GAUSSIAN User's Manual

Boris Kozintsev

August 17, 1999

This is the documentation for the GAUSSIAN library, a collection of C routines for generating stationary Gaussian random fields over regular grids, and estimation of discrete random fields obtained by quantization (clipping) of such Gaussian fields.

The library was written in 1998-99 during my Ph.D. studies at the Mathematics Department, University of Maryland, where I was fortunate to have Professor Benjamin Kedem as an advisor. I would like to extend my most deep gratitude to him. I also appreciate the support from the NASA Grant NAG52783.

For more information on the underlying theory and algorithms see the paper “Generation of ‘Similar’ Images From a Given Discrete Image” by B. Kozintsev and B. Kedem. See also the ISR Technical Report Ph.D. 99-3, `http://www.isr.umd.edu/TechReports/ISR/1999/PhD_99-3/PhD_99-3.phtml`.

For an example of some of the GAUSSIAN library capabilities, see the interactive CGI script for visualization of clipped Gaussian random fields at `http://www.math.umd.edu/~bak/gaussian/generate.cgi`.

Contents

Chapter 1

Mathematical Definitions

1.1 Stationary Gaussian fields

First we define a *regular grid.* A regular grid $\mathcal{S} \subset \mathcal{R}^2$ is a set of points $\mathcal{S} = \{\mathbf{s}_{ij}\}$ such that for fixed horizontal and vertical steps x and y,

$$\begin{aligned} &\mathbf{s}_{ij} = \mathbf{s}_{00} + iy\mathbf{v}_2 + ix\mathbf{v}_1, \quad i = 0, \ldots, n_1 - 1, \quad j = 0, \ldots, n_2 - 1, \\ &\text{where} \quad \mathbf{v}_1 = (1, 0) \text{ and } \mathbf{v}_2 = (0, 1). \end{aligned}$$

Next we define a *stationary* zero-mean *Gaussian random field* $\mathbf{Z}$ over $\mathcal{S}$ with covariance function

$$r : \mathcal{R}^2 \to \mathcal{R}.$$

Having such a field means that for any finite set of points

$$\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_n \in \mathcal{S}$$

we have random variables

$$Z(\mathbf{s}_1), Z(\mathbf{s}_2), \ldots, Z(\mathbf{s}_n),$$

which are jointly normal with mean zero and covariances

$$\mathrm{Cov}(Z(\mathbf{s}_i), Z(\mathbf{s}_j)) = r(\mathbf{s}_i - \mathbf{s}_j).$$

1.2 Covariance Functions

The GAUSSIAN library has the following four built-in covariance functions.

Exponential correlation:

$$r_{\boldsymbol{\theta}}(l) = \theta_1^{l^{\theta_2}},$$

where $\theta_1 \in (0, 1)$ and $\theta_2 \in (0, 2]$.

Matérn correlation:

$$r_{\boldsymbol{\theta}}(l) = \begin{cases} \dfrac{1}{2^{\theta_2 - 1}\Gamma(\theta_2)} \left(\dfrac{l}{\theta_1}\right)^{\theta_2} \mathcal{K}_{\theta_2}\left(\dfrac{l}{\theta_1}\right), & l \neq 0, \\ 1, & l = 0, \end{cases}$$

where $\theta_1 > 0$, $\theta_2 > 0$ and $\mathcal{K}_{\theta_2}$ is a modified Bessel function of the third kind of order θ_2.

Rational quadratic correlation:

$$r_{\boldsymbol{\theta}}(l) = \left(1 + \frac{l^2}{\theta_1^2}\right)^{-\theta_2},$$

where $\theta_1 > 0$ and $\theta_2 > 0$.

Spherical correlation:

$$r_{\theta}(l) = \begin{cases} 1 - \dfrac{3}{2}\left(\dfrac{l}{\theta}\right) + \dfrac{1}{2}\left(\dfrac{l}{\theta}\right)^3, & l \leq \theta, \\ 0, & \text{otherwise}, \end{cases}$$

where $\theta > 0$.

Note that these functions depend on scalar l, the distance between the points at which the covariance is computed, which means that the resulting fields will be *isotropic*. Note also that here $r(0) = 1$. These two properties, although present, are not used in any special way by the library. Therefore, if you want to add your own covariance function, neither of them has to hold. You may add a non-isotropic covariance function having an arbitrary positive value at zero.

1.3 Clipping

Now we define discrete random fields and clipping. We say that we have a *discrete random field* $\mathbf{X}$ over $\mathcal{S}$ if at each point $\mathbf{s} \in \mathcal{S}$ there is a discrete random variable $X(\mathbf{s})$ which takes values $0, 1, 2, \ldots, k$.

In the GAUSSIAN library, discrete fields are modeled as clipped Gaussian fields. That is, we assume that for a vector of thresholds $\mathbf{c} = (c_1, c_2, \ldots, c_k)$, the field $\mathbf{X}$ is a quantization of $\mathbf{Z}$ at levels $\mathbf{c}$. At each point $\mathbf{s} \in \mathcal{S}$,

$$\begin{aligned} &X(\mathbf{s}) = 0 \text{ whenever } Z(\mathbf{s}) \leq c_1, \\ &X(\mathbf{s}) = j \text{ whenever } c_j < Z(\mathbf{s}) \leq c_{j+1}, \quad j = 1, \ldots, k-1, \\ &X(\mathbf{s}) = k \text{ whenever } Z(\mathbf{s}) > c_k. \end{aligned}$$

Chapter 2

Installation

2.1 FFTW Library

The computations in the GAUSSIAN library are based on the fast Fourier transform, which is performed using the FFTW library by M. Frigo and S. G. Johnson. The name "FFTW" is an acronym for "Fastest Fourier Transform in the West." FFTW is free software, released under the GNU General Public License (GPL). FFTW is available for download at `http://www.fftw.org/`. If you want to install and use GAUSSIAN, you also have to install FFTW.

To install FFTW, follow the instructions on the FFTW website. In a nutshell, you must download the file `fftw-2.1.2.tar.gz` (the exact file name depends upon the current version). Then you have to unpack it by typing

```
$ gunzip fftw-2.1.2.tar.gz
$ tar -xvf fftw-2.1.2.tar
```

This creates a directory `fftw-2.1.2` containing the files necessary to build the library. Now type

```
$ cd fftw-2.1.2
$ ./configure
$ make
```

This compiles the library. Note that when building the library under Solaris, the correct C compiler must be used – see FFTW FAQ at `http://www.fftw.org/faq/section2.html#solarisSucks`.

If the compilation is successful, among other things that happen, the following two files are created:

```
~/fftw-2.1.2/fftw/fftw.h
~/fftw-2.1.2/fftw/.libs/libfftw.a
```

You can now copy these files to some convenient location by typing, for example,

```
$ mkdir ~/FF
$ cp ~/fftw-2.1.2/fftw/fftw.h ~/FF
$ cp ~/fftw-2.1.2/fftw/.libs/libfftw.a ~/FF
```

and then delete the `fftw-2.1.2` directory tree.

In case of problems with building the FFTW library, one could try runnning

```
$ make distclean
$ ./configure --with-gcc
$ make
```

2.2 GAUSSIAN Library

Once you installed FFTW, you should download and build GAUSSIAN. Download the file `gaussian.tar.gz`, and type the following commands:

```
$ gunzip gaussian.tar.gz
$ tar -xvf gaussian.tar
$ cd GAUSSIAN
$ make FFTW_LIB=/home6/bak/FF FFTW_H=/home6/bak/FF
```

In the last command, you should change the italic text to match your FFTW installation. `FFTW_LIB` and `FFTW_H` are the directories where the files `libfftw.a` and `fftw.h`, respectively, are.

This will compile the library and the demonstration programs. You will have the directories

`LIB` with the GAUSSIAN library and its source;

`DEMOS` with the demonstration programs;

`DOC` with this manual;

`CGI` with the CGI script from `http://www.math.umd.edu/~bak/gaussian/generate.cgi`.

In case of problems one could try running

```
$ make clean
$ make CC=gcc FFTW_LIB=/home6/bak/FF FFTW_H=/home6/bak/FF
```

On DEC Alpha, to get the correct results one sometimes has to run

```
$ make clean
$ make CC="cc -std" FFTW_LIB=/home6/bak/FF FFTW_H=/home6/bak/FF
```

2.3 Linking Your Programs Against GAUSSIAN

The program that uses functions from the GAUSSIAN library must `#include "gaussian.h"` and `"fftw.h"`, and should be linked against `gaussian.a`, `libfftw.a`, and `libm.a`, in this order. Suppose the following code is stored in `useless.c`:

```
#include <stdio.h>
#include "fftw.h"
#include "gaussian.h"
int main (int argc, char **argv) {
   ghandle handle;
   handle = create_ghandle (16, 16, FFTW_ESTIMATE);
   if (handle) {
        printf ("Success!\n");
        destroy_ghandle (handle);
   }
   else printf ("Not enough memory.\n");
   return 0;
}
```

To compile it, create the following `Makefile`:

```
FFTW_H       = directory with fftw.h
FFTW_LIB     = directory with libfftw.a
GAUSSIAN_H   = directory with gaussian.h
GAUSSIAN_LIB = directory with gaussian.a
CFLAGS       = -I${FFTW_H} -I${GAUSSIAN_H} -O
LIBS = ${GAUSSIAN_LIB}/gaussian.a ${FFTW_LIB}/libfftw.a -lm
.c:
	${CC} ${CFLAGS} ${LDFLAGS} $< ${LIBS} -o $@
```

Note that the last line begins with the TAB character.

Now you can type `make useless` to compile the program.

Chapter 3

Running the Demonstration Programs

There are four demonstration programs that come with the GAUSSIAN library: `field_demo`, `em_demo`, `z_given_x_demo`, and `num2ppm_demo`.

The program `field_demo` illustrates generation of Gaussian and clipped Gaussian random fields. Its functionality is identical to the CGI program at `http://www. math.umd.edu/~bak/gaussian/generate.cgi`, except that its memory is not limited by the web server and hence it is able to generate larger fields.

The program `em_demo` illustrates estimation of the covariance parameters in the unobserved Gaussian field from its observed clipped version by the EM algorithm. The input for the program is a discrete random field.

The program `z_given_x_demo` also takes discrete random field as an input and generates a Gaussian field that would produce the given discrete field after clipping at the specified levels.

The program `num2ppm_demo` performs a conversion from the numerical format to the PPM (picture) format. The PPM images can be viewed by the shareware XV program available at `http://www.trilon.com/xv/`, and by other programs.

3.1 File Formats

3.1.1 PPM Image File Format

The GAUSSIAN library stores images as PPM (`P6`) files. Their format is as follows:

- A "magic number" for identifying the file type. A ppm file's magic number is the two characters `P6`.
- Whitespace (blanks, TABs, CRs, LFs).

- A width, formatted as ASCII characters in decimal.
- Whitespace.
- A height, again in ASCII decimal.
- Whitespace.
- The maximum color-component value, again in ASCII decimal (less or equal to 255).
- Whitespace.
- Width × height pixels, starting at the top-left corner of the pixmap, proceeding in normal English reading order. Each pixel takes three bytes, each between 0 and the specified maximum value. The three values for each pixel represent red, green, and blue, respectively; a value of 0 means that color is off, and the maximum value means that color is maxxed out.

For additional information, see `http://hegel.ittc.ukans.edu/topics/linux/man-pages/man5/ppm.5.html`.

3.1.2 Numeric File Formats

In addition to creating PPM (image) files, the GAUSSIAN library can output fields in the numeric form, in one of the two formats.

The first format, referred to as `'3'`, includes a number of comment lines starting with `#`, followed by lines of X Y Z values, like in this example of a 2×3 field:

```
# Gaussian 2x3 r.f. with Spherical (5.000000) correlation
# and mean 0.
   0    0       -0.246575
   0    1        0.237055
   0    2       -0.322522
   1    0       -0.593353
   1    1        0.027527
   1    2        0.129377
```

The second format, referred to as `'Z'`, stores Z values only, like in this version of the same field:

```
# Gaussian 2x3 r.f. with Spherical (5.000000) correlation
# and mean 0.
    -0.246575  0.237055  -0.322522  -0.593353
     0.027527  0.129377
```

You should choose one or another format depending on the software to which you intend to feed the resulting file.

3.1.3 Initialization Files

The programs `field_demo`, `em_demo`, and `z_given_x_demo` obtain their input from the initialization files. These are plain text files which can be edited with any text editor. The lines starting with the # character are comments. All the other lines have the form

```
parameter = value
```

or

```
parameter : value
```

You should edit the values to the right of the semicolons or equality signs, and leave the left hand sides unchanged. The best way to create a new initialization file is to copy one of the files distributed with the library and make the necessary changes. The meaning of the parameters for each program is explained in the next three sections.

Note that the library provides functions for reading such files, so you can use a similar technique in your own projects.

3.2 Unconditional Field Generation (`field_demo`)

The `field_demo` program takes its input from a text initialization file. To run the program, type

```
$ field_demo field.ini
```

In the command above, `field.ini` is the sample initialization file provided with the program. Its entries have the following meaning.

-
  ```
  wn1 = 32
  wn2 = 32
  ```

  These two lines specify the height and width of the grid for the desired random field.

-
  ```
  n1 = 128
  n2 = 128
  ```

  The values `n1` and `n2` specify the height and the width of the grid that will actually be used for random field generation, and should not be smaller than `wn1` and `wn2` respectively. You may have to specify high values for `n1` and `n2` to achieve positive embedding of the covariance matrix. Generally, `n1` and `n2` should be large enough so that the covariance between the field values on the opposite sides of the grid is almost zero. Upon successful generation, a `wn1` × `wn2` portion will be taken from the upper left corner of the larger `n1` × `n2` field. Note that high values of `n1` and `n2` make the program use more memory and work slower.

-
  ```
  covtype = s
  theta1 = 10
  theta2 = 1
  ```

  These lines define the covariance structure of the field. The possible values for `covtype` are

  `m` for Matérn correlation,

  `e` for exponential correlation,

  `r` for rational quatdratic correlation, and

  `s` for spherical correlation.

  See Section 1.2 for definitions of these covariance functions. For spherical correlation, the parameter `theta2` is ignored.

-
  ```
  nlevels=2
  c1=0.33
  c2=0.33
  ```

  These parameters determine how the generated Gaussian field will be clipped to obtain a discrete field. If you are interested in the Gaussian field only and not in the discrete field, they are not important to you.

  The value of `nlevels` is the number of clipping thresholds (equals k in the definition of clipping in Section 1.3). One threshold produces a binary image, two thresholds produce a three-color image, and so on. To produce a three color image, like in this example, the values of the Gaussian field below the first threshold will be assigned the first color in the list of colors (see below), the values between the thresholds will be assigned the second color, and the remaining values which are above the second threshold will be assigned the third color. The thresholds will be chosen depending on the values of `c1` and `c2` that specify the desired proportions of the first and second color respectively. Together they define the proportion of the third color (`1-c1-c2`). In our example, the resulting three color image will have equal number of pixels of each color.

  In general, for `nlevels=`*j* you must add more lines so that you have entries `c1`, `c2`,..., `c`*j*, the sum of which is less than 1.

-
  ```
  numeric file format   : 3
  ```

  This line determines the format in which the Gaussian and discrete fields will be output (see Section 3.1). The possible values are

  `3` for X Y Z values, and

  `Z` for Z values only.

If you only want image files (PPM), this parameter is not important.

- `numerical file for a Gaussian field : a.txt`

  This specifies the name of the file into which the Gaussian field will be output in the numeric form, according to the above `numeric file format` parameter. If left blank, the file will not be created.

- `numerical file for a discrete field : ad.txt`

  This specifies the name of the file into which the discrete (clipped Gaussian) field will be output in the numeric form, according to the above `numeric file format` parameter. If left blank, the file will not be created.

- `ppm file for a Gaussian field : a.ppm`

  This specifies the name of the image file (PPM format) into which the Gaussian field will be output in the gray levels form. If left blank, the file will not be created.

- `ppm file for a discrete field : ad.ppm`

  This specifies the name of the image file (PPM format) into which the discrete (clipped Gaussian) field will be output in colors specified below. If left blank, the file will not be created.

-
  ```
  red0=101
  green0=178
  blue0=232
  ..........
  red13=201
  green13=107
  blue13=6
  ```

  The rest of the file defines 14 colors for use in the discrete PPM file. Each color is determined by its red, green, and blue levels which are integers between 0 and 255.

3.3 Estimation (em_demo)

The em_demo program takes its input from a text initialization file. To run the program, type

```
$ em_demo em.ini
```

In the command above, `em.ini` is the sample initialization file provided with the program. Its entries have the following meaning.

-
  ```
  Discrete data file name: ad.txt
  Discrete data file format: 3
  ```

  These lines determine the name and the format of the file with the given discrete field. The possible values for format are

  `3` for X Y Z values,

  `Z` for Z values only, and

  `p` for PPM file.

-
  ```
  n1 = 32
  n2 = 32
  ```

  These lines determine the height and width of the grid. They will be ignored if the input file is in the PPM format, since this format includes the image dimensions.

-
  ```
  covtype = s
  theta1 = 1
  theta2 = 1
  ```

  These lines define the covariance structure of the unobserved Gaussian field, and the starting values for $\boldsymbol{\theta}$ in the EM algorithm. The possible values for `covtype` are

  `m` for Matérn correlation,

  `e` for exponential correlation,

  `r` for rational quatdratic correlation, and

  `s` for spherical correlation.

  See Section 1.2 for definitions of these correlation functions. For spherical correlation, the parameter `theta2` is ignored.

-
  ```
  nlevels=2
  threshold1=-0.284290
  threshold2= 0.486082
  ```

  These parameters determine how the unobserved Gaussian field was clipped to obtain the given discrete field. The value of `nlevels` is the number of clipping thresholds (equals k in the definition of clipping in Section 1.3). One threshold produces a binary image, two thresholds produce a three-color image, and so on. To produce a three color image, like in this example, the values of the Gaussian field below `-0.284290` have been assigned the first code (see the next entry), the values between `-0.284290` and `0.486082` have been assigned the second code, and the remaining values which are above `0.486082` have been assigned the third code.

In general, for `nlevels=`*j* you must add more lines so that you have entries `threshold1`, `threshold2`,..., `threshold`*j* with increasing values.

-
  ```
  code1=0
  code2=1
  ```

  These lines determine how the discrete field is coded. The entry `code`*j* is the value in the discrete field that resulted from the value in the Gaussian field between the (*j*`-1`)-st and *j*-th threshold, or between $-\infty$ and `threshold1` for *j*=1.

  For numeric files, the codes are Z-values. For PPM files, the codes are the levels of red (for simplicity it is assumed that all the colors used in the discrete image have distinct red levels).

  In general, for `nlevels=`*j* you must add more lines so that you have entries `code1`, `code2`,..., `code`*j*. Note that the number of codes is less than the number of colors in the image by one. This is because all the codes but one together with the given image define the remaining code implicitly.

-
  ```
  em_steps=300
  gibbs_iterations=10
  ```

  These lines determine the number of EM algorithm steps to perform, and the number of Gibbs iterations to perform at each step.

-
  ```
  optimizaion=0
  ```

  This line is used for the spherical covariance only and determines whether to use FMINBR or HOOKE optimization in one dimension. The value of `0` specifies to use the FMINBR procedure (recommended).

-
  ```
  log file name: em.log
  ```

  This line gives the name of the file to which the EM algorithm trajectory will be written to. You should plot the values in this file to assess the convergence.

3.4 Conditional Field Generation (`z_given_x_demo`)

The `z_given_x_demo` program generates a Gaussian field **Z** given the discrete field **X**, the covariance structure, and the clipping levels, such that **Z** would result in **X** after clipping. The program takes its input from a text initialization file. To run the program, type

```
$ z_given_x_demo z_given_x.ini
```

In the command above, `z_given_x.ini` is the sample initialization file provided with the program. Its entries have the following meaning.

-
  ```
  Discrete data file name: ad.txt
  Discrete data file format: 3
  ```

  These lines determine the name and format of the file with the given discrete field. The possible values for format are

  `3` for X Y Z values,

  `Z` for Z values only, and

  `p` for PPM file.

-
  ```
  n1 = 32
  n2 = 32
  ```

  These lines determine the height and width of the grid. They will be ignored if the input file is in the PPM format, since this format includes the image dimensions.

-
  ```
  covtype = s
  theta1 = 10
  theta2 = 1
  ```

  These lines define the covariance structure of the unobserved Gaussian field, and the starting values for $\boldsymbol{\theta}$ in the EM algorithm. The possible values for `covtype` are

  `m` for Matérn correlation,

  `e` for exponential correlation,

  `r` for rational quatdratic correlation, and

  `s` for spherical correlation.

  See Section 1.2 for definitions of these correlation functions. For spherical correlation, the parameter `theta2` is ignored.

-
  ```
  nlevels=2
  threshold1=-0.284290
  threshold2= 0.486082
  ```

  These parameters determine how the unobserved Gaussian field was clipped to obtain the given discrete field. The value of `nlevels` is the number of clipping thresholds (equals k in the definition of clipping in Section 1.3).

One threshold produces a binary image, two thresholds produce a three-color image, and so on. To produce a three color image, like in this example, the values of the Gaussian field below `-0.284290` have been assigned the first code (see the next entry), the values between `-0.284290` and `0.486082` have been assigned the second code, and the remaining values which are above `0.486082` have been assigned the third code.

In general, for `nlevels=`*j* you must add more lines so that you have entries `threshold1`, `threshold2`,..., `threshold`*j* with increasing values.

-
  ```
  code1=0
  code2=1
  ```

  These lines determine how the discrete field is coded. The entry `code`*j* is the value in the discrete field that resulted from the value in the Gaussian field between the (*j*`-1`)-st and *j*-th threshold, or between $-\infty$ and `threshold1` for *j*=1.

  For numeric files, the codes are Z-values. For PPM files, the codes are the levels of red (for simplicity it is assumed that all the colors used in the discrete image have distinct red levels).

  In general, for `nlevels=`*j* you must add more lines so that you have entries `code1`, `code2`,..., `code`*j*. Note that the number of codes is less than the number of colors in the image by one. This is because all the codes but one together with the given image define the remaining code implicitly.

-
  ```
  gibbs_iterations=10
  ```

  This line determines the number of Gibbs iterations to perform.

-
  ```
  numerical file for a Gaussian field : a.txt
  ```

  This specifies the name of the file into which the Gaussian field will be output in the numeric form. If left blank, the file will not be created.

-
  ```
  numerical file for a discrete field : a1.txt
  ```

  This specifies the name of the file into which the discrete (clipped Gaussian) field will be output in the numeric form, according to the above `numeric file format` parameter. This discrete field is provided for testing purposes. You may compare it to the input discrete field to verify that they are identical. If the parameter value is left blank, the file will not be created.

-
  ```
  ppm file for a Gaussian field : a.ppm
  ```

This specifies the name of the image file (PPM format) into which the Gaussian field will be output in the gray levels form. If left blank, the file will not be created.

-
  ```
  ppm file for a discrete field : ad1.ppm
  ```

  This specifies the name of the image file (PPM format) into which the discrete (clipped Gaussian) field will be output in colors specified below. This discrete field is provided for testing purposes. You may compare it to the input discrete field to verify that they are identical. If the parameter value is left blank, the file will not be created.

-
  ```
    red0=101
    green0=178
    blue0=232
    ..........
    red13=201
    green13=107
    blue13=6
  ```

  The rest of the file defines 14 colors for use in the discrete PPM file. Each color is determined by its red, green, and blue levels which are integers between 0 and 255.

3.5 Converting Numbers to Pictures (num2ppm_demo)

The num2ppm_demo performs a conversion from one of the two numerical formats to the PPM format. This program does not use initialization files and takes input interactively from the keyboard. Suppose that you want to convert the file a.txt which has a 128 × 128 field stored in the X-Y-Z form. You would enter

```
$ num2ppm_demo
Input file name [none]: a.txt
File format (<3> for X Y Z values, <Z> for Z values alone) [3]: 3
Output file name [a.txt.ppm]: a.ppm
Grid height [256]: 128
Grid height [128]: 128
Created file 'a.ppm'
$
```

By this point, the desired PPM file is created. Note that what you type is typeset in italics; the values in the brackets are the default answers.

Chapter 4

Programmer's Guide

4.1 General information

The GAUSSIAN library consists of the following files:

- rv.c – functions for generating univariate random variables;
- fields.c – functions for processing random fields;
- formats.c – functions for reading and writing files in different formats;
- z.c – functions for evaluating the Gaussian distribution function and its inverse;
- fminbr.c – one-dimensional optimization;
- hooke.c – multidimensional optimization;
- rkbesl.c – Bessel function evaluation;
- gaussian.h – function declarations.

4.1.1 How The Random Fields Are Stored

The GAUSSIAN library utilizes FFTW for computations. Therefore, we follow the FFTW conventions. By including `<fftw.h>`, you will have access to the following definitions:

```
typedef double FFTW_REAL;
typedef struct {
    FFTW_REAL re, im;
} FFTW_COMPLEX;
#define c_re(c) ((c).re)
#define c_im(c) ((c).im)
```

The rectangular fields are stored as one-dimensional arrays of `FFTW_COMPLEX` numbers. Each array therefore can hold two fields – one in its real and one in its imaginary part. The storage order within an array is row major. In particular, to allocate memory for an `nrows` × `ncols` field, we use

```
FFTW_COMPLEX *f;
f = (FFTW_COMPLEX *) malloc (nrows * ncols
                                    * sizeof (FFTW_COMPLEX));
```

The (10, 15)-th element of the field stored in the real part of `*f` is then given by `c_re(f[15 + 10*ncols])`.

4.1.2 Random Number Generation

All the GAUSSIAN random number algorithms are based on the standard `drand48()` function from `stdlib`. If you do not initialize it, you will get the same random number sequence each time you run the program. This could be useful for debugging and experiments. However, if you want a new random sequence for each program run, insert the following call at the beginning of your code:

```
srand48(time(NULL));
```

4.2 Library Functions

4.2.1 Univariate Random Variables (`rv.c`)

There are five functions in the GAUSSIAN library for univariate random number generation.

- `double runif (double rmin, double rmax);`

  `runif` returns a realization of a random variable uniformly distributed between `rmin` and `rmax`.

- `int discrv (int v[], double p[], int n);`

  `discrv` returns a realization of a discrete random variable taking `n` different values `v[]` with probabilities `p[]`.

- `double n01 ();`

  `n01` returns a realization of a $N(0,1)$ random variable.

- `double normal (double mu, double sigma2);`

  `normal` returns a realization of a $N(\mu,\sigma^2)$ random variable.

-
  ```
  double trunc_normal (double mu, double sigma2,
                       double a, double b);
  ```

  `trunc_normal` returns a realization of $Z \sim N(\mu,\sigma^2)$ conditional on $Z \in [a,b)$.

4.2.2 File I/O (`formats.c`)

Four file formats are used in the GAUSSIAN library: two file formats for storing random fields as numerical data (as X Y Z triples and as Z values only), one image format (PPM), and a format for initialization files. There are 10 functions to work with these formats.

-
  ```
  int write_clipped_ppm (FFTW_COMPLEX *inpic,
                         int which,
                         int nlevels,
                         double *c,
                         int *r,
                         int *g,
                         int *b,
                         char *fname,
                         int rows,
                         int cols);
  ```

  This function is used to clip a Gaussian field and write it into an image file in the PPM format. The parameters have the following meaning:

  - ◇ `inpic` – a `rows` ×`cols` array of `FFTW_COMPLEX` where the Gaussian field is stored;
  - ◇ `which` – specifies whether to use the real (`which`==0) or imaginary (`which`==1) part of `inpic`;
  - ◇ `nlevels` – the number of clipping thresholds in the array `c` (equals $k - 1$ in the definition of clipping in Section 1.2). One threshold produces a binary image, two thresholds produce three-color image, and so on;
  - ◇ `c` – the array of the clipping thresholds, having `nlevels` elements;
  - ◇ `r`, `g`, `b` – levels of red, green, and blue defining the colors to use in the PPM file. Each array has (`nlevels`+1) element;
  - ◇ `fname` – the name of the PPM file to create;
  - ◇ `rows`, `cols` – the dimensions of the field.

  The function returns `0` for success, `-1` if the file cannot be written.

-
  ```
  int write_clipped_numeric (FFTW_COMPLEX *inpic,
                             int which,
                             char format,
                             int nlevels,
                             double *c,
                             char *fname,
                             int rows,
                             int cols,
                             char *comment);
  ```

This function is used to clip a Gaussian field and write it into a file in one of the two numeric file formats. The parameters have the following meaning:

- ◇ `inpic` – a `rows` ×`cols` array of `FFTW_COMPLEX` where the Gaussian field is stored;
- ◇ `which` – specifies whether to use the real (`which`==0) or imaginary (`which`==1) part of `inpic`;
- ◇ `format` – specifies the numeric file format. The value of `'Z'` outputs the field values only row-wise (i.e. in the English reading order). The value of `'3'` outputs the field as `X Y Z` triples;
- ◇ `nlevels` – the number of clipping thresholds in the array `c` (equals $k-1$ in the definition of clipping in Section 1.2). One threshold produces a binary image, two thresholds produce three-color image, and so on;
- ◇ `c` – the array of the clipping thresholds, having `nlevels` elements;
- ◇ `fname` – the name of the file to create;
- ◇ `rows`, `cols` – the dimensions of the field;
- ◇ `comment` – the comment line to put at the beginning of the file.

The function returns `0` for success, `-1` if the file cannot be written.

-
```
int write_numeric (FFTW_COMPLEX *inpic,
                   int which,
                   char format,
                   char *fname,
                   int rows,
                   int cols,
                   char *comment);
```

This function is used to write a field into a file in one of the two numeric file formats. The parameters have the following meaning:

- ◇ `inpic` – a `rows` ×`cols` array of `FFTW_COMPLEX` where the Gaussian field is stored;
- ◇ `which` – specifies whether to use the real (`which`==0) or imaginary (`which`==1) part of `inpic`;
- ◇ `format` – specifies the numeric file format. The value of `'Z'` outputs the field values only row-wise (i.e. in the English reading order). The value of `'3'` outputs the field as `X Y Z` triples;
- ◇ `fname` – the name of the file to create;
- ◇ `rows`, `cols` – the dimensions of the field;
- ◇ `comment` – the comment line to put at the beginning of the file.

The function returns `0` for success, `-1` if the file cannot be written.

-
  ```
  int write_ppm(FFTW_COMPLEX *m,
                int which,
                char *fname,
                int rows,
                int cols);
  ```

  This function is used to write a field into a PPM image file as grey levels. The parameters have the following meaning:

  ◇ `m` – a `rows` ×`cols` array of `FFTW_COMPLEX` where the Gaussian field is stored;

  ◇ `which` – specifies whether to use the real (`which`==0) or imaginary (`which`==1) part of `inpic`;

  ◇ `fname` – the name of the file to create;

  ◇ `rows`, `cols` – the dimensions of the field;

  ◇ `comment` – the comment line to put at the beginning of the file.

  The function returns `0` for success, `-1` if the file cannot be written.

-
  ```
  int read_numeric (FFTW_COMPLEX *m,
                    int which,
                    char format,
                    char *fname,
                    int rows,
                    int cols);
  ```

  This function is used to read a field from a data file written in one of the two numeric formats. The parameters have the following meaning:

  ◇ `m` – a `rows` ×`cols` initialized array of `FFTW_COMPLEX` where the field will be stored.

  ◇ `which` – specifies whether to use the real (`which`==0) or imaginary (`which`==1) part of `m`;

  ◇ `format` – specifies the numeric file format. The possible values are `'3'` and `'Z'` and have the same meaning as in the `write_numeric` function;

  ◇ `fname` – the name of the file to read;

  ◇ `rows`, `cols` – the dimensions of the field;

  The function returns `0` for success, `-1` if the file cannot be opened for reading.

-
  ```
  int get_constraints_ppm (char *fname,
                           int *n1,
                           int *n2,
                           int nlevels,
                           double *c,
  ```

```
                              int *r,
                              double **a,
                              double **b);
```

This function is used to collect information about a random field **Z** from its clipped version **X**. Namely, from the PPM image of **X** and the clipping thresholds we construct two fields **a** and **b** such that $\mathbf{a} \le \mathbf{Z} < \mathbf{b}$ coordinate-wise. The possible values of the components of **a** and **b** are the thresholds and the values of `-1000` and `1000` that are used as surrogate $-\infty$ and ∞. The parameters of the function have the following meaning.

- ◇ `fname` – the name of the PPM file to read;
- ◇ `n1`, `n2` – the variables into which to put the image dimensions read from the PPM file;
- ◇ `nlevels` – the number of clipping thresholds in the array `c` (equals $k - 1$ in the definition of clipping in Section 1.2). One threshold produces a binary image, two thresholds produce three-color image, and so on;
- ◇ `c` – the array of the clipping thresholds, having `nlevels` elements;
- ◇ `r` – the array with (`nlevels`+1) elements of the levels of red in the colors used in the PPM file. It is assumed that all the colors have different levels of red.
- ◇ `a`, `b` – pointer variables which will be initialized to point to the new arrays with `(*n1)*(*n2)` elements each with the fields **a** and **b**. The memory is obtained by a call to `malloc`.

The function returns `0` for success, `-1` if the file cannot be opened for reading.

-
  ```
  int get_constraints_num (char *fname,
                           char file_format,
                           int n1,
                           int n2,
                           int nlevels,
                           double *c,
                           int *codes,
                           double **a,
                           double **b);
  ```

This function is used to collect information about a random field **Z** from its clipped version **X**. Namely, from the file containing **X** in one of the two numeric formats, and from the clipping thresholds we construct two fields **a** and **b** such that $\mathbf{a} \le \mathbf{Z} < \mathbf{b}$ coordinate-wise. The possible values of the components of **a** and **b** are the thresholds and the values of `-1000` and `1000` that are used as surrogate $-\infty$ and ∞. The parameters of the function have the following meaning.

- ◇ `fname` – the name of the file to read;
- ◇ `file_format` – specifies the numeric file format. The possible values are `'3'` and `'Z'` and have the same meaning as in the `write_numeric` function;
- ◇ `n1`, `n2` – the variables specifying the image dimensions;
- ◇ `nlevels` – the number of clipping thresholds in the array `c` (equals $k-1$ in the definition of clipping in Section 1.2). One threshold produces a binary image, two thresholds produce three-color image, and so on;
- ◇ `c` – the array of the clipping thresholds, having `nlevels` elements;
- ◇ `r` – the array with (`nlevels`+1) elements of the levels of red in the colors used in the PPM file. It is assumed that all the colors have different levels of red.
- ◇ `a`, `b` – pointer variables which will be initialized to point to the new arrays with `n1*n2` elements each with the fields **a** and **b**. The memory is obtained by a call to `malloc`.

The function returns `0` for success, `-1` if the file cannot be opened for reading.

- `int getint_ini (char *fname, char *lhs);`

  This function returns the integer value from the string in the initialization file `fname` specified by `lhs`.

- `double getdouble_ini (char *fname, char *lhs);`

  This function returns the double precision value from the string in the initialization file `fname` specified by `lhs`.

- `char *getstring_ini (char *fname, char *lhs);`

  This function returns a pointer to a newly allocated string read from the line in the initialization file `fname` specified by `lhs`.

4.2.3 Random Fields (`fields.c`)

All the functions dealing with random fields use the following structure defined in `<gaussian.h>`:

```
struct ghandle_struct{
       FFTW_COMPLEX *w;
       FFTW_COMPLEX *k;
       int n1;
       int n2;
       fftwnd_plan plan;
       fftwnd_plan planb;
       int status;
};
typedef struct ghandle_struct *ghandle;
```

The structure contains field dimensions `n1` and `n2`, pointers to two arrays allocated to store the embedded versions of the field, FFTW information ("plan") required to perform the forward and backward Fourier transforms of these arrays, and the error status. To create a `ghandle`, you must provide the numbers `n1` and `n2` to the following function.

- `ghandle create_ghandle(int n1, int n2, int method);`

  ◇ `n1` and `n2` are the field's dimensions;

  ◇ `method` is the option that specifies how the handle is created. The possible values are the constants `FFTW_ESTIMATE` and `FFTW_MEASURE` defined in `<fftw.h>`. If you use `FFTW_ESTIMATE`, the `ghandle` will be created relatively fast. However, it will have sub-optimal characteristics in terms of the speed of the Fourier transform. On the other hand, if you use `FFTW_MEASURE`, it will take some time to create a `ghandle` with the best possible performance on your particular machine. Therefore, whether to use `FFTW_ESTIMATE` or `FFTW_MEASURE` depends on how many operations you intend to perform with a given `ghandle`.

    Other than performance, the `method` parameter does not affect anything.

  The function returns a pointer to the newly created structure, or `NULL` if the structure cannot be created because of insufficient memory.

The `ghandle`'s that are no longer needed are destroyed by the following function.

- `int destroy_ghandle(ghandle handle);`

  ◇ `ghandle` is a pointer to the structure to be destroyed.

  The function always returns 0.

All the covariance models in the Gaussian library are defined in the following function:

-
  ```
  double cov (int i,
              int j,
              int cols,
              double *theta,
              char covtype);
  ```

  The parameters of the function have the following meaning.

- ◇ `i` and `j` are the numbers of the grid points at which the covariance is to be computed. The points are numbered in the English reading order starting at zero in the upper left corner of the grid;
- ◇ `cols` is the number of columns in the grid;
- ◇ `theta` is the array of covariance function parameters;
- ◇ `covtype` is the covariance function type. Currently the possible values are

  `'m'` for Matérn correlation,
  `'e'` for exponential correlation,
  `'r'` for rational quatdratic correlation, and
  `'s'` for spherical correlation,
  see Section 1.2 for definitions.

The function returns the covariance between the point #`i` and point #`j`. The return value of 10 for the Matérn correlation signifies that an error occurred while evaluating the Bessel function.

If you want to add new covariance models, you should modify this function.

-
  ```
  FFTW_COMPLEX *fill_in_encompassing_matrix
            (ghandle handle, double *theta, char covtype);
  ```

  This function populates the first column of **C** – the *block circulant with circulant blocks* covariance matrix of the encompassing field.

  - ◇ `handle` is the handle returned by the `create_ghandle` function;
  - ◇ `theta` is the array of covariance function parameters;
  - ◇ `covtype` is the the covariance function type. Currently the possible values are

    `'m'` for Matérn correlation,
    `'e'` for exponential correlation,
    `'r'` for rational quatdratic correlation, and
    `'s'` for spherical correlation,
    see Section 1.2 for definitions.

  The function puts the first column of **C** into the real part of `handle->k` and returns a pointer to its first element, or `NULL` in case when the Matérn correlation cannot be computed.

-
  ```
  FFTW_COMPLEX *generate_w (ghandle handle);
  ```

  This function generates two independent realizations of the encompassing Gaussian field **W** with the given first column of **C** – the *block circulant with circulant blocks* covariance matrix. The first column is stored in the real part of `handle->k`, and is usually obtained usually by a call to `fill_in_encompassing_matrix`. The function puts the two resulting realization of **W** into real and imaginary parts of `handle->k` and returns a pointer to its first element, or `NULL` in case of failure (negative embedding).

-
  ```
  FFTW_COMPLEX *generate_w_given_x (ghandle handle,
                                    int gibbs_iterations,
                                    double *enc_a,
                                    double *enc_b,
                                    double *theta,
                                    char covtype);
  ```

  `generate_w_given_x` generates a realization of the encompassing Gaussian field $\mathbf{W}$ conditional on $\mathbf{a} < \mathbf{W} \leq \mathbf{b}$ component-wise. The parameters are as follows.

  ◇ `handle` is the handle returned by the `create_ghandle` function;

  ◇ `gibbs_iterations` is the number of Gibbs iterations to perform;

  ◇ `enc_a` and `enc_b` are the arrays of the constrains, each of `4*handle->n1*handle->n2` elements;

  ◇ `theta` is the array of covariance function parameters;

  ◇ `covtype` is the the covariance function type. Currently the possible values are

  `'m'` for Matérn correlation,
  `'e'` for exponential correlation,
  `'r'` for rational quatdratic correlation, and
  `'s'` for spherical correlation,
  see Section 1.2 for definitions.

  The function puts the resulting realization of $\mathbf{W}$ into the real part of `handle->w` and returns a pointer to its first element, or `NULL` in case of failure (negative embedding).

-
  ```
  FFTW_COMPLEX *z_from_w (ghandle handle);
  ```

  This function recovers the field $\mathbf{Z}$ from the encompassing field $\mathbf{W}$. The encompassing fields must be stored in the real and imaginary parts of `handle->k`. The resulting `handle->n1` $\times$ `handle->n2` fields $\mathbf{Z}$ will be put into the corresponding parts of `handle->w`. The function returns a pointer to its first element.

-
  ```
  FFTW_COMPLEX *generate_z (ghandle handle,
                            double *theta,
                            char covtype);
  ```

  This function generates two independent realizations of the Gaussian field $\mathbf{Z}$.

  ◇ `handle` is the handle returned by the `create_ghandle` function;

  ◇ `theta` is the array of the covariance parameters;

  ◇ `covtype` is the covariance function type. Currently the possible values are

`'m'`	for Matérn correlation,
`'e'`	for exponential correlation,
`'r'`	for rational quatdratic correlation, and
`'s'`	for spherical correlation,

see Section 1.2 for definitions.

The function puts the two resulting realization of **Z** into real and imaginary parts of `handle->w`, and returns a pointer to its first element, or `NULL` in case of failure (negative embedding).

-
  ```
  int write_field (int n1,
                   int n2,
                   int wn1,
                   int wn2,
                   double *theta,
                   char covtype,
                   char *fname_gaussian_ppm,
                   char *fname_gaussian_num,
                   char *fname_clipped_ppm,
                   char *fname_clipped_num,
                   char format,
                   int nlevels,
                   int *r, int *g, int *b,
                   double *c,
                   int thresholds);
  ```

This function is used to generate Gaussian and clipped Gaussian random fields and output them to files. Its use is demonstrated in the `field_demo` program. The parameters have the following meaning.

◇ `n1`, `n2`, `wn1`, and `wn2` specify the height and the width of the grid that will actually be used for random field generation, and of the grid that will be output, respectively. You may have to specify high values for `n1` and `n2` to achieve positive embedding of the covariance matrix. Generally, `n1` and `n2` should be large enough so that the covariance between the field values on the opposite sides of the grid is almost zero.

Upon successful generation, a `wn1` × `wn2` portion will be taken from the upper left corner of the larger `n1` × `n2` field. Note that high values of `n1` and `n2` make the program use more memory and work slower;

◇ `theta` is an array of the covariance parameters;

◇ `covtype` determines the covariance family used. Currently the available choices are

`m`	for Matérn correlation,
`e`	for exponential correlation,

`r` for rational quatdratic correlation, and

`s` for spherical correlation,

see Section 1.2 for definitions.

- ◇ `fname_gaussian_ppm` is the name of the image file (PPM format) into which the Gaussian field will be output as grey levels. If `NULL`, the file will not be created.
- ◇ `fname_gaussian_num` is the name of the file into which the Gaussian field will be output in the numeric form, according to the `format` parameter. If `NULL`, the file will not be created.
- ◇ `fname_clipped_ppm` is the name of the image file (PPM format) into which the discrete (clipped Gaussian) field will be output in colors according to `r`, `g`, and `b` parameters. If `NULL`, the file will not be created.
- ◇ `fname_clipped_num` is the name of file into which the discrete (clipped Gaussian) field will be output in the numeric form, according to the `format` parameter. If `NULL`, the file will not be created.
- ◇ `format` specifies the format in which the Gaussian and discrete fields will be output (see Section 3.1). The possible values are

  `3` for X Y Z values, and

  `Z` for Z values only.

  If you only want the images (PPM), this parameter is not important.
- ◇ `nlevels` is the number of clipping thresholds (equals k in the definition of clipping in Section 1.3).
- ◇ `r`, `g`, and `b` are the arrays of `nlevels` elements each, which determine the colors used in the discrete PPM file. If you are only interested in a discrete random field, these parameters are not important and can be specified as `NULL`'s.
- ◇ `c` is the array of `nlevels` thresholds or percentages (depending upon the `thresholds` parameter). These determine how the generated Gaussian field will be clipped to obtain a discrete field.

  For example, to produce a three color image, `nlevels` should be set to `2`, and two thresholds or percentage values should be supplied through `c`.

  If you are providing the thresholds, set `thresholds` parameter to `1`. In this case, the values of the Gaussian field below the first threshold will be assigned the first color in the list of colors (see `r`, `g`, `b`), the values between the thresholds will be assigned the second color, and the remaining values which are above the second threshold will be assigned the third color.

  Otherwise, if you are providing the percentages, set the `thresholds` parameter to `0`. Then the thresholds will be chosen after the Gaussian image is generated in such a way that the resulting discrete image will have exactly the specified percentage of each color.

If you are interested in the Gaussian field only and not in the discrete field, this parameter is not important and can be specified as `NULL`.

◇ `thresholds` specifies what is provided in the `c` array. The value of `1` stands for actual thresholds for use when clipping, and the value of `0` stands for the requested percentages of each color, based on which the thresholds will be determined after the Gaussian image is generated.

The function returns one of the following:

`0` – success;

`-1` – negative embedding;

`-2` – out of memory;

`-3` – can't compute correlation (for Matérn only).

- `double minus_likelihood (double *theta, int n);`

This function returns minus log likelihood of the encompassing Gaussian field **W**.

◇ `theta` is the covariance parameter vector (array);

◇ `n` is the number of elements in `theta`.

Before the function is called, the global variable `likelihood_params` must be initialized. It is declared in `gaussian.h` as follows:

```
struct likelihood_params_struct{
         char covtype;
         ghandle handle;
};
struct likelihood_params_struct *likelihood_params;
```

Here `likelihood_params->covtype` is one of the following:

`'m'` for Matérn correlation,

`'e'` for exponential correlation,

`'r'` for rational quatdratic correlation, and

`'s'` for spherical correlation,

see Section 1.2 for definitions, and `likelihood_params->handle` is the current working handle. The Fourier transform of the data for which the likelihood is computed is stored in `likelihood_params->handle->w`.

-
  ```
  int EM (double *a,
          double *b,
          int n1,
          int n2,
          int gibbs_iterations,
          int em_iterations,
          char covtype,
          double *theta,
          int ntheta,
          int usehooke,
          char *log_fname);
  ```

  This function estimates the covariance parameter vector $\boldsymbol{\theta}$ in the Gaussian model from the observed clipped data by the SEM algorithm. The use of the function is demonstrated in the `em_demo` program. The parameters have the following meaning.

  ◇ `a` and `b` are `n1` $\times$ `n2` arrays of constrains given by the observed data. For the unobserved Gaussian field $\mathbf{Z}$, we know that $\mathbf{a} < \mathbf{Z} \leq \mathbf{b}$ coordinate-wise;

  ◇ `n1` and `n2` are the dimensions of the discrete field;

  ◇ `gibbs_iterations` is the number of Gibbs iterations to perform at each SEM step;

  ◇ `em_iterations` is the number of SEM steps to perform;

  ◇ `covtype` determines the covariance family used. Currently the available choices are

    `m` for Matérn correlation,

    `e` for exponential correlation,

    `r` for rational quatdratic correlation, and

    `s` for spherical correlation,

    see Section 1.2 for definitions;

  ◇ `theta` is the starting value for $\boldsymbol{\theta}$ in the SEM algorithm. The array has `ntheta` elements;

  ◇ `ntheta` is the dimension of $\boldsymbol{\theta}$;

  ◇ `usehooke` determines whether the HOOKE (`usehooke==1`) or FMINBR (`usehooke==0`) optimization routines should be used in case `ntheta==1`. It is recommended to use FMINBR;

  ◇ `log_fname` is the name of the file to which the SEM path will be written. You should plot the values from this file to evaluate the convergence.

  The function returns `0` for success or `-1` in case of the out of memory condition.

4.3 Other Routines

The files `z.c`, `fminbr.c`, `hooke.c`, and `rkbesl.c` were downloaded from Netlib, `http://www.netlib.org`. See the comments sections of these files for documentation.