gstools.krige.ExtDrift property), 407
has_const_mean (gstools.krige.Krige property), 371
has_const_mean (gstools.krige.Ordinary property), 389
has_const_mean (gstools.krige.Simple property), 380
has_const_mean (gstools.krige.Universal property), 398

has_ppf (gstools.covmodel.Circular property), 225
has_ppf (gstools.covmodel.CovModel property), 128
has_ppf (gstools.covmodel.Cubic property), 201
has_ppf (gstools.covmodel.Exponential property), 154
has_ppf (gstools.covmodel.Gaussian property), 141
has_ppf (gstools.covmodel.HyperSpherical property), 249
has_ppf (gstools.covmodel.JBessel property), 274
has_ppf (gstools.covmodel.Linear property), 213
has_ppf (gstools.covmodel.Matern property), 166
has_ppf (gstools.covmodel.Rational property), 190
has_ppf (gstools.covmodel.Spherical property), 237
has_ppf (gstools.covmodel.Stable property), 177
has_ppf (gstools.covmodel.SuperSpherical property), 262
has_ppf (gstools.covmodel.TPLExponential property), 300
has_ppf (gstools.covmodel.TPLGaussian property), 286
has_ppf (gstools.covmodel.TPLSimple property), 327
has_ppf (gstools.covmodel.TPLStable property), 314
HyperSpherical (class in gstools.covmodel), 241

inc_beta() (in module gstools.tools), 425
inc_gamma() (in module gstools.tools), 424
IncComprRandMeth (class in gstools.field.generator), 352
int_drift_no (gstools.krige.Detrended property), 416
int_drift_no (gstools.krige.ExtDrift property), 407
int_drift_no (gstools.krige.Krige property), 371
int_drift_no (gstools.krige.Ordinary property), 389
int_drift_no (gstools.krige.Simple property), 380
int_drift_no (gstools.krige.Universal property), 398
integral_scale (gstools.covmodel.Circular property), 226
integral_scale (gstools.covmodel.CovModel property), 128
integral_scale (gstools.covmodel.Cubic property), 202
integral_scale (gstools.covmodel.Exponential property), 154
integral_scale (gstools.covmodel.Gaussian property), 141
integral_scale (gstools.covmodel.HyperSpherical property), 250
integral_scale (gstools.covmodel.JBessel property), 274
integral_scale (gstools.covmodel.Linear property), 214
integral_scale (gstools.covmodel.Matern property), 166
integral_scale (gstools.covmodel.Rational property), 190
integral_scale (gstools.covmodel.Spherical property), 238
integral_scale (gstools.covmodel.Stable property), 178
integral_scale (gstools.covmodel.SuperSpherical property), 262
integral_scale (gstools.covmodel.TPLExponential property), 301
integral_scale (gstools.covmodel.TPLGaussian property), 287
integral_scale (gstools.covmodel.TPLSimple property), 327
is_isotropic (gstools.covmodel.CovModel property), 226
is_isotropic (gstools.covmodel.Circular property), 226
is_isotropic (gstools.covmodel.CovModel property), 129
is_isotropic (gstools.covmodel.Cubic property), 202
is_isotropic (gstools.covmodel.Exponential property), 154
is_isotropic (gstools.covmodel.Gaussian property), 141
is_isotropic (gstools.covmodel.HyperSpherical property), 250
is_isotropic (gstools.covmodel.JBessel property), 274
is_isotropic (gstools.covmodel.Linear property), 214
is_isotropic (gstools.covmodel.Matern property), 166
is_isotropic (gstools.covmodel.Rational property), 190
is_isotropic (gstools.covmodel.Spherical property), 238
is_isotropic (gstools.covmodel.Stable property), 178
is_isotropic (gstools.covmodel.SuperSpherical property), 262
is_isotropic (gstools.covmodel.TPLExponential property), 301
is_isotropic (gstools.covmodel.TPLGaussian property), 287
is_isotropic (gstools.covmodel.TPLSimple property), 327
is_isotropic (gstools.covmodel.TPLStable property), 315
iso_arg (gstools.covmodel.Circular property), 226
iso_arg (gstools.covmodel.CovModel property), 226
iso_arg (gstools.covmodel.Cubic property), 202
iso_arg (gstools.covmodel.Exponential property), 154
iso_arg (gstools.covmodel.Gaussian property), 141
iso_arg (gstools.covmodel.HyperSpherical property), 250
iso_arg (gstools.covmodel.JBessel property), 274
iso_arg (gstools.covmodel.Linear property), 214
iso_arg (gstools.covmodel.Matern property), 166
iso_arg (gstools.covmodel.Rational property), 190
iso_arg (gstools.covmodel.Spherical property), 238
iso_arg (gstools.covmodel.Stable property), 178
iso_arg (gstools.covmodel.SuperSpherical property), 262
iso_arg (gstools.covmodel.TPLExponential property), 301
iso_arg (gstools.covmodel.TPLGaussian property), 287
iso_arg (gstools.covmodel.TPLSimple property), 327
iso_arg (gstools.covmodel.TPLStable property), 315
iso_arg_list (gstools.covmodel.Circular property), 226
iso_arg_list (gstools.covmodel.CovModel property), 226
iso_arg_list (gstools.covmodel.HyperSpherical property), 250
iso_arg_list (gstools.covmodel.JBessel property), 274
iso_arg_list (gstools.covmodel.Linear property), 214
iso_arg_list (gstools.covmodel.Matern property), 166
iso_arg_list (gstools.covmodel.Rational property), 190
iso_arg_list (gstools.covmodel.Spherical property), 238
iso_arg_list (gstools.covmodel.Stable property), 178
iso_arg_list (gstools.covmodel.SuperSpherical property), 262
iso_arg_list (gstools.covmodel.TPLExponential property), 301
iso_arg_list (gstools.covmodel.TPLGaussian property), 287
iso_arg_list (gstools.covmodel.TPLSimple property), 327
iso_arg_list (gstools.covmodel.TPLStable property), 315
iso_arg_list (gstools.covmodel.CovModel prop-
Index
len_scale_vec (gstools.covmodel.Gaussian property), 142
len_scale_vec (gstools.covmodel.HyperSpherical property), 251
len_scale_vec (gstools.covmodel.IBessel property), 275
len_scale_vec (gstools.covmodel.Linear property), 215
len_scale_vec (gstools.covmodel.Matern property), 167
len_scale_vec (gstools.covmodel.Rational property), 191
len_scale_vec (gstools.covmodel.Spherical property), 239
len_scale_vec (gstools.covmodel.Stable property), 179
len_scale_vec (gstools.covmodel.SuperSpherical property), 263
len_scale_vec (gstools.covmodel.TPLExponential property), 302
len_scale_vec (gstools.covmodel.TPLGaussian property), 288
len_scale_vec (gstools.covmodel.TPLSimple property), 328
len_scale_vec (gstools.covmodel.TPLequalproperty), 315
len_up (gstools.covmodel.TPLExponential property), 302
len_up (gstools.covmodel.TPLGaussian property), 288
len_up (gstools.covmodel.TPLStable property), 316
len_up_rescaled (gstools.covmodel.TPLExponential property), 302
len_up_rescaled (gstools.covmodel.TPLGaussian property), 288
len_up_rescaled (gstools.covmodel.TPLStable property), 316
likelihood() (gstools.normalizer.BoxCox method), 448
likelihood() (gstools.normalizer.BoxCoxShift method), 451
likelihood() (gstools.normalizer.LogNormal method), 445
likelihood() (gstools.normalizer.Manly method), 460
likelihood() (gstools.normalizer.Modulus method), 457
likelihood() (gstools.normalizer.Normalizer method), 442
likelihood() (gstools.normalizer.YeoJohnson method), 454
Linear (class in gstools.covmodel), 205
ln_spectral_rad_pdf() (gstools.covmodel.Circular method), 223
ln_spectral_rad_pdf() (gstools.covmodel.CovModel method), 126
ln_spectral_rad_pdf() (gstools.covmodel.Cubic method), 199
ln_spectral_rad_pdf() (gstools.covmodel.Exponential method), 151
ln_spectral_rad_pdf() (gstools.covmodel.Gaussian method), 138
ln_spectral_rad_pdf() (gstools.covmodel.HyperSpherical method), 247
ln_spectral_rad_pdf() (gstools.covmodel.IBessel method), 271
ln_spectral_rad_pdf() (gstools.covmodel.Linear method), 211
ln_spectral_rad_pdf() (gstools.covmodel.Matern method), 163
ln_spectral_rad_pdf() (gstools.covmodel.Rational method), 187
ln_spectral_rad_pdf() (gstools.covmodel.Spherical method), 235
ln_spectral_rad_pdf() (gstools.covmodel.Stable method), 175
ln_spectral_rad_pdf() (gstools.covmodel.SuperSpherical method), 259
ln_spectral_rad_pdf() (gstools.covmodel.TPLExponential method), 298
ln_spectral_rad_pdf() (gstools.covmodel.TPLGaussian method), 284
ln_spectral_rad_pdf() (gstools.covmodel.TPLSimple method), 324
ln_spectral_rad_pdf() (gstools.covmodel.TPLStable method), 312
loglikelihood() (gstools.normalizer.BoxCox method), 448
loglikelihood() (gstools.normalizer.BoxCoxShift method), 451
loglikelihood() (gstools.normalizer.LogNormal method), 445
loglikelihood() (gstools.normalizer.Manly method), 460
loglikelihood() (gstools.normalizer.Modulus method), 457
loglikelihood() (gstools.normalizer.Normalizer method), 442
loglikelihood() (gstools.normalizer.YeoJohnson method), 454
LogNormal (class in gstools.normalizer), 444
M
main_axes() (gstools.covmodel.Circular method), 223
main_axes() (gstools.covmodel.CovModel method), 126
<table>
<thead>
<tr>
<th>Function</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>main_axes()</td>
<td>199</td>
</tr>
<tr>
<td>(gstools.covmodel.Cubic method), 151</td>
<td></td>
</tr>
<tr>
<td>main_axes()</td>
<td>138</td>
</tr>
<tr>
<td>(gstools.covmodel.Gaussian method), 247</td>
<td></td>
</tr>
<tr>
<td>main_axes()</td>
<td>271</td>
</tr>
<tr>
<td>(gstools.covmodel.JBessel method), 211</td>
<td></td>
</tr>
<tr>
<td>main_axes()</td>
<td>163</td>
</tr>
<tr>
<td>(gstools.covmodel.Matern method), 187</td>
<td></td>
</tr>
<tr>
<td>main_axes()</td>
<td>175</td>
</tr>
<tr>
<td>(gstools.covmodel.Stable method), 235</td>
<td></td>
</tr>
<tr>
<td>main_axes()</td>
<td>175</td>
</tr>
<tr>
<td>(gstools.covmodel.Superspherical method), 312</td>
<td></td>
</tr>
<tr>
<td>Manly (class in gstools.normalizer), 459</td>
<td></td>
</tr>
<tr>
<td>MasterRNG (class in gstools.random), 420</td>
<td></td>
</tr>
<tr>
<td>Matern (class in gstools.covmodel), 157</td>
<td></td>
</tr>
<tr>
<td>matrix_anisometrize()</td>
<td>430</td>
</tr>
<tr>
<td>(in module gstools.tools), 429</td>
<td></td>
</tr>
<tr>
<td>matrix_anisotropify()</td>
<td>429</td>
</tr>
<tr>
<td>(in module gstools.tools), 429</td>
<td></td>
</tr>
<tr>
<td>matrix_rotate()</td>
<td>428</td>
</tr>
<tr>
<td>(in module gstools.tools), 429</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>345</td>
</tr>
<tr>
<td>(gstools.field.CondSRF property), 350</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>338</td>
</tr>
<tr>
<td>(gstools.field.SRF property), 338</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>407</td>
</tr>
<tr>
<td>(gstools.krige.Detrended property), 407</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>371</td>
</tr>
<tr>
<td>(gstools.krige.Krige property), 371</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>389</td>
</tr>
<tr>
<td>(gstools.krige.Ordinary property), 389</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>380</td>
</tr>
<tr>
<td>(gstools.krige.Simple property), 380</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td>399</td>
</tr>
<tr>
<td>(gstools.krige.Universal property), 399</td>
<td></td>
</tr>
<tr>
<td>mesh_type</td>
<td>408</td>
</tr>
<tr>
<td>(gstools.krige.ExtDrift property), 408</td>
<td></td>
</tr>
<tr>
<td>mesh_type</td>
<td>371</td>
</tr>
<tr>
<td>(gstools.krige.Krige property), 371</td>
<td></td>
</tr>
<tr>
<td>mesh_type</td>
<td>389</td>
</tr>
<tr>
<td>(gstools.krige.Ordinary property), 389</td>
<td></td>
</tr>
<tr>
<td>mesh_type</td>
<td>380</td>
</tr>
<tr>
<td>(gstools.krige.Simple property), 380</td>
<td></td>
</tr>
<tr>
<td>mesh_type</td>
<td>399</td>
</tr>
<tr>
<td>(gstools.krige.Universal property), 399</td>
<td></td>
</tr>
<tr>
<td>mode_no</td>
<td>354</td>
</tr>
<tr>
<td>(gstools.field.generator.IncomprRandMeth prop-</td>
<td></td>
</tr>
<tr>
<td>erty), 354</td>
<td></td>
</tr>
<tr>
<td>mode_no</td>
<td>356</td>
</tr>
<tr>
<td>(gstools.field.generator.RandMeth property), 356</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>345</td>
</tr>
<tr>
<td>(gstools.field.CondSRF property), 345</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>350</td>
</tr>
<tr>
<td>(gstools.field.Field property), 350</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>354</td>
</tr>
<tr>
<td>(gstools.field.generator.IncomprRandMeth prop-</td>
<td></td>
</tr>
<tr>
<td>erty), 354</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>356</td>
</tr>
<tr>
<td>(gstools.field.generator.RandMeth property), 356</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>338</td>
</tr>
<tr>
<td>(gstools.field.SRF property), 338</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>416</td>
</tr>
<tr>
<td>(gstools.krige.Detrended property), 416</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>408</td>
</tr>
<tr>
<td>(gstools.krige.ExtDrift property), 408</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>371</td>
</tr>
<tr>
<td>(gstools.krige.Krige property), 371</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>389</td>
</tr>
<tr>
<td>(gstools.krige.Ordinary property), 389</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>380</td>
</tr>
<tr>
<td>(gstools.krige.Simple property), 380</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>399</td>
</tr>
<tr>
<td>(gstools.krige.Universal property), 399</td>
<td></td>
</tr>
<tr>
<td>module</td>
<td>117</td>
</tr>
<tr>
<td>gstools, 120</td>
<td></td>
</tr>
<tr>
<td>gstools.covmodel, 120</td>
<td></td>
</tr>
<tr>
<td>gstools.covmodel.plot, 331</td>
<td></td>
</tr>
<tr>
<td>gstools.field, 333</td>
<td></td>
</tr>
<tr>
<td>gstools.field.generator, 352</td>
<td></td>
</tr>
<tr>
<td>gstools.field.upsampling, 358</td>
<td></td>
</tr>
<tr>
<td>gstools.krige, 363</td>
<td></td>
</tr>
<tr>
<td>gstools.normalizer, 441</td>
<td></td>
</tr>
<tr>
<td>gstools.random, 418</td>
<td></td>
</tr>
<tr>
<td>gstools.tools, 422</td>
<td></td>
</tr>
<tr>
<td>gstools.transform, 432</td>
<td></td>
</tr>
<tr>
<td>gstools.vario gram, 359</td>
<td></td>
</tr>
<tr>
<td>Modulus (class in gstools.normalizer), 456</td>
<td></td>
</tr>
</tbody>
</table>

N

<table>
<thead>
<tr>
<th>Function</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>227</td>
</tr>
<tr>
<td>(gstools.covmodel.Circular property), 130</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>203</td>
</tr>
<tr>
<td>(gstools.covmodel.Cubic property), 203</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>155</td>
</tr>
<tr>
<td>(gstools.covmodel.Exponential property), 155</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>142</td>
</tr>
<tr>
<td>(gstools.covmodel.Gaussian property), 142</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>251</td>
</tr>
<tr>
<td>(gstools.covmodel.HyperSpherical property), 251</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>275</td>
</tr>
<tr>
<td>(gstools.covmodel.JBessel property), 275</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>215</td>
</tr>
<tr>
<td>(gstools.covmodel.Linear property), 215</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>167</td>
</tr>
<tr>
<td>(gstools.covmodel.Matern property), 167</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>191</td>
</tr>
<tr>
<td>(gstools.covmodel.Rational property), 191</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>239</td>
</tr>
<tr>
<td>(gstools.covmodel.Spherical property), 239</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>179</td>
</tr>
<tr>
<td>(gstools.covmodel.Stable property), 179</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>263</td>
</tr>
<tr>
<td>(gstools.covmodel.Superspherical property), 263</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>288</td>
</tr>
<tr>
<td>(gstools.covmodel.TPLGaussian property), 288</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>328</td>
</tr>
<tr>
<td>(gstools.covmodel.TPLSimple property), 328</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>316</td>
</tr>
<tr>
<td>(gstools.covmodel.TPLStable property), 316</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>333</td>
</tr>
<tr>
<td>(gstools.field.CondSRF property), 333</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>350</td>
</tr>
<tr>
<td>(gstools.field.Field property), 350</td>
<td></td>
</tr>
</tbody>
</table>

Index 497
name (gtools.field.generator.IncomprRandMeth property), 354
name (gtools.field.generator.RandMeth property), 356
name (gtools.field.SRF property), 338
name (gtools.krige.Detrended property), 416
name (gtools.krige.ExtDrift property), 408
name (gtools.krige.Krige property), 371
name (gtools.krige.Ordinary property), 389
name (gtools.krige.Simple property), 381
name (gtools.krige.Universal property), 399
nugget (gtools.covmodel.Circular property), 227
nugget (gtools.covmodel.CovModel property), 130
nugget (gtools.covmodel.Cubic property), 203
nugget (gtools.covmodel.Exponential property), 155
nugget (gtools.covmodel.Gaussian property), 142
nugget (gtools.covmodel.HyperSpherical property), 251
nugget (gtools.covmodel.JBessel property), 275
nugget (gtools.covmodel.Linear property), 215
nugget (gtools.covmodel.Matern property), 167
nugget (gtools.covmodel.Rational property), 191
nugget (gtools.covmodel.Spherical property), 239
nugget (gtools.covmodel.Stable property), 179
nugget (gtools.covmodel.SuperSpherical property), 263
nugget (gtools.covmodel.TPLEXponential property), 302
nugget (gtools.covmodel.TPLGaussian property), 288
nugget (gtools.covmodel.TPLSimple property), 328
nugget (gtools.covmodel.TPLStable property), 316
nugget_bounds (gtools.covmodel.Circular property), 227
nugget_bounds (gtools.covmodel.CovModel property), 130
nugget_bounds (gtools.covmodel.Cubic property), 203
nugget_bounds (gtools.covmodel.Exponential property), 155
nugget_bounds (gtools.covmodel.Gaussian property), 142
nugget_bounds (gtools.covmodel.HyperSpherical property), 251
nugget_bounds (gtools.covmodel.JBessel property), 275
nugget_bounds (gtools.covmodel.Linear property), 215
nugget_bounds (gtools.covmodel.Matern property), 167
nugget_bounds (gtools.covmodel.Rational property), 191
nugget_bounds (gtools.covmodel.Spherical property), 239
nugget_bounds (gtools.covmodel.Stable property), 179
nugget_bounds (gtools.covmodel.SuperSpherical property), 263
nugget_bounds (gtools.covmodel.TPLEXponential property), 302
nugget_bounds (gtools.covmodel.TPLGaussian property), 288
nugget_bounds (gtools.covmodel.TPLSimple property), 328
nugget_bounds (gtools.covmodel.TPLStable property), 316
nugget_bounds (gtools.covmodel.Circular property), 227
nugget_bounds (gtools.covmodel.CovModel property), 130
nugget_bounds (gtools.covmodel.Cubic property), 203
nugget_bounds (gtools.covmodel.Exponential property), 155
nugget_bounds (gtools.covmodel.Gaussian property), 142
nugget_bounds (gtools.covmodel.HyperSpherical property), 251
nugget_bounds (gtools.covmodel.JBessel property), 275
nugget_bounds (gtools.covmodel.Linear property), 215
nugget_bounds (gtools.covmodel.Matern property), 167
nugget_bounds (gtools.covmodel.Rational property), 191
nugget_bounds (gtools.covmodel.Spherical property), 239
nugget_bounds (gtools.covmodel.Stable property), 179
nugget_bounds (gtools.covmodel.SuperSpherical property), 263
nugget_bounds (gtools.covmodel.TPLEXponential property), 302
nugget_bounds (gtools.covmodel.TPLGaussian property), 288

498

Index
GeoStatTools Documentation, Release 1.3.5

nugget_bounds (gstools.covmodel.TPLSimple prop- opt_arg_bounds (gstools.covmodel.TPLSimple property), 328
erty), 328
nugget_bounds (gstools.covmodel.TPLStable prop- opt_arg_bounds (gstools.covmodel.TPLStable property), 316
erty), 316
Ordinary (class in gstools.krige), 382

O

opt_arg (gstools.covmodel.Circular property), 227
opt_arg (gstools.covmodel.CovModel property), 130
opt_arg (gstools.covmodel.Cubic property), 203
opt_arg (gstools.covmodel.Exponential property), 155
opt_arg (gstools.covmodel.Gaussian property), 143
opt_arg (gstools.covmodel.HyperSpherical property),
251
opt_arg (gstools.covmodel.JBessel property), 275
opt_arg (gstools.covmodel.Linear property), 215
opt_arg (gstools.covmodel.Matern property), 167
opt_arg (gstools.covmodel.Rational property), 191
opt_arg (gstools.covmodel.Spherical property), 239
opt_arg (gstools.covmodel.Stable property), 179
opt_arg (gstools.covmodel.SuperSpherical property),
263
opt_arg (gstools.covmodel.TPLExponential property),
302
opt_arg (gstools.covmodel.TPLGaussian property),
288
opt_arg (gstools.covmodel.TPLSimple property), 328
opt_arg (gstools.covmodel.TPLStable property), 316
opt_arg_bounds (gstools.covmodel.Circular property), 227
opt_arg_bounds (gstools.covmodel.CovModel property), 130
opt_arg_bounds (gstools.covmodel.Cubic property),
203
opt_arg_bounds
(gstools.covmodel.Exponential
property), 155
opt_arg_bounds (gstools.covmodel.Gaussian property), 143
opt_arg_bounds (gstools.covmodel.HyperSpherical
property), 251
opt_arg_bounds (gstools.covmodel.JBessel property), 275
opt_arg_bounds (gstools.covmodel.Linear property),
215
opt_arg_bounds (gstools.covmodel.Matern property),
167
opt_arg_bounds (gstools.covmodel.Rational property), 191
opt_arg_bounds (gstools.covmodel.Spherical property), 239
opt_arg_bounds (gstools.covmodel.Stable property),
179
opt_arg_bounds (gstools.covmodel.SuperSpherical
property), 263
opt_arg_bounds (gstools.covmodel.TPLExponential
property), 303
opt_arg_bounds
(gstools.covmodel.TPLGaussian
property), 289

Index

P
percentile_scale()
(gstools.covmodel.Circular
method), 223
percentile_scale() (gstools.covmodel.CovModel
method), 126
percentile_scale()
(gstools.covmodel.Cubic
method), 199
percentile_scale() (gstools.covmodel.Exponential
method), 151
percentile_scale() (gstools.covmodel.Gaussian
method), 138
percentile_scale()
(gstools.covmodel.HyperSpherical method),
247
percentile_scale()
(gstools.covmodel.JBessel
method), 271
percentile_scale()
(gstools.covmodel.Linear
method), 211
percentile_scale()
(gstools.covmodel.Matern
method), 163
percentile_scale()
(gstools.covmodel.Rational
method), 187
percentile_scale() (gstools.covmodel.Spherical
method), 235
percentile_scale()
(gstools.covmodel.Stable
method), 175
percentile_scale()
(gstools.covmodel.SuperSpherical method),
259
percentile_scale()
(gstools.covmodel.TPLExponential method),
298
percentile_scale()
(gstools.covmodel.TPLGaussian
method),
284
percentile_scale() (gstools.covmodel.TPLSimple
method), 324
percentile_scale() (gstools.covmodel.TPLStable
method), 312
plot() (gstools.covmodel.Circular method), 223
plot() (gstools.covmodel.CovModel method), 126
plot() (gstools.covmodel.Cubic method), 199
plot() (gstools.covmodel.Exponential method), 151
plot() (gstools.covmodel.Gaussian method), 138
plot() (gstools.covmodel.HyperSpherical method),
247
plot() (gstools.covmodel.JBessel method), 271
plot() (gstools.covmodel.Linear method), 211
plot() (gstools.covmodel.Matern method), 163
plot() (gstools.covmodel.Rational method), 187
plot() (gstools.covmodel.Spherical method), 235
plot() (gstools.covmodel.Stable method), 175
499


pykrige_angle (gstools.covmodel.TPLExponential property), 303
pykrige_angle (gstools.covmodel.TPLGaussian property), 289
pykrige_angle (gstools.covmodel.TPLSimple property), 329
pykrige_angle (gstools.covmodel.TPLStable property), 317
pykrige_angle_x (gstools.covmodel.Circular property), 227
pykrige_angle_x (gstools.covmodel.CovModel property), 130
pykrige_angle_x (gstools.covmodel.Cubic property), 203
pykrige_angle_x (gstools.covmodel.Exponential property), 155
pykrige_angle_x (gstools.covmodel.Gaussian property), 143
pykrige_angle_x (gstools.covmodel.HyperSpherical property), 251
pykrige_angle_x (gstools.covmodel.JBessel property), 276
pykrige_angle_x (gstools.covmodel.Linear property), 215
pykrige_angle_x (gstools.covmodel.Matern property), 168
pykrige_angle_x (gstools.covmodel.Rational property), 192
pykrige_angle_x (gstools.covmodel.Spherical property), 239
pykrige_angle_x (gstools.covmodel.Stable property), 179
pykrige_angle_x (gstools.covmodel.SuperSpherical property), 264
pykrige_angle_x (gstools.covmodel.TPLExponential property), 303
pykrige_angle_x (gstools.covmodel.TPLGaussian property), 289
pykrige_angle_x (gstools.covmodel.TPLSimple property), 329
pykrige_angle_x (gstools.covmodel.TPLStable property), 317
pykrige_angle_x (gstools.covmodel.TPLSimple property), 203
pykrige_angle_y (gstools.covmodel.Matern property), 168
pykrige_angle_y (gstools.covmodel.Rational property), 192
pykrige_angle_y (gstools.covmodel.Spherical property), 239
pykrige_angle_y (gstools.covmodel.Stable property), 179
pykrige_angle_y (gstools.covmodel.SuperSpherical property), 264
pykrige_angle_y (gstools.covmodel.TPLExponential property), 303
pykrige_angle_y (gstools.covmodel.TPLGaussian property), 289
pykrige_angle_y (gstools.covmodel.TPLSimple property), 329
pykrige_angle_y (gstools.covmodel.TPLStable property), 317
pykrige_angle_y (gstools.covmodel.TPLSimple property), 203
pykrige_angle_z (gstools.covmodel.Matern property), 168
pykrige_angle_z (gstools.covmodel.Rational property), 192
pykrige_angle_z (gstools.covmodel.Spherical property), 239
pykrige_angle_z (gstools.covmodel.Stable property), 179
pykrige_angle_z (gstools.covmodel.SuperSpherical property), 264
pykrige_angle_z (gstools.covmodel.TPLExponential property), 303
pykrige_angle_z (gstools.covmodel.TPLGaussian property), 289
pykrige_angle_z (gstools.covmodel.TPLSimple property), 329
pykrige_angle_z (gstools.covmodel.TPLStable property), 317
pykrige_anis (gstools.covmodel.Circular property), 227
pykrige_anis (gstools.covmodel.CovModel property), 130
pykrige_anis (gstools.covmodel.Cubic property), 203
pykrige_anis (gstools.covmodel.Exponential property), 156
pykrige_anis (gstools.covmodel.Gaussian property), 143
pykrige_anis (gstools.covmodel.HyperSpherical property), 251
pykrige_anis (gstools.covmodel.JBessel property), 276
pykrige_anis (gstools.covmodel.Linear property), 215
pykrige_anis (gstools.covmodel.Matern property), 168
pykrige_anis (gstools.covmodel.Rational property), 192
pykrige_anis (gstools.covmodel.Spherical property), 239
pykrige_anis (gstools.covmodel.Stable property), 179
pykrige_anis (gstools.covmodel.Superspherical property), 264
pykrige_anis (gstools.covmodel.TPLExponential property), 303
pykrige_anis (gstools.covmodel.TPLGaussian property), 289
pykrige_anis (gstools.covmodel.TPLSimple property), 329
pykrige_anis (gstools.covmodel.TPLStable property), 317
pykrige_anis_y (gstools.covmodel.Circular property), 228
pykrige_anis_y (gstools.covmodel.CovModel property), 130
pykrige_anis_y (gstools.covmodel.Cubic property), 204
pykrige_anis_y (gstools.covmodel.Exponential property), 289
pykrige_anis_y (gstools.covmodel.TPLSimple property), 329
pykrige_anis_y (gstools.covmodel.TPLStable property), 317
pykrige_anis_z (gstools.covmodel.Circular property), 228
pykrige_anis_z (gstools.covmodel.CovModel property), 130
pykrige_anis_z (gstools.covmodel.Cubic property), 204
pykrige_anis_z (gstools.covmodel.Exponential property), 156
pykrige_anis_z (gstools.covmodel.HyperSpherical property), 252
pykrige_anis_z (gstools.covmodel.JBessel property), 276
pykrige_anis_z (gstools.covmodel.Linear property), 216
pykrige_anis_z (gstools.covmodel.Matern property), 168
pykrige_anis_z (gstools.covmodel.Rational property), 192
pykrige_anis_z (gstools.covmodel.Spherical property), 240
pykrige_anis_z (gstools.covmodel.Stable property), 180
pykrige_anis_z (gstools.covmodel.Superspherical property), 264
pykrige_anis_z (gstools.covmodel.TPLExponential property), 303
pykrige_anis_z (gstools.covmodel.TPLGaussian property), 289
pykrige_anis_z (gstools.covmodel.TPLSimple property), 329
pykrige_anis_z (gstools.covmodel.TPLStable property), 317
pykrige_kwargs (gstools.covmodel.Circular property), 228
pykrige_kwargs (gstools.covmodel.CovModel property), 130
pykrige_kwargs (gstools.covmodel.Cubic property), 204
pykrige_kwargs (gstools.covmodel.Exponential property), 156
pykrige_kwargs (gstools.covmodel.Gaussian property), 143
pykrige_kwargs (gstools.covmodel.HyperSpherical property), 252
pykrige_kwargs (gstools.covmodel.JBessel property), 276
pykrige_kwargs (gstools.covmodel.Linear property), 216
pykrige_kwargs (gstools.covmodel.Matern property), 168
pykrige_kwargs (gstools.covmodel.Rational property), 192
pykrige_kwargs (gstools.covmodel.Spherical property), 240
pykrige_kwargs (gstools.covmodel.Stable property), 180
pykrige_kwargs (gstools.covmodel.SuperSpherical property), 264
pykrige_kwargs (gstools.covmodel.TPLExponential property), 303
pykrige_kwargs (gstools.covmodel.TPLGaussian property), 289
pykrige_kwargs (gstools.covmodel.TPLSimple property), 329
pykrige_vario() (gstools.covmodel.Circular method), 223
pykrige_vario() (gstools.covmodel.CovModel method), 126
pykrige_vario() (gstools.covmodel.Cubic method), 199
pykrige_vario() (gstools.covmodel.Exponential method), 151
pykrige_vario() (gstools.covmodel.Gaussian method), 139
pykrige_vario() (gstools.covmodel.HyperSpherical method), 247
pykrige_vario() (gstools.covmodel.JBessel method), 272
pykrige_vario() (gstools.covmodel.Linear method), 211
pykrige_vario() (gstools.covmodel.Matern method), 164
pykrige_vario() (gstools.covmodel.Rational method), 188
pykrige_vario() (gstools.covmodel.Spherical method), 235
pykrige_vario() (gstools.covmodel.Stable method), 175
pykrige_vario() (gstools.covmodel.SuperSpherical method), 260
pykrige_vario() (gstools.covmodel.TPLExponential method), 298
pykrige_vario() (gstools.covmodel.TPLGaussian method), 284
pykrige_vario() (gstools.covmodel.TPLSimple method), 325
pykrige_vario() (gstools.covmodel.TPLStable method), 312
rescale (gstools.covmodel.Exponential property), 156
rescale (gstools.covmodel.Gaussian property), 143
rescale (gstools.covmodel.HyperSpherical property), 252
rescale (gstools.covmodel.JBessel property), 276
rescale (gstools.covmodel.Linear property), 216
rescale (gstools.covmodel.Matern property), 168
rescale (gstools.covmodel.Rational property), 192
rescale (gstools.covmodel.Spherical property), 240
rescale (gstools.covmodel.Stable property), 180
rescale (gstools.covmodel.SuperSpherical property), 264
rescale (gstools.covmodel.TPLExponential property), 303
rescale (gstools.covmodel.TPLGaussian property), 289
rescale (gstools.covmodel.TPLSimple property), 329
rescale (gstools.covmodel.TPLStable property), 317
reset_seed() (gstools.field.generator.IncomprRandMeth method), 353
reset_seed() (gstools.field.generator.RandMeth method), 356
RNG (class in gstools.random), 418
rotated_main_axes() (in module gstools.tools), 427
rotation_planes() (in module gstools.tools), 428
S
sample_dist() (gstools.random.RNG method), 418
sample_ln_pdf() (gstools.random.RNG method), 418
sample_sphere() (gstools.random.RNG method), 418
sampling (gstools.field.generator.IncomprRandMeth property), 354
sampling (gstools.field.generator.RandMeth property), 356
seed (gstools.field.generator.IncomprRandMeth property), 354
seed (gstools.field.generator.RandMeth property), 356
seed (gstools.random.MasterRNG property), 420
seed (gstools.random.RNG property), 419
set_angles() (in module gstools.tools), 427
set_anis() (in module gstools.tools), 427
set_arg_bounds() (gstools.covmodel.Circular method), 224
set_arg_bounds() (gstools.covmodel.CovModel method), 126
set_arg_bounds() (gstools.covmodel.Cubic method), 200
set_arg_bounds() (gstools.covmodel.Exponential method), 151
set_arg_bounds() (gstools.covmodel.Gaussian method), 139
set_arg_bounds() (gstools.covmodel.HyperSpherical method), 248
set_arg_bounds() (gstools.covmodel.JBessel method), 272

Index
set_arg_bounds() (gstools.covmodel.Linear method), 212
set_arg_bounds() (gstools.covmodel.Matern method), 164
set_arg_bounds() (gstools.covmodel.Rational method), 188
set_arg_bounds() (gstools.covmodel.Spherical method), 236
set_arg_bounds() (gstools.covmodel.Stable method), 176
set_arg_bounds() (gstools.covmodel.SuperSpherical method), 260
set_arg_bounds() (gstools.covmodel.TPLExponential method), 285
set_arg_bounds() (gstools.covmodel.TPLGaussian method), 325
set_arg_bounds() (gstools.covmodel.TPLSimple method), 312
set_condition() (gstools.krige.Detrended method), 413
set_condition() (gstools.krige.ExtDrift method), 404
set_condition() (gstools.krige.Krige method), 368
set_condition() (gstools.krige.Ordinary method), 386
set_condition() (gstools.krige.Simple method), 395
set_condition() (gstools.krige.Universal method), 396
set_drift_functions() (gstools.krige.Detrended method), 413
set_drift_functions() (gstools.krige.ExtDrift method), 405
set_drift_functions() (gstools.krige.Krige method), 368
set_drift_functions() (gstools.krige.Ordinary method), 387
set_drift_functions() (gstools.krige.Simple method), 378
set_drift_functions() (gstools.krige.Universal method), 396
set_generator() (gstools.field.CondSRF method), 343
set_generator() (gstools.field.SRF method), 336
set_pos() (gstools.field.CondSRF method), 343
set_pos() (gstools.field.Field method), 349
set_pos() (gstools.field.SRF method), 337
set_pos() (gstools.krige.Detrended method), 414
set_pos() (gstools.krige.ExtDrift method), 405
set_pos() (gstools.krige.Krige method), 368
set_pos() (gstools.krige.Ordinary method), 387
set_pos() (gstools.krige.Simple method), 378
set_pos() (gstools.krige.Universal method), 396
sill (gstools.covmodel.Circular property), 228
sill (gstools.covmodel.CovModel property), 130
sill (gstools.covmodel.Cubic property), 204
sill (gstools.covmodel.Exponential property), 156
sill (gstools.covmodel.Gaussian property), 143
sill (gstools.covmodel.HyperSpherical property), 252
sill (gstools.covmodel.JBessel property), 276
sill (gstools.covmodel.Linear property), 216
sill (gstools.covmodel.Matern property), 168
sill (gstools.covmodel.Rational property), 192
sill (gstools.covmodel.Spherical property), 240
sill (gstools.covmodel.Stable property), 180
sill (gstools.covmodel.SuperSpherical property), 264
sill (gstools.covmodel.TPLExponential property), 303
sill (gstools.covmodel.TPLGaussian property), 289
sill (gstools.covmodel.TPLSimple property), 329
sill (gstools.covmodel.TPLStable property), 317
Simple (class in gstools.krige), 373
spectral_density() (gstools.covmodel.Circular method), 224
spectral_density() (gstools.covmodel.CovModel method), 126
spectral_density() (gstools.covmodel.Cubic method), 200
spectral_density() (gstools.covmodel.Exponential method), 152
spectral_density() (gstools.covmodel.Gaussian method), 139
spectral_density() (gstools.covmodel.HyperSpherical method), 248
spectral_density() (gstools.covmodel.JBessel method), 272
spectral_density() (gstools.covmodel.Linear method), 212
spectral_density() (gstools.covmodel.Matern method), 164
spectral_density() (gstools.covmodel.Rational method), 188
spectral_density() (gstools.covmodel.Spherical method), 236
spectral_density() (gstools.covmodel.Stable method), 176
spectral_density() (gstools.covmodel.SuperSpherical method), 260
spectral_density() (gstools.covmodel.TPLExponential method), 299
spectral_density() (gstools.covmodel.TPLGaussian method), 285
spectral_density() (gstools.covmodel.TPLSimple method), 325
spectral_density() (gstools.covmodel.TPLStable method), 317
spectral_rad_cdf() (gstools.covmodel.Exponential method), 152
spectral_rad_cdf() (gstools.covmodel.Gaussian method), 139
spectral_rad_pdf() (gstools.covmodel.Circular method), 224
Index

224 spectral_rad_pdf() (gstools.covmodel.CovModel method), 127
spectral_rad_pdf() (gstools.covmodel.Cubic method), 200
spectral_rad_pdf() (gstools.covmodel.Exponential method), 152
spectral_rad_pdf() (gstools.covmodel.Gaussian method), 139
spectral_rad_pdf() (gstools.covmodel.HyperSpherical method), 248
spectral_rad_pdf() (gstools.covmodel.JBessel method), 272
spectral_rad_pdf() (gstools.covmodel.Linear method), 212
spectral_rad_pdf() (gstools.covmodel.Matern method), 164
spectral_rad_pdf() (gstools.covmodel.Rational method), 188
spectral_rad_pdf() (gstools.covmodel.Spherical method), 236
spectral_rad_pdf() (gstools.covmodel.Stable method), 176
spectral_rad_pdf() (gstools.covmodel.SuperSpherical method), 260
spectral_rad_pdf() (gstools.covmodel.TPLExponential method), 285
spectral_rad_pdf() (gstools.covmodel.TPLGaussian method), 325
spectral_rad_pdf() (gstools.covmodel.TPLSimple method), 313
Spherical (class in gstools.covmodel), 229
SRF (class in gstools.field), 333
Stable (class in gstools.covmodel), 169
standard_bins() (in module gstools.variogram), 362
structured() (gstools.field.CondSRF method), 343
structured() (gstools.field.Field method), 349
structured() (gstools.krige.Detrended method), 414
structured() (gstools.krige.ExtDrift method), 405
structured() (gstools.krige.Krige method), 369
structured() (gstools.krige.Ordinary method), 387
structured() (gstools.krige.Simple method), 378
structured() (gstools.krige.Universal method), 396
SuperSpherical (class in gstools.covmodel), 253
T

to_pyvista() (gstools.field.CondSRF method), 343
to_pyvista() (gstools.field.Field method), 349
to_pyvista() (gstools.field.SRF method), 337
to_pyvista() (gstools.krige.Detrended method), 414
to_pyvista() (gstools.krige.ExtDrift method), 405
to_pyvista() (gstools.krige.Krige method), 369
to_pyvista() (gstools.krige.Ordinary method), 387
to_pyvista() (gstools.krige.Simple method), 378
to_pyvista() (gstools.krige.Universal method), 396
to_vtk() (in module gstools.tools), 423
to_vtk_structured() (in module gstools.tools), 423
to_vtk_unstructured() (in module gstools.tools), 423
tpl_exp_spec_dens() (in module gstools.tools), 425
tpl_gau_spec_dens() (in module gstools.tools), 426
TPLExponential (class in gstools.covmodel), 291
TPLGaussian (class in gstools.covmodel), 277
TPLSimple (class in gstools.covmodel), 318
TPLStable (class in gstools.covmodel), 305
tplstable_cor() (in module gstools.tools), 425
transform() (gstools.field.CondSRF method), 344
transform() (gstools.field.Field method), 349
transform() (gstools.field.SRF method), 337
transform() (gstools.krige.Detrended method), 414
transform() (gstools.krige.ExtDrift method), 405
transform() (gstools.krige.Krige method), 369
transform() (gstools.krige.Ordinary method), 387
transform() (gstools.krige.Simple method), 378
transform() (gstools.krige.Universal method), 396
trend (gstools.field.CondSRF property), 345
trend (gstools.field.Field property), 351
trend (gstools.field.SRF property), 339
trend (gstools.krige.Detrended property), 417
trend (gstools.krige.ExtDrift property), 408
trend (gstools.krige.Krige property), 371
unbiased (gstools.krige.Detrended property), 417
unstructured() (gstools.field.ConDSRF method), 344

update() (gstools.field.generator.IncomprRandMeth method), 353
upsampling (gstools.field.SRF property), 339
upsampling_func() (gstools.field.SRF method), 337

V

var_factor() (gstools.covmodel.Stable property), 180
var_factor() (gstools.covmodel.Cubic method), 200
var_factor() (gstools.covmodel.Circular property), 224
var_factor() (gstools.covmodel.Detrended property), 417
var_factor() (gstools.covmodel.Cubic method), 200
var_factor() (gstools.covmodel.Stable property), 176

Index
var_factor() (gstools.covmodel.SuperSpherical method), 260
var_factor() (gstools.covmodel.TPLExponential method), 299
var_factor() (gstools.covmodel.TPLGaussian method), 285
var_factor() (gstools.covmodel.TPLSimple method), 325
var_factor() (gstools.covmodel.TPLStable method), 313
var_estimate() (in module gstools.variogram), 359
var_estimate_axis() (in module gstools.variogram), 361
vario_axis() (gstools.covmodel.TPLGaussian method), 285
vario_axis() (gstools.covmodel.TPLSimple method), 325
vario_axis() (gstools.covmodel.TPLStable method), 313
vario_estimate() (in module gstools.variogram), 359
vario_estimate_axis() (in module gstools.variogram), 361
vario_nugget() (gstools.covmodel.Circular method), 224
vario_nugget() (gstools.covmodel.CovModel method), 127
vario_nugget() (gstools.covmodel.Cubic method), 200
vario_nugget() (gstools.covmodel.Exponential method), 152
vario_nugget() (gstools.covmodel.Gaussian method), 140
vario_nugget() (gstools.covmodel.HyperSpherical method), 248
vario_nugget() (gstools.covmodel.JBessel method), 272
vario_nugget() (gstools.covmodel.Linear method), 212
vario_nugget() (gstools.covmodel.Matern method), 164
vario_nugget() (gstools.covmodel.Rational method), 188
vario_nugget() (gstools.covmodel.Spherical method), 236
vario_nugget() (gstools.covmodel.Stable method), 176
vario_nugget() (gstools.covmodel.TPLExponential method), 299
vario_nugget() (gstools.covmodel.TPLGaussian method), 285
vario_nugget() (gstools.covmodel.TPLSimple method), 325
vario_nugget() (gstools.covmodel.TPLStable method), 313
vario_estimate() (in module gstools.variogram), 359
vario_estimate_axis() (in module gstools.variogram), 361
vario_nugget() (gstools.covmodel.Circular method), 224
vario_nugget() (gstools.covmodel.CovModel method), 127
vario_nugget() (gstools.covmodel.Cubic method), 200
vario_nugget() (gstools.covmodel.Exponential method), 152
vario_nugget() (gstools.covmodel.Gaussian method), 140
vario_nugget() (gstools.covmodel.HyperSpherical method), 248
vario_nugget() (gstools.covmodel.JBessel method), 272
vario_nugget() (gstools.covmodel.Linear method), 212
vario_nugget() (gstools.covmodel.Matern method), 164
vario_nugget() (gstools.covmodel.Rational method), 188
vario_nugget() (gstools.covmodel.Spherical method), 236
vario_nugget() (gstools.covmodel.Stable method), 176
vario_nugget() (gstools.covmodel.TPLExponential method), 299
vario_nugget() (gstools.covmodel.TPLGaussian method), 285
vario_nugget() (gstools.covmodel.TPLSimple method), 325
vario_nugget() (gstools.covmodel.TPLStable method), 313
vario_estimate() (in module gstools.variogram), 359
vario_estimate_axis() (in module gstools.variogram), 361
vario_nugget() (gstools.covmodel.Circular method), 224
vario_nugget() (gstools.covmodel.CovModel method), 127
vario_nugget() (gstools.covmodel.Cubic method), 200
vario_nugget() (gstools.covmodel.Exponential method), 152
vario_nugget() (gstools.covmodel.Gaussian method), 140
vario_nugget() (gstools.covmodel.HyperSpherical method), 248
vario_nugget() (gstools.covmodel.JBessel method), 272