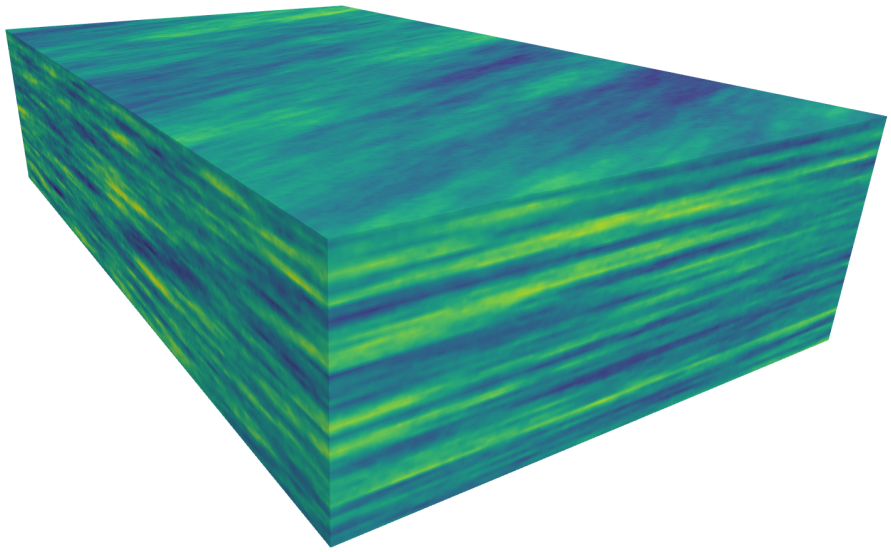
Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

Note

Simulation of Gaussian Random Fields Using the Fast Fourier Transform



Note no SAND/04/2018

Authors Vegard Kvernelv
Daniel Barker
Petter Abrahamsen

Date February 28, 2018

Norwegian Computing Center

Norsk Regnesentral (Norwegian Computing Center, NR) is a private, independent, non-profit foundation established in 1952. NR carries out contract research and development projects in information and communication technology and applied statistical-mathematical modelling. The clients include a broad range of industrial, commercial and public service organisations in the national as well as the international market. Our scientific and technical capabilities are further developed in co-operation with The Research Council of Norway and key customers. The results of our projects may take the form of reports, software, prototypes, and short courses. A proof of the confidence and appreciation our clients have in us is given by the fact that most of our new contracts are signed with previous customers.

Norsk Regnesentral
Norwegian Computing Center
Postboks 114, Blindern
NO-0314 Oslo, Norway

Besøksadresse
Office address
Gaustadalléen 23
NO-0373 Oslo, Norway

Telefon · telephone
(+47) 22 85 25 00
Telefaks · telefax
(+47) 22 69 76 60

Internett · internet
www.nr.no
E-post · e-mail
nr@nr.no

Title	**Simulation of Gaussian Random Fields Using the Fast Fourier Transform**
Authors	**Vegard Kvernelv , Daniel Barker , Petter Abrahamsen**
Date	February 28, 2018
Publication number	SAND/04/2018

Abstract

This report documents a Python package called `nrlib` for simulating a Gaussian random field (GRF) realization represented on a regular grid. The method exploit the numerical accuracy and efficiency of the Fast Fourier transform (FFT) to generate samples fast with close to perfect statistical properties.

Keywords	stochastic simulation, Gaussian random field, FFT
Target group	
Availability	Open
Project	
Project number	190510
Research field	
Number of pages	33

Contents

1 Background

This report documents a Python package called `nrlib` for simulating a Gaussian random field (GRF) realization represented on a regular grid. The method exploit the Fast Fourier transform (FFT) to generate samples fast with close to perfect statistical properties.

The basic idea is to generate independent Gaussian random variables on a regular grid and use a spatial filter to smooth the independent random variables to obtain a spatially correlated GRF. The FFT is used to speed up the smoothing since a filtering process (convolution) is a simple cell-by-cell multiplication in the Fourier domain. A representation of the spatial convolution filter in the Fourier domain is easily obtained from the FFT of any stationary correlation function. The properties of GRF is briefly discussed in Section A and the simulation method is explained in Section B.

The `nrlib` package is an interface to the C++ library NRLib using the Boost Python framework. It is possible to implement the package in a pure Python library using `numpy`, but due to efficiency requirements a C++ based solution was chosen. Figure 1.1 shows a comparison between a `numpy` implementation and the chosen C++ based solution.

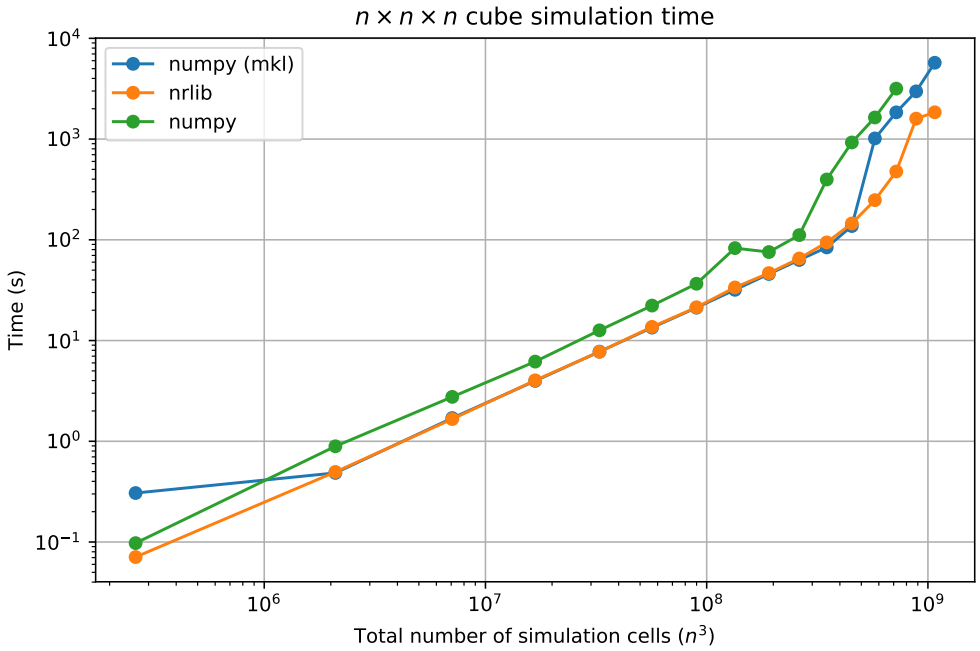


Figure 1.1. Comparison of speed using a `numpy` implementation and the chosen C++ based `nrlib` solution. The `numpy (mkl)` numbers are obtained using a `numpy` distribution that uses Intel MKL. The trend is linear except for large grids where memory limitations comes into effect. These results were retrieved on a laptop with 32 GB RAM.

The `nrlib` package is described in Section 2 including a tutorial in Section 2.1 and some details on installation and compilation in Section 2.2. The `nrlib` API is documentet in Section 3.

The appendix contains a mathematical description of the simulation method. Section A provides some background on GRFs and Section B explains the FFT simulation algorithm in detail. Section C discuss why the grids representing the simulated GRFs need to be padded to provide

perfect results.

1.1 Literature

The basic references to simulation of GRF using spectral methods are Ripley (1987), Dietrich and Newsam (1993) and Wood and Chan (1994). Other relevant references are Chilés and Delfiner (1997), Mantoglou and Wilson (1982), Pardo-Igúzquiza and Chica-Olmo (1993), Pardo-Igúzquiza and Chica-Olmo (1994a), Pardo-Igúzquiza and Chica-Olmo (1994b), and Goff and Jennings, Jr. (1999).

2 The `nrlib` Package

The `nrlib` package consists of a Python module that is built on top of a C++ library called NRLib using Boost Python. For the user, it is intended to behave like a regular Python package. However, there are differences between the data types and structures in C++ and Python, and therefore not all conversions from C++ objects to Python objects are straightforward. Most types can be exposed to Python in such a way that they are indistinguishable from Python data types, but in many cases the work to get there may be significant.

NRLib exposes functions for simulating GRFs, utilizing the FFT-algorithm for speed. Moreover, the C++ library uses FFT-algorithms implemented in Intel MKL to get optimized performance. The main functions of the module are `simulate` and `variogram`. The full API overview can be found in Section 3.

The purpose of the two main functions is to first use `variogram` to create a variogram that can be passed to `simulate`. An option would have been to have all the variogram related parameters as arguments to `simulate`, but since the variogram parameters does not depend on the simulation grid parameters, it is a natural de-coupling of the procedure.

2.1 Tutorial

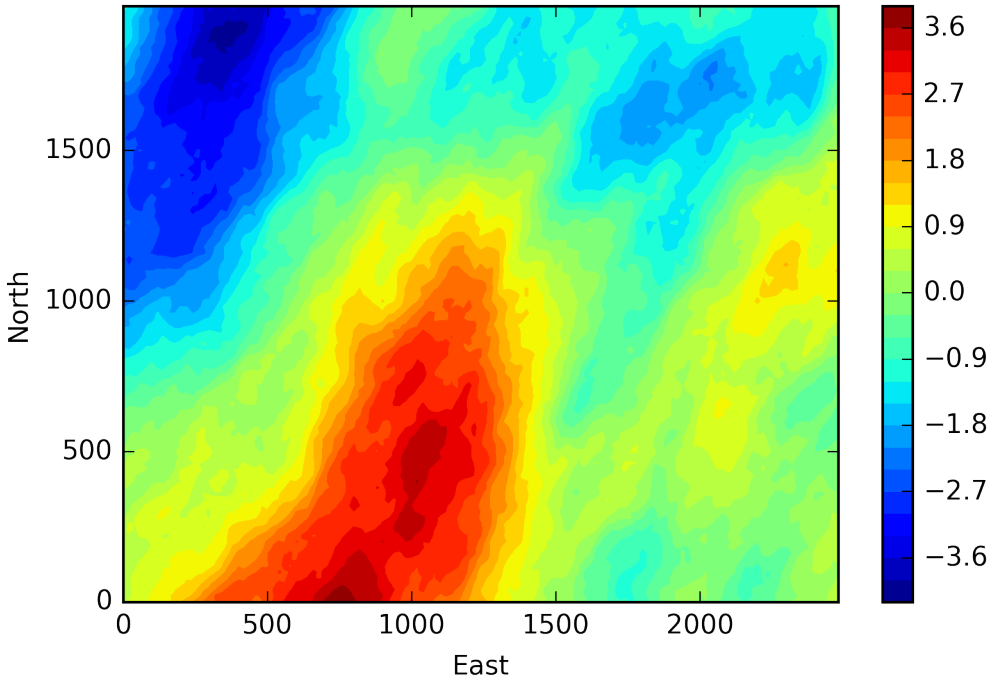
This section introduces the core methods of NRLib and shows a simple example of how they can be used along with some explanations. The entire example:

```
In [ ]: import nrlib
        import numpy as np
        import matplotlib.pyplot as plt

        v = nrlib.variogram('general_exponential', 2000.0, 1000.0, azimuth=30.0)
        nx, ny = 100, 125
        dx, dy = 20.0, 20.0
        nrlib.seed(42)
        f_vector = nrlib.simulate(v, nx, dx, ny, dy)
        f = f_vector.reshape((nx, ny), order='F')

        x = np.arange(0, nx) * dx
        y = np.arange(0, ny) * dy
        plt.contourf(y, x, f, 30)
        plt.colorbar()
        plt.xlabel('East')
        plt.ylabel('North')
```

```
plt.show()
```



The first part contains import statements. `nrlib` is an obvious import. `numpy` is used to define the grid and `matplotlib` is used for visualization.

```
In [ ]: import nrlib
        import numpy as np
        import matplotlib.pyplot as plt
```

Next, the variogram is created using the `variogram`-factory function. The variogram is anisotropic since the range is 2000 in the main direction and 1000 in the perpendicular direction. Also, an azimuth angle is used to rotate the variogram.

```
In [ ]: v = nrlib.variogram('general_exponential', 2000.0, 1000.0, azimuth=30.0)
```

Next, the grid parameters are set and a seed is provided. The seed is not mandatory, and would have been set automatically if `nrlib.seed` had not been explicitly called. Note that if the seed is set automatically, it uses the current wall clock time with second precision. Further, the seed is not reset unless `nrlib` is unloaded.

For multi-threaded applications it is recommended to explicitly set the seed for each thread, unless the implications of second-precision seeding are fully understood. For instance, when using the built-in `multiprocessing` module, each thread may have its own instance of `nrlib`, depending on how the subprocess is started. This means that threads will fetch seeds independently, but threads that start within the same wall clock second will fetch the same seed and thus yield the same realization (given that simulation parameters are the same).

```
In [ ]: nx, ny = 100, 100
        dx, dy = 20.0, 20.0
        nrlib.seed(42)
```

The simulation is separated into two steps; simulation and reshaping. The simulation returns a one-dimensional `numpy.ndarray`. The reshaping step is necessary when the number of dimensions is greater than 1, since the output from `simulate` will always be one-dimensional. The output array uses Fortran ordering, which is supported by the `reshape`-function.

```
In [ ]: f_vector = nrlib.simulate(v, nx, dx, ny, dy)
        f = f_vector.reshape((nx, ny), order='F')
```

We want to plot the simulated field using matplotlib's `contourf`, and we therefore need to define the values for the two axes.

```
In [ ]: x = np.arange(0, nx) * dx
        y = np.arange(0, ny) * dy
```

Finally, the solution is plotted and the axes are labeled. Note that azimuth is the angle clockwise from the first axis towards the second. When interpreting x in the above as North and y as East, the definition of azimuth is the degrees from the North direction, taken clockwise, which is the same as in a standard North-East-Down coordinate system.

```
In [ ]: plt.contourf(y, x, f, 30)
        plt.colorbar()
        plt.xlabel('East')
        plt.ylabel('North')
        plt.show()
```

If we instead want left-handed indexing, we can replace the `azimuth` argument in the above with $90 - \texttt{azimuth}$. This is in particular relevant when relating the realization to grids in RMS, which by default uses left-handed indexing.

2.2 Compilation and Technical Details

This section decribes how to start using `nrlib`. However, some details are likely to change over time and the content may not be up-to-date.

`nrlib` is not a pure Python package, but a C++ library that exposes Python bindings using the Boost Python framework. Since it is a C++ library, it needs to be compiled for the target architecture. In addition to Boost Python, `nrlib` depends on Intel MKL.

`nrlib` may be provided in two ways: as a precompiled Python-importable package or as a `pip`-installable package. Installing the package via `pip` is considered more robust, but is also more tedious. Using the precompiled version is simpler, but may be problematic if the system architecture is different from the architecture on which the package was compiled.

2.2.1 Using a precompiled version

The precompiled version will be named something similar to `nrlib.so` or perhaps `nrlib.cpython-34m.so` on Linux, and `nrlib.pyd` on Windows. The only requirement when using a precompiled version of `nrlib`:

1. `nrlib` must be available to the active Python interpreter

This can be solved by having the package in the working directory, copying it to the DLL subfolder (or equivalent) of the Python installation, or by utilizing the PYTHONPATH environment

variable. nrlib is currently being linked statically to Intel MKL and the necessary Boost libraries, and dynamic libraries are therefore not needed.

2.2.2 Compiling nrlib

Compiling nrlib from source requires Boost and Intel MKL. In particular, the Boost Python libraries must be precompiled against the Python version that nrlib should be used with. Compiling against a different minor version (e.g. compiling Boost using 3.4 and import nrlib in 3.6) seem to work, but compiling against different major versions (e.g. 3.4 and 2.7) will not work.

Once Intel MKL is downloaded and installed, set the environment variable MKL_ROOT to the mkl subfolder. This subfolder should contain the folders lib, include and bin, and a few others.

nrlib may be distributed with the necessary Boost dependencies, with headers in the boost-folder and compiled libraries in the lib-folder. If these are incompatible with the system architecture, Boost should be compiled from scratch for the particular system. When compiling a separate Boost version, set the environment variable BOOST_ROOT such that either $BOOST_ROOT/boost or $BOOST_ROOT/include/boost contains the headers, and $BOOST_ROOT/lib contains the compiled libraries. On Linux, make sure that Boost is compiled with the -fPIC flag. Boost is linked statically with nrlib, but since nrlib is a shared library, dropping the flag will lead to linker errors.

nrlib is compiled using pip by running pip install [--user] . in the same directory as setup.py. The pip-argument --user is optional. --user installs the package to the user's home directory instead of the Python-distribution. In some cases (such as when integrating with RMS), this is necessary.

Once compiled, unit tests are available in the tests folder. These unittests can be run as regular Python scripts, but a perhaps easier approach is to use a testing framework such as nose, and type nosetests tests to run all tests. The package can be uninstalled by typing pip uninstall nrlib.

Consult setup.py and comments therein for further details.

3 API Overview

The documentation found in this API was generated using Python built-in function help and thus shows the docstrings of the various functions. All functions support being called with positional arguments, keyword arguments or a combination of both. In the case of positional arguments, see the examples or fetch the docstrings in a Python interpreter for the order of the arguments.

3.1 nrlib.variogram

```
Factory function for creating a particular variogram. The variogram is always
defined in three directions, but for simulation in fewer dimensions than three,
only the corresponding number of directions are used.

Parameters
----------
type: string
    A string representing the type of variogram. The following types are supported:
    gaussian, exponential, general_exponential, spherical, matern32, matern52,
```

```
    matern72 and constant.
main_range: float
    Range of the variogram in the main direction.
perp_range, depth_range: floats, optional
    Parameters representing the range of the variogram in the two directions
    perpendicular to main_range. If any of these are zero, the default is to set
    the value to the same as main_range.
azimuth: float, optional
    Lateral orientation of the variogram in degrees. Default is 0.0.
dip: float, optional
    Dip direction of the variogram in degrees. Default is 0.0
power: float, optional
    Power of the exponent for the general_exponential variogram, which is the only
    variogram type this is used for. Default is 1.5.

Returns
-------
out: Variogram
    An instance of the class nrlib.Variogram.

Examples
--------
>>> nrlib.variogram('gaussian', 1000.0)

Specifying dip

>>> nrlib.variogram('matern52', 1000.0, dip=45.0)

Multiple directions

>>> nrlib.variogram('general_exponential', 1000.0, 500.0, 250.0, power=1.8)
```

3.2 nrlib.simulate

```
Simulates a Gaussian random field with the corresponding variogram in one, two or
three dimensions. The random generator seed may be set by using nrlib.seed.

Parameters
----------
variogram: nrlib.Variogram
    An instance of nrlib.Variogram (see nrlib.variogram).
nx, ny, nz: int
    Grid size of the simulated field. Only nx is required. Setting ny and/or nz to
    a value less than or equal to 1 reduces the dimension. Default is ny = 1 and
    nz = 1.
dx, dy, dz: float
    Grid resolution in x, y and z directions. dx is always required. dy and dz are
    required if respectively ny and nz are greater than 1.
```

```
Returns
-------
out: numpy.ndarray
    One-dimensional array with the simulation result. Uses Fortran ordering if the
    simulation is multi-dimensional.

Examples
--------
>>> v = nrlib.variogram('gaussian', 250.0, 125.0)
>>> nx, dx = 10, 100.0
>>> z = nrlib.simulate(v, nx, dx)
>>> z
array([-1.29924289, -1.51172913, -1.2935657 , -0.80779427,  0.22217236,
1.26740091,  0.66094991, -0.77396656,  0.01523847,  0.44392584])

Multi-dimensional simulation

>>> nx, ny = 100, 200
>>> dx, dy = 10.0, 5.0
>>> z = nrlib.simulate(v, nx, dx, ny, dy)
>>> z_np = z.reshape((nx, ny), order='F')
>>> z_np.shape
(100,200)
```

3.3 nrlib.seed

```
seed( (int)arg1) -> None :

Sets the current simulation seed. If this has not been set when calling
nrlib.simulate, it is set to the current time with second precision. Be wary of
the latter, in particular if nrlib is used in a parallel-processing context.

Examples
--------
>>> nrlib.seed(123)


seed() -> int :

Gets the current simulation seed. Throws RunTimeError if the seed has not been set
yet.

Examples
--------
>>> nrlib.seed(123)
>>> nrlib.seed()
123
```

3.4 nrlib.simulation_size

```
Function for determining the grid size after padding in order to assess the
complexity of the problem. Returns bindings to a vector with up to three elements
with the number of grid cells after the grid has been padded. Signature is the
same as nrlib.simulate.

Examples
--------
>>> v = nrlib.variogram('spherical', 250.0, 125.0)
>>> nx, ny, dx, dy = 100, 100, 10.0, 10.0
>>> list(nrlib.simulation_size(v, nx, dx, ny, dy))
[126, 113]
```

3.5 nrlib.advanced.simulate

```
Same as nrlib.simulate, but with a few additional advanced and
experimental settings.

Parameters
----------
variogram, nx, ny, nz, dx, dy, dz:
    See nrlib.simulate
padx, pady, padz: int
    Grid padding as a number of cells. In nrlib.simulate, these are set
    automatically to the values returned by nrlib.simulation_size.
sx, sy, sz: float
    Gaussian smoothing parameters to reduce the range. The parameters are the
    values of the smoothing kernel at one variogram range and MUST therefore be
    greater than 0 and less than 1. A value close to or greater than 1 means no
    smoothing.

Returns
-------
out: numpy.ndarray
    See nrlib.simulate
```

A Gaussian Random Fields

Let $z(\mathbf{x})$ denote a random variable in the point $\mathbf{x} \in \mathbb{R}^D$ where $D = 1, 2$ or 3[1]. The function $z(\mathbf{x})$ is a GRF if $z(\mathbf{x})$ has a Gaussian distribution at any point $\mathbf{x}$. It is fully specified by its mean function $\mathrm{E}\{z(\mathbf{x})\} = \mu(\mathbf{x})$ and the covariance function between any two points:

$$\mathrm{Cov}\{z(\mathbf{x}_1), z(\mathbf{x}_2)\} = c(\mathbf{x}_1, \mathbf{x}_2). \tag{A.1}$$

The covariance function is not arbitrary. It must be symmetric and positive definite. A positive definite function ensures that the covariance matrix Σ with elements

$$\Sigma_{ij} = c(\mathbf{x}_i, \mathbf{x}_j) \quad \text{for any set of locations} \quad x_i, x_j \in \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \tag{A.2}$$

is a positive definite matrix. This means that the n-dimensional vector $\mathbf{z} = [z(\mathbf{x}_1), z(\mathbf{x}_2), \ldots, z(\mathbf{x}_n)]'$ has a multi Gaussian distribution with covariance matrix Σ.

If $z(\mathbf{x})$ is *stationary*, then $\mu(\mathbf{x}) = \mu$, $\sigma(\mathbf{x}) = \sigma$ and

$$c(\mathbf{x}_1, \mathbf{x}_2) = c(\mathbf{h}), \tag{A.3}$$

where $\mathbf{h} = \mathbf{x}_2 - \mathbf{x}_1$. It is common to call $c(\mathbf{h})$ a (stationary) covariance function. It is closely related to the stationary *correlation function*, $\rho(\mathbf{x})$, by:

$$c(\mathbf{h}) = \sigma^2 \, \rho(\mathbf{h}), \tag{A.4}$$

where σ is the standard deviation of the GRF.

If a stationary $z(\mathbf{x})$ is *isotropic*, then the covariance function only depend on the distance:

$$c(\mathbf{x}_1, \mathbf{x}_2) = c(h), \tag{A.5}$$

where $h = \|\mathbf{h}\|$.

A.1 Variograms

The variogram function, $\gamma(\mathbf{h})$, is closely related to the (stationary) covariance function:

$$\gamma(\mathbf{h}) = \sigma^2 - c(\mathbf{h}). \tag{A.6}$$

If the GRF is isotropic this simplifies to

$$\gamma(h) = \sigma^2 - c(h). \tag{A.7}$$

The significance of isotropic covariance and variogram functions is that it is straight forward to verify that these are positive definite. This is sufficient to guarantee that the GRF is properly defined. Moreover, it is simple to make non-stationary GRF by using an isotropic GRF: Assume that $z'(\mathbf{x})$ is an isotropic GRF with unit variance and correlation function $\rho(h)$. Then the GRF

$$z(\mathbf{x}) = \mu(\mathbf{x}) + \sigma(\mathbf{x}) \, z'(\mathbf{x}), \tag{A.8}$$

has expectation and covariance

$$\mathrm{E}\{z(\mathbf{x})\} = \mu(\mathbf{x}) \tag{A.9}$$

$$\mathrm{Cov}\{z(\mathbf{x}_1), z(\mathbf{x}_2)\} = \sigma(\mathbf{x}_1) \, \sigma(\mathbf{x}_2) \, \rho(h). \tag{A.10}$$

Name	Correlation function	Parameter
Spherical	$1 - d(1.5 - 0.5\,d^2)$, for $d \leq 1$, otherwise 0.	
General Exponential	$\exp(-3\,d^p)$	$0 < p \leq 2$
Gaussian	$\exp(-3\,d^2)$	
Exponential	$\exp(-3\,d)$	
Matern32	$\exp(-s)(1+s)$	$s = 4.744\,d$
Matern52	$\exp(-s)(1+s+\frac{s^2}{3})$	$s = 5.918\,d$
Matern72	$\exp(-s)(1+s+\frac{2s^2}{5}+\frac{s^3}{15})$	$s = 6.877\,d$

Table A.1. Correlation functions available in the `nrlib` package. The relative distance is $d = h/R \geq 0.0$ where h is distance and R is the correlation range. The constants of the Matérn correlation functions are chosen such that the correlation is 0.05 when $d = 1$.

The two functions $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$ can be any function but it is common to choose $\sigma(\mathbf{x}) \geq 0$ so it can be interpreted as the local standard deviation.

The types of correlation functions (variograms) that are available in `nrlib` package are shown in Table A.1. A given correlation function is associated with a range, which is the distance at which the correlation function is (approximately) 0.05. The only exception to this is the spherical correlation function, in which case the range is where the correlation becomes 0. Table A.1 shows the formulas for the correlation functions with relative distance d, which is the distance-to-range ratio.

A.2 Anisotropy

The `nrlib` package supports Euclidean distance and distance relative to an ellipsoid defined by two angles (dip and azimuth) where the three main axes can have different lengths. This corresponds to rotation and squeezing/stretching along ellipsoid axes of an isotropic GRF. Such fields are often called *anisotropic*.

Anisotropy in three dimensions can be specified as an ellipsoid with orientation. The three principal axes are often specified as main range, perpendicular range and vertical range, with each range parallel to the corresponding axis. Another terminology is the parallel, normal and vertical range). Further, the orientation is specified using the azimuth and dip angle. The azimuth angle is the angle between the first and second coordinate axis, and the dip angle is the angle from the plane defined be the first two coordinate axes. Typically, the coordinate axes are North-East-Down.

1. D can be any positve integer but $D > 3$ is not considered by the `nrlib` package.

B Stochastic Simulation

Simulating a spatially correlated GRF can be done by linear transformations of independent Gaussian distributed variables, that is, white noise. Let $\mathbf{w}$ be a vector consisting of white noise. Then, $\mathbf{z}$, defined by

$$\mathbf{z} = L\mathbf{w}, \tag{B.1}$$

is a multivariate Gaussian distributed vector with covariance matrix

$$\Sigma = LL'. \tag{B.2}$$

The elements of Σ depend on the covariance function according to Eq. A.2. Note that L is not uniquely defined by Eq. B.2.

Finding a matrix L that satisfy Eq. B.2 can be done using the eigenvalue decomposition. If $\Sigma = V\Lambda V'$ is the eigenvalue decomposition, then

$$L = V\sqrt{\Lambda} \tag{B.3}$$

satisfies Eq. B.2.

A faster alternative is to use the Cholesky factorization that utilize that Σ is symmetric and positive definite.

The complexity of the eigenvalue decomposition and the Cholesky factorization is n^3, where n is the length of the random vectors $\mathbf{z}$ and $\mathbf{w}$. For most grids even the Cholesky factorization will require enormous computer resources since the number of variables could be many millions or even billions.

B.1 Smoothing White Noise using Convolution

Another way to simulate GRF realization with a specific correlation function is to use a convolution filter. Assuming that we can find a convolution filter, f, such that

$$z(\mathbf{x}) = \int f(\mathbf{x} - \mathbf{h})\, w(\mathbf{h})\, d\mathbf{h}, \tag{B.4}$$

where $w(\mathbf{h})$ is Gaussian white noise, and $z(\mathbf{x})$ is a GRF with a covariance function $c(\mathbf{h})$. Eq. B.4 is a convolution of white noise and is conveniently written

$$z = f * w \tag{B.5}$$

A convolution is a time consuming numerical operation since performing a (discretized) D-dimensional numerical integral involves a huge number of operations to obtain any accuracy. However, the Convolution theorem of Fourier transforms simplifies the procedure. It states that

$$\mathcal{F}(f * w) = \mathcal{F}(f) \cdot \mathcal{F}(w), \tag{B.6}$$

where $\mathcal{F}$ is the Fourier transform and $\cdot$ is the element-wise (complex) product. Combining Eq. B.5 and Eq. B.6 with $z = \mathcal{F}^{-1}\big(\mathcal{F}(z)\big)$ gives

$$z(\mathbf{x}) = \mathcal{F}^{-1}\big(\mathcal{F}(f) \cdot \mathcal{F}(w)\big). \tag{B.7}$$

The Convolution theorem also holds for the discrete Fourier transform that are numerically solved by the robust, accurate and efficient FFT algorithm. Generating white noise on a regular grid is

simple so the missing piece is the filter, f, or more precisely, the Fourier transform of the filter: $\mathcal{F}(f)$.

Again, the Fourier transform provides a simple solution. Assuming

$$f * f = c, \tag{B.8}$$

then it can be shown that

$$\mathrm{Cov}\{z(\mathbf{x}), z(\mathbf{x}+\mathbf{h})\} = \mathrm{Cov}\{(f*w)(\mathbf{x}), (f*w)(\mathbf{x}+\mathbf{h})\} = (f*f)(\mathbf{h}) = c(\mathbf{h}). \tag{B.9}$$

So applying the Convolution theorem to Eq. B.8 gives $\mathcal{F}(f)\cdot\mathcal{F}(f) = \mathcal{F}(c)$ so that $\mathcal{F}(f) = \sqrt{\mathcal{F}(c)}$. Replacing this in Eq. B.7 gives

$$\boxed{z(\mathbf{x}) = \mathcal{F}^{-1}\big(\sqrt{\mathcal{F}(c)}\cdot\mathcal{F}(w)\big).} \tag{B.10}$$

This is the simulation algorithm used by `nrlib` where $\mathcal{F}$ is using the very efficient FFT algorithm. The Fourier transform of the covariance function is real since the covariance function is symmetric. Moreover the Fourier transform is non-negative since the covariance function is assumed positive definite. This is a fundamental property of positive definite functions known as Bochners theorem. In practice, numerical inaccuracies can introduce (minute) negative Fourier coefficients and (minute) imaginary Fourier coefficients.

B.2 Convolution using FFT

The FFT algorithm requires a regular $D = 1, 2$ or 3 dimensional grid to represent the covariance function, the white noise, and the final simulated GRF itself. All these grids are identical in size. The complexity of the FFT algorithm is of order $n \log n$ where n is the number of grid cells. For all practical purposes the CPU time scales linearly with grid size. So increasing grid resolution or grid size comes at a prize. This is particularly notable in three dimensions where grids can include hundreds of millions of cells. Note however that Eq. B.10 is correct, independent of the grid resolution, since the convolution theorem holds for finite grids.

The discrete Fourier transform introduces periodicity. This means that simulated realizations become identical on opposite edges. To avoid this, a padding must be introduced. The padding is an extension of the grid that is removed after the final inverse Fourier transform. A reasonable assumption is that the padding must be approximately one correlation range to avoid unwanted spatial correlations on opposite edges of the grid. The padding is a nuisance since it increase the size of the grids and thus adds to the CPU time. It is therefore important to use as small padding as possible.

The periodicity introduces a second problem: Most covariance functions fail to be positive definite on a cyclic finite space. The consequence is that some of the Fourier coefficients in $\mathcal{F}(c)$ become negative. This is handled by setting all negative coefficients to zero when calculating the filter:[2]

$$\mathcal{F}(f) = \sqrt{\max\big(\mathcal{F}(c), 0\big)}. \tag{B.11}$$

This effectively alter the correlation function into a positive definite function on the cyclic finite space. An illustration of this is shown in Figure C.2. Usually, the negative coefficients are minute and this have no practical consequence but in cases where the range(s) is large compared to the grid length(s) simulated realizations get visual artifacts (Figure C.1) and the variance becomes too

2. It is possible to keep the negative coefficients of $\mathcal{F}(c)$. The final result, $z(x)$, should still be purely real due to symmetries in $\mathcal{F}(w)$. However, setting the negative coefficients to zero provides correlations that are closer to the specified correlation function.

small. Again, the solution is to provide enough padding so that the grid is large compared to the range.

In conclusion, we may have to perform simulation on a larger grid than desired in order to have a well-defined correlation function. In the following, we will refer to the *realization grid* as the grid that represents the realization and which has the desired size. The *simulation grid* is the grid with extra padding. The extra padding must be large enough to avoid

1. cyclicity in the simulated realizations
2. significant negative Fourier coefficients in $\mathcal{F}(c)$.

On the other hand the padding should be as small as possible to keep the need for computer resources to a minimum. The quantification of large enough is addressed in the next section.

C Simulation Grid Size

We want a method for establishing the minimum required simulation grid size for a given set of simulation parameters. Simulation parameters refers to the size of the realization grid and the range of the correlation function. Note that the grid resolution itself, the spacing between different grid cells, does not influence this analysis.

Without loss of generality, we consider isotropic correlation functions and simulation on a line. Given a realization grid length, l_r, we seek the smallest value for the padding length, l_p, such that a simulation grid of length $l_s = l_r + l_p$ satisfy the two criteria above. To avoid cyclicity, it is sufficient that the correlation between the edges of the realization grid is practically zero when wrapping around the grid. This amounts to

$$l_p \gtrsim R, \tag{C.1}$$

where R is the correlation range. This means that the correlation is approximately 0.05 from edge to edge which is sufficiently close to zero.

The second criteria is related to the correlation range R versus the simulation grid length, l_s. The assumption is that the correlation range, R, must be smaller than a range factor, α, of the simulation grid length:

$$R_{\text{max}} < \alpha\, l_s. \tag{C.2}$$

This maximum range factor criteria is different for different correlation functions. For the spherical correlation function where correlations drop to zero at R it is reasonable to assume that $R_{\text{max}} \approx 0.5\, l_s$ will provide good results. For correlation functions where correlations never drop to zero, it is reasonable to believe that they need a smaller range factor. Figure C.4 contains range factors for several correlation functions.

C.1 Determining Maximum Correlation Range for a Given Simulation Grid Length

Figure C.1 shows two simulated realizations of GRFs with general exponential correlation functions with power 1.5. Note the cyclicity since the the whole simulation grid is displayed. The left figure looks fine whereas the right figure displays striping artifacts along the grid directions. The correlation range in the left figure is $R = 0.25\, l_s$ and in the right figure $R = 0.75\, l_s$. As expected,

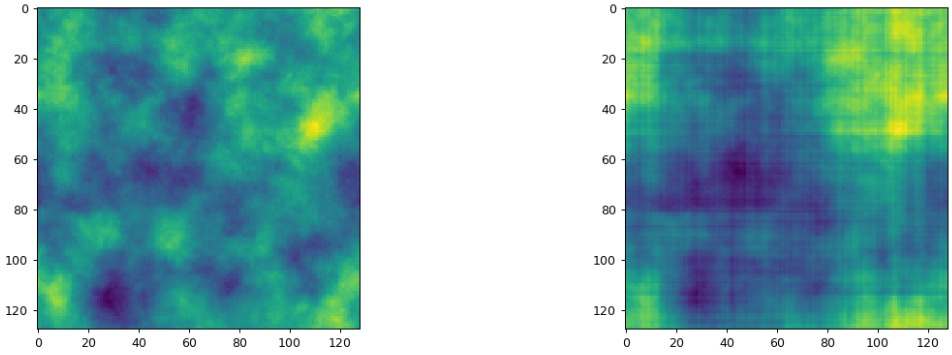


Figure C.1. Simulated fields with varying range, R, relative to simulation grid lengths, l_s. The general exponential correlation function with power 3/2 has been used. The whole simulation grid is displayed so the opposite edges are almost identical. Visual artifacts are apparent in the rightmost picture.

the correlation range must be significantly smaller than the simulation grid length to give good results.

To investigate this further consider the covariance function obtained when removing the negative fourier coefficients:

$$c' = \mathcal{F}^{-1}\big(\max(\mathcal{F}(c), 0)\big). \tag{C.3}$$

This is the covariance function corresponding to the modified filter in Eq. B.11. Recall that negative coefficients are caused by a non-positive definite covariance function on the cyclic domain implied by the FFT-algorithm. Figure C.2 compare this modified correlation function, c', to the original correlation function, c, for $R = 0.75\, l_s$. It is the observed discrepancies that leads to the striping effects. Figure C.3 compares c' to c for $R = 0.25\, l_s$. In this case, the error is visually undetectable.

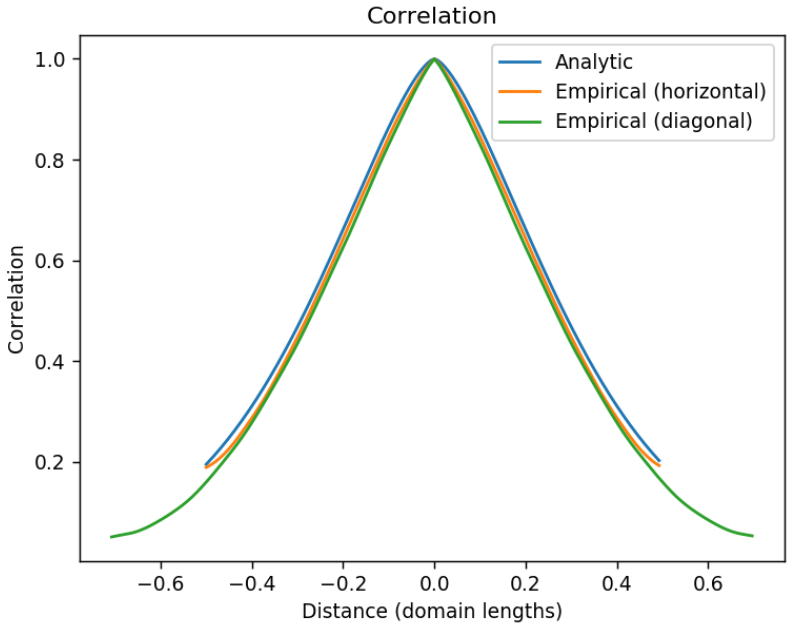


Figure C.2. General exponential correlation function with $p = 1.5$ and $R = 0.75\, l_s$. Comparison of modified covariance function c' (see Eq. C.3) with analytical covariance function c. Note the difference of diagonal correlation compared with horizontal.

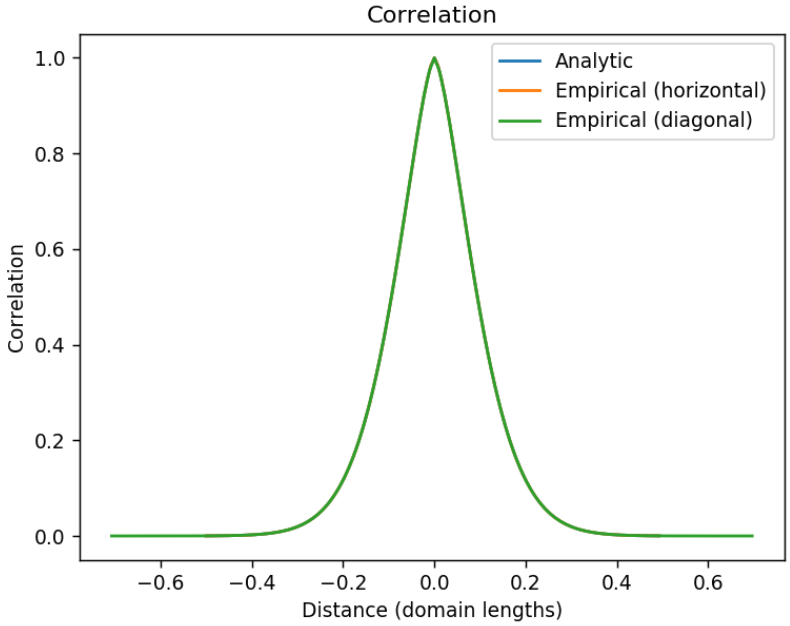


Figure C.3. General exponential variogram with $p = 1.5$ and $R = 0.25\, l_s$. Comparison of modified covariance function c' (see Eq. C.3) with analytical covariance function c. In this case c' and c are indistinguishable.

C.2 Quantitative Results

To make a quantitative assessment consider the L_2 norm of the difference between c' and c:

$$\text{error}^2 = \int_{-l_s/2}^{l_s/2} \big(c(x) - c'(x)\big)^2 dx. \tag{C.4}$$

Figure C.4 shows the error for increasing range values. For this specific correlation function, the error rises dramatically when the range surpasses 0.3 times the simulation grid length, l_s. This indicates that any realization should be padded in such a way that $R < 0.3\, l_s$.

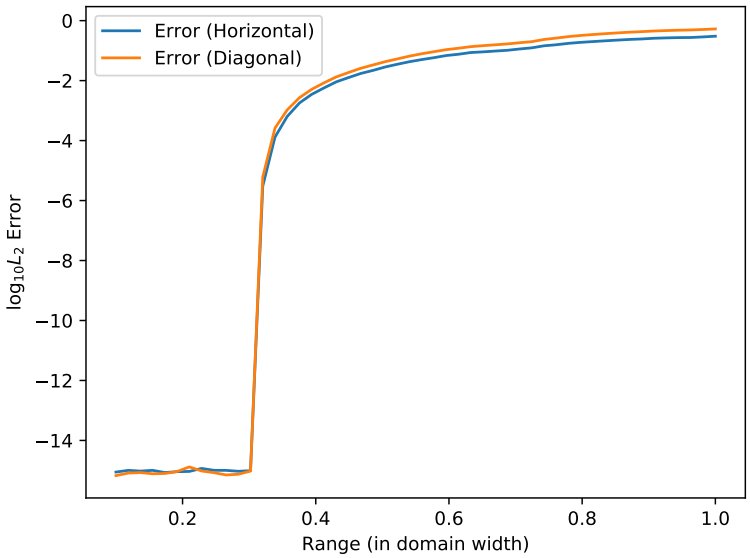


Figure C.4. Error for a general exponential correlation function ($p = 1.5$) as a function of correlation range. The error is given in log-scale.

Section E shows similar figures for the other correlation functions. While most of them have a clear cutoff value, the Gaussian correlation function does not. The recommended maximum range relative to simulation grid lengths for several correlation functions, determined by finding the intersection with error $= 10^{-12}$ on a 64×64 grid, rounded down to two decimals, is given in Table C.1.

Correlation function	Maximum range factor α	Illustration of ϵ
Spherical	0.52	Figure E.1
Exponential	0.43	Figure E.2
General exponential ($p = 1.5$)	0.31	Figure E.3
Gaussian	0.15	Figure E.4
Matern32	0.25	Figure E.5
Matern52	0.21	Figure E.6
Matern72	0.19	Figure E.7

Table C.1. Recommended maximum range factor, α, relative to simulation grid lengths, l_s, for different correlation functions. Maximum range factor criteria is $R_{\text{max}} = \alpha\, l_s$.

The padding must be at least one correlation range to avoid cyclicity. In many cases this padding

is sufficient to meet the maximum range factor criteria, Eq. C.2. Otherwise extra padding must be added to meet the maximum range factor criteria Eq. C.2.

In the case of anisotropic correlation functions, the padding should be determined for each dimension.

C.3 Adjust Padding to Optimize FFT Performance

Classic FFT algorithms require the number of grid nodes in each dimension to be 2^p where p is any positive integer. The p may be different for each dimension. So in three dimensions, the worst case scenario is that the grid must be almost doubled in all three directions. This amounts to increasing CPU time and storage by a factor eight. `nrlib` uses the FFT algorithm in Intel MKL. It has restrictions on the number of grid nodes in each dimension but the requirement is more forgiving. The number of grid nodes in each direction must be represented by

$$2^p\, 3^q\, 5^r\, 7^s \quad \text{where} \quad p, q, r = 0, 1, 2, \ldots \quad \text{and} \quad s = 0, 1. \tag{C.5}$$

Figure C.5 illustrates how many additional grid nodes that is needed to fullfill this restriction. The ideal grid length is the straight line whereas the necessary number of grid nodes to satisfy Eq. C.5 is the red line. It is clear that the extra grid nodes needed are relatively small and will have little impact on performance. This is further illustrated in Figure C.6 that shows the relative increase in the number of grid nodes. The worst cases are for small grids. For more than 200 grid nodes (in one direction) the relative increase in grid nodes are seldom more than 4 %.

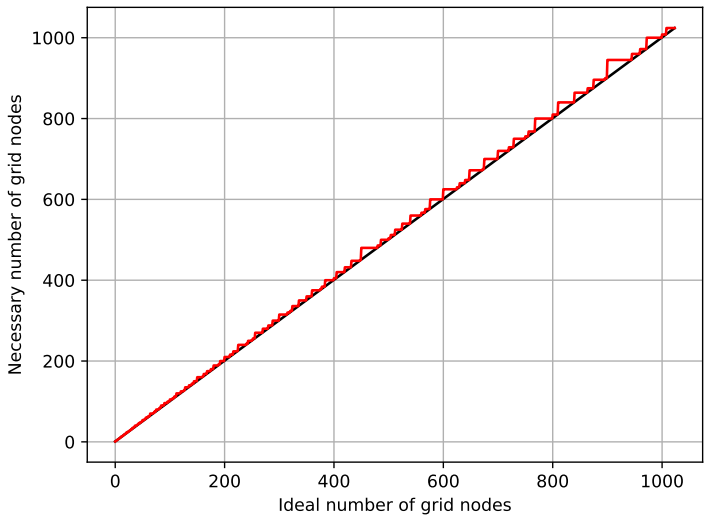


Figure C.5. Red line is number of grid nodes versus number of grid nodes required to comply with the criteria in Eq. C.5.

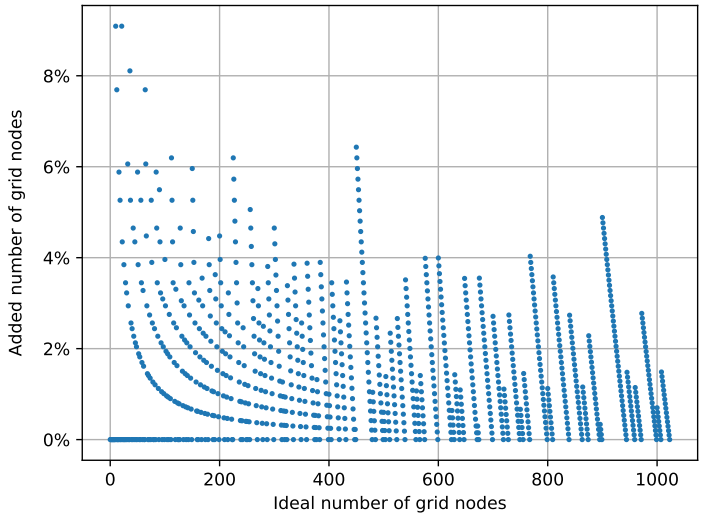


Figure C.6. Relative increase in number of grid nodes needed to comply with criteria Eq. C.5.

D Variogram Estimation

For a set of points, labeled x_i, with corresponding values z_i, the empirical isotropic variogram for z at range d is

$$\gamma(d) = \frac{1}{2N} \sum_{i,j \in X_d} |z_i - z_j|^2, \tag{D.1}$$

where X_d is the set of all pairs, i, j, such that $|x_i - x_j| = d$. N is the number of elements in X_d. The division by 2 ensures an unbiased estimator of the variogram. For a grid in an arbitrary number of dimensions with n points there are $n(n-1)/2$ connections between cells, and thus as many measurements of $z_i - z_j$. For a regular grid, several of these connections will contribute measurements to the same *distance group*, X_d, but the larger d is, the fewer the measurements. For instance, in 1D there will be only one measurement for d equal to the realization grid size. To control for this variance at a given distance d, one can create multiple realizations on the same grid or expand the grid.

In the following analysis, we have generated multiple realization and taken the mean over all realizations to find the empirical variogram. For the sake of implementational simplicity, we have only taken measurements of the difference in z when one of the points is fixed in the origin. The primary purpose of doing these analysis is to verify that the code works as intended.

The parameters are given in Table D.1. This is expected to be unbiased since the amount of padding is sufficient according to the previous tests. Figure D.1 shows the result. It is pretty much as expected. The empirical variogram is relatively accurate, with a slight trend towards overestimation at short distances (less than ∼20).

Parameter	**Value**
Variogram type	Exponential
Dimensions	2
Range	100
Grid size	100
Padding	300
Grid Resolution	1.0
Realizations	5000

Table D.1. Simulation parameters for variogram estimation of a simple case.

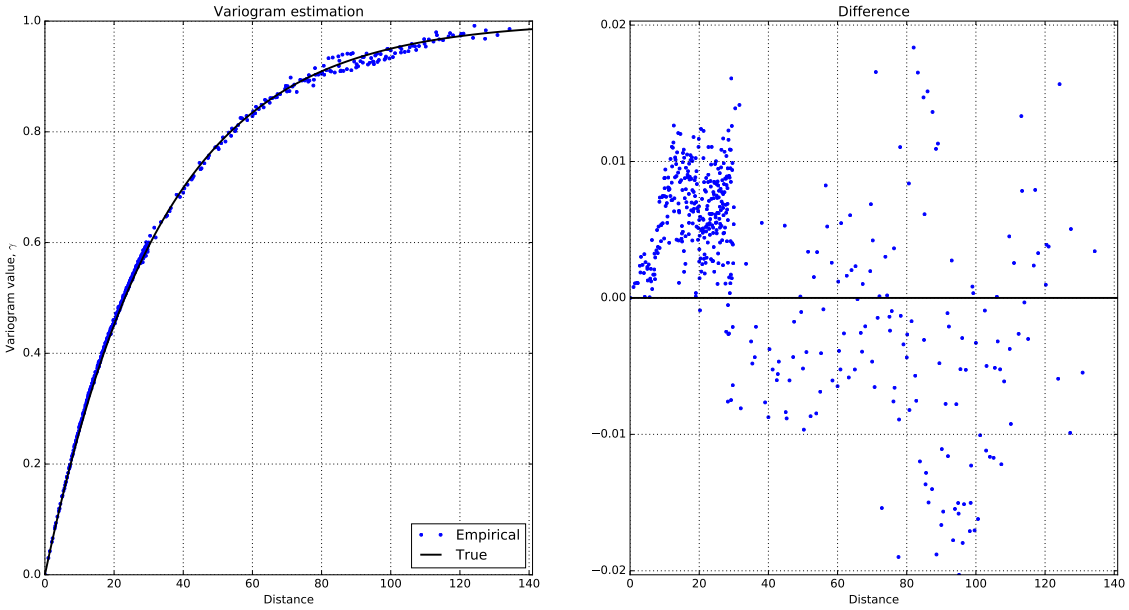


Figure D.1. Variogram estimation for a simple case. Parameters given in Table D.1.

E Correlation Function Error vs Correlation Range

This appendix contains figures showing the Error (Eq. C.4) for the correlation functions in Table A.1. The figures shows the Error as a function of correlation range relative to the simulation grid length.

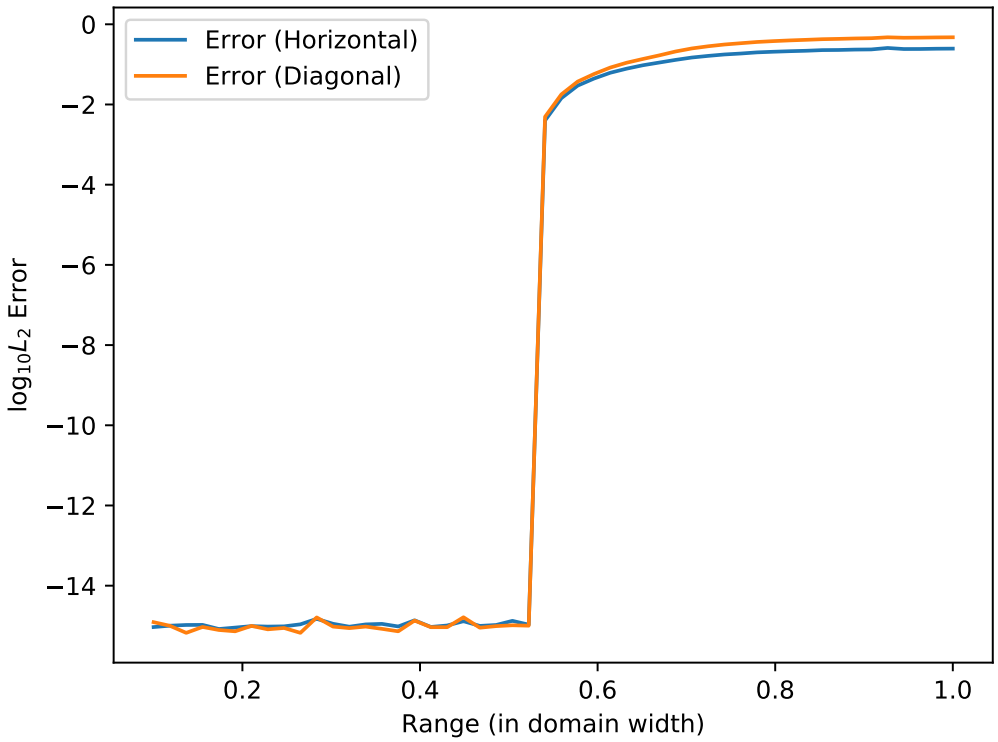


Figure E.1. Calculated overall correlation error for the spherical variogram as function of range compared with domain width ($N = 64$). The error is given in log-scale.

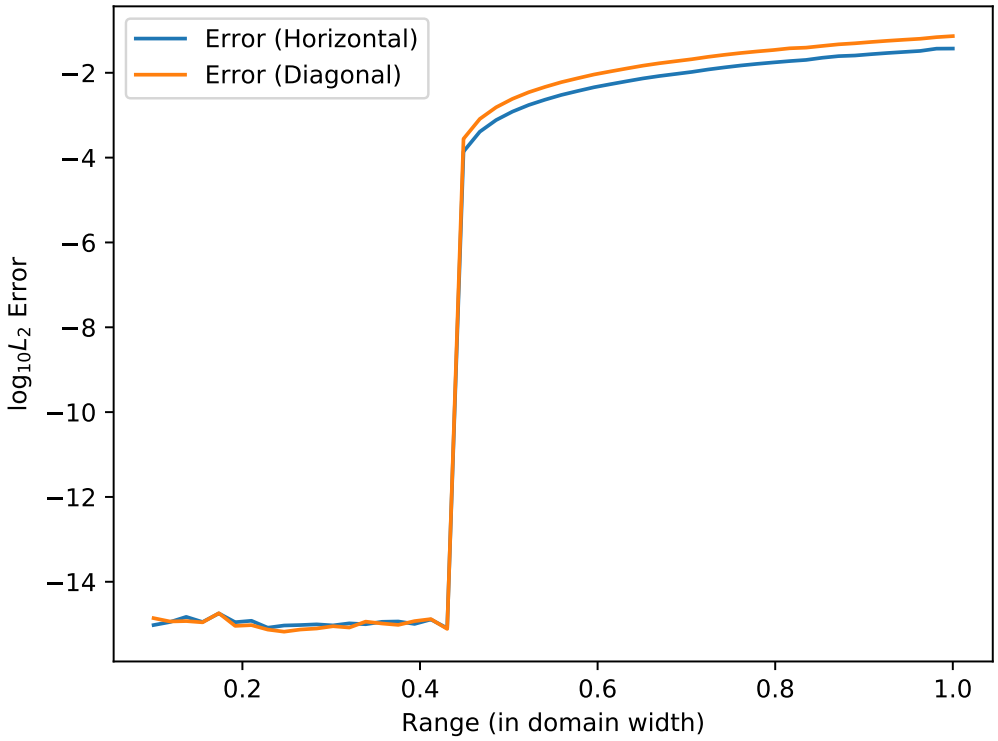


Figure E.2. Calculated overall correlation error for the exponential variogram as function of range compared with domain width ($N = 64$). The error is given in log-scale.

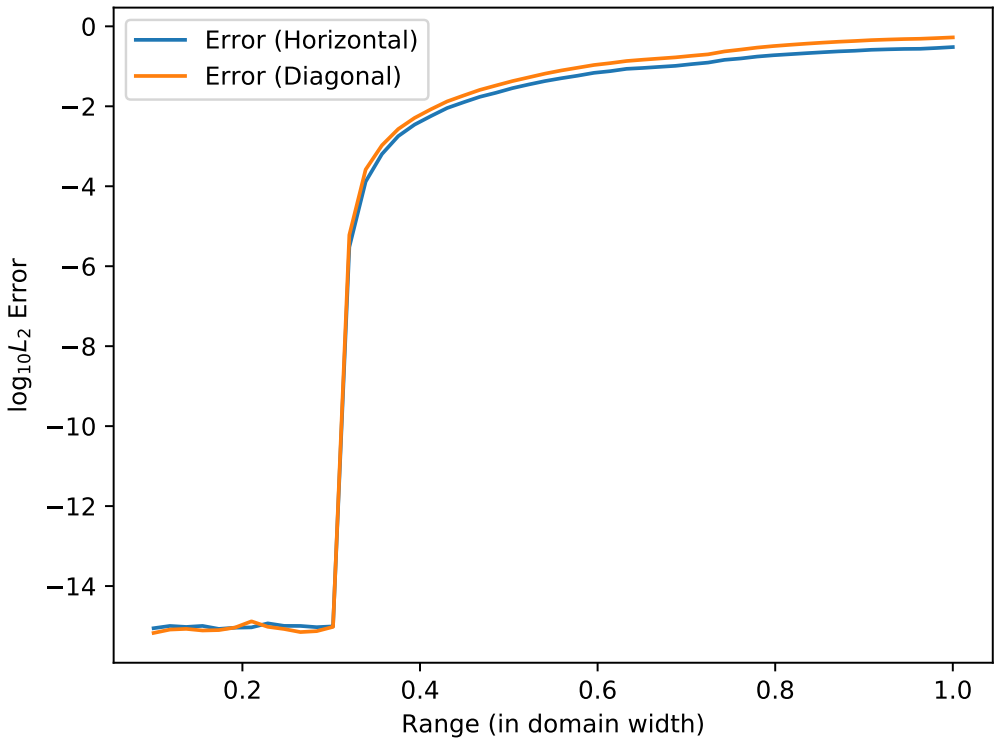


Figure E.3. Calculated overall correlation error for a general exponential variogram, $p = \frac{3}{2}$, as function of range compared with domain width ($N = 64$). The error is given in log-scale.

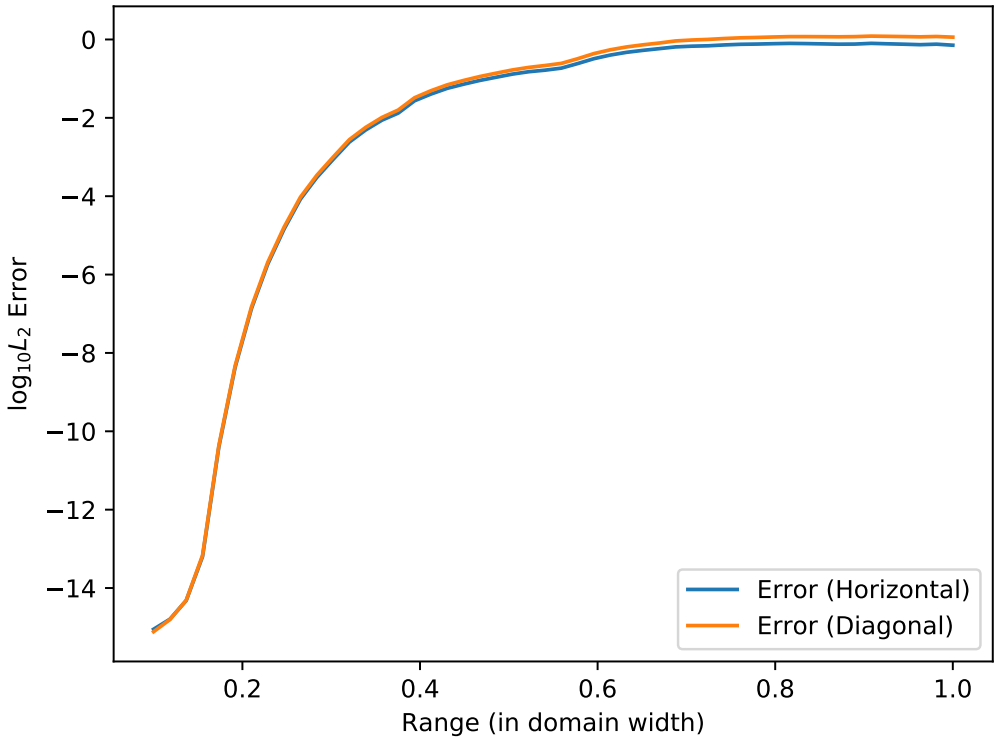


Figure E.4. Calculated overall correlation error for the Gaussian variogram as function of range compared with domain width ($N = 64$). The error is given in log-scale.

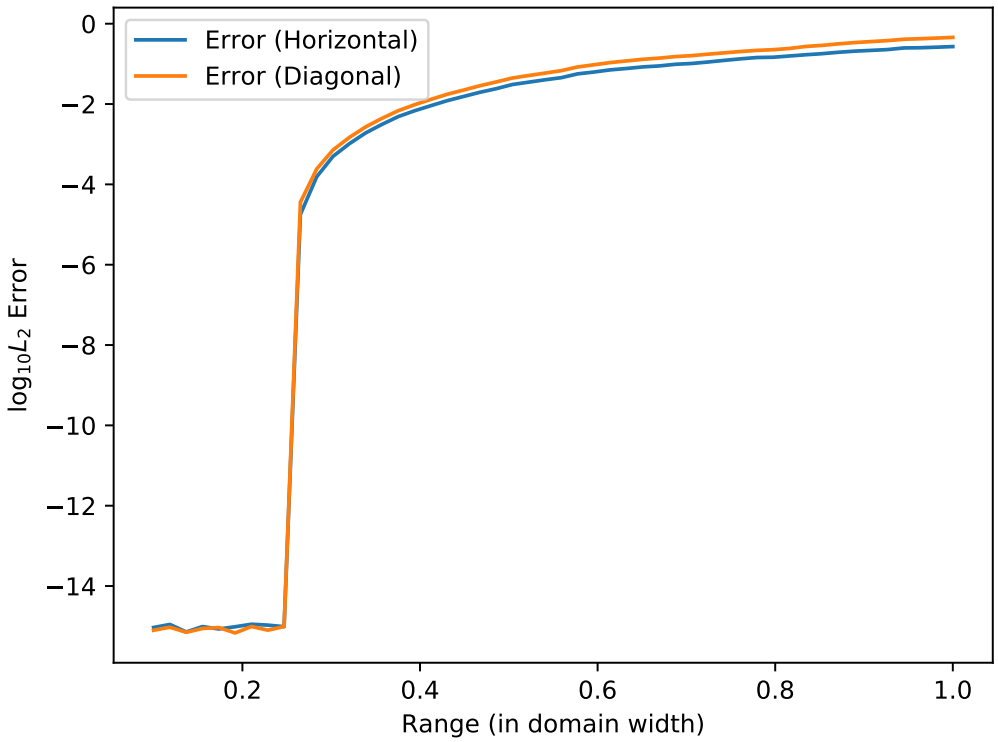


Figure E.5. Calculated overall correlation error for the Matern32 variogram as function of range compared with domain width ($N = 64$). The error is given in log-scale.

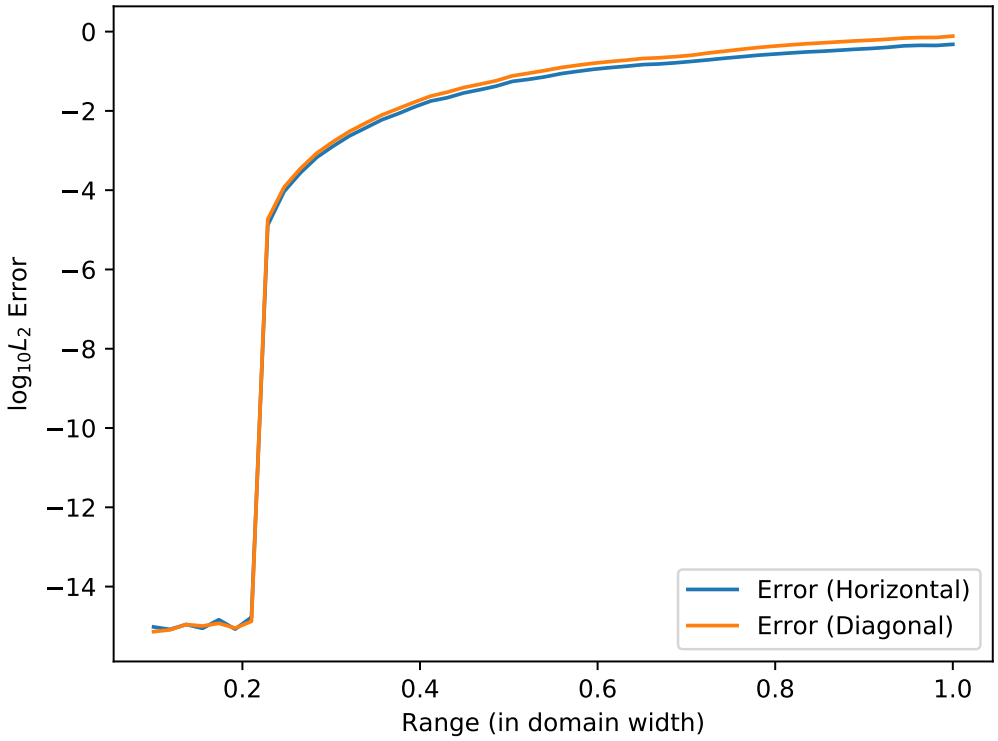


Figure E.6. Calculated overall correlation error for the Matern52 variogram as function of range compared with domain width ($N = 64$). The error is given in log-scale.

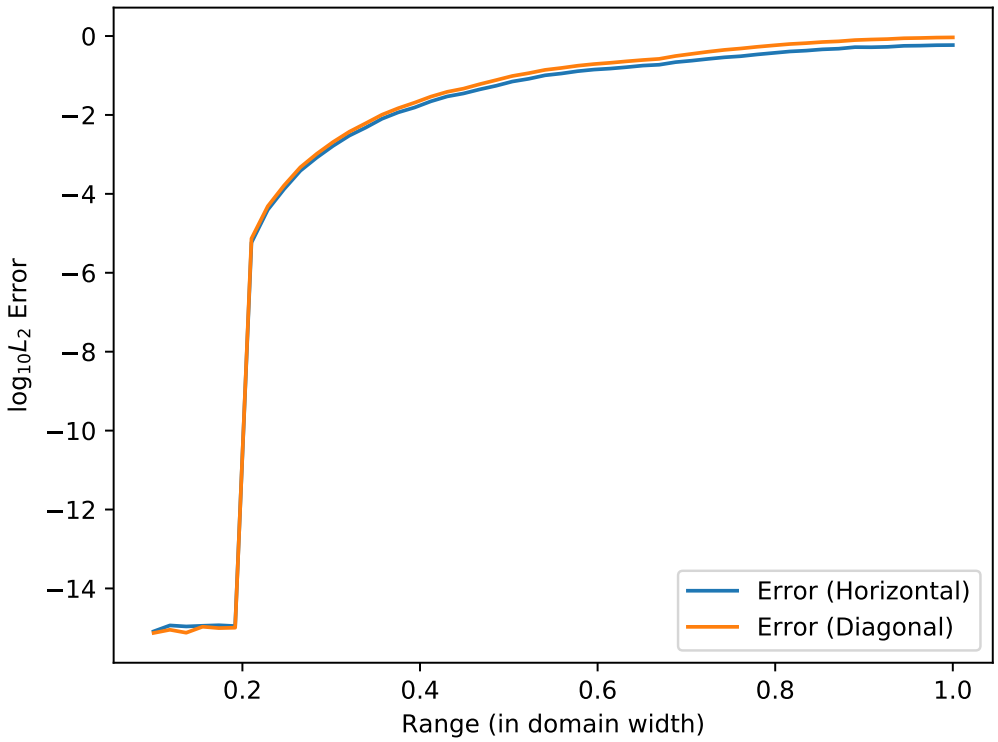


Figure E.7. Calculated overall correlation error for the Matern72 variogram as function of range compared with domain width ($N = 64$). The error is given in log-scale.

References

Chilés, J.-P. and Delfiner, P. (1997). Discrete exact simulation by the Fourier method. In Baafi, E. Y. and Schofield, N. A., editors, *Geostatistics Wollongong '96, Proceedings of the Fifth International Geostatistical Congress*, pages 258–269, Wollongong, Australia. Kluwer Academic Publishers. 6

Dietrich, C. R. and Newsam, G. N. (1993). A fast and exact method for multidimensional gaussian stochastic simulations. *Water Resources Research*, 29(8):2861–2869. Available from: http://dx.doi.org/10.1029/93WR01070. 6

Goff, J. H. and Jennings, Jr., J. W. (1999). Improvement on Fourier-based unconditional and conditional simulations for band limited fractal (von Kármán) statistical models. *Math. Geol.*, 31(6):627–649. 6

Mantoglou, A. and Wilson, J. (1982). The turning bands method for simulation of random fields using line generation by a spectral method. *Water Resources Res.*, 5:1379–1394. 6

Pardo-Igúzquiza, E. and Chica-Olmo, M. (1993). The fourier integral method: An efficient spectral method for simulation of random fields. *Math. Geol.*, 25(2):177–217. 6

Pardo-Igúzquiza, E. and Chica-Olmo, M. (1994a). SPECSIM: a program for simulating random fields by an improved spectral approach. *Computers & Geosciences*, 20(4):597–613. 6

Pardo-Igúzquiza, E. and Chica-Olmo, M. (1994b). Spectral simulation of multivariate stationary random functions using covariance fourier transforms. *Math. Geol.*, 26(3):277–299. 6

Ripley, B. D. (1987). *Stochastic Simulation*. John Wiley & Sons, New York. 6

Wood, A. T. A. and Chan, G. (1994). Simulation of stationary gaussian processes in $[0, 1]^d$. *Journal of Computational and Graphical Statistics*, 3(4):409–432. Available from: http://dx.doi.org/10.1080/10618600.1994.10474655. 6

List of Figures

List of Tables

Index