

UNU.RAN User Manual

Generating non-uniform random numbers
Version 0.7.0, 6 August 2004

Josef Leydold
Wolfgang Hörmann
Erich Janka
Roman Karawatzki
Günter Tirler

Copyright © 2000–2003 Institut fuer Statistik, WU Wien.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Table of Contents

UNURAN – Universal Non-Uniform RANdom number generators	1
1 Introduction	3
1.1 Usage of this document	3
1.2 Installation	3
1.3 Using the library	4
1.4 Concepts of UNURAN	5
1.5 Contact the authors	9
2 Examples	11
2.1 As short as possible	12
2.2 As short as possible (String API)	14
2.3 Select a method	15
2.4 Select a method (String API)	17
2.5 Arbitrary distributions	18
2.6 Arbitrary distributions (String API)	20
2.7 Change parameters of the method	22
2.8 Change parameters of the method (String API)	24
2.9 Change uniform random generator	26
2.10 Change uniform random generator (String API)	28
2.11 Sample pairs of antithetic random variates	30
2.12 Sample pairs of antithetic random variates (String API)	33
2.13 More examples	35
3 String Interface	37
3.1 Syntax of String Interface	37
3.2 Distribution String	39
3.2.1 Keys for Distribution String	39
3.3 Function String	42
3.4 Method String	45
3.4.1 Keys for Method String	45
3.5 Uniform RNG String	52
4 Handling distribution objects	53
4.1 Functions for all kinds of distribution objects	55
4.2 Continuous univariate distributions	57
4.3 Continuous univariate order statistics	64
4.4 Continuous empirical univariate distributions	67
4.5 Continuous multivariate distributions	68
4.6 Continuous univariate full conditional distribution	76
4.7 Continuous empirical multivariate distributions	78
4.8 MATRix distributions	79
4.9 Discrete univariate distributions	80

5 Methods for generating non-uniform random variates 85

5.1	Routines for all generator objects	85
5.2	AUTO – Select method automatically	87
5.3	Methods for continuous univariate distributions	88
5.3.1	AROU – Automatic Ratio-Of-Uniforms method	92
5.3.2	CSTD – Continuous STandarD distributions	95
5.3.3	HINV – Hermite interpolation based INVersion of CDF	97
5.3.4	HRB – Hazard Rate Bounded	100
5.3.5	HRD – Hazard Rate Decreasing	101
5.3.6	HRI – Hazard Rate Increasing	102
5.3.7	NINV – Numerical INVersion	103
5.3.8	NROU – Naive Ratio-Of-Uniforms method	106
5.3.9	SROU – Simple Ratio-Of-Uniforms method	108
5.3.10	SSR – Simple Setup Rejection	112
5.3.11	TABL – a TABLe method with piecewise constant hats	115
5.3.12	TDR – Transformed Density Rejection	120
5.3.13	TDRGW – Transformed Density Rejection - Gilks&Wild variant	125
5.3.14	UTDR – Universal Transformed Density Rejection	128
5.4	Methods for continuous empirical univariate distributions	131
5.4.1	EMPK – EMPirical distribution with Kernel smoothing	134
5.4.2	EMPL – EMPirical distribution with Linear interpolation	137
5.5	Methods for continuous multivariate distributions	138
5.5.1	BALL – ball-sampler	139
5.5.2	NORTA – NORmal To Anything	141
5.5.3	VAROU – Multivariate Adaptive Ratio-Of-Uniforms method	142
5.5.4	VMT – Vector Matrix Transformation	143
5.5.5	VNROU – Multivariate Naive Ratio-Of-Uniforms method	144
5.5.6	WALK – Metropolis random-walk sampler	146
5.6	Markov chain samplers for continuous multivariate distributions	147
5.6.1	GIBBS – Markov Chain - GIBBS sampler	148
5.6.2	HITRO – Markov Chain - HIT-and-run sampler with Ratio-Of-uniforms	151
5.7	Methods for continuous empirical multivariate distributions	156
5.7.1	VEMPK – (Vector) EMPirical distribution with Kernel smoothing	158
5.8	Methods for discrete univariate distributions	159
5.8.1	DARI – discrete automatic rejection inversion	162
5.8.2	DAU – (Discrete) Alias-Urn method	165
5.8.3	DGT – (Discrete) Guide Table method (indexed search)	166
5.8.4	DSROU – Discrete Simple Ratio-Of-Uniforms method	168
5.8.5	DSS – (Discrete) Sequential Search method	170
5.8.6	DSTD – Discrete STandarD distributions	171
5.9	Methods for random matrices	173
5.9.1	MCORR – Random CORRelation matrix	174
5.10	Methods for uniform univariate distributions	175
5.10.1	UNIF – wrapper for UNIFORM random number generator	176

6 Using uniform random number generators 177

6.1	Simple interface for uniform random number generators	182
6.2	Interface to GSL uniform random number generators	183
6.3	Interface to GSL generators for quasi-random points	183
6.4	Interface to Otmar Lendl’s pseudo-random number generators	184
6.5	Interface to L’Ecuyer’s RNGSTREAMS random number generators	185
6.6	Combine point set generator with random shifts	185

7	UNURAN Library of standard distributions	187
7.1	UNURAN Library of continuous univariate distributions	189
7.1.1	beta – Beta distribution	189
7.1.2	cauchy – Cauchy distribution	189
7.1.3	chi – Chi distribution	189
7.1.4	chisquare – Chisquare distribution	190
7.1.5	exponential – Exponential distribution	190
7.1.6	extremeI – Extreme value type I (Gumbel-type) distribution	190
7.1.7	extremeII – Extreme value type II (Frechet-type) distribution	190
7.1.8	gamma – Gamma distribution	191
7.1.9	laplace – Laplace distribution	191
7.1.10	logistic – Logistic distribution	191
7.1.11	lomax – Lomax distribution (Pareto distribution of second kind)	192
7.1.12	normal – Normal distribution	192
7.1.13	pareto – Pareto distribution (of first kind)	193
7.1.14	powerexponential – Powerexponential (Subbotin) distribution	193
7.1.15	rayleigh – Rayleigh distribution	193
7.1.16	student – Student's t distribution	193
7.1.17	triangular – Triangular distribution	194
7.1.18	uniform – Uniform distribution	194
7.1.19	weibull – Weibull distribution	194
7.2	UNURAN Library of continuous multivariate distributions	195
7.2.1	copula – Copula (distribution with uniform marginals)	195
7.2.2	multicauchy – Multicauchy distribution	195
7.2.3	multiexponential – Multiexponential distribution	195
7.2.4	multinormal – Multinormal distribution	195
7.2.5	multistudent – Multistudent distribution	195
7.3	UNURAN Library of discrete univariate distributions	197
7.3.1	binomial – Binomial distribution	197
7.3.2	geometric – Geometric distribution	197
7.3.3	hypergeometric – Hypergeometric distribution	197
7.3.4	logarithmic – Logarithmic distribution	198
7.3.5	negativebinomial – Negative Binomial distribution	198
7.3.6	poisson – Poisson distribution	198
7.4	UNURAN Library of random matrices	199
7.4.1	correlation – Random correlation matrix	199
8	Error handling	201
8.1	Error reporting	201
8.2	Output streams	204
9	Debugging	205
10	Testing	207
11	Miscellaneous	211
11.1	Mathematics	211

Appendix A	A Short Introduction to Random Variate Generation	213
A.1	The Inversion Method	213
A.2	The Rejection Method	214
A.3	The Composition Method	215
A.4	The Ratio-of-Uniforms Method	216
A.5	Inversion for Discrete Distributions	217
A.6	Indexed Search (Guide Table Method)	218
Appendix B	Glossary	219
Appendix C	Bibliography	221
Appendix D	Function Index	223

UNURAN – Universal Non-Uniform RANdom number generators

UNURAN (Universal Non-Uniform RANdom Number generator) is a collection of algorithms for generating non-uniform pseudorandom variates as a library of C functions designed and implemented by the ARVAG (Automatic Random VARIate Generation) project group in Vienna, and released under the GNU Public License (GPL). It is especially designed for such situations where

- a non-standard distribution or a truncated distribution is needed.
- experiments with different types of distributions are made.
- random variates for variance reduction techniques are used.
- fast generators of predictable quality are necessary.

Of course it is also well suited for standard distributions. However due to its more sophisticated programming interface it might not be as easy to use if you only look for a generator for the standard normal distribution. (Although UNURAN provides generators that are superior in many aspects to those found in quite a number of other libraries.)

UNURAN implements several methods for generating random numbers. The choice depends primary on the information about the distribution can be provided and – if the user is familiar with the different methods – on the preferences of the user.

The design goals of UNURAN are to provide *reliable*, *portable* and *robust* (as far as this is possible) functions with a consistent and easy to use interface. It is suitable for all situation where experiments with different distributions including non-standard distributions. For example it is no problem to replace the normal distribution by an empirical distribution in a model.

Since originally designed as a library for so called black-box or universal algorithms its interface is different from other libraries. (Nevertheless it also contains special generators for standard distributions.) It does not provide subroutines for random variate generation for particular distributions. Instead it uses an object-oriented interface. Distributions and generators are treated as independent objects. This approach allows one not only to have different methods for generating non-uniform random variates. It is also possible to choose the method which is optimal for a given situation (e.g. speed, quality of random numbers, using for variance reduction techniques, etc.). It also allows to sample from non-standard distribution or even from distributions that arise in a model and can only be computed in a complicated subroutine.

Sampling from a particular distribution requires the following steps:

1. Create a distribution object. (Objects for standard distributions are available in the library)
2. Choose a method.
3. Initialize the generator, i.e., create the generator object. If the choosen method is not suitable for the given distribution (or if the distribution object contains too little information about the distribution) the initialization routine fails and produces an error message. Thus the generator object does (probably) not produce false results (random variates of a different distribution).
4. Use this generator object to sample from the distribution.

There are four types of objects that can be manipulated independently:

- **Distribution objects:** hold all information about the random variates that should be generated. The following types of distributions are available:
 - Continuous and Discrete distributions
 - Empirical distributions
 - Multivariate distributions

Of course a library of standard distributions is included (and these can be further modified to get, e.g., truncated distributions). Moreover the library provides subroutines to build almost arbitrary distributions.

- **Generator objects:** hold the generators for the given distributions. It is possible to build independent generator objects for the same distribution object which might use the same or different methods for generation. (If the chosen method is not suitable for the given method, a NULL pointer is returned in the initialization step).
- **Parameter objects:** Each transformation method requires several parameters to adjust the generator to a given distribution. The parameter object holds all this information. When created it contains all necessary default settings. It is only used to create a generator object and destroyed immediately. Although there is no need to change these parameters or even know about their existence for “usual distributions”, they allow a fine tuning of the generator to work with distributions with some awkward properties. The library provides all necessary functions to change these default parameters.
- **Uniform Random Number Generators:** All generator objects need one (or more) streams of uniform random numbers that are transformed into random variates of the given distribution. These are given as pointers to appropriate functions or structures (objects). Two generator objects may have their own uniform random number generators or share a common one. Any functions that produce uniform (pseudo-) random numbers can be used. We suggest Otmar Lendl’s PRNG library.

1 Introduction

1.1 Usage of this document

We designed this document in a way such that one can use UNURAN with reading as little as necessary. Read [Section 1.2 \[Installation\]](#), page 3 for the instructions to install the library. [Section 1.4 \[Concepts of UNURAN\]](#), page 5, describes the basics of UNURAN. It also has a short guideline for choosing an appropriate method. In [Chapter 2 \[Examples\]](#), page 11 examples are given that can be copied and modified. They also can be found in the directory ‘examples’ in the source tree.

Further information are given in consecutive chapters. [Chapter 4 \[Handling distribution objects\]](#), page 53, describes how to create and manipulate distribution objects. [Chapter 7 \[standard distributions\]](#), page 187, describes predefined distribution objects that are ready to use. [Chapter 5 \[Methods\]](#), page 85 describes the various methods in detail. For each of possible distribution classes (continuous, discrete, empirical, multivariate) there exists a short overview section that can be used to choose an appropriate method followed by sections that describe each of the particular methods in detail. These are merely for users with some knowledge about the methods who want to change method-specific parameters and can be ignored by others.

Abbreviations and explanation of some basic terms can be found in [Appendix B \[Glossary\]](#), page 219.

1.2 Installation

UNURAN was developed on an Intel architecture under Linux with the GNU C compiler.

Uniform random number generator

It can be used with any uniform random number generator but (at the moment) some features work best with Otmar Lendl’s *prng* library (see <http://statistik.wu-wien.ac.at/prng/> for description and downloading). For more details on using uniform random number in UNURAN see [Chapter 6 \[Using uniform random number generators\]](#), page 177.

UNURAN

1. First unzip and untar the package and change to the directory:

```
tar zxvf unuran-0.7.0.tar.gz
cd unuran-0.7.0
```

2. Edit the file ‘src/unuran_config.h’.
3. Set sources of uniform random numbers in file ‘src/unuran_config.h’.

Set UNUR_URNG_DEFAULT_TYPE to one of the given types. The setting in the distributed file is a built-in generator, so no other libraries need to be installed.

You can change the default uniform generators for the chosen type.

UNURAN provides wrapper functions for some libraries for uniform random number generators. If you want to use these libraries, then uncomment the corresponding UNURAN_HAS_ macro. Notice that these libraries have to be installed before running UNURAN’s configure script. Moreover, it is important that these libraries have to be linked when making an executable.

Important: Do **not** set UNUR_URNG_TYPE to other values but UNUR_URNG_GENERIC unless you have serious reasons. With UNURAN-0.7.0 these other types are depreciated and not tested any more. They are only provided for backwards compatibility if this is required.

4. Run a configuration script:

```
sh ./configure --prefix=<prefix>
```

where <prefix> is the root of the installation tree. When omitted ‘/usr/local’ is used.

Use `configure --help` to get a list of other options.

Important: You must install PRNG or RngStreams *before* `configure` is executed.

Important: UNU.RAN now relies on some aspects of IEEE 754 compliant floating point arithmetic. In particular, 1./0. and 0./0. must result in `infinity` and `NaN` (not a number), respectively, and must not cause a floating point exception. For almost all modern computing architecture this is implemented in hardware. For others there should be a special compiler flag to get this feature (e.g. `-MIEEE` on DEC alpha).

5. Compile and install the library:

```
make
make install
```

This installs the following files:

```
$(prefix)/include/unuran.h
$(prefix)/include/unuran_config.h
$(prefix)/include/unuran_tests.h
$(prefix)/info/unuran.info
$(prefix)/lib/libunuran.a
$(prefix)/lib/libunuran.so
```

(However, the names of the libraries may vary with your OS.)

Obviously `$(prefix)/include` and `$(prefix)/lib` must be in the search path of your compiler. You can use environment variables to add these directories to the search path. If you are using the bash type (or add to your profile):

```
export LIBRARY_PATH="<prefix>/lib"
export C_INCLUDE_PATH="<prefix>/include"
```

If you want to make a shared library, then making such a library can be enabled using

```
sh ./configure --enable-shared
```

If you want to link against the shared library make sure that it can be found when executing the binary that links to the library. If it is not installed in the usual path, then the easiest way is to set the `LD_LIBRARY_PATH` environment variable. See any operating system documentation about shared libraries for more information, such as the `ld(1)` and `ld.so(8)` manual pages.

6. Documentation in various formats (PS, PDF, info, dvi, HTML, plain text) can be found in directory ‘doc’.
7. You can run some tests my

```
make check
```

However, this test suite requires the usage of prng. It might happen that some of the tests might fail due to roundoff errors or the mysteries of floating point arithmetic, since we have used some extreme settings to test the library.

1.3 Using the library

ANSI C Compliance

The library is written in ANSI C and is intended to conform to the ANSI C standard. It should be portable to any system with a working ANSI C compiler.

The library does not rely on any non-ANSI extensions in the interface it exports to the user. Programs you write using UNU.RAN can be ANSI compliant. Extensions which can be used in a way compatible with pure ANSI C are supported, however, via conditional compilation. This allows the library to take advantage of compiler extensions on those platforms which support them.

To avoid namespace conflicts all exported function names and variables have the prefix `unur_`, while exported macros have the prefix `UNUR_`.

Compiling and Linking

If you want to use the library you must include the UNURAN header file

```
#include <unuran.h>
```

If you also need the test routines then also add

```
#include <unuran_tests.h>
```

If these header files are not installed on the standard search path of your compiler you will also need to provide its location to the preprocessor as a command line flag. The default location of the 'unuran.h' is '/usr/local/include'. A typical compilation command for a source file 'app.c' with the GNU C compiler gcc is,

```
gcc -I/usr/local/include -c app.c
```

This results in an object file 'app.o'. The default include path for gcc searches '/usr/local/include' automatically so the -I option can be omitted when UNURAN is installed in its default location.

The library is installed as a single file, 'libunuran.a'. A shared version of the library is also installed on systems that support shared libraries. The default location of these files is '/usr/local/lib'. To link against the library you need to specify the main library. The following example shows how to link an application with the library (and the the PRNG library if you decide to use this source of uniform pseudo-random numbers),

```
gcc app.o -lunuran -lprng -lm
```

Shared Libraries

To run a program linked with the shared version of the library it may be necessary to define the shell variable LD_LIBRARY_PATH to include the directory where the library is installed. For example,

```
LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH ./app
```

To compile a statically linked version of the program instead, use the -static flag in gcc,

```
gcc -static app.o -lunuran -lprng -lm
```

Compatibility with C++

The library header files automatically define functions to have **extern "C"** linkage when included in C++ programs.

1.4 Concepts of UNURAN

UNURAN is a C library for generating non-uniformly distributed random variates. Its emphasis is on the generation of non-standard distribution and on streams of random variates of special purposes. It is designed to provide a consistent tool to sample from distributions with various properties. Since there is no universal method that fits for all situations, various methods for sampling are implemented.

UNURAN solves this complex task by means of an object oriented programming interface. Three basic objects are used:

- distribution object UNUR_DISTR
Hold all information about the random variates that should be generated.
- generator object UNUR_GEN
Hold the generators for the given distributions. Two generator objects are completely independent of each other. They may share a common uniform random number generator or have their owns.
- parameter object UNUR_PAR
Hold all information for creating a generator object. It is necessary due to various parameters and switches for each of these generation methods.

For programming notice that the parameter objects only hold pointer to arrays but do not have their own copy of such an array. Especially, if a dynamically allocated array is used it *must not* be freed until the generator object has been created!

The idea behind these structures is to make distributions, choosing a generation method and sampling to orthogonal (ie. independent) functions of the library. The parameter object is only introduced due to the necessity to deal with various parameters and switches for each of these generation methods which are required to adjust the algorithms to unusual distributions with extreme properties but have default values that are suitable for most applications. These parameters and the data for distributions are set by various functions.

Once a generator object has been created sampling (from the univariate continuous distribution) can be done by the following command:

```
double x = unur_sample_cont(generator);
```

Analogous commands exist for discrete and multivariate distributions. For detailed examples that can be copied and modified see [Chapter 2 \[Examples\]](#), page 11.

Distribution objects

All information about a distribution are stored in objects (structures) of type `UNUR_DISTR`. UNURAN has five different types of distribution objects:

<code>cont</code>	Continuous univariate distributions.
<code>cvec</code>	Continuous multivariate distributions.
<code>discr</code>	Discrete univariate distributions.
<code>cemp</code>	Continuous empirical univariate distribution, ie. given by sample.
<code>cvemp</code>	Continuous empirical multivariate distribution, ie. given by sample.
<code>matr</code>	Matrix distributions.

Distribution objects can be created from scratch by the following call

```
distr = unur_distr_<type>_new();
```

where `<type>` is one of the five possible types from the above table. Notice that these commands only create an *empty* object which still must be filled by means of calls for each type of distribution object (see [Chapter 4 \[Handling distribution objects\]](#), page 53). The naming scheme of these functions is designed to indicate the corresponding type of the distribution object and the task to be performed. It is demonstrated on the following example.

```
unur_distr_cont_set_pdf(distr, mypdf);
```

This command stores a PDF named `mypdf` in the distribution object `distr` which must have the type `cont`.

Of course UNURAN provides an easier way to use standard distribution. Instead of using `unur_distr_<type>_new` calls and functions `unur_distr_<type>_set_<...>` for setting data objects for standard distribution can be created by a single call. Eg. to get an object for the normal distribution with mean 2 and standard deviation 5 use

```
double parameter[2] = {2.0, 5.0};
UNUR_DISTR *distr = unur_distr_normal(parameter, 2);
```

For a list of standard distributions see [Chapter 7 \[Standard distributions\]](#), page 187.

Generation methods

The information a distribution object must contain depends heavily on the method chosen for sampling random variates.

Brackets indicate optional information while a tilde indicates that only an approximation must be provided. See [Appendix B \[Glossary\]](#), page 219, for unfamiliar terms.

Methods for **continuous univariate distributions**

sample with `unur_sample_cont`

method	PDF	dPDF	mode	area	other
AROU	x	x	[x]		T-concave
CSTD					build-in standard distribution
HINV	[x]	[x]			CDF
NINV	[x]				CDF
SROU			x	x	T-concave
SSR			x	x	T-concave
TABLE	x		x	[~]	all local extrema
TDR	x	x			T-concave
UTDR	x		x	~	T-concave

Methods for **continuous empirical univariate distributions**

sample with `unur_sample_cont`

EMPK: Requires an observed sample. EMPL: Requires an observed sample.

Methods for **continuous multivariate distributions**

sample with `unur_sample_vec`

VMT: Requires the mean vector and the covariance matrix. VNROU: Requires the PDF.

Methods for **continuous empirical multivariate distributions**

sample with `unur_sample_vec`

VEMPK: Requires an observed sample.

Methods for **discrete univariate distributions**

sample with `unur_sample_discr`

method	PMF	PV	mode	sum	other
DARI	x		x	~	T-concave
DAU	[x]	x			
DGT	[x]	x			
DSTD					build-in standard distribution
DSS	[x]	x		x	

Because of tremendous variety of possible problems, UNURAN provides many methods. All information for creating an generator object have to collected in a parameter first. For example if the task is to sample from a continuous distribution the method AROU might be a good choice. Then the call

```
UNUR_PAR *par = unor_arou_new(distribution);
```

creates an parameter object `par` with a pointer to the distribution object and default values for all necessary parameters for method AROU. Other methods can be used by replacing `arou` with the name of the desired methods (in lower case letters):

```
UNUR_PAR *par = unor_<method>_new(distribution);
```

This sets the default values for all necessary parameters for the chosen methods. These are suitable for almost all applications. Nevertheless it is possible to control the behaviour of the method using corresponding `set` calls for each method. This might be necessary to adjust the algorithm for an unusual distribution with extreme properties, or just for fine tuning the performance of the algorithm. The following example demonstrates how to change the maximum number of iterations for method NINV to the value 50:

```
unur_ninv_set_max_iteration(par, 50);
```

All available methods are described in details in [Chapter 5 \[Methods\]](#), page 85.

Creating a generator object

Now it is possible to create a generator object:

```
UNUR_GEN *generator = unor_init(par);
if (generator == NULL) exit(EXIT_FAILURE);
```

Important: You must always check whether `unur_init` has been executed successfully. Otherwise the NULL pointer is returned which causes a segmentation fault when used for sampling.

Important: The call of `unur_init` **destroys** the parameter object!

Moreover it is recommended to call `unur_init` immediately after the parameter object `par` has created and modified.

An existing generator object is a rather static construct. Nevertheless some of the parameters can still be modified by `chg` calls, e.g.

```
unur_ninv_chg_max_iteration(gen, 30);
```

Notice that it is important *when* pararameters are changed because different functions must be used:

To change the parameters *before* creating the generator object, the function name includes the term `set` and the first argument must be of type `UNUR_PAR`.

To change the parameters for an *existing* generator object, the function name includes the term `chg` and the first argument must be of type `UNUR_GEN`.

For details see [Chapter 5 \[Methods\]](#), page 85.

Sampling

You can now use your generator object in any place of your program to sample from your distribution. You only have take about the type of number it computes: `double`, `int` or a vector (array of `doubles`). Notice that at this point it does not matter whether you are sampling from a gamma distribution, a truncated normal distribution or even an empirical distribution.

Destroy

When you do not need your generator object any more, you should destroy it:

```
unur_free(generator);
```

Uniform random numbers

Each generator object can have its own uniform random number generator or share one with others. When created a parameter object the pointer for the uniform random number generator is set to the default generator. However it can be changed at any time to any other generator:

```
unur_set_urng(par, urng);
```

or

```
unur_chg_urng(generator, urng);
```

respectively. See [Chapter 6 \[Using uniform random number generators\]](#), page 177, for details.

1.5 Contact the authors

If you have any problems with UNURAN, suggestions how to improve the library or find a bug, please contact us via email unuran@statistik.wu-wien.ac.at.

For news please visit out homepage at <http://statistik.wu-wien.ac.at/unuran/>.

2 Examples

The examples in this chapter should compile cleanly and can be found in the directory ‘`examples`’ of the source tree of UNURAN. Assuming that UNURAN as well as the PRNG libraries have been installed properly (see [Section 1.2 \[Installation\]](#), [page 3](#)) each of these can be compiled (using the GCC in this example) with

```
gcc -Wall -O2 -o example example.c -lunuran -lprng -lm
```

Remark: `-lprng` must be omitted when the PRNG library is not installed. Then however some of the examples might not work.

The library uses three objects: `UNUR_DISTR`, `UNUR_PAR` and `UNUR_GEN`. It is not important to understand the details of these objects but it is important not to change the order of their creation. The distribution object can be destroyed *after* the generator object has been made. (The parameter object is freed automatically by the `unur_init` call.) It is also important to check the result of the `unur_init` call. If it has failed the `NULL` pointer is returned and causes a segmentation fault when used for sampling.

We give all examples with the UNURAN standard API and the more convenient string API.

2.1 As short as possible

Select a distribution and let UNURAN do all necessary steps.

```

/* ----- */
/* File: example0.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

int main()
{
    int i; /* loop variable */
    double x; /* will hold the random number */

    /* Declare the three UNURAN objects. */
    UNUR_DISTR *distr; /* distribution object */
    UNUR_PAR *par; /* parameter object */
    UNUR_GEN *gen; /* generator object */

    /* Use a predefined standard distribution: */
    /* Gaussian with mean zero and standard deviation 1. */
    /* Since this is the standard form of the distribution, */
    /* we need not give these parameters. */
    distr = unur_distr_normal(NULL, 0);

    /* Use method AUTO: */
    /* Let UNURAN select a suitable method for you. */
    par = unur_auto_new(distr);

    /* Now you can change some of the default settings for the */
    /* parameters of the chosen method. We don't do it here. */

    /* Create the generator object. */
    gen = unur_init(par);

    /* Notice that this call has also destroyed the parameter */
    /* object 'par' as a side effect. */

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* It is possible to reuse the distribution object to create */
    /* another generator object. If you do not need it any more, */
    /* it should be destroyed to free memory. */
    unur_distr_free(distr);

    /* Now you can use the generator object 'gen' to sample from */
    /* the standard Gaussian distribution. */
    /* Eg.: */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* When you do not need the generator object any more, you */
    /* can destroy it. */
    unur_free(gen);

```

```
    exit (EXIT_SUCCESS);  
} /* end of main() */  
/* ----- */
```

2.2 As short as possible (String API)

Select a distribution and let UNURAN do all necessary steps.

```

/* ----- */
/* File: example0_str.c */
/* ----- */
/* String API. */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

int main()
{
    int    i;        /* loop variable */
    double x;        /* will hold the random number */

    /* Declare UNURAN generator object. */
    UNUR_GEN *gen;    /* generator object */

    /* Create the generator object. */
    /* Use a predefined standard distribution: */
    /*   Standard Gaussian distribution. */
    /* Use method AUTO: */
    /*   Let UNURAN select a suitable method for you. */
    gen = unur_str2gen("normal()");

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Now you can use the generator object 'gen' to sample from */
    /* the standard Gaussian distribution. */
    /* Eg.: */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* When you do not need the generator object any more, you */
    /* can destroy it. */
    unur_free(gen);

    exit (EXIT_SUCCESS);
} /* end of main() */

/* ----- */

```

2.3 Select a method

Select method AROU and use it with default parameters.

```

/* ----- */
/* File: example1.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

int main()
{
    int i; /* loop variable */
    double x; /* will hold the random number */

    /* Declare the three UNURAN objects. */
    UNUR_DISTR *distr; /* distribution object */
    UNUR_PAR *par; /* parameter object */
    UNUR_GEN *gen; /* generator object */

    /* Use a predefined standard distribution: */
    /* Gaussian with mean zero and standard deviation 1. */
    /* Since this is the standard form of the distribution, */
    /* we need not give these parameters. */
    distr = unur_distr_normal(NULL, 0);

    /* Choose a method: AROU. */
    /* For other (suitable) methods replace "arou" with the */
    /* respective name (in lower case letters). */
    par = unur_arou_new(distr);

    /* Now you can change some of the default settings for the */
    /* parameters of the chosen method. We don't do it here. */

    /* Create the generator object. */
    gen = unur_init(par);

    /* Notice that this call has also destroyed the parameter */
    /* object 'par' as a side effect. */

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* It is possible to reuse the distribution object to create */
    /* another generator object. If you do not need it any more, */
    /* it should be destroyed to free memory. */
    unur_distr_free(distr);

    /* Now you can use the generator object 'gen' to sample from */
    /* the standard Gaussian distribution. */
    /* Eg.: */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* When you do not need the generator object any more, you */
    /* can destroy it. */
}

```

```
unur_free(gen);  
  
exit (EXIT_SUCCESS);  
  
} /* end of main() */  
  
/* ----- */
```

2.4 Select a method (String API)

Select method AROU and use it with default parameters.

```

/* ----- */
/* File: example1_str.c */
/* ----- */
/* String API. */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

int main()
{
    int i; /* loop variable */
    double x; /* will hold the random number */

    /* Declare UNURAN generator object. */
    UNUR_GEN *gen; /* generator object */

    /* Create the generator object. */
    /* Use a predefined standard distribution: */
    /* Standard Gaussian distribution. */
    /* Choose a method: AROU. */
    /* For other (suitable) methods replace "arou" with the */
    /* respective name. */
    gen = unur_str2gen("normal() & method=arou");

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Now you can use the generator object 'gen' to sample from */
    /* the standard Gaussian distribution. */
    /* Eg.: */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* When you do not need the generator object any more, you */
    /* can destroy it. */
    unur_free(gen);

    exit (EXIT_SUCCESS);
} /* end of main() */

/* ----- */

```

2.5 Arbitrary distributions

If you want to sample from a non-standard distribution, UNURAN might be exactly what you need. Depending on the information is available, a method must be chosen for sampling, see [Section 1.4 \[Concepts\]](#), page 5 for an overview and [Chapter 5 \[Methods\]](#), page 85 for details.

```

/* ----- */
/* File: example2.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* In this example we build a distribution object from scratch */
/* and sample from this distribution. */
/* ----- */
/* We use method TDR (Transformed Density Rejection) which */
/* required a PDF and the derivative of the PDF. */
/* ----- */

/* Define the PDF and dPDF of our distribution. */
/* ----- */
/* Our distribution has the PDF */
/* ----- */
/*          / 1 - x*x  if |x| <= 1
/* f(x) = <
/*          \ 0         otherwise
/* ----- */

/* The PDF of our distribution: */
double mypdf( double x, const UNUR_DISTR *distr )
/* The second argument ('distr') can be used for parameters */
/* for the PDF. (We do not use parameters in our example.) */
{
    if (fabs(x) >= 1.)
        return 0.;
    else
        return (1.-x*x);
} /* end of mypdf() */

/* The derivative of the PDF of our distribution: */
double mydpdf( double x, const UNUR_DISTR *distr )
{
    if (fabs(x) >= 1.)
        return 0.;
    else
        return (-2.*x);
} /* end of mydpdf() */

/* ----- */

int main()
{
    int    i;        /* loop variable */
    double x;        /* will hold the random number */

    /* Declare the three UNURAN objects. */
    UNUR_DISTR *distr; /* distribution object */
    UNUR_PAR   *par;   /* parameter object */
    UNUR_GEN   *gen;   /* generator object */

    /* Create a new distribution object from scratch. */

```



```

/* It is a continuous distribution, and we need a PDF and the */
/* derivative of the PDF. Moreover we set the domain.          */

/* Get empty distribution object for a continuous distribution */
distr = unur_distr_cont_new();

/* Assign the PDF and dPDF (defined above).                    */
unur_distr_cont_set_pdf( distr, mypdf );
unur_distr_cont_set_dpdpf( distr, mydpdf );

/* Set the domain of the distribution (optional for TDR).      */
unur_distr_cont_set_domain( distr, -1., 1. );

/* Choose a method: TDR.                                       */
par = unur_tdr_new(distr);

/* Now you can change some of the default settings for the    */
/* parameters of the chosen method. We don't do it here.      */

/* Create the generator object.                                 */
gen = unur_init(par);

/* Notice that this call has also destroyed the parameter     */
/* object 'par' as a side effect.                               */

/* It is important to check if the creation of the generator  */
/* object was successful. Otherwise 'gen' is the NULL pointer  */
/* and would cause a segmentation fault if used for sampling.  */
if (gen == NULL) {
    fprintf(stderr, "ERROR: cannot create generator object\n");
    exit (EXIT_FAILURE);
}

/* It is possible to reuse the distribution object to create   */
/* another generator object. If you do not need it any more,   */
/* it should be destroyed to free memory.                      */
unur_distr_free(distr);

/* Now you can use the generator object 'gen' to sample from   */
/* the distribution. Eg.:                                       */
for (i=0; i<10; i++) {
    x = unur_sample_cont(gen);
    printf("%f\n",x);
}

/* When you do not need the generator object any more, you    */
/* can destroy it.                                             */
unur_free(gen);

exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */

```

2.6 Arbitrary distributions (String API)

If you want to sample from a non-standard distribution, UNURAN might be exactly what you need. Depending on the information is available, a method must be chosen for sampling, see [Section 1.4 \[Concepts\]](#), [page 5](#) for an overview and [Chapter 5 \[Methods\]](#), [page 85](#) for details.

```

/* ----- */
/* File: example2_str.c                               */
/* ----- */
/* String API.                                         */
/* ----- */

/* Include UNURAN header file.                         */
#include <unuran.h>

/* ----- */

/* In this example we use a generic distribution object */
/* and sample from this distribution.                   */
/* ----- */
/* The PDF of our distribution is given by              */
/* ----- */
/*          / 1 - x*x if |x| <= 1                      */
/* f(x) = <                                         */
/*          \ 0      otherwise                        */
/* ----- */
/* We use method TDR (Transformed Density Rejection) which */
/* required a PDF and the derivative of the PDF.          */
/* ----- */

int main()
{
    int    i;      /* loop variable */
    double x;      /* will hold the random number */

    /* Declare UNURAN generator object. */
    UNUR_GEN *gen; /* generator object */

    /* Create the generator object. */
    /* Use a generic continuous distribution. */
    /* Choose a method: TDR. */
    gen = unur_str2gen("distr = cont; pdf=\"1-x*x\"; domain=(-1,1) & method=tdr");

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Now you can use the generator object 'gen' to sample from */
    /* the distribution. Eg.: */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* When you do not need the generator object any more, you */
    /* can destroy it. */
    unur_free(gen);

    exit (EXIT_SUCCESS);
}

```

```
} /* end of main() */
```

```
/* ----- */
```

2.7 Change parameters of the method

Each method for generating random numbers allows several parameters to be modified. If you do not want to use default values, it is possible to change them. The following example illustrates how to change parameters. For details see [Chapter 5 \[Methods\]](#), page 85.

```

/* ----- */
/* File: example3.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

int main()
{
    int i; /* loop variable */
    double x; /* will hold the random number */

    double fparams[2]; /* array for parameters for distribution */

    /* Declare the three UNURAN objects. */
    UNUR_DISTR *distr; /* distribution object */
    UNUR_PAR *par; /* parameter object */
    UNUR_GEN *gen; /* generator object */

    /* Use a predefined standard distribution: */
    /* Gaussian with mean 2. and standard deviation 0.5. */
    fparams[0] = 2.;
    fparams[1] = 0.5;
    distr = unur_distr_normal( fparams, 2 );

    /* Choose a method: TDR. */
    par = unur_tdr_new(distr);

    /* Change some of the default parameters. */

    /* We want to use  $T(x)=\log(x)$  for the transformation. */
    unur_tdr_set_c( par, 0. );

    /* We want to have the variant with immediate acceptance. */
    unur_tdr_set_variant_ia( par );

    /* We want to use 10 construction points for the setup */
    unur_tdr_set_cpoints ( par, 10, NULL );

    /* Create the generator object. */
    gen = unur_init(par);

    /* Notice that this call has also destroyed the parameter */
    /* object 'par' as a side effect. */

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* It is possible to reuse the distribution object to create */
    /* another generator object. If you do not need it any more, */
    /* it should be destroyed to free memory. */
    unur_distr_free(distr);

```

```
/* Now you can use the generator object 'gen' to sample from */
/* the distribution. Eg.: */
for (i=0; i<10; i++) {
    x = unur_sample_cont(gen);
    printf("%f\n",x);
}

/* It is possible with method TDR to truncate the distribution */
/* for an existing generator object ... */
unur_tdr_chg_truncated( gen, -1., 0. );

/* ... and sample again. */
for (i=0; i<10; i++) {
    x = unur_sample_cont(gen);
    printf("%f\n",x);
}

/* When you do not need the generator object any more, you */
/* can destroy it. */
unur_free(gen);

exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */
```

2.8 Change parameters of the method (String API)

Each method for generating random numbers allows several parameters to be modified. If you do not want to use default values, it is possible to change them. The following example illustrates how to change parameters. For details see [Chapter 5 \[Methods\]](#), page 85.

```

/* ----- */
/* File: example3_str.c                               */
/* ----- */
/* String API.                                         */
/* ----- */

/* Include UNURAN header file.                         */
#include <unuran.h>

/* ----- */

int main()
{
    int    i;          /* loop variable          */
    double x;          /* will hold the random number */

    /* Declare UNURAN generator object.                 */
    UNUR_GEN *gen;     /* generator object       */

    /* Create the generator object.                     */
    /* Use a predefined standard distribution:           */
    /* Gaussian with mean 2. and standard deviation 0.5. */
    /* Choose a method: TDR with parameters             */
    /* c = 0: use T(x)=log(x) for the transformation;   */
    /* variant "immediate acceptance";                 */
    /* number of construction points = 10.              */
    gen = unur_str2gen("normal(2,0.5) & method=tdr; c=0.; variant=ia; cpoints=10");

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Now you can use the generator object 'gen' to sample from */
    /* the distribution. Eg.:                                     */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* It is possible with method TDR to truncate the distribution */
    /* for an existing generator object ...                         */
    unur_tdr_chg_truncated( gen, -1., 0. );

    /* ... and sample again.                                       */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* When you do not need the generator object any more, you */
    /* can destroy it.                                           */
    unur_free(gen);

    exit (EXIT_SUCCESS);
}

```

```
} /* end of main() */
```

```
/* ----- */
```

2.9 Change uniform random generator

All generator object use the same default uniform random number generator by default. This can be changed to any generator of your choice such that each generator object has its own random number generator or can share it with some other objects. It is also possible to change the default generator at any time. See [Chapter 6 \[Using uniform random number generators\]](#), [page 177](#), for details.

The following example shows how the uniform random number generator can be set or changed for a generator object. It requires the PRNG library to be installed and used. Otherwise the example must be modified accordingly.

```
/* ----- */
/* File: example4.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* This example makes use of the PRNG library (see */
/* http://statistik.wu-wien.ac.at/prng/) for generating */
/* uniform random numbers. */
/* To compile this example you must have set */
/* */
/* #define UNUR_URNG_TYPE UNUR_URNG_GENERIC */
/* #define UNURAN_HAS_PRNG 1 */
/* */
/* in 'src/unuran_config.h'. */
/* (Of course the executable has to be linked against the */
/* prng library.) */
/* ----- */

int main()
{
    #if UNUR_URNG_TYPE == UNUR_URNG_GENERIC && defined(UNURAN_HAS_PRNG)

        int i; /* loop variable */
        double x; /* will hold the random number */
        double fparams[2]; /* array for parameters for distribution */

        /* Declare the three UNURAN objects. */
        UNUR_DISTR *distr; /* distribution object */
        UNUR_PAR *par; /* parameter object */
        UNUR_GEN *gen; /* generator object */

        /* Declare objects for uniform random number generators. */
        UNUR_URNG *urng1, *urng2; /* uniform generator objects */

        /* PRNG only: */
        /* Make a object for uniform random number generator. */
        /* For details see http://statistik.wu-wien.ac.at/prng/ */
        /* We use the Mersenne Twister. */
        urng1 = unur_urng_prng_new("mt19937(1237)");
        if (urng1 == NULL) exit (EXIT_FAILURE);

        /* Use a predefined standard distribution: */
        /* Beta with parameters 2 and 3. */
        fparams[0] = 2.;
        fparams[1] = 3.;
        distr = unur_distr_beta( fparams, 2 );

        /* Choose a method: TDR. */
    }
}
```



```

    par = unur_tdr_new(distr);

    /* Set uniform generator in parameter object */
    unur_set_urng( par, urng1 );

    /* Create the generator object. */
    gen = unur_init(par);

    /* Notice that this call has also destroyed the parameter */
    /* object 'par' as a side effect. */

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* It is possible to reuse the distribution object to create */
    /* another generator object. If you do not need it any more, */
    /* it should be destroyed to free memory. */
    unur_distr_free(distr);

    /* Now you can use the generator object 'gen' to sample from */
    /* the distribution. Eg.: */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* Now we want to switch to a different uniform random number */
    /* generator. */
    /* Now we use an ICG (Inversive Congruential Generator). */
    urng2 = unur_urng_prng_new("icg(2147483647,1,1,0)");
    if (urng2 == NULL) exit (EXIT_FAILURE);
    unur_chg_urng( gen, urng2 );

    /* ... and sample again. */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* When you do not need the generator object any more, you */
    /* can destroy it. */
    unur_free(gen);

    /* We also should destroy the uniform random number generators.*/
    unur_urng_free(urng1);
    unur_urng_free(urng2);

    exit (EXIT_SUCCESS);

#else
    printf("You must use the PRNG library to run this example!\n\n");
    exit (77); /* exit code for automake check routines */
#endif

} /* end of main() */

/* ----- */

```

2.10 Change uniform random generator (String API)

All generator object use the same default uniform random number generator by default. This can be changed to any generator of your choice such that each generator object has its own random number generator or can share it with some other objects. It is also possible to change the default generator at any time. See [Chapter 6 \[Using uniform random number generators\]](#), [page 177](#), for details.

The following example shows how the uniform random number generator can be set or changed for a generator object. It requires the PRNG library to be installed and used. Otherwise the example must be modified accordingly.

```
/* ----- */
/* File: example4_str.c                               */
/* ----- */
/* String API.                                         */
/* ----- */

/* Include UNURAN header file.                         */
#include <unuran.h>

/* ----- */

/* This example makes use of the PRNG library (see      */
/* http://statistik.wu-wien.ac.at/prng/) for generating */
/* uniform random numbers.                             */
/* To compile this example you must have set           */
/* ----- */
/* #define UNUR_URNG_TYPE UNUR_URNG_GENERIC            */
/* #define UNURAN_HAS_PRNG 1                          */
/* ----- */
/* in 'src/unuran_config.h'.                          */
/* (Of course the executable has to be linked against the */
/* prng library.)                                       */

/* It also works with necessary modifications with other uniform */
/* random number generators.                                  */

/* ----- */

int main()
{
    #if UNUR_URNG_TYPE == UNUR_URNG_GENERIC && defined(UNURAN_HAS_PRNG)

        int    i;          /* loop variable                */
        double x;          /* will hold the random number */

        /* Declare UNURAN generator object.                */
        UNUR_GEN *gen;    /* generator object            */

        /* Declare objects for uniform random number generators. */
        UNUR_URNG *urng1, *urng2; /* uniform generator objects */

        /* Create the generator object.                    */
        /* Use a predefined standard distribution:          */
        /* Beta with parameters 2 and 3.                   */
        /* Choose a method: TDR.                            */
        /* Use the Mersenne Twister for unifrom random number */
        /* generator (requires PRNG library).               */
        gen = unur_str2gen("beta(2,3) & method=tdr & urng = mt19937(1237)");

        /* It is important to check if the creation of the generator */
        /* object was successful. Otherwise 'gen' is the NULL pointer */
        /* and would cause a segmentation fault if used for sampling. */
        if (gen == NULL) {
```

```

        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Now you can use the generator object 'gen' to sample from
    /* the distribution. Eg.:
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* Now we want to switch to a different uniform random number
    /* generator.
    /* Now we use an ICG (Inversive Congruential Generator).
    urng2 = unur_urng_prng_new("icg(2147483647,1,1,0)");
    if (urng2 == NULL) exit (EXIT_FAILURE);

    /* Change uniform random number generator.
    /* Notice however that we should save the pointer to uniform
    /* random number generator in the generator object.
    urng1 = unur_chg_urng( gen, urng2 );

    /* ... and sample again.
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }

    /* When you do not need the generator object any more, you
    /* can destroy it.
    unur_free(gen);

    /* We also should destroy the uniform random number generators.*/
    unur_urng_free(urng1);
    unur_urng_free(urng2);

    exit (EXIT_SUCCESS);

#else
    printf("You must use the PRNG library to run this example!\n\n");
    exit (77);    /* exit code for automake check routines */
#endif

} /* end of main() */

/* ----- */

```

2.11 Sample pairs of antithetic random variates

Using Method TDR it is easy to sample pairs of antithetic random variates.

```

/* ----- */
/* File: example_anti.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* Example how to sample from two streams of antithetic random */
/* variates from Gaussian N(2,5) and Gamma(4) distribution, resp.*/

/* ----- */

/* This example makes use of the PRNG library (see */
/* http://statistik.wu-wien.ac.at/prng/) for generating */
/* uniform random numbers. */
/* To compile this example you must have set */
/* */
/* #define UNUR_URNG_TYPE UNUR_URNG_GENERIC */
/* #define UNURAN_HAS_PRNG 1 */
/* */
/* in 'src/unuran_config.h'. */
/* (Of course the executable has to be linked against the */
/* prng library.) */
/* ----- */

int main()
{
    #if UNUR_URNG_TYPE == UNUR_URNG_GENERIC && defined(UNURAN_HAS_PRNG)

        int i; /* loop variable */
        double xn, xg; /* will hold the random number */
        double fparams[2]; /* array for parameters for distribution */

        /* Declare the three UNURAN objects. */
        UNUR_DISTR *distr; /* distribution object */
        UNUR_PAR *par; /* parameter object */
        UNUR_GEN *gen_normal, *gen_gamma;
        /* generator objects */

        /* Declare objects for uniform random number generators. */
        UNUR_URNG *urng1, *urng2; /* uniform generator objects */

        /* PRNG only: */
        /* Make a object for uniform random number generator. */
        /* For details see http://statistik.wu-wien.ac.at/prng/. */

        /* The first generator: Gaussian N(2,5) */

        /* uniform generator: We use the Mersenne Twister. */
        urng1 = unur_urng_prng_new("mt19937(1237)");
        if (urng1 == NULL) exit (EXIT_FAILURE);

        /* UNURAN generator object for N(2,5) */
        fparams[0] = 2.;
        fparams[1] = 5.;
        distr = unur_distr_normal( fparams, 2 );

        /* Choose method TDR with variant PS. */
    
```

```

par = unur_tdr_new( distr );
unur_tdr_set_variant_ps( par );

/* Set uniform generator in parameter object. */
unur_set_urng( par, urng1 );

/* Set auxilliary uniform random number generator. */
/* We use the default generator. */
unur_use_urng_aux_default( par );

/* Alternatively you can create and use your own auxilliary */
/* uniform random number generator: */
/* UNUR_URNG *urng_aux; */
/* urng_aux = unur_urng_prng_new("tt800"); */
/* if (urng_aux == NULL) exit (EXIT_FAILURE); */
/* unur_set_urng_aux( par, urng_aux ); */

/* Create the generator object. */
gen_normal = unur_init(par);
if (gen_normal == NULL) {
    fprintf(stderr, "ERROR: cannot create generator object\n");
    exit (EXIT_FAILURE);
}

/* Destroy distribution object (gen_normal has its own copy). */
unur_distr_free(distr);

/* The second generator: Gamma(4) with antithetic variates. */

/* uniform generator: We use the Mersenne Twister. */
urng2 = unur_urng_prng_new("anti(mt19937(1237))");
if (urng2 == NULL) exit (EXIT_FAILURE);

/* UNURAN generator object for gamma(4) */
fparams[0] = 4.;
distr = unur_distr_gamma( fparams, 1 );

/* Choose method TDR with variant PS. */
par = unur_tdr_new( distr );
unur_tdr_set_variant_ps( par );

/* Set uniform generator in parameter object. */
unur_set_urng( par, urng2 );

/* Set auxilliary uniform random number generator. */
/* We use the default generator. */
unur_use_urng_aux_default( par );

/* Alternatively you can create and use your own auxilliary */
/* uniform random number generator (see above). */
/* Notice that both generator objects gen_normal and */
/* gen_gamma can share the same auxilliary URNG. */

/* Create the generator object. */
gen_gamma = unur_init(par);
if (gen_gamma == NULL) {
    fprintf(stderr, "ERROR: cannot create generator object\n");
    exit (EXIT_FAILURE);
}

/* Destroy distribution object (gen_normal has its own copy). */
unur_distr_free(distr);

```

```

/* Now we can sample pairs of negatively correlated random    */
/* variates. E.g.:                                           */
for (i=0; i<10; i++) {
    xn = unur_sample_cont(gen_normal);
    xg = unur_sample_cont(gen_gamma);
    printf("%g, %g\n",xn,xg);
}

/* When you do not need the generator objects any more, you   */
/* can destroy it.                                           */
unur_free(gen_normal);
unur_free(gen_gamma);

/* We also should destroy the uniform random number generators.*/
unur_urng_free(urng1);
unur_urng_free(urng2);

exit (EXIT_SUCCESS);

#else
    printf("You must use the PRNG library to run this example!\n\n");
    exit (77); /* exit code for automake check routines */
#endif

} /* end of main() */

/* ----- */

```

2.12 Sample pairs of antithetic random variates (String API)

Using Method TDR it is easy to sample pairs of antithetic random variates.

```

/* ----- */
/* File: example_anti_str.c */
/* ----- */
/* String API. */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* Example how to sample from two streams of antithetic random */
/* variates from Gaussian N(2,5) and Gamma(4) distribution, resp.*/

/* ----- */

/* This example makes use of the PRNG library (see */
/* http://statistik.wu-wien.ac.at/prng/) for generating */
/* uniform random numbers. */
/* To compile this example you must have set */
/* */
/* #define UNUR_URNG_TYPE UNUR_URNG_GENERIC */
/* #define UNURAN_HAS_PRNG 1 */
/* */
/* in 'src/unuran_config.h'. */
/* (Of course the executable has to be linked against the */
/* prng library.) */

/* It also works with necessary modifications with other uniform */
/* random number generators. */

/* ----- */

int main()
{
    #if UNUR_URNG_TYPE == UNUR_URNG_GENERIC && defined(UNURAN_HAS_PRNG)

        int i; /* loop variable */
        double xn, xg; /* will hold the random number */

        /* Declare UNURAN generator objects. */
        UNUR_GEN *gen_normal, *gen_gamma;

        /* PRNG only: */
        /* Make a object for uniform random number generator. */
        /* For details see http://statistik.wu-wien.ac.at/prng/. */

        /* Create the first generator: Gaussian N(2,5) */
        gen_normal = unur_str2gen("normal(2,5) & method=tdr; variant_ps & urng=mt19937(1237)");
        if (gen_normal == NULL) {
            fprintf(stderr, "ERROR: cannot create generator object\n");
            exit (EXIT_FAILURE);
        }
        /* Set auxilliary uniform random number generator. */
        /* We use the default generator. */
        unur_chgto_urng_aux_default(gen_normal);

        /* The second generator: Gamma(4) with antithetic variates. */
        gen_gamma = unur_str2gen("gamma(4) & method=tdr; variant_ps & urng=anti(mt19937(1237))");
        if (gen_gamma == NULL) {
            fprintf(stderr, "ERROR: cannot create generator object\n");
            exit (EXIT_FAILURE);
        }
    }
}

```

```

    }
    unur_chgto_urng_aux_default(gen_gamma);

    /* Now we can sample pairs of negatively correlated random    */
    /* variates. E.g.:                                           */
    for (i=0; i<10; i++) {
        xn = unur_sample_cont(gen_normal);
        xg = unur_sample_cont(gen_gamma);
        printf("%g, %g\n",xn,xg);
    }

    /* When you do not need the generator objects any more, you    */
    /* can destroy it.                                           */

    /* But first we have to destroy the uniform random number    */
    /* generators.                                               */
    unur_urng_free(unur_get_urng(gen_normal));
    unur_urng_free(unur_get_urng(gen_gamma));

    unur_free(gen_normal);
    unur_free(gen_gamma);

    exit (EXIT_SUCCESS);

#else
    printf("You must use the PRNG library to run this example!\n\n");
    exit (77);    /* exit code for automake check routines */
#endif

} /* end of main() */

/* ----- */

```


2.13 More examples

See [Section 5.3 \[Methods for continuous univariate distributions\]](#), page 88.

See [Section 5.4 \[Methods for continuous empirical univariate distributions\]](#), page 131.

See [Section 5.7 \[Methods for continuous empirical multivariate distributions\]](#), page 156.

See [Section 5.8 \[Methods for discrete univariate distributions\]](#), page 159.

3 String Interface

The string interface (string API) provided by the `unur_str2gen` call is the easiest way to use UNURAN. This function takes a character string as its argument. The string is parsed and the information obtained is used to create a generator object. It returns `NULL` if this fails, either due to a syntax error, or due to invalid data. In both cases `unur_error` is set to the corresponding error codes (see [Section 8.1 \[Error reporting\]](#), page 201). Additionally there exists the call `unur_str2distr` that only produces a distribution object.

Notice that the string interface does not implement all features of the UNURAN library. For trickier tasks it might be necessary to use the UNURAN calls.

In [Chapter 2 \[Examples\]](#), page 11, all examples are given using both the UNURAN standard API and this convenient string API. The corresponding program codes are equivalent.

Function reference

`UNUR_GEN* unur_str2gen (const char* string)`

Get a generator object for the distribution, method and uniform random number generator as described in the given *string*. See [Section 3.1 \[Syntax of String Interface\]](#), page 37, for details.

`UNUR_DISTR* unur_str2distr (const char* string)`

Get a distribution object for the distribution described in *string*. See [Section 3.1 \[Syntax of String Interface\]](#), page 37, and [Section 3.2 \[Distribution String\]](#), page 39, for details. However, only the block for the distribution object is allowed.

3.1 Syntax of String Interface

The given string holds information about the requested distribution and (optional) about the sampling method and the uniform random number generator invoked. The interpretation of the string is not case-sensitive, all white spaces are ignored.

The string consists of up to three blocks, separated by ampersands `&`.

Each block consists of `<key>=<value>` pairs, separated by semicolons `;`.

The first key in each block is used to indicate each block. We have three different blocks with the following (first) keys:

<code>distr</code>	definition of the distribution (see Section 3.2 [Distribution String] , page 39).
<code>method</code>	description of the transformation method (see Section 3.4 [Method String] , page 45).
<code>urng</code>	uniform random number generation (see Section 3.5 [Uniform RNG String] , page 52).

The `distr` block must be the very first block and is obligatory. All the other blocks are optional and can be arranged in arbitrary order.

For details see the following description of each block.

In the following example

```
distr = normal(3.,0.75); domain = (0,inf) & method = tdr; c = 0
```

we have a distribution block for the truncated normal distribution with mean 3 and standard deviation 0.75 on domain (0,infinity); and block for choosing method TDR with parameter `c` set to 0.

The `<key>=<value>` pairs that follow the first (initial) pair in each block are used to set parameters. The name of the parameter is given by the `<key>` string. It is deduced from the UNURAN set calls by taking the part after `..._set_`. The `<value>` string holds the parameters to be set, separated by commata `,`. There are three types of parameters:

string "..."

i.e. any sequence of characters enclosed by double quotes "...",

list (...,...)

i.e. list of *numbers*, separated by commata ,, enclosed in parenthesis (...), and

number

a sequence of characters that is not enclosed by quotes "..." or parenthesis (...). It is interpreted as float or integer depending on the type of the corresponding parameter.

The <value> string (including the character =) can be omitted when no argument is required.

At the moment not all **set** calls are supported. The syntax for the <value> can be directly derived from the corresponding **set** calls. To simplify the syntax additional shortcuts are possible. The following table lists the parameters for the **set** calls that are supported by the string interface; the entry in parenthesis gives the type of the argument as <value> string:

int (*number*):

The *number* is interpreted as an integer. **true** and **on** are transformed to 1, **false** and **off** are transformed to 0. A missing argument is interpreted as 1.

int, int (*number, number* or *list*):

The two numbers or the first two entries in the list are interpreted as a integers. **inf** and **-inf** are transformed to INT_MAX and INT_MIN respectively, i.e. the largest and smallest integers that can be represented by the computer.

unsigned (*number*):

The *number* is interpreted as an unsigned hexadecimal integer.

double (*number*):

The number is interpreted as a floating point number. **inf** is transformed to UNUR_INFINITY.

double, double (*number, number* or *list*):

The two numbers or the first two entries in the list are interpreted as a floating point numbers. **inf** is transformed to UNUR_INFINITY. However using **inf** in the list might not work for all versions of C. Then it is recommended to use two single numbers instead of a list.

int, double* ([*number,*] *list* or *number*):

- The list is interpreted as a double array. The (first) number as its length. If it is less than the actual size of the array only the first entries of the array are used.
- If only the list is given (i.e., if the first number is omitted), the first number is set to the actual size of the array.
- If only the number is given (i.e., if the list is omitted), the NULL pointer is used instead an array as argument.

double*, int (*list* [,*number*]):

The list is interpreted as a double array. The (second) number as its length. If the length is omitted, it is replaced by the actual size of the array. (Only in the distribution block!)

char* (*string*):

The character string is passed as is to the corresponding set call.

Notice that missing entries in a list of numbers are interpreted as 0. E.g, a the list (1,,3) is read as (1,0,3), the list (1,2,) as (1,2,0).

The the list of **key** strings in [Section 3.2.1 \[Keys for Distribution String\]](#), page 39, and [Section 3.4.1 \[Keys for Method String\]](#), page 45, for further details.

3.2 Distribution String

The `distr` block must be the very first block and is obligatory. For that reason the keyword `distr` is optional and can be omitted (together with the `=` character). Moreover it is ignored while parsing the string. However, to avoid some possible confusion it has to start with the letter `d` (if it is given at all).

The value of the `distr` key is used to get the distribution object, either via a `unur_distr_<value>` call for a standard distribution via a `unur_distr_<value>_new` call to get an object of a generic distribution. However not all generic distributions are supported yet.

The parameters for the standard distribution are given as a list. There must not be any character (other than white space) between the name of the standard distribution and the opening parenthesis (of this list. E.g., to get a beta distribution, use

```
distr = beta(2,4)
```

To get an object for a discrete distribution with probability vector (0.5,0.2,0.3), use

```
distr = discr; pv = (0.5,0.2,0.3)
```

It is also possible to set a PDF, PMF, or CDF using a string. E.g., to create a continuous distribution with PDF proportional to $\exp(-\sqrt{2+(x-1)^2} + (x-1))$ and domain (0,inf) use

```
distr = cont; pdf = "exp(-sqrt(2+(x-1)^2) + (x-1))"
```

Notice: If this string is used in an `unur_str2distr` or `unur_str2gen` call the double quotes " must be protected by `\`". Alternatively, single quotes may be used instead

```
distr = cont; pdf = 'exp(-sqrt(2+(x-1)^2) + (x-1))'
```

For the details of function strings see [Section 3.3 \[Function String\]](#), page 42.

3.2.1 Keys for Distribution String

List of standard distributions see [Chapter 7 \[Standard distributions\]](#), page 187

- `[distr =] beta(...)` \Rightarrow see [Section 7.1.1 \[beta\]](#), page 189
- `[distr =] binomial(...)` \Rightarrow see [Section 7.3.1 \[binomial\]](#), page 197
- `[distr =] cauchy(...)` \Rightarrow see [Section 7.1.2 \[cauchy\]](#), page 189
- `[distr =] chi(...)` \Rightarrow see [Section 7.1.3 \[chi\]](#), page 189
- `[distr =] chisquare(...)` \Rightarrow see [Section 7.1.4 \[chisquare\]](#), page 190
- `[distr =] exponential(...)` \Rightarrow see [Section 7.1.5 \[exponential\]](#), page 190
- `[distr =] extremeI(...)` \Rightarrow see [Section 7.1.6 \[extremeI\]](#), page 190
- `[distr =] extremeII(...)` \Rightarrow see [Section 7.1.7 \[extremeII\]](#), page 190
- `[distr =] gamma(...)` \Rightarrow see [Section 7.1.8 \[gamma\]](#), page 191
- `[distr =] geometric(...)` \Rightarrow see [Section 7.3.2 \[geometric\]](#), page 197
- `[distr =] hypergeometric(...)` \Rightarrow see [Section 7.3.3 \[hypergeometric\]](#), page 197
- `[distr =] laplace(...)` \Rightarrow see [Section 7.1.9 \[laplace\]](#), page 191
- `[distr =] logarithmic(...)` \Rightarrow see [Section 7.3.4 \[logarithmic\]](#), page 198
- `[distr =] logistic(...)` \Rightarrow see [Section 7.1.10 \[logistic\]](#), page 191
- `[distr =] lomax(...)` \Rightarrow see [Section 7.1.11 \[lomax\]](#), page 192
- `[distr =] negativebinomial(...)` \Rightarrow see [Section 7.3.5 \[negativebinomial\]](#), page 198
- `[distr =] normal(...)` \Rightarrow see [Section 7.1.12 \[normal\]](#), page 192
- `[distr =] pareto(...)` \Rightarrow see [Section 7.1.13 \[pareto\]](#), page 193
- `[distr =] poisson(...)` \Rightarrow see [Section 7.3.6 \[poisson\]](#), page 198
- `[distr =] powerexponential(...)` \Rightarrow see [Section 7.1.14 \[powerexponential\]](#), page 193
- `[distr =] rayleigh(...)` \Rightarrow see [Section 7.1.15 \[rayleigh\]](#), page 193
- `[distr =] student(...)` \Rightarrow see [Section 7.1.16 \[student\]](#), page 193

- [distr =] triangular(...) ⇒ see [Section 7.1.17 \[triangular\]](#), page 194
- [distr =] uniform(...) ⇒ see [Section 7.1.18 \[uniform\]](#), page 194
- [distr =] weibull(...) ⇒ see [Section 7.1.19 \[weibull\]](#), page 194

List of generic distributions see [Chapter 4 \[Handling Distribution Objects\]](#), page 53

- [distr =] cemp ⇒ see [Section 4.4 \[CEMP\]](#), page 67
- [distr =] cont ⇒ see [Section 4.2 \[CONT\]](#), page 57
- [distr =] discr ⇒ see [Section 4.9 \[DISCR\]](#), page 80

Notice: Order statistics for continuous distributions (see [Section 4.3 \[CORDER\]](#), page 64) are supported by using the key `orderstatistics` for distributions of type `CONT`.

List of keys that are available via the String API. For description see the corresponding UNURAN set calls.

- All distribution types

`name = "<string>"`
 ⇒ see [\[unur_distr_set_name\]](#), page 55

- `cemp` (*Distribution Type*) (see [Section 4.4 \[CEMP\]](#), page 67)

`data = (<list>) [, <int>]`
 ⇒ see [\[unur_distr_cemp_set_data\]](#), page 67

- `cont` (*Distribution Type*) (see [Section 4.2 \[CONT\]](#), page 57)

`cdf = "<string>"`
 ⇒ see [\[unur_distr_cont_set_cdfstr\]](#), page 59

`center = <double>`
 ⇒ see [\[unur_distr_cont_set_center\]](#), page 62

`domain = <double>, <double> | (<list>)`
 ⇒ see [\[unur_distr_cont_set_domain\]](#), page 60

`hr = "<string>"`
 ⇒ see [\[unur_distr_cont_set_hrstr\]](#), page 61

`logpdf = "<string>"`
 ⇒ see [\[unur_distr_cont_set_logpdfstr\]](#), page 60

`mode = <double>`
 ⇒ see [\[unur_distr_cont_set_mode\]](#), page 62

`pdf = "<string>"`
 ⇒ see [\[unur_distr_cont_set_pdfstr\]](#), page 59

`pdfarea = <double>`
 ⇒ see [\[unur_distr_cont_set_pdfarea\]](#), page 62

`pdfparams = (<list>) [, <int>]`
 ⇒ see [\[unur_distr_cont_set_pdfparams\]](#), page 59

`orderstatistics = <int>, <int> | (<list>)`

Make order statistics for given distribution. The first parameter gives the sample size, the second parameter its rank. (see see [\[unur_distr_corder_new\]](#), page 64)

- `discr` (*Distribution Type*) (see Section 4.9 [DISCR], page 80)

`cdf = "<string>"`

⇒ see [unur_distr_discr_set_cdfstr], page 81

`domain = <int>, <int> | (<list>)`

⇒ see [unur_distr_discr_set_domain], page 82

`mode [= <int>]`

⇒ see [unur_distr_discr_set_mode], page 83

`pmf = "<string>"`

⇒ see [unur_distr_discr_set_pmfstr], page 81

`pmfparams = (<list>) [, <int>]`

⇒ see [unur_distr_discr_set_pmfparams], page 82

`pmfsum = <double>`

⇒ see [unur_distr_discr_set_pmfsum], page 83

`pv = (<list>) [, <int>]`

⇒ see [unur_distr_discr_set_pv], page 80

3.3 Function String

In unuran it is also possible to define functions (e.g. CDF or PDF) as strings. As you can see in Example 2 (Section 2.6 [Example_2_str], page 20) it is very easy to define the PDF of a distribution object by means of a string. The possibilities using this string interface are more restricted than using a pointer to a routine coded in C (Section 2.5 [Example_2], page 18). But the differences in evaluation time is small. When a distribution object is defined using this string interface then of course the same conditions on the given density or CDF must be satisfied for a chosen method as for the standard API. This string interface can be used for both within the UNURAN string API using the `unur_str2gen` call, and for calls that define the density or CDF for a particular distribution object as done with (e.g.) the call `unur_distr_cont_set_pdfstr`. Here is an example for the latter case:

```
unur_distr_cont_set_pdfstr(distr, "1-x*x");
```

Syntax

The syntax for the function string is case insensitive, white spaces are ingnored. The expressions are similar to most programming languages and mathematical programs (see also the examples below). It is especially influenced by C. The usual preceedence rules are used (from highest to lowest preceedence: functions, power, multiplication, addition, relation operators). Use parentheses in case of doubt or when these preceedences should be changed.

Relation operators can be used as indicator functions, i.e. the term $(x>1)$ is evaluted as 1 if this relation is satisfied, and as 0 otherwise.

The first unknown symbol (letter or word) is interpreted as the variable of the function. It is recommended to use `x`. Only one variable can be used.

Important: The symbol `e` is used twice, for Euler's constant ($= 2.7182\dots$) and as exponent. The multiplication operator `*` must not be omitted, i.e. `2 x` is interpreted as the string `2x` (which will result in a syntax error).

List of symbols

Numbers

Numbers are composed using digits and, optionally, a sign, a decimal point, and an exponent indicated by `e`.

Symbol	Explanation	Examples
0...9	<i>digits</i>	2343
.	<i>decimal point</i>	165.567
-	<i>negative sign</i>	-465.223
e	<i>exponet</i>	13.2e-4 (=0.00132)

Constants

<code>pi</code>	$\pi = 3.1415\dots$	<code>3*pi+2</code>
<code>e</code>	<i>Euler's constant</i>	<code>3*e+2</code> (= 10.15...; do not cofuse with <code>3e2</code> = 300)
<code>inf</code>	<i>infinity</i>	(used for domains)

Special symbols

(<i>opening parenthesis</i>	<code>2*(3+x)</code>
)	<i>closing parenthesis</i>	<code>2*(3+x)</code>
,	<i>(argument) list separator</i>	<code>mod(13,2)</code>

Relation operators (Indicator functions)

<	<i>less than</i>	<code>(x<1)</code>
=	<i>equal</i>	<code>(2=x)</code>
==	<i>same as =</i>	<code>(x==3)</code>
>	<i>greater than</i>	<code>(x>0)</code>
<=	<i>less than or equal</i>	<code>(x<=1)</code>
!=	<i>not equal</i>	<code>(x!0)</code>
<>	<i>same as !=</i>	<code>(x<>pi)</code>
>=	<i>greater or equal</i>	<code>(x>=1)</code>

Arithmetic operators

+	<i>addition</i>	<code>2+x</code>
-	<i>subtraction</i>	<code>2-x</code>
*	<i>multiplication</i>	<code>2*x</code>
/	<i>division</i>	<code>x/2</code>
^	<i>power</i>	<code>x^2</code>

Functions

mod	<code>mod(m,n)</code> <i>remainder of division m over n</i>	<code>mod(x,2)</code>
exp	<i>exponential function (same as e^x)</i>	<code>exp(-x^2)</code> (same as $e^{(-x^2)}$)
log	<i>natural logarithm</i>	<code>log(x)</code>
sin	<i>sine</i>	<code>sin(x)</code>
cos	<i>cosine</i>	<code>cos(x)</code>
tan	<i>tangent</i>	<code>tan(x)</code>
sec	<i>secant</i>	<code>sec(x*2)</code>
sqrt	<i>square root</i>	<code>sqrt(2*x)</code>
abs	<i>absolute value</i>	<code>abs(x)</code>
sgn	<i>sign function</i>	<code>sign(x)*3</code>

Variable

x	<i>variable</i>	3*x^2
---	-----------------	-------

Examples

$$1.231+7.9876*x-1.234e-3*x^2+3.335e-5*x^3$$

$$\sin(2*\pi*x)+x^2$$

$$\exp(-((x-3)/2.1)^2)$$

It is also possible to define functions using different terms on separate domains. However, instead of constructs using `if ... then ... else ...` indicator functions are available.

For example to define the density of triangular distribution with domain (-1,1) and mode 0 use

$$(x>-1)*(x<0)*(1+x) + (x>=0)*(x<1)*(1-x)$$

3.4 Method String

The key `method` is obligatory, it must be the first key and its value is the name of a method suitable for the chosen standard distribution. E.g., if method AROU is chosen, use

```
method = arou
```

Of course the all following keys depend on the method chosen at first. All corresponding `set` calls of UNURAN are available and the key is the string after the `unur_<methodname>_set_` part of the command. E.g., UNURAN provides the command `unur_arou_set_max_sqhratio` to set a parameter of method AROU. To call this function via the string-interface, the key `max_sqhratio` can be used:

```
max_sqhratio = 0.9
```

Additionally the keyword `debug` can be used to set debugging flags (see [Chapter 9 \[Debugging\]](#), page 205, for details).

If this block is omitted, a suitable default method is used. Notice however that the default method may change in future versions of UNURAN.

3.4.1 Keys for Method String

List of methods and keys that are available via the String API. For description see the corresponding UNURAN set calls.

- `method = arou` ⇒ `unur_arou_new` (see [Section 5.3.1 \[AROU\]](#), page 92)

```
center = <double>
```

⇒ see [\[unur_arou_set_center\]](#), page 93

```
cpoints = <int> [, (<list>)] | (<list>)
```

⇒ see [\[unur_arou_set_cpoints\]](#), page 93

```
darsfactor = <double>
```

⇒ see [\[unur_arou_set_darsfactor\]](#), page 92

```
guidefactor = <double>
```

⇒ see [\[unur_arou_set_guidefactor\]](#), page 93

```
max_segments [= <int>]
```

⇒ see [\[unur_arou_set_max_segments\]](#), page 93

```
max_sqhratio = <double>
```

⇒ see [\[unur_arou_set_max_sqhratio\]](#), page 92

```
pedantic [= <int>]
```

⇒ see [\[unur_arou_set_pedantic\]](#), page 93

```
usecenter [= <int>]
```

⇒ see [\[unur_arou_set_usecenter\]](#), page 93

```
usedars [= <int>]
```

⇒ see [\[unur_arou_set_usedars\]](#), page 92

```
verify [= <int>]
```

⇒ see [\[unur_arou_set_verify\]](#), page 93

- `method = auto` ⇒ `unur_auto_new` (see [Section 5.2 \[AUTO\]](#), page 87)

```
logss [= <int>]
```

⇒ see [\[unur_auto_set_logss\]](#), page 87

- `method = cstd` ⇒ `unur_cstd_new` (see [Section 5.3.2 \[CSTD\]](#), page 95)

```
variant = <unsigned>
```

⇒ see [\[unur_cstd_set_variant\]](#), page 95

- `method = dari` \Rightarrow `unur_dari_new` (see Section 5.8.1 [DARI], page 162)
 - `cpfactor = <double>`
 - \Rightarrow see `[unur_dari_set_cpfactor]`, page 163
 - `squeeze [= <int>]`
 - \Rightarrow see `[unur_dari_set_squeeze]`, page 162
 - `tablesize [= <int>]`
 - \Rightarrow see `[unur_dari_set_tablesize]`, page 163
 - `verify [= <int>]`
 - \Rightarrow see `[unur_dari_set_verify]`, page 163
- `method = dau` \Rightarrow `unur_dau_new` (see Section 5.8.2 [DAU], page 165)
 - `urnfactor = <double>`
 - \Rightarrow see `[unur_dau_set_urnfactor]`, page 165
- `method = dgt` \Rightarrow `unur_dgt_new` (see Section 5.8.3 [DGT], page 166)
 - `guidefactor = <double>`
 - \Rightarrow see `[unur_dgt_set_guidefactor]`, page 166
 - `variant = <unsigned>`
 - \Rightarrow see `[unur_dgt_set_variant]`, page 167
- `method = dsrou` \Rightarrow `unur_dsrou_new` (see Section 5.8.4 [DSROU], page 168)
 - `cdfatmode = <double>`
 - \Rightarrow see `[unur_dsrou_set_cdfatmode]`, page 168
 - `verify [= <int>]`
 - \Rightarrow see `[unur_dsrou_set_verify]`, page 169
- `method = dstd` \Rightarrow `unur_dstd_new` (see Section 5.8.6 [DSTD], page 171)
 - `variant = <unsigned>`
 - \Rightarrow see `[unur_dstd_set_variant]`, page 171
- `method = empk` \Rightarrow `unur_empk_new` (see Section 5.4.1 [EMPK], page 134)
 - `beta = <double>`
 - \Rightarrow see `[unur_empk_set_beta]`, page 135
 - `kernel = <unsigned>`
 - \Rightarrow see `[unur_empk_set_kernel]`, page 135
 - `positive [= <int>]`
 - \Rightarrow see `[unur_empk_set_positive]`, page 136
 - `smoothing = <double>`
 - \Rightarrow see `[unur_empk_set_smoothing]`, page 136
 - `varcor [= <int>]`
 - \Rightarrow see `[unur_empk_set_varcor]`, page 136
- `method = gibbs` \Rightarrow `unur_gibbs_new` (see Section 5.6.1 [GIBBS], page 148)

- burnin [= <int>]
 ⇒ see [\[unur_gibbs_set_burnin\]](#), page 150
- c = <double>
 ⇒ see [\[unur_gibbs_set_c\]](#), page 149
- thinning [= <int>]
 ⇒ see [\[unur_gibbs_set_thinning\]](#), page 149
- variant_coordinate
 ⇒ see [\[unur_gibbs_set_variant_coordinate\]](#), page 149
- variant_random_direction
 ⇒ see [\[unur_gibbs_set_variant_random_direction\]](#), page 149
- method = hinv ⇒ [unur_hinv_new](#) (see Section 5.3.3 [HINV], page 97)
 - boundary = <double>, <double> | (<list>)
 ⇒ see [\[unur_hinv_set_boundary\]](#), page 98
 - cpoints = (<list>), <int>
 ⇒ see [\[unur_hinv_set_cpoints\]](#), page 98
 - guidefactor = <double>
 ⇒ see [\[unur_hinv_set_guidefactor\]](#), page 99
 - max_intervals [= <int>]
 ⇒ see [\[unur_hinv_set_max_intervals\]](#), page 99
 - order [= <int>]
 ⇒ see [\[unur_hinv_set_order\]](#), page 98
 - u_resolution = <double>
 ⇒ see [\[unur_hinv_set_u_resolution\]](#), page 98
- method = hitro ⇒ [unur_hitro_new](#) (see Section 5.6.2 [HITRO], page 151)
 - adaptive_multiplier = <double>
 ⇒ see [\[unur_hitro_set_adaptive_multiplier\]](#), page 154
 - burnin [= <int>]
 ⇒ see [\[unur_hitro_set_burnin\]](#), page 155
 - r = <double>
 ⇒ see [\[unur_hitro_set_r\]](#), page 154
 - thinning [= <int>]
 ⇒ see [\[unur_hitro_set_thinning\]](#), page 155
 - use_adaptiveline [= <int>]
 ⇒ see [\[unur_hitro_set_use_adaptiveline\]](#), page 153
 - use_adaptiverectangle [= <int>]
 ⇒ see [\[unur_hitro_set_use_adaptiverectangle\]](#), page 153
 - use_boundingrectangle [= <int>]
 ⇒ see [\[unur_hitro_set_use_boundingrectangle\]](#), page 153
 - v = <double>
 ⇒ see [\[unur_hitro_set_v\]](#), page 154
 - variant_coordinate
 ⇒ see [\[unur_hitro_set_variant_coordinate\]](#), page 153

variant_random_direction

⇒ see [\[unur_hitro_set_variant_random_direction\]](#), page 153

- method = hrb ⇒ `unur_hrb_new` (see Section 5.3.4 [HRB], page 100)

upperbound = `<double>`

⇒ see [\[unur_hrb_set_upperbound\]](#), page 100

verify [= `<int>`]

⇒ see [\[unur_hrb_set_verify\]](#), page 100

- method = hrd ⇒ `unur_hrd_new` (see Section 5.3.5 [HRD], page 101)

verify [= `<int>`]

⇒ see [\[unur_hrd_set_verify\]](#), page 101

- method = hri ⇒ `unur_hri_new` (see Section 5.3.6 [HRI], page 102)

p0 = `<double>`

⇒ see [\[unur_hri_set_p0\]](#), page 102

verify [= `<int>`]

⇒ see [\[unur_hri_set_verify\]](#), page 102

- method = ninv ⇒ `unur_ninv_new` (see Section 5.3.7 [NINV], page 103)

max_iter [= `<int>`]

⇒ see [\[unur_ninv_set_max_iter\]](#), page 104

start = `<double>`, `<double>` | (`<list>`)

⇒ see [\[unur_ninv_set_start\]](#), page 104

table [= `<int>`]

⇒ see [\[unur_ninv_set_table\]](#), page 104

usenewton

⇒ see [\[unur_ninv_set_usenewton\]](#), page 104

useregula

⇒ see [\[unur_ninv_set_useregula\]](#), page 104

x_resolution = `<double>`

⇒ see [\[unur_ninv_set_x_resolution\]](#), page 104

- method = nrou ⇒ `unur_nrou_new` (see Section 5.3.8 [NROU], page 106)

center = `<double>`

⇒ see [\[unur_nrou_set_center\]](#), page 107

r = `<double>`

⇒ see [\[unur_nrou_set_r\]](#), page 107

u = `<double>`, `<double>` | (`<list>`)

⇒ see [\[unur_nrou_set_u\]](#), page 107

v = `<double>`

⇒ see [\[unur_nrou_set_v\]](#), page 107

verify [= `<int>`]

⇒ see [\[unur_nrou_set_verify\]](#), page 107

- `method = srou` \Rightarrow `unur_srou_new` (see Section 5.3.9 [SROU], page 108)
 - `cdfatmode = <double>`
 \Rightarrow see `[unur_srou_set_cdfatmode]`, page 109
 - `pdfatmode = <double>`
 \Rightarrow see `[unur_srou_set_pdfatmode]`, page 109
 - `r = <double>`
 \Rightarrow see `[unur_srou_set_r]`, page 109
 - `usemirror [= <int>]`
 \Rightarrow see `[unur_srou_set_usemirror]`, page 110
 - `usesqueeze [= <int>]`
 \Rightarrow see `[unur_srou_set_usesqueeze]`, page 109
 - `verify [= <int>]`
 \Rightarrow see `[unur_srou_set_verify]`, page 110
- `method = ssr` \Rightarrow `unur_ssr_new` (see Section 5.3.10 [SSR], page 112)
 - `cdfatmode = <double>`
 \Rightarrow see `[unur_ssr_set_cdfatmode]`, page 113
 - `pdfatmode = <double>`
 \Rightarrow see `[unur_ssr_set_pdfatmode]`, page 113
 - `usesqueeze [= <int>]`
 \Rightarrow see `[unur_ssr_set_usesqueeze]`, page 113
 - `verify [= <int>]`
 \Rightarrow see `[unur_ssr_set_verify]`, page 113
- `method = tabl` \Rightarrow `unur_tabl_new` (see Section 5.3.11 [TABL], page 115)
 - `areafraction = <double>`
 \Rightarrow see `[unur_tabl_set_areafraction]`, page 116
 - `boundary = <double>, <double> | (<list>)`
 \Rightarrow see `[unur_tabl_set_boundary]`, page 118
 - `cpoints = <int> [, (<list>)] | (<list>)`
 \Rightarrow see `[unur_tabl_set_cpoints]`, page 116
 - `darsfactor = <double>`
 \Rightarrow see `[unur_tabl_set_darsfactor]`, page 117
 - `guidefactor = <double>`
 \Rightarrow see `[unur_tabl_set_guidefactor]`, page 118
 - `max_intervals [= <int>]`
 \Rightarrow see `[unur_tabl_set_max_intervals]`, page 118
 - `max_sqhratio = <double>`
 \Rightarrow see `[unur_tabl_set_max_sqhratio]`, page 117
 - `nstp [= <int>]`
 \Rightarrow see `[unur_tabl_set_nstp]`, page 116
 - `pedantic [= <int>]`
 \Rightarrow see `[unur_tabl_set_pedantic]`, page 119

slopes = (<list>), <int>

⇒ see [\[unur_tabl_set_slopes\]](#), page 118

usedars [= <int>]

⇒ see [\[unur_tabl_set_usedars\]](#), page 117

useear [= <int>]

⇒ see [\[unur_tabl_set_useear\]](#), page 116

variant_ia [= <int>]

⇒ see [\[unur_tabl_set_variant_ia\]](#), page 116

variant_splitmode = <unsigned>

⇒ see [\[unur_tabl_set_variant_splitmode\]](#), page 117

verify [= <int>]

⇒ see [\[unur_tabl_set_verify\]](#), page 119

- method = tdr ⇒ [unur_tdr_new](#) (see Section 5.3.12 [TDR], page 120)

c = <double>

⇒ see [\[unur_tdr_set_c\]](#), page 121

center = <double>

⇒ see [\[unur_tdr_set_center\]](#), page 123

cpoints = <int> [, (<list>)] | (<list>)

⇒ see [\[unur_tdr_set_cpoints\]](#), page 122

darsfactor = <double>

⇒ see [\[unur_tdr_set_darsfactor\]](#), page 122

guidefactor = <double>

⇒ see [\[unur_tdr_set_guidefactor\]](#), page 124

max_intervals [= <int>]

⇒ see [\[unur_tdr_set_max_intervals\]](#), page 123

max_sqratio = <double>

⇒ see [\[unur_tdr_set_max_sqratio\]](#), page 123

pedantic [= <int>]

⇒ see [\[unur_tdr_set_pedantic\]](#), page 124

reinit_ncpoints [= <int>]

⇒ see [\[unur_tdr_set_reinit_ncpoints\]](#), page 122

reinit_percentiles = <int> [, (<list>)] | (<list>)

⇒ see [\[unur_tdr_set_reinit_percentiles\]](#), page 122

usecenter [= <int>]

⇒ see [\[unur_tdr_set_usecenter\]](#), page 123

usedars [= <int>]

⇒ see [\[unur_tdr_set_usedars\]](#), page 121

usemode [= <int>]

⇒ see [\[unur_tdr_set_usemode\]](#), page 123

variant_gw

⇒ see [\[unur_tdr_set_variant_gw\]](#), page 121

- variant_ia
⇒ see [\[unur_tdr_set_variant_ia\]](#), page 121
- variant_ps
⇒ see [\[unur_tdr_set_variant_ps\]](#), page 121
- verify [= <int>]
⇒ see [\[unur_tdr_set_verify\]](#), page 124
- method = tdrwg ⇒ [unur_tdrwg_new](#) (see Section 5.3.13 [TDRGW], page 125)
 - cpoints = <int> [, (<list>)] | (<list>)
⇒ see [\[unur_tdrwg_set_cpoints\]](#), page 126
 - max_intervals [= <int>]
⇒ see [\[unur_tdrwg_set_max_intervals\]](#), page 126
 - pedantic [= <int>]
⇒ see [\[unur_tdrwg_set_pedantic\]](#), page 126
 - reinit_ncpoints [= <int>]
⇒ see [\[unur_tdrwg_set_reinit_ncpoints\]](#), page 126
 - reinit_percentiles = <int> [, (<list>)] | (<list>)
⇒ see [\[unur_tdrwg_set_reinit_percentiles\]](#), page 126
 - verify [= <int>]
⇒ see [\[unur_tdrwg_set_verify\]](#), page 126
- method = utdr ⇒ [unur_utdr_new](#) (see Section 5.3.14 [UTDR], page 128)
 - cpfactor = <double>
⇒ see [\[unur_utdr_set_cpfactor\]](#), page 129
 - deltafactor = <double>
⇒ see [\[unur_utdr_set_deltafactor\]](#), page 129
 - pdfatmode = <double>
⇒ see [\[unur_utdr_set_pdfatmode\]](#), page 128
 - verify [= <int>]
⇒ see [\[unur_utdr_set_verify\]](#), page 129
- method = vempk ⇒ [unur_vempk_new](#) (see Section 5.7.1 [VEMPK], page 158)
 - smoothing = <double>
⇒ see [\[unur_vempk_set_smoothing\]](#), page 158
 - varcor [= <int>]
⇒ see [\[unur_vempk_set_varcor\]](#), page 158
- method = vnrou ⇒ [unur_vnrou_new](#) (see Section 5.5.5 [VNROU], page 144)
 - r = <double>
⇒ see [\[unur_vnrou_set_r\]](#), page 145
 - v = <double>
⇒ see [\[unur_vnrou_set_v\]](#), page 145
 - verify [= <int>]
⇒ see [\[unur_vnrou_set_verify\]](#), page 145

3.5 Uniform RNG String

The value of the `urng` key is passed to the PRNG interface (see [PRNG manual](#) for details). However it only works when using the PRNG library is enabled, see [Section 1.2 \[Installation\]](#), [page 3](#) for details. There are no other keys.

IMPORTANT: UNURAN creates a new uniform random number generator for the generator object. The pointer to this uniform generator has to be read and saved via a `unur_get_urng` call in order to clear the memory *before* the UNURAN generator object is destroyed.

If this block is omitted the UNURAN default generator is used (which *must not* be destroyed).

4 Handling distribution objects

Objects of type `UNUR_DISTR` are used for handling distributions. All data about a distribution are stored in this object. UNURAN provides functions that return instances of such objects for standard distributions (see [Chapter 7 \[Standard distributions\], page 187](#)). It is then possible to change these distribution objects by various set calls. Moreover, it is possible to build a distribution object entirely from scratch. For this purpose there exists `unur_distr_<type>_new` calls that return an empty object of this type for each object type (eg. univariate continuous) which can be filled with the appropriate set calls.

UNURAN distinguishes between several types of distributions, each of which has its own sets of possible parameters (for details see the corresponding sections):

- continuous univariate distributions
- continuous univariate order statistics
- continuous empirical univariate distributions
- continuous multivariate distributions
- continuous empirical multivariate distributions
- matrix distributions
- discrete univariate distributions

Notice that there are essential data about a distribution, eg. the PDF, a list of (shape, scale, location) parameters for the distribution, and the domain of (the possibly truncated) distribution. And there exist parameters that are/can be derived from these, eg. the mode of the distribution or the area below the given PDF (which need not be normalized for many methods). UNURAN keeps track of parameters which are known. Thus if one of the essential parameters is changed all derived parameters are marked as unknown and must be set again if these are required for the chosen generation method. Additionally to set calls there are calls for updating derived parameters for objects provided by the UNURAN library of standard distributions (one for each parameter to avoid computational overhead since not all parameters are required for all generator methods).

All parameters of distribution objects can be read by corresponding get calls.

Every generator object has its own copy of a distribution object which is accessible by a `unur_get_distr` call. Thus the parameter for this distribution can be read. However, **never** extract the distribution object out of a generator object and run one of the set calls on it to modify the distribution. (How should the poor generator object know what has happened?) Instead there exist calls for each of the generator methods that change particular parameters of the internal copy of the distribution object.

How To Use

UNURAN collects all data required for a particular generation method in a *distribution object*. There are two ways to get an instance of a distributions object:

1. Build a distribution from scratch, by means of the corresponding `unur_distr_<type>_new` call, where `<type>` is the type of the distribution as listed in the below subsections.
2. Use the corresponding `unur_distr_<name>_new` call to get prebuild distribution from the UNURAN library of standard distributions. Here `<name>` is the name of the standard distribution in [Chapter 7 \[Standard distributions\], page 187](#).

In either cases the corresponding `unur_distr_<type>_set_<param>` calls to set the necessary parameters `<param>` (case 1), or change the values of the standard distribution in case 2 (if this makes sense for you). In the latter case `<type>` is the type to which the standard distribution

belongs to. These `set` calls return `UNUR_SUCCESS` when the corresponding parameter has been set successfully. Otherwise an error code is returned.

The parameters of a distribution are divided into *essential* and *derived* parameters.

Notice, that there are some restrictions in setting parameters to avoid possible confusions. Changing essential parameters marks derived parameters as `unknown`. Some of the parameters cannot be changed any more when already set; some parameters block each others. In such a case a new instance of a distribution object has to be build.

Additionally `unur_distr_<type>_upd_<param>` calls can be used for updating derived parameters for objects provided by the UNURAN library of standard distributions.

All parameters of a distribution object get be read by means of `unur_distr_<type>_get_<param>` calls.

Every distribution object be identified by its `name` which is a string of arbitrary characters provided by the user. For standard distribution it is automatically set to `<name>` in the corresponding `new` call. It can be changed to any other string.

4.1 Functions for all kinds of distribution objects

The calls in this section can be applied to all distribution objects.

- Destroy **free** an instance of a generator object.
- Ask for the **type** of a generator object.
- Ask for the **dimension** of a generator object.
- Deal with the **name** (identifier string) of a generator object.

Function reference

void unur_distr_free (*UNUR_DISTR* distribution*)

Destroy the *distribution* object.

int unur_distr_set_name (*UNUR_DISTR* distribution, const char* name*)

const char* unur_distr_get_name (*const UNUR_DISTR* distribution*)

Set and get *name* of *distribution*. The *name* can be an arbitrary character string. It can be used to identify generator objects for the user. It is used by UNURAN when printing information of the distribution object into a log files.

int unur_distr_get_dim (*const UNUR_DISTR* distribution*)

Get number of components of a random vector (its dimension) the *distribution*.

For univariate distributions it returns dimension 1.

For matrix distributions it returns the number of components (i.e., number of rows times number of columns). When the respective numbers of rows and columns are needed use **unur_distr_matr_get_dim** instead.

unsigned int unur_distr_get_type (*const UNUR_DISTR* distribution*)

Get type of *distribution*. Possible types are

UNUR_DISTR_CONT

univariate continuous distribution

UNUR_DISTR_CEMP

empirical continuous univariate distribution (i.e. a sample)

UNUR_DISTR_CVEC

continuous multivariate distribution

UNUR_DISTR_CVEMP

empirical continuous multivariate distribution (i.e. a vector sample)

UNUR_DISTR_DISCR

discrete univariate distribution

UNUR_DISTR_MATR

matrix distribution

Alternatively the **unur_distr_is_<TYPE>** calls can be used.

int unur_distr_is_cont (*const UNUR_DISTR* distribution*)

TRUE if *distribution* is a continuous univariate distribution.

int unur_distr_is_cvec (*const UNUR_DISTR* distribution*)

TRUE if *distribution* is a continuous multivariate distribution.

```
int unur_distr_is_cemp (const UNUR_DISTR* distribution)
    TRUE if distribution is an empirical continuous univariate distribution, i.e. a sample.

int unur_distr_is_cvemp (const UNUR_DISTR* distribution)
    TRUE if distribution is an empirical continuous multivariate distribution.

int unur_distr_is_discr (const UNUR_DISTR* distribution)
    TRUE if distribution is a discrete univariate distribution.

int unur_distr_is_matr (const UNUR_DISTR* distribution)
    TRUE if distribution is a matrix distribution.
```

4.2 Continuous univariate distributions

The calls in this section can be applied to continuous univariate distributions.

- Create a **new** instance of a continuous univariate distribution.
- Handle and evaluate distribution function (CDF, **cdf**), probability density function (PDF, **pdf**) and the derivative of the density function (**dpdf**). The following is important:
 - . **pdf** need not be normalized, i.e., any integrable nonnegative function can be used.
 - . **dpdf** must be the derivative of the function provided as **pdf**.
 - . **cdf** must be a distribution function, i.e. it must be monotonically increasing with range $[0,1]$.
 - . If **cdf** and **pdf** are used together for a particular generation method, then **pdf** must be the derivative of the **cdf**, i.e., it must be normalized.
- Handle and evaluate the logarithm of the probability density function (**logPDF**, **logpdf**) and the derivative of the logarithm of the density function (**dlogpdf**).

Some methods use the logarithm of the density if available.

- Set (and change) parameters (**pdfparams**) and the area below the graph (**pdfarea**) of the given density.
- Set the **mode** of the distribution.
- Set the **center** of the distribution. It is used by some generation methods to adjust the parameters of the generation algorithms to gain better performance. It can be seen as the location of the “central part” of the distribution.
- Some generation methods require the hazard rate (**hr**) of the distribution instead of its **pdf**.
- Alternatively, **cdf**, **pdf**, **dpdf**, and **hr** can be provided as **strings** instead of function pointers.
- Set the **domain** of the distribution. Notice that the library also can handle truncated distributions, i.e., distributions that are derived from (standard) distributions by simply restricting its domain to a subset. However, there is a subtle difference between changing the domain of a distribution object by a **unur_distr_cont_set_domain** call and changing the (truncated) domain for an existing generator object. The domain of the distribution object is used to create the generator object with hats, squeezes, tables, etc. Whereas truncating the domain of an existing generator object need not necessarily require a recomputation of these data. Thus by a **unur_<method>_chg_truncated** call (if available) the sampling region is restricted to the subset of the domain of the given distribution object. However, generation methods that require a recreation of the generator object when the domain is changed have a **unur_<method>_chg_domain** call instead. For these calls there are of course no restrictions on the given domain (i.e., it is possible to increase the domain of the distribution) (see [Chapter 5 \[Methods\]](#), page 85, for details).

Function reference

UNUR_DISTR* **unur_distr_cont_new** (*void*)

Create a new (empty) object for univariate continuous distribution.

Essential parameters

```
int unur_distr_cont_set_pdf (UNUR_DISTR* distribution,
                             UNUR_FUNCT_CONT* pdf)
int unur_distr_cont_set_dpdf (UNUR_DISTR* distribution,
                              UNUR_FUNCT_CONT* dpdf)
```

```
int unur_distr_cont_set_cdf (UNUR_DISTR* distribution,
                           UNUR_FUNCT_CONT* cdf)
```

Set respective pointer to the probability density function (PDF), the derivative of the probability density function (dPDF) and the cumulative distribution function (CDF) of the *distribution*. Each of these function pointers must be of type `double funct(double x, const UNUR_DISTR *distr)`.

Due to the fact that some of the methods do not require a normalized PDF the following is important:

- The given CDF must be the cumulative distribution function of the (non-truncated) distribution. If a distribution from the UNURAN library of standard distributions (see [Chapter 7 \[Standard distributions\], page 187](#)) is truncated, there is no need to change the CDF.
- If both the CDF and the PDF are used (for a method or for order statistics), the PDF must be the derivative of the CDF. If a truncated distribution for one of the standard distributions from the UNURAN library of standard distributions is used, there is no need to change the PDF.
- If the area below the PDF is required for a given distribution it must be given by the `unur_distr_cont_set_pdfarea` call. For a truncated distribution this must be of course the integral of the PDF in the given truncated domain. For distributions from the UNURAN library of standard distributions this is done automatically by the `unur_distr_cont_upd_pdfarea` call.

It is important to note that all these functions must return a result for all values of *x*. Eg., if the domain of a given PDF is the interval $[-1,1]$, then the given function must return 0.0 for all points outside this interval. In case of an overflow the PDF should return `UNUR_INFINITY`.

It is not possible to change such a function. Once the PDF or CDF is set it cannot be overwritten. This also holds when the logPDF is given or when the PDF is given by the `unur_distr_cont_set_pdfstr` or `unur_distr_cont_set_logpdfstr` call. A new distribution object has to be used instead.

```
UNUR_FUNCT_CONT* unur_distr_cont_get_pdf (const UNUR_DISTR*
                                           distribution)
```

```
UNUR_FUNCT_CONT* unur_distr_cont_get_dpdf (const UNUR_DISTR*
                                             distribution)
```

```
UNUR_FUNCT_CONT* unur_distr_cont_get_cdf (const UNUR_DISTR*
                                           distribution)
```

Get the respective pointer to the PDF, the derivative of the PDF and the CDF of the *distribution*. The pointer is of type `double funct(double x, const UNUR_DISTR *distr)`. If the corresponding function is not available for the distribution, the NULL pointer is returned.

```
double unur_distr_cont_eval_pdf (double x, const UNUR_DISTR*
                                 distribution)
```

```
double unur_distr_cont_eval_dpdf (double x, const UNUR_DISTR*
                                  distribution)
```

```
double unur_distr_cont_eval_cdf (double x, const UNUR_DISTR*
                                 distribution)
```

Evaluate the PDF, derivative of the PDF and the CDF, respectively, at *x*. Notice that *distribution* must not be the NULL pointer. If the corresponding function is not available for the distribution, `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

IMPORTANT: In the case of a truncated standard distribution these calls always return the respective values of the *untruncated* distribution!


```

int unur_distr_cont_set_logpdf (UNUR_DISTR* distribution,
                               UNUR_FUNCT_CONT* logpdf)
int unur_distr_cont_set_dlogpdf (UNUR_DISTR* distribution,
                                 UNUR_FUNCT_CONT* dlogpdf)
UNUR_FUNCT_CONT* unur_distr_cont_get_logpdf (const UNUR_DISTR*
                                             distribution)
UNUR_FUNCT_CONT* unur_distr_cont_get_dlogpdf (const UNUR_DISTR*
                                              distribution)
double unur_distr_cont_eval_logpdf (double x, const UNUR_DISTR*
                                    distribution)
double unur_distr_cont_eval_dlogpdf (double x, const UNUR_DISTR*
                                    distribution)

```

Analogous calls for the logarithm of the density function.

```

int unur_distr_cont_set_pdfstr (UNUR_DISTR* distribution, const char*
                               pdfstr)

```

This function provides an alternative way to set a PDF and its derivative of the *distribution*. *pdfstr* is a character string that contains the formula for the PDF, see [Section 3.3 \[Function String\]](#), page 42, for details. See also the remarks for the `unur_distr_cont_set_pdf` call.

It is not possible to call this function twice or to call this function after a `unur_distr_cont_set_pdf` call.

```

int unur_distr_cont_set_cdfstr (UNUR_DISTR* distribution, const char*
                               cdfstr)

```

This function provides an alternative way to set a CDF; analogously to the `unur_distr_cont_set_pdfstr` call.

```

char* unur_distr_cont_get_pdfstr (const UNUR_DISTR* distribution)
char* unur_distr_cont_get_dpdpstr (const UNUR_DISTR* distribution)
char* unur_distr_cont_get_cdfstr (const UNUR_DISTR* distribution)

```

Get pointer to respective string for PDF, derivative of PDF, and CDF of *distribution* that is given as string (instead of a function pointer). This call allocates memory to produce this string. It should be freed when it is not used any more.

```

int unur_distr_cont_set_pdfparams (UNUR_DISTR* distribution, const
                                   double* params, int n_params)

```

Sets array of parameters for *distribution*. There is an upper limit for the number of parameters `n_params`. It is given by the macro `UNUR_DISTR_MAXPARAMS` in ‘`unuran_config.h`’. (It is set to 5 by default but can be changed to any appropriate nonnegative number.) If `n_params` is negative or exceeds this limit no parameters are copied into the distribution object and `unur_errno` is set to `UNUR_ERR_DISTR_NPARAMS`.

For standard distributions from the UNURAN library the parameters are checked. Moreover, the domain is updated automatically unless it has been changed before by a `unur_distr_cont_set_domain` call. If the given parameters are invalid for the standard distribution, then no parameters are set and an error code is returned. Notice that the given parameter list for such a distribution is handled in the same way as in the corresponding `new` calls, i.e. optional parameters for the PDF that are not present in the given list are (re-)set to their default values.

```
int unur_distr_cont_get_pdfparams (const UNUR_DISTR* distribution, const
    double** params)
```

Get number of parameters of the PDF and set pointer *params* to array of parameters. If no parameters are stored in the object, an error code is returned and *params* is set to NULL.

Important: Do **not** change the entries in *params*!

```
int unur_distr_cont_set_pdfparams_vec (UNUR_DISTR* distribution, int
    par, const double* param_vec, int n_param_vec)
```

This function provides an interface for additional vector parameters for a continuous *distribution*.

It sets the parameter with number *par*. *par* indicates directly which of the parameters is set and must be a number between 0 and UNUR_DISTR_MAXPARAMS-1 (the upper limit of possible parameters defined in 'unuran_config.h'; it is set to 5 but can be changed to any appropriate nonnegative number.)

The entries of a this parameter are given by the array *param_vec* of size *n_param_vec*.

If *param_vec* is NULL then the corresponding entry is cleared.

If an error occurs no parameters are copied into the parameter object *unur_errno* is set to UNUR_ERR_DISTR_DATA.

```
int unur_distr_cont_get_pdfparams_vec (const UNUR_DISTR* distribution,
    int par, const double** param_vecs)
```

Get parameter of the PDF with number *par*. The pointer to the parameter array is stored in *param_vecs*, its size is returned by the function. If the requested parameter is not set, then an error code is returned and *params* is set to NULL.

Important: Do **not** change the entries in *param_vecs*!

```
int unur_distr_cont_set_logpdfstr (UNUR_DISTR* distribution, const
    char* logpdfstr)
```

```
char* unur_distr_cont_get_logpdfstr (const UNUR_DISTR* distribution)
char* unur_distr_cont_get_dlogpdfstr (const UNUR_DISTR* distribution)
```

Analogous calls for the logarithm of the density function.

```
int unur_distr_cont_set_domain (UNUR_DISTR* distribution, double left,
    double right)
```

Set the left and right borders of the domain of the distribution. This can also be used to truncate an existing distribution. For setting the boundary to $\pm\infty$ use \pm UNUR_INFINITY. If *right* is not strictly greater than *left* no domain is set and *unur_errno* is set to UNUR_ERR_DISTR_SET.

Important: For some technical reasons it is assumed that the density is unimodal and thus monotone on either side of the mode! This is used in the case when the given mode is outside of the original domain. Then the mode is set to the corresponding boundary of the new domain.

```
int unur_distr_cont_get_domain (const UNUR_DISTR* distribution, double*
    left, double* right)
```

Get the left and right borders of the domain of the distribution. If the domain is not set \pm UNUR_INFINITY is assumed and returned. No error is reported in this case.

```
int unur_distr_cont_get_truncated (const UNUR_DISTR* distribution,
                                   double* left, double* right)
```

Get the left and right borders of the (truncated) domain of the distribution. For non-truncated distribution this call is equivalent to the `unur_distr_cont_get_domain` call.

This call is only useful in connection with a `unur_get_distr` call to get the boundaries of the sampling region of a generator object.

```
int unur_distr_cont_set_hr (UNUR_DISTR* distribution,
                           UNUR_FUNCT_CONT* hazard)
```

Set pointer to the hazard rate (HR) of the *distribution*.

The *hazard rate* (or failure rate) is a mathematical way of describing aging. If the lifetime X is a random variable with density $f(x)$ and CDF $F(x)$ the hazard rate $h(x)$ is defined as $h(x) = f(x) / (1-F(x))$. In other words, $h(x)$ represents the (conditional) rate of failure of a unit that has survived up to time x with probability $1-F(x)$. The key distribution is the exponential distribution as it has constant hazard rate of value 1. Hazard rates tending to infinity describe distributions with sub-exponential tails whereas distributions with hazard rates tending to zero have heavier tails than the exponential distribution.

It is important to note that all these functions must return a result for all floats x . In case of an overflow the PDF should return `UNUR_INFINITY`.

Important: Do not simply use $f(x) / (1-F(x))$, since this is numerically very unstable and results in numerical noise if $F(x)$ is (very) close to 1. Moreover, if the density $f(x)$ is known a generation method that uses the density is more appropriate.

It is not possible to change such a function. Once the HR is set it cannot be overwritten. This also holds when the HR is given by the `unur_distr_cont_set_hrstr` call. A new distribution object has to be used instead.

```
UNUR_FUNCT_CONT* unur_distr_cont_get_hr (const UNUR_DISTR*
                                           distribution)
```

Get the pointer to the hazard rate of the *distribution*. The pointer is of type `double funct(double x, const UNUR_DISTR *distr)`. If the corresponding function is not available for the distribution, the NULL pointer is returned.

```
double unur_distr_cont_eval_hr (double x, const UNUR_DISTR* distribution)
```

Evaluate the hazard rate at x . Notice that *distribution* must not be the NULL pointer. If the corresponding function is not available for the distribution, `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

```
int unur_distr_cont_set_hrstr (UNUR_DISTR* distribution, const char*
                               hrstr)
```

This function provides an alternative way to set a hazard rate and its derivative of the *distribution*. *hrstr* is a character string that contains the formula for the HR, see [Section 3.3 \[Function String\]](#), page 42, for details. See also the remarks for the `unur_distr_cont_set_hr` call.

It is not possible to call this function twice or to call this function after a `unur_distr_cont_set_hr` call.

```
char* unur_distr_cont_get_hrstr (const UNUR_DISTR* distribution)
```

Get pointer to string for HR of *distribution* that is given via the string interface. This call allocates memory to produce this string. It should be freed when it is not used any more.

Derived parameters

The following parameters **must** be set whenever one of the essential parameters has been set or changed (and the parameter is required for the chosen method).

`int unur_distr_cont_set_mode (UNUR_DISTR* distribution, double mode)`

Set mode of *distribution*. The *mode* must be contained in the domain of *distribution*. Otherwise the mode is not set and `unur_errno` is set to `UNUR_ERR_DISTR_SET`. Notice that the mode is adjusted when the domain is set, see the remark for the `unur_distr_cont_set_domain` call.

`int unur_distr_cont_upd_mode (UNUR_DISTR* distribution)`

Recompute the mode of the *distribution*. This call works properly for distribution objects from the UNURAN library of standard distributions when the corresponding function is available. Otherwise a (slow) numerical mode finder based on Brent's algorithm is used. If it fails `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

`double unur_distr_cont_get_mode (UNUR_DISTR* distribution)`

Get mode of *distribution*. If the mode is not marked as known, `unur_distr_cont_upd_mode` is called to compute the mode. If this is not successful `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`. (There is no difference between the case where no routine for computing the mode is available and the case where no mode exists for the distribution at all.)

`int unur_distr_cont_set_center (UNUR_DISTR* distribution, double center)`

Set center of the *distribution*. The center is used by some methods to shift the distribution in order to decrease numerical round-off error. If not given explicitly a default is used.

Important: This call does not check whether the center is contained in the given domain. Similarly `unur_distr_cont_set_domain` does not adjust the center properly.

Default: The mode, if set by a `unur_distr_cont_set_mode` or `unur_distr_cont_upd_mode` call; otherwise 0.

`double unur_distr_cont_get_center (const UNUR_DISTR* distribution)`

Get center of the *distribution*. It always returns some point as there always exists a default for the center, see `unur_distr_cont_set_center`.

`int unur_distr_cont_set_pdfarea (UNUR_DISTR* distribution, double area)`

Set the area below the PDF. If *area* is non-positive, no area is set and `unur_errno` is set to `UNUR_ERR_DISTR_SET`.

For a distribution object created by the UNURAN library of standard distributions you always should use the `unur_distr_cont_upd_pdfarea`. Otherwise there might be ambiguous side-effects.

`int unur_distr_cont_upd_pdfarea (UNUR_DISTR* distribution)`

Recompute the area below the PDF of the distribution. It only works for distribution objects from the UNURAN library of standard distributions when the corresponding function is available. Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

This call sets the normalization constant such that the given PDF is the derivative of a given CDF, i.e. the area is 1. However, for truncated distributions the area is smaller than 1.

The call does not work for distributions from the UNURAN library of standard distributions with truncated domain when the CDF is not available.

```
double unur_distr_cont_get_pdfarea (UNUR_DISTR* distribution)
```

Get the area below the PDF of the distribution. If this area is not known, `unur_distr_cont_upd_pdfarea` is called to compute it. If this is not successful `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`.

4.3 Continuous univariate order statistics

These are special cases of a continuous univariate distributions and thus they have most of these parameters (with the exception that functions cannot be changed). Additionally,

- there is a call to extract the underlying distribution,
- and a call to handle the **rank** of the order statistics.

Function reference

`UNUR_DISTR* unur_distr_corder_new (const UNUR_DISTR* distribution, int n, int k)`

Create an object for order statistics of sample size *n* and rank *k*. *distribution* must be a pointer to a univariate continuous distribution. The resulting generator object is of the same type as of a `unur_distr_cont_new` call. (However, it cannot be used to make an order statistics out of an order statistics.)

To have a PDF for the order statistics, the given distribution object must contain a CDF and a PDF. Moreover, it is assumed that the given PDF is the derivative of the given CDF. Otherwise the area below the PDF of the order statistics is not computed correctly.

Important: There is no warning when the computed area below the PDF of the order statistics is wrong.

`const UNUR_DISTR* unur_distr_corder_get_distribution (const UNUR_DISTR* distribution)`

Get pointer to distribution object for underlying distribution.

Essential parameters

`int unur_distr_corder_set_rank (UNUR_DISTR* distribution, int n, int k)`

Change sample size *n* and rank *k* of order statistics. In case of invalid data, no parameters are changed. The area below the PDF can be set to that of the underlying distribution by a `unur_distr_corder_upd_pdfarea` call.

`int unur_distr_corder_get_rank (const UNUR_DISTR* distribution, int* n, int* k)`

Get sample size *n* and rank *k* of order statistics. In case of error an error code is returned.

Additionally most of the set and get calls for continuous univariate distributions work. The most important exceptions are that the PDF and CDF cannot be changed and `unur_distr_cont_upd_mode` uses in any way a (slow) numerical method that might fail.

`UNUR_FUNCT_CONT* unur_distr_corder_get_pdf (UNUR_DISTR* distribution)`

`UNUR_FUNCT_CONT* unur_distr_corder_get_dpdf (UNUR_DISTR* distribution)`

`UNUR_FUNCT_CONT* unur_distr_corder_get_cdf (UNUR_DISTR* distribution)`

Get the respective pointer to the PDF, the derivative of the PDF and the CDF of the distribution, respectively. The pointer is of type `double funct(double x, UNUR_DISTR *distr)`. If the corresponding function is not available for the distribution, the NULL pointer is returned. See also `unur_distr_cont_get_pdf`. (Macro)

`double unur_distr_corder_eval_pdf (double x, UNUR_DISTR* distribution)`

`double unur_distr_corder_eval_dpdf (double x, UNUR_DISTR* distribution)`

`double unur_distr_corder_eval_cdf (double x, UNUR_DISTR* distribution)`

Evaluate the PDF, derivative of the PDF. and the CDF, respectively, at *x*. Notice that *distribution* must not be the NULL pointer. If the corresponding function is not available for

the distribution, `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. See also `unur_distr_cont_eval_pdf`. (Macro)

IMPORTANT: In the case of a truncated standard distribution these calls always return the respective values of the *untruncated* distribution!

```
int unir_distr_corder_set_pdfparams (UNUR_DISTR* distribution, double*
                                     params, int n_params)
```

Set array of parameters for underlying distribution. See `unur_distr_cont_set_pdfparams` for details. (Macro)

```
int unir_distr_corder_get_pdfparams (UNUR_DISTR* distribution, double**
                                     params)
```

Get number of parameters of the PDF of the underlying distribution and set pointer *params* to array of parameters. See `unur_distr_cont_get_pdfparams` for details. (Macro)

```
int unir_distr_corder_set_domain (UNUR_DISTR* distribution, double
                                  left, double right)
```

Set the left and right borders of the domain of the distribution. See `unur_distr_cont_set_domain` for details. (Macro)

```
int unir_distr_corder_get_domain (UNUR_DISTR* distribution, double*
                                  left, double* right)
```

Get the left and right borders of the domain of the distribution. See `unur_distr_cont_get_domain` for details. (Macro)

```
int unir_distr_corder_get_truncated (UNUR_DISTR* distribution, double*
                                     left, double* right)
```

Get the left and right borders of the (truncated) domain of the distribution. See `unur_distr_cont_get_truncated` for details. (Macro)

Derived parameters

The following parameters **must** be set whenever one of the essential parameters has been set or changed (and the parameter is required for the chosen method).

```
int unir_distr_corder_set_mode (UNUR_DISTR* distribution, double mode)
```

Set mode of distribution. See also `unur_distr_corder_set_mode`. (Macro)

```
double unir_distr_corder_upd_mode (UNUR_DISTR* distribution)
```

Recompute the mode of the distribution numerically. Notice that this routine is slow and might not work properly in every case. See also `unur_distr_cont_upd_mode` for further details. (Macro)

```
double unir_distr_corder_get_mode (UNUR_DISTR* distribution)
```

Get mode of distribution. See `unur_distr_cont_get_mode` for details. (Macro)

```
int unir_distr_corder_set_pdfarea (UNUR_DISTR* distribution, double
                                   area)
```

Set the area below the PDF. See `unur_distr_cont_set_pdfarea` for details. (Macro)

`double unur_distr_corder_upd_pdfarea (UNUR_DISTR* distribution)`

Recompute the area below the PDF of the distribution. It only works for order statistics for distribution objects from the UNURAN library of standard distributions when the corresponding function is available. `unur_distr_cont_upd_pdfarea` assumes that the PDF of the underlying distribution is normalized, i.e. it is the derivative of its CDF. Otherwise the computed area is wrong and there is **no** warning about this failure. See `unur_distr_cont_upd_pdfarea` for further details. (Macro)

`double unur_distr_corder_get_pdfarea (UNUR_DISTR* distribution)`

Get the area below the PDF of the distribution. See `unur_distr_cont_get_pdfarea` for details. (Macro)

4.4 Continuous empirical univariate distributions

Empirical univariate distributions are just lists of numbers. Thus there are only calls to insert these data. How these data are used to sample from the empirical distribution depends from the chosen generation method.

Function reference

`UNUR_DISTR* unur_distr_cemp_new (void)`

Create a new (empty) object for empirical univariate continuous distribution.

Essential parameters

`int unur_distr_cemp_set_data (UNUR_DISTR* distribution, const double* sample, int n_sample)`

Set observed sample for empirical distribution.

`int unur_distr_cemp_read_data (UNUR_DISTR* distribution, const char* filename)`

Read data from file '*filename*'. It reads the first number from each line. Numbers are parsed by means of the C standard routine `strtod`. Lines that do not start with `+`, `-`, `.`, or a digit are ignored. (Beware of lines starting with a blank!)

In case of an error (file cannot be opened, invalid string for double in line) no data are copied into the distribution object and an error code is returned.

`int unur_distr_cemp_get_data (const UNUR_DISTR* distribution, const double** sample)`

Get number of samples and set pointer *sample* to array of observations. If no sample has been given, an error code is returned and *sample* is set to `NULL`.

Important: Do **not** change the entries in *sample*!

4.5 Continuous multivariate distributions

The following calls handle multivariate distributions. However, the requirements of particular generation methods is not as unique as for univariate distributions. Moreover, random vector generation methods are still under development. The below functions are a first attempt to handle this situation.

Notice that some of the parameters – when given carelessly – might contradict to others. For example: Some methods require the marginal distribution and some methods need a standardized form of the marginal distributions, where the actual mean and variance is stored in the mean vector and the covariance matrix, respectively.

We also have to mention that some methods might abuse some of the parameters. For example, method VMT (see [Section 5.5.4 \[VMT\]](#), [page 143](#)) uses standard marginal distributions. However, the marginal distribution of the generated vectors might be transformed and thus differ from the given marginal distributions. Please read the description of the chosen sampling method carefully.

The following kind of calls exists:

- Create a **new** instance of a continuous multivariate distribution;
- Handle and evaluate probability density function (PDF, **pdf**) and the gradient of the density function (**dpdf**). The following is important:
 - . **pdf** need not be normalized, i.e., any integrable nonnegative function can be used.
 - . **dpdf** must be the derivate of the function provided as **pdf**.
- Handle and evaluate the logarithm of the probability density function (logPDF, **logpdf**) and the gradient of the logarithm of the density function (**dlogpdf**).
Some methods use the logarithm of the density if available.
- Set (and change) parameters (**pdfparams**) and the volume below the graph (**pdfvol**) of the given density.
- Set **mode** and **mean** of the distribution.
- Set the **center** of the distribution. It is used by some generation methods to adjust the parameters of the generation algorithms to gain better performance. It can be seen as the location of the “central part” of the distribution.
- Handle the **covariance** matrix of the distribution and its **cholesky** and **inverse** matrices.
- Set the **rankcorrelation** matrix of the distribution.
- Deal with **marginal** distributions.

Function reference

UNUR_DISTR* `unur_distr_cvec_new (int dim)`

Create a new (empty) object for multivariate continuous distribution. *dim* is the number of components of the random vector (i.e. its dimension). It is also possible to use dimension 1. Notice, however, that this is treated as a distribution of random vectors with only one component and not as a distribution of real numbers. For the latter `unur_distr_cont_new` should be used to create an object for a univariate distribution.

Essential parameters

int `unur_distr_cvec_set_pdf (UNUR_DISTR* distribution,
UNUR_FUNCT_CVEC* pdf)`

Set respective pointer to the PDF of the *distribution*. This function must be of type `double funct(const double *x, const UNUR_DISTR *distr)`, where *x* must be a pointer to a double array of appropriate size (i.e. of the same size as given to the `unur_distr_cvec_new` call).

It is not necessary that the given PDF is normalized, i.e. the integral need not be 1. Nevertheless the volume below the PDF can be provided by a `unur_distr_cvec_set_pdfvol` call.

It is not possible to change the PDF. Once the PDF is set it cannot be overwritten. This also holds when the logPDF is given. A new distribution object has to be used instead.

```
int unur_distr_cvec_set_dpdf (UNUR_DISTR* distribution,
                             UNUR_VFUNCT_CVEC* dpdf)
```

Set pointer to the gradient of the PDF. The type of this function must be `int funct(double *result, const double *x, const UNUR_DISTR *distr)`, where *result* and *x* must be pointers to double arrays of appropriate size (i.e. of the same size as given to the `unur_distr_cvec_new` call). The gradient of the PDF is stored in the array *result*. The function should return an error code in case of an error and must return `UNUR_SUCCESS` otherwise.

The given function must be the gradient of the function given by a `unur_distr_cvec_set_pdf` call.

It is not possible to change the gradient of the PDF. Once the dPDF is set it cannot be overwritten. This also holds when the gradient of the logPDF is given. A new distribution object has to be used instead.

```
int unur_distr_cvec_set_pdpdf (UNUR_DISTR* distribution,
                              UNUR_FUNCTD_CVEC* pdpdf)
```

Set pointer to partial derivatives of the PDF. The type of this function must be `double funct(const double *x, int coord, const UNUR_DISTR *distr)`, where *x* must be a pointer to a double array of appropriate size (i.e. of the same size as given to the `unur_distr_cvec_new` call). *coord* is the coordinate for which the partial derivative should be computed.

Notice that *coord* must be an integer from $\{0, \dots, \text{dim}-1\}$.

It is not possible to change the partial derivative of the PDF. Once the pdPDF is set it cannot be overwritten. This also holds when the partial derivative of the logPDF is given. A new distribution object has to be used instead.

```
UNUR_FUNCT_CVEC* unur_distr_cvec_get_pdf (const UNUR_DISTR*
                                           distribution)
```

Get the pointer to the PDF of the *distribution*. The pointer is of type `double funct(const double *x, const UNUR_DISTR *distr)`. If the corresponding function is not available for the *distribution*, the NULL pointer is returned.

```
UNUR_VFUNCT_CVEC* unur_distr_cvec_get_dpdf (const UNUR_DISTR*
                                              distribution)
```

Get the pointer to the gradient of the PDF of the *distribution*. The pointer is of type `int double funct(double *result, const double *x, const UNUR_DISTR *distr)`. If the corresponding function is not available for the *distribution*, the NULL pointer is returned.

```
double unur_distr_cvec_eval_pdf (const double* x, UNUR_DISTR*
                                 distribution)
```

Evaluate the PDF of the *distribution* at *x*. *x* must be a pointer to a double array of appropriate size (i.e. of the same size as given to the `unur_distr_cvec_new` call) that contains the vector for which the function has to be evaluated.

Notice that *distribution* must not be the NULL pointer. If the corresponding function is not available for the *distribution*, `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

```
int unur_distr_cvec_eval_dpdpf (double* result, const double* x,
    UNUR_DISTR* distribution)
```

Evaluate the gradient of the PDF of the *distribution* at *x*. The result is stored in the double array *result*. Both *result* and *x* must be pointer to double arrays of appropriate size (i.e. of the same size as given to the `unur_distr_cvec_new` call).

Notice that *distribution* must not be the NULL pointer. If the corresponding function is not available for the *distribution*, an error code is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA` (*result* is left unmodified).

```
double unur_distr_cvec_eval_pdpdf (const double* x, int coord,
    UNUR_DISTR* distribution)
```

Evaluate the partial derivative of the PDF of the *distribution* at *x* for the coordinate *coord*. *x* must be a pointer to a double array of appropriate size (i.e. of the same size as given to the `unur_distr_cvec_new` call) that contains the vector for which the function has to be evaluated.

Notice that *coord* must be an integer from $\{0, \dots, \text{dim}-1\}$.

Notice that *distribution* must not be the NULL pointer. If the corresponding function is not available for the *distribution*, `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

```
int unur_distr_cvec_set_logpdf (UNUR_DISTR* distribution,
    UNUR_FUNCT_CVEC* logpdf)
int unur_distr_cvec_set_dlogpdf (UNUR_DISTR* distribution,
    UNUR_VFUNCT_CVEC* dlogpdf)
int unur_distr_cvec_set_pdlogpdf (UNUR_DISTR* distribution,
    UNUR_FUNCTD_CVEC* pdlogpdf)
UNUR_FUNCT_CVEC* unur_distr_cvec_get_logpdf (const UNUR_DISTR*
    distribution)
UNUR_VFUNCT_CVEC* unur_distr_cvec_get_dlogpdf (const UNUR_DISTR*
    distribution)
double unur_distr_cvec_eval_logpdf (const double* x, UNUR_DISTR*
    distribution)
int unur_distr_cvec_eval_dlogpdf (double* result, const double* x,
    UNUR_DISTR* distribution)
double unur_distr_cvec_eval_pdlogpdf (const double* x, int coord,
    UNUR_DISTR* distribution)
```

Analogous calls for the logarithm of the density function.

```
int unur_distr_cvec_set_mean (UNUR_DISTR* distribution, const double*
    mean)
```

Set mean vector for multivariate *distribution*. *mean* must be a pointer to an array of size `dim`, where `dim` is the dimension returned by `unur_distr_get_dim`. A NULL pointer for *mean* is interpreted as the zero vector $(0, \dots, 0)$.

```
const double* unur_distr_cvec_get_mean (const UNUR_DISTR* distribution)
```

Get the mean vector of the *distribution*. The function returns a pointer to an array of size `dim`. If the mean vector is not marked as known the NULL pointer is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`.

Important: Do **not** modify the array that holds the mean vector!

```
int unur_distr_cvec_set_covar (UNUR_DISTR* distribution, const double*
                             covar)
```

Set covariance matrix for multivariate *distribution*. *covar* must be a pointer to an array of size `dim` x `dim`, where `dim` is the dimension returned by `unur_distr_get_dim`. The rows of the matrix have to be stored consecutively in this array.

covar must be a variance-covariance matrix of the *distribution*, i.e. it must be symmetric and positive definit and its diagonal entries (i.e. the variance of the components of the random vector) must be strictly positive. The Cholesky factor is computed (and stored) to verify the positive definiteness condition. Notice that the inverse of the given covariance matrix is automatically computed when it is requested by some routine. Notice that the computation of this inverse matrix is unstable in case of high correlations and/or high dimensions. Thus it might fail and methods that require this inverse cannot be used. As an alternative the inverse of the covariance matrix can be directly set by a `unur_distr_cvec_set_covar_inv` call.

A NULL pointer for *covar* is interpreted as the identity matrix.

Important: This entry is abused in some methods which do not require the covariance matrix. It is then used to perform some transformation to obtain better performance.

Important: In case of an error (e.g. because *covar* is not a valid covariance matrix) an error code is returned. Moreover, the covariance matrix is not set and is marked as unknown. A previously set covariance matrix is then no longer available.

Remark: UNU.RAN does not check whether the an eventually set covariance matrix and a rank-correlation matrix do not contradict each other.

```
int unur_distr_cvec_set_covar_inv (UNUR_DISTR* distribution, const
                                  double* covar_inv)
```

Set inverse of the covariance matrix for multivariate *distribution*. *covar_inv* must be a pointer to an array of size `dim` x `dim`, where `dim` is the dimension returned by `unur_distr_get_dim`. The rows of the matrix have to be stored consecutively in this array.

covar_inv must be symmetric and positive definit. Only the symmetry of the matrix is checked.

A NULL pointer for *covar_inv* is interpreted as the identity matrix.

Important: In case of an error (because *covar_inv* is not symetric) an error code is returned. Moreover, the inverse of the covariance matrix is not set and is marked as unknown. A previously set inverse matrix is then no longer available.

Remark: UNU.RAN does not check whether the given matrix is positive definit.

Remark: UNU.RAN does not check whether the matrix *covar_inv* is the inverse of the eventually set covariance matrix.

```
const double* unur_distr_cvec_get_covar (const UNUR_DISTR*
                                         distribution)
const double* unur_distr_cvec_get_cholesky (const UNUR_DISTR*
                                             distribution)
const double* unur_distr_cvec_get_covar_inv (UNUR_DISTR*
                                              distribution)
```

Get covariance matrix of *distribution*, its Cholesky factor, and its inverse, respectively. The function returns a pointer to an array of size `dim` x `dim`. The rows of the matrix are stored consecutively in this array. If the requested matrix is not marked as known the NULL pointer is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`.

Important: Do **not** modify the array that holds the covariance matrix!

Remark: The inverse of the covariance matrix is computed if it is not already stored.

```
int unur_distr_cvec_set_rankcorr (UNUR_DISTR* distribution, const
    double* rankcorr)
```

Set rank-correlation matrix (Spearman's correlation) for multivariate *distribution*. *rankcorr* must be a pointer to an array of size *dim* x *dim*, where *dim* is the dimension returned by `unur_distr_get_dim`. The rows of the matrix have to be stored consecutively in this array.

rankcorr must be a rank-correlation matrix of the *distribution*, i.e. it must be symmetric and positive definite and its diagonal entries must be equal to 1.

The Cholesky factor is computed (and stored) to verify the positive definiteness condition.

A NULL pointer for *rankcorr* is interpreted as the identity matrix.

Important: In case of an error (e.g. because *rankcorr* is not a valid rank-correlation matrix) an error code is returned. Moreover, the rank-correlation matrix is not set and is marked as unknown. A previously set rank-correlation matrix is then no longer available.

Remark: UNU.RAN does not check whether the an eventually set covariance matrix and a rank-correlation matrix do not contradict each other.

```
const double* unur_distr_cvec_get_rankcorr (const UNUR_DISTR*
    distribution)
```

```
const double* unur_distr_cvec_get_rk_cholesky (const UNUR_DISTR*
    distribution)
```

Get rank-correlation matrix and its cholesky factor, respectively, of *distribution*. The function returns a pointer to an array of size *dim* x *dim*. The rows of the matrix are stored consecutively in this array. If the requested matrix is not marked as known the NULL pointer is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`.

Important: Do **not** modify the array that holds the rank-correlation matrix!

```
int unur_distr_cvec_set_marginals (UNUR_DISTR* distribution,
    UNUR_DISTR* marginal)
```

```
int unur_distr_cvec_set_stdmarginals (UNUR_DISTR* distribution,
    UNUR_DISTR* marginal)
```

Sets marginal distribution and standardized marginal distributions of the given *distribution* to the same *marginal* distribution object. The *marginal* distribution must be an instance of a continuous univariate distribution object. In conjunction with `unur_distr_cvec_set_covar` and `unur_distr_cvec_set_mean` the standardized marginals must be used, i.e., they should have mean 0 and standard deviation 1 (if both exist for the given marginal distribution). Notice that the marginal distribution is copied into the *distribution* object.

```
int unur_distr_cvec_set_marginal_array (UNUR_DISTR* distribution,
    UNUR_DISTR** marginals)
```

```
int unur_distr_cvec_set_stdmarginal_array (UNUR_DISTR* distribution,
    UNUR_DISTR** marginals)
```

Analogously to the above `unur_distr_cvec_set_marginals` and `unur_distr_cvec_set_stdmarginals` calls. However, now an array *marginals* of the pointers to each of the marginal distributions must be given. It **must** be an array of size *dim*, where *dim* is the dimension returned by `unur_distr_get_dim`. *Notice:* Local copies for each of the entries are stored in the *distribution* object. If some of these entries are identical (i.e. contain the same pointer), then for each of these a new copy is made.

```
int unur_distr_cvec_set_marginal_list (UNUR_DISTR* distribution, ...)
```



```
int unur_distr_cvec_set_stdmarginal_list (UNUR_DISTR* distribution,
    ...)
```

Similar to the above `unur_distr_cvec_set_marginal_array` and `unur_distr_cvec_set_stdmarginal_array` calls. However, now the pointers to the particular marginal distributions can be given as parameter and does not require an array of pointers. Additionally the given distribution objects are immediately destroyed. Thus calls like `unur_distr_normal` can be used as arguments. (With `unur_distr_cvec_set_marginal_array` the result of such call has to be stored in a pointer since it has to be freed afterwards to avoid memory leaks!)

If one of the given pointer to marginal distributions is the `NULL` pointer then the marginal distributions of *distribution* are not set (or previous settings are not changed) and an error code is returned.

Important: All distribution objects given in the argument list are destroyed!

```
const UNUR_DISTR* unur_distr_cvec_get_marginal (const UNUR_DISTR*
    distribution, int n)
const UNUR_DISTR* unur_distr_cvec_get_stdmarginal (const UNUR_DISTR*
    distribution, int n)
```

Get pointer to the *n*-th (standardized) marginal distribution object from the given multivariate *distribution*. If this does not exist, `NULL` is returned. The marginal distributions are enumerated from 1 to `dim`, where `dim` is the dimension returned by `unur_distr_get_dim`.

```
int unur_distr_cvec_set_pdfparams (UNUR_DISTR* distribution, const
    double* params, int n_params)
```

Sets array of parameters for *distribution*. There is an upper limit for the number of parameters `n_params`. It is given by the macro `UNUR_DISTR_MAXPARAMS` in ‘`unuran_config.h`’. (It is set to 5 by default but can be changed to any appropriate nonnegative number.) If *n_params* is negative or exceeds this limit no parameters are copied into the distribution object and `unur_errno` is set to `UNUR_ERR_DISTR_NPARAMS`.

For standard distributions from the UNURAN library the parameters are checked. Moreover, the domain is updated automatically unless it has been changed before by a `unur_distr_cont_set_domain` call. If the given parameters are invalid for the standard distribution, then no parameters are set and an error code is returned. Notice that the given parameter list for such a distribution is handled in the same way as in the corresponding `new` calls, i.e. optional parameters for the PDF that are not present in the given list are (re-)set to their default values.

```
int unur_distr_cvec_get_pdfparams (const UNUR_DISTR* distribution, const
    double** params)
```

Get number of parameters of the PDF and set pointer *params* to array of parameters. If no parameters are stored in the object, an error code is returned and *params* is set to `NULL`.

Important: Do **not** change the entries in *params*!

```
int unur_distr_cvec_set_pdfparams_vec (UNUR_DISTR* distribution, int
    par, const double* param_vec, int n_params)
```

This function provides an interface for additional vector parameters for a multivariate *distribution* besides mean vector and covariance matrix which have their own calls.

It sets the parameter with number *par*. *par* indicates directly which of the parameters is set and must be a number between 0 and `UNUR_DISTR_MAXPARAMS-1` (the upper limit of possible parameters defined in ‘`unuran_config.h`’; it is set to 5 but can be changed to any appropriate nonnegative number.)

The entries of a this parameter are given by the array *param_vec* of size *n_params*. Notice that using this interface an $(n \times m)$ -matrix has to be stored in an array of length *n_params* = *n* times *m*; where the rows of the matrix are stored consecutively in this array.

Due to great variety of possible parameters for a multivariate *distribution* there is no simpler interface.

If *param_vec* is NULL then the corresponding entry is cleared.

If an error occurs no parameters are copied into the parameter object *unur_errno* is set to UNUR_ERR_DISTR_DATA.

```
int unur_distr_cvec_get_pdfparams_vec (const UNUR_DISTR* distribution,
                                       int par, const double** param_vecs)
```

Get parameter of the PDF with number *par*. The pointer to the parameter array is stored in *param_vecs*, its size is returned by the function. If the requested parameter is not set, then an error code is returned and *params* is set to NULL.

Important: Do **not** change the entries in *param_vecs*!

Derived parameters

The following paramters **must** be set whenever one of the essential parameters has been set or changed (and the parameter is required for the chosen method).

```
int unur_distr_cvec_set_mode (UNUR_DISTR* distribution, const double*
                             mode)
```

Set mode of the *distribution*. *mode* must be a pointer to an array of the size returned by *unur_distr_get_dim*. A NULL pointer for *mode* is interpreted as the zero vector (0,...,0).

```
const double* unur_distr_cvec_get_mode (const UNUR_DISTR* distribution)
```

Get mode of the *distribution*. The function returns a pointer to an array of the size returned by *unur_distr_get_dim*. If the mode is not marked as known the NULL pointer is returned and *unur_errno* is set to UNUR_ERR_DISTR_GET. (There is no difference between the case where no routine for computing the mode is available and the case where no mode exists for the *distribution* at all.)

Important: Do **not** modify the array that holds the mode!

```
int unur_distr_cvec_set_center (UNUR_DISTR* distribution, const double*
                               center)
```

Set center of the *distribution*. *center* must be a pointer to an array of the size returned by *unur_distr_get_dim*. A NULL pointer for *center* is interpreted as the zero vector (0,...,0).

The center is used by some methods to shift the distribution in order to decrease numerical round-off error. If not given explicitly a default is used.

Default: The mode, if given by a *unur_distr_cvec_set_mode* call; else the mean, if given by a *unur_distr_cvec_set_mean* call; otherwise the null vector (0,...,0).

```
const double* unur_distr_cvec_get_center (UNUR_DISTR* distribution)
```

Get center of the *distribution*. The function returns a pointer to an array of the size returned by *unur_distr_get_dim*. It always returns some point as there always exists a default for the center, see *unur_distr_cvec_set_center*. *Important:* Do **not** modify the array that holds the center!


```
int unur_distr_cvec_set_pdfvol (UNUR_DISTR* distribution, double  
                                volume)
```

Set the volume below the PDF. If *vol* is non-positive, no volume is set and `unur_errno` is set to `UNUR_ERR_DISTR_SET`.

```
double unur_distr_cvec_get_pdfvol (const UNUR_DISTR* distribution)
```

Get the volume below the PDF of the *distribution*. If this volume is not known, `unur_distr_cont_upd_pdfarea` is called to compute it. If this is not successful `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`.

4.6 Continuous univariate full conditional distribution

Full conditional distribution for a given continuous multivariate distribution. The condition is a position vector and either a variable that is varied or a vector that indicates the direction on which the random vector can variate.

There is a subtle difference between using direction vector and using the k -th variable. When a direction vector is given the PDF of the conditional distribution is defined by $f(t) = PDF(pos + t \cdot dir)$. When a variable is selected the full conditional distribution with all other variables fixed is used.

This is a special case of a continuous univariate distribution and thus they have most of these parameters (with the exception that functions cannot be changed). Additionally,

- there is a call to extract the underlying multivariate distribution,
- and a call to handle the variables that are fixed and the direction for changing the random vector.

This distribution type is primarily used for evaluation the conditional distribution and its derivative (as required for, e.g., the Gibbs sampler). The density is not normalized (i.e. does not integrate to one). Mode and area are not available and it does not make sense to use any call to set or change parameters except the ones given below.

Function reference

UNUR_DISTR* `unur_distr_condi_new` (*const UNUR_DISTR** *distribution*, *const double** *pos*, *const double** *dir*, *int* *k*)

Create an object for full conditional distribution for the given *distribution*. The condition is given by a position vector *pos* and either the k -th variable that is varied or the vector *dir* that contains the direction on which the random vector can variate.

distribution must be a pointer to a multivariate continuous distribution. *pos* must be a pointer to an array of size *dim*, where *dim* is the dimension of the underlying distribution object. *dir* must be a pointer to an array if size *dim* or NULL. *k* must be in the range 0, ..., *dim*-1. If the k -th variable is used, *dir* must be set to NULL.

Notice: There is a subtle difference between using direction vector *dir* and using the k -th variable. When *dir* is given, the current position *pos* is mapped into 0 of the conditional distribution and the derivative is taken from the function $PDF(pos+t*dir)$ w.r.t. t . On the other hand, when the coordinate k is used (i.e., when *dir* is set to NULL), the full conditional distribution of the distribution is considered (as used for the Gibbs sampler). In particular, the current point is just projected into the one-dimensional subspace without mapping it into the point 0.

Notice: If a coordinate k is used, then the k -th partial derivative is used if it is available. Otherwise the gradient is computed and the k -th component is returned.

The resulting generator object is of the same type as of a `unur_distr_cont_new` call.

int `unur_distr_condi_set_condition` (*struct unur_distr** *distribution*, *const double** *pos*, *const double** *dir*, *int* *k*)

Set/change condition for conditional *distribution*. Change values of fixed variables to *pos* and use direction *dir* or k -th variable of conditional *distribution*.

pos must be a pointer to an array of size *dim*, where *dim* is the dimension of the underlying distribution object. *dir* must be a pointer to an array if size *dim* or NULL. *k* must be in the range 0, ..., *dim*-1. If the k -th variable is used, *dir* must be set to NULL.

Notice: There is a subtle difference between using direction vector *dir* and using the k -th variable. When *dir* is given, the current position *pos* is mapped into 0 of the conditional

distribution and the derivative is taken from the function $\text{PDF}(\text{pos}+t*\text{dir})$ w.r.t. t . On the other hand, when the coordinate k is used (i.e., when dir is set to `NULL`), the full conditional distribution of the distribution is considered (as used for the Gibbs sampler). In particular, the current point is just projected into the one-dimensional subspace without mapping it into the point 0.

```
int unur_distr_condi_get_condition (struct unur_distr* distribution,
                                   const double** pos, const double** dir, int* k)
```

Get condition for conditional *distribution*. The values for the fixed variables are stored in *pos*, which must be a pointer to an array of size `dim`. The condition is stored in *dir* and *k*, respectively.

Important: Do **not** change the entries in *pos* and *dir*!

```
const UNUR_DISTR* unur_distr_condi_get_distribution (const UNUR_DISTR*
                                                    distribution)
```

Get pointer to distribution object for underlying distribution.

4.7 Continuous empirical multivariate distributions

Empirical multivariate distributions are just lists of vectors (with the same dimension). Thus there are only calls to insert these data. How these data are used to sample from the empirical distribution depends from the chosen generation method.

Function reference

`UNUR_DISTR* unur_distr_cvemp_new (int dim)`

Create a new (empty) object for an empirical multivariate continuous distribution. *dim* is the number of components of the random vector (i.e. its dimension). It must be at least 2; otherwise `unur_distr_cemp_new` should be used to create an object for an empirical univariate distribution.

Essential parameters

`int unur_distr_cvemp_set_data (UNUR_DISTR* distribution, const double* sample, int n_sample)`

Set observed sample for empirical *distribution*. *sample* is an array of doubles of size *dim* x *n_sample*, where *dim* is the dimension of the *distribution* returned by `unur_distr_get_dim`. The data points must be stored consecutively in *sample*.

`int unur_distr_cvemp_read_data (UNUR_DISTR* distribution, const char* filename)`

Read data from file '*filename*'. It reads the first *dim* numbers from each line, where *dim* is the dimension of the *distribution* returned by `unur_distr_get_dim`. Numbers are parsed by means of the C standard routine `strtod`. Lines that do not start with +, -, ., or a digit are ignored. (Beware of lines starting with a blank!)

In case of an error (file cannot be opened, too few entries in a line, invalid string for double in line) no data are copied into the distribution object and an error code is returned.

`int unur_distr_cvemp_get_data (const UNUR_DISTR* distribution, const double** sample)`

Get number of samples and set pointer *sample* to array of observations. If no sample has been given, an error code is returned and *sample* is set to NULL. If successful *sample* points to an array of length *dim* x *n_sample*, where *dim* is the dimension of the distribution returned by `unur_distr_get_dim` and *n_sample* the return value of the function.

Important: Do **not** modify the array *sample*.

4.8 MATRix distributions

Distributions for random matrices. Notice that UNURAN uses arrays of `doubles` to handle matrices. The rows of the matrix are stored consecutively.

Function reference

`UNUR_DISTR* unur_distr_matr_new (int n_rows, int n_cols)`

Create a new (empty) object for a matrix distribution. *n_rows* and *n_cols* are the respective numbers of rows and columns of the random matrix (i.e. its dimensions). It is also possible to have only one number for rows and/or columns. Notice, however, that this is treated as a distribution of random matrices with only one row or column or component and not as a distribution of vectors or real numbers. For the latter `unur_distr_cont_new` or `unur_distr_cvec_new` should be used to create an object for a univariate distribution and a multivariate (vector) distribution, respectively.

Essential parameters

`int unur_distr_matr_get_dim (const UNUR_DISTR* distribution, int* n_rows, int* n_cols)`

Get number of rows and columns of random matrix (its dimension). It returns the total number of components. If successful `UNUR_SUCCESS` is returned.

4.9 Discrete univariate distributions

The calls in this section can be applied to discrete univariate distributions.

- Create a **new** instance of a discrete univariate distribution.
- Handle and evaluate distribution function (CDF, **cdf**) and probability mass function (PMF, **pmf**). The following is important:
 - . **pmf** need not be normalized, i.e., any summable nonnegative function on the set of integers can be used.
 - . **cdf** must be a distribution function, i.e. it must be monotonically increasing with range $[0,1]$.
 - . If **cdf** and **pdf** are used together for a particular generation method, then **pmf** must be normalized, i.e. it must sum to 1.
- Alternatively, **cdf** and **pdf** can be provided as **strings** instead of function pointers.
- Some generation methods require a (finite) probability vector (PV, **pv**), i.e. an array of **doubles**. It can be automatically computed if the **pmf** is given but **pv** is not.
- Set (and change) parameters (**pmfparams**) and the total sum (**pmfsum**) of the given PMF or PV.
- Set the **mode** of the distribution.
- Set the **domain** of the distribution.

Function reference

UNUR_DISTR* **unur_distr_discr_new** (*void*)

Create a new (empty) object for a univariate discrete distribution.

Essential parameters

There are two interfaces for discrete univariate distributions: Either provide a (finite) probability vector (PV). Or provide a probability mass function (PMF). For the latter case there are also a couple of derived parameters that are not required when a PV is given.

It is not possible to set both a PMF and a PV directly. However, the PV can be computed from the PMF (or the CDF if no PMF is available) by means of a **unur_distr_discr_make_pv** call. If both the PV and the PMF are given in the distribution object it depends on the generation method which of these is used.

int **unur_distr_discr_set_pv** (*UNUR_DISTR** *distribution*, *const double** *pv*, *int* *n_pv*)

Set finite probability vector (PV) for the *distribution*. It is not necessary that the entries in the given PV sum to 1. *n_pv* must be positive. However, there is no testing whether all entries in *pv* are non-negative.

If no domain has been set, then the left boundary is set to 0, by default. If *n_pv* is too large, e.g. because left boundary + *n_pv* exceeds the range of integers, then the call fails.

Notice that it is not possible to set both a PV and a PMF. (E.g., it is not possible to set a PV for a *distribution* from UNURAN library of standard distributions.)

int **unur_distr_discr_make_pv** (*UNUR_DISTR** *distribution*)

Compute a PV when a PMF is given. However, when the domain is not given or is too large and the sum over the PMF is given then the (right) tail of the *distribution* is chopped off such that the probability for the tail region is less than 1.e-8. If the sum over the PMF is not given a PV of maximal length is computed.

The maximal size of the created PV is bounded by the macro `UNUR_MAX_AUTO_PV` that is defined in `'unuran_config.h'`.

If successful, the length of the generated PV is returned. If the sum over the PMF on the chopped tail is not negligible small (i.e. greater than 1.e-8 or unknown) than the negative of the length of the PV is returned and `unur_errno` is set to `UNUR_ERR_DISTR_SET`.

Notice that the left boundary of the PV is set to 0 by default when a discrete distribution object is created from scratch.

If computing a PV fails for some reasons, an error code is returned and `unur_errno` is set to `UNUR_ERR_DISTR_SET`.

```
int unur_distr_discr_get_pv (const UNUR_DISTR* distribution, const
                             double** pv)
```

Get length of PV of the *distribution* and set pointer *pv* to array of probabilities. If no PV is given, an error code is returned and *pv* is set to NULL.

(It does not call `unur_distr_discr_make_pv` !)

```
int unur_distr_discr_set_pmf (UNUR_DISTR* distribution,
                              UNUR_FUNCT_DISCR* pmf)
```

```
int unur_distr_discr_set_cdf (UNUR_DISTR* distribution,
                              UNUR_FUNCT_DISCR* cdf)
```

Set respective pointer to the PMF and the CDF of the *distribution*. These functions must be of type `double funct(int k, const UNUR_DISTR *distr)`.

It is important to note that all these functions must return a result for all integers *k*. E.g., if the domain of a given PMF is the interval $\{1,2,3,\dots,100\}$, than the given function must return 0.0 for all points outside this interval.

The default domain for the PMF or CDF is $[0, \text{INT_MAX}]$. The domain can be changed using a `unur_distr_discr_set_domain` call.

It is not possible to change such a function. Once the PMF or CDF is set it cannot be overwritten. A new distribution object has to be used instead.

Notice that it not possible to set both a PV and a PMF, i.e. it is not possible to use this call after a `unur_distr_discr_set_pv` call.

```
double unur_distr_discr_eval_pv (int k, const UNUR_DISTR* distribution)
double unur_distr_discr_eval_pmf (int k, const UNUR_DISTR* distribution)
double unur_distr_discr_eval_cdf (int k, const UNUR_DISTR* distribution)
```

Evaluate the PV, PMF, and the CDF, respectively, at *k*. Notice that *distribution* must not be the NULL pointer. If no PV is set for the *distribution*, then `unur_distr_discr_eval_pv` behaves like `unur_distr_discr_eval_pmf`. If the corresponding function is not available for the *distribution*, `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

IMPORTANT: In the case of a truncated standard distribution these calls always return the respective values of the *untruncated* distribution!

```
int unur_distr_discr_set_pmfstr (UNUR_DISTR* distribution, const char*
                                pmfstr)
```

This function provides an alternative way to set a PMF of the *distribution*. *pmfstr* is a character string that contains the formula for the PMF, see [Section 3.3 \[Function String\]](#), [page 42](#), for details. See also the remarks for the `unur_distr_discr_set_pmf` call.

It is not possible to call this funtion twice or to call this function after a `unur_distr_discr_set_pmf` call.

```
int unur_distr_discr_set_cdfstr (UNUR_DISTR* distribution, const char*
                                cdfstr)
```

This function provides an alternative way to set a CDF; analogously to the `unur_distr_discr_set_pmfstr` call.

```
char* unur_distr_discr_get_pmfstr (const UNUR_DISTR* distribution)
char* unur_distr_discr_get_cdfstr (const UNUR_DISTR* distribution)
```

Get pointer to respective string for PMF and CDF of *distribution* that is given via the string interface. This call allocates memory to produce this string. It should be freed when it is not used any more.

```
int unur_distr_discr_set_pmfparams (UNUR_DISTR* distribution, const
                                    double* params, int n_params)
```

Set array of parameters for *distribution*. There is an upper limit for the number of parameters *n_params*. It is given by the macro `UNUR_DISTR_MAXPARAMS` in ‘`unuran_config.h`’. (It is set to 5 but can be changed to any appropriate nonnegative number.) If *n_params* is negative or exceeds this limit no parameters are copied into the *distribution* object and `unur_errno` is set to `UNUR_ERR_DISTR_NPARAMS`.

For standard distributions from the UNURAN library the parameters are checked. Moreover, the domain is updated automatically unless it has been changed before by a `unur_distr_discr_set_domain` call. If the given parameters are invalid for the standard distribution, then no parameters are set and an error code is returned. Notice that the given parameter list for such a distribution is handled in the same way as in the corresponding `new` calls, i.e. optional parameters for the PDF that are not present in the given list are (re-)set to their default values.

Important: Integer parameter must be given as doubles.

```
int unur_distr_discr_get_pmfparams (const UNUR_DISTR* distribution, const
                                    double** params)
```

Get number of parameters of the PMF and set pointer *params* to array of parameters. If no parameters are stored in the object, an error code is returned and *params* is set to `NULL`.

```
int unur_distr_discr_set_domain (UNUR_DISTR* distribution, int left, int
                                right)
```

Set the left and right borders of the domain of the *distribution*. This can also be used to truncate an existing distribution. For setting the boundary to $\pm\infty$ use `INT_MIN` and `INT_MAX`, respectively. If *right* is not strictly greater than *left* no domain is set and `unur_errno` is set to `UNUR_ERR_DISTR_SET`. It is allowed to use this call to increase the domain. If the PV of the discrete distribution is used, then the right boundary is ignored (and internally set to *left* + size of PV - 1). Notice that `INT_MIN` and `INT_MAX` are interpreted as (minus/plus) infinity.

Default: [0, `INT_MAX`].

```
int unur_distr_discr_get_domain (const UNUR_DISTR* distribution, int*
                                left, int* right)
```

Get the left and right borders of the domain of the *distribution*. If the domain is not set explicitly the interval [`INT_MIN`, `INT_MAX`] is assumed and returned. When a PV is given then the domain is set automatically to [0, size of PV - 1].

Derived parameters

The following parameters **must** be set whenever one of the essential parameters has been set or changed (and the parameter is required for the chosen method).

`int unur_distr_discr_set_mode (UNUR_DISTR* distribution, int mode)`
 Set mode of *distribution*.

`int unur_distr_discr_upd_mode (UNUR_DISTR* distribution)`
 Recompute the mode of the *distribution*. This call works properly for distribution objects from the UNURAN library of standard distributions when the corresponding function is available. Otherwise a (slow) numerical mode finder is used. If it fails `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

`int unur_distr_discr_get_mode (UNUR_DISTR* distribution)`
 Get mode of *distribution*. If the mode is not marked as known, `unur_distr_discr_upd_mode` is called to compute the mode. If this is not successful `INT_MAX` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`. (There is no difference between the case where no routine for computing the mode is available and the case where no mode exists for the distribution at all.)

`int unur_distr_discr_set_pmfsum (UNUR_DISTR* distribution, double sum)`
 Set the sum over the PMF. If *sum* is non-positive, no sum is set and `unur_errno` is set to `UNUR_ERR_DISTR_SET`.

For a distribution object created by the UNURAN library of standard distributions you always should use the `unur_distr_discr_upd_pmfsum`. Otherwise there might be ambiguous side-effects.

`int unur_distr_discr_upd_pmfsum (UNUR_DISTR* distribution)`
 Recompute the sum over the PMF of the *distribution*. In most cases the normalization constant is recomputed and thus the sum is 1. This call only works for distribution objects from the UNURAN library of standard distributions when the corresponding function is available. Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`.

The call does not work for distributions from the UNURAN library of standard distributions with truncated domain when the CDF is not available.

`double unur_distr_discr_get_pmfsum (UNUR_DISTR* distribution)`
 Get the sum over the PMF of the *distribution*. If this sum is not known, `unur_distr_discr_upd_pmfsum` is called to compute it. If this is not successful `UNUR_INFINITY` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`.

5 Methods for generating non-uniform random variates

5.1 Routines for all generator objects

Routines for all generator objects.

Function reference

`UNUR_GEN* unur_init (UNUR_PAR* parameters)`

Initialize a generator object. All necessary information must be stored in the parameter object.

Important: If an error has occurred a NULL pointer is return. This must not be used for the sampling routines (this causes a segmentation fault).

Always check whether the call was successful or not!

Important: This call destroys the *parameter* object automatically. Thus it is not necessary/allowed to free it.

`int unur_sample_discr (UNUR_GEN* generator)`

`double unur_sample_cont (UNUR_GEN* generator)`

`void unur_sample_vec (UNUR_GEN* generator, double* vector)`

`void unur_sample_matr (UNUR_GEN* generator, double* matrix)`

Sample from generator object. The three routines depend on the type of the generator object (discrete or continuous univariate distribution, multivariate distribution, or random matrix).

Notice: UNURAN uses arrays of doubles to handle matrices. There the rows of the matrix are stored consecutively.

Important: These routines do **not** check whether *generator* is an invalid NULL pointer.

`void unur_free (UNUR_GEN* generator)`

Destroy (free) the given generator object.

`int unur_get_dimension (const UNUR_GEN* generator)`

Get the number of dimension of a (multivariate) distribution. For a univariate distribution 1 is return.

`const char* unur_get_genid (const UNUR_GEN* generator)`

Get identifier string for generator. If UNUR_ENABLE_GENID is not defined in 'unuran_config.h' then only the method used for the generator is returned.

`const UNUR_DISTR* unur_get_distr (const UNUR_GEN* generator)`

Get pointer to distribution object from generator object. This function should be used with extreme care. **Never** manipulate the distribution object returned by this call. (How should the poor generator object know what you have done?)

`int unur_set_use_distr_privatecopy (UNUR_PAR* parameters, int use_privatecopy)`

Set flag whether the generator object should make a private copy of the given distribution object or just stores the pointer to this distribution object. Values for *use_privatecopy*:

TRUE make a private copy (default)

FALSE do not make a private copy and store pointer to given (external) distribution object.

By default, generator objects keep their own private copy of the given distribution object. Thus the generator object can be handled independently from other UNURAN objects (with uniform random number generators as the only exception). When the generator object is initialized the given distribution object is cloned and stored.

However, in some rare situations it can be useful when only the pointer to the given distribution object is stored without making a private copy. A possible example is when only one random variate has to be drawn from the distribution. This behavior can be achieved when *use_localcopy* is set to **FALSE**.

Warning! Using a pointer to the external distribution object instead of a private copy must be done with **extreme care!** When the distribution object is changed or freed then the generator object does not work any more, might cause a segmentation fault, or (even worse) produces garbage. On the other hand, when the generator object is initialized or used to draw a random sampling the distribution object may be changed.

Notice: The prototypes of all `unur_<method>_new` calls use a `const` qualifier for the distribution argument. However, if *use_privatecopy* is set to **FALSE** this qualifier is discarded and the distribution might be changed.

Default: *use_privatecopy* is **TRUE**.

5.2 AUTO – Select method automatically

AUTO selects a an appropriate method for the given distribution object automatically. There are no parameters for this method, yet. But it is planned to give some parameter to describe the task for which the random variate generator is used for and thus make the choice of the generating method more appropriate. Notice that the required sampling routine for the generator object depends on the type of the given distribution object.

The chosen method also depends on the sample size for which the generator object will be used. If only a few random variates the order of magnitude of the sample size should be set via a `unur_auto_set_logss` call.

IMPORTANT: This is an experimental version and the method chosen may change in future releases of UNURAN.

For an example see [Section 2.1 \[Example: As short as possible\]](#), page 12.

How To Use

Create a generator object for the given distribution object.

Function reference

`UNUR_PAR* unsur_auto_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unsur_auto_set_logss (UNUR_PAR* parameters, int logss)`

Set the order of magnitude for the size of the sample that will be generated by the generator, i.e., the the common logarithm of the sample size.

Default is 10.

Notice: This feature will be used in future releases of UNURAN only.

5.3 Methods for continuous univariate distributions

Overview of methods

Methods for **continuous univariate distributions**

sample with `unur_sample_cont`

method	PDF	dPDF	mode	area	other
AROU	x	x	[x]		T-concave
CSTD					build-in standard distribution
HINV	[x]	[x]			CDF
NINV	[x]				CDF
SROU	x		x	x	T-concave
SSR	x		x	x	T-concave
TABLE	x		x	[~]	all local extrema
TDR	x	x			T-concave
UTDR	x		x	~	T-concave

Example

```

/* ----- */
/* File: example_cont.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* Example how to sample from a continuous univariate */
/* distribution. */
/* We build a distribution object from scratch and sample. */
/* ----- */

/* Define the PDF and dPDF of our distribution. */
/* Our distribution has the PDF */
/* f(x) = < 1 - x*x if |x| <= 1 */
/* \ 0 otherwise */
/* The PDF of our distribution: */
double mypdf( double x, const UNUR_DISTR *distr )
/* The second argument ('distr') can be used for parameters */
/* for the PDF. (We do not use parameters in our example.) */
{
    if (fabs(x) >= 1.)
        return 0.;
    else
        return (1.-x*x);
} /* end of mypdf() */

/* The derivative of the PDF of our distribution: */
double mydpdf( double x, const UNUR_DISTR *distr )
{
    if (fabs(x) >= 1.)

```

```

    return 0.;
else
    return (-2.*x);
} /* end of mydpdf() */

/* ----- */

int main()
{
    int    i;        /* loop variable */
    double x;        /* will hold the random number */

    /* Declare the three UNURAN objects. */
    UNUR_DISTR *distr; /* distribution object */
    UNUR_PAR   *par;   /* parameter object */
    UNUR_GEN   *gen;   /* generator object */

    /* Create a new distribution object from scratch. */

    /* Get empty distribution object for a continuous distribution */
    distr = unur_distr_cont_new();

    /* Fill the distribution object -- the provided information */
    /* must fulfill the requirements of the method choosen below. */
    unur_distr_cont_set_pdf(distr, mypdf); /* PDF */
    unur_distr_cont_set_dpdf(distr, mydpdf); /* its derivative */
    unur_distr_cont_set_mode(distr, 0.); /* mode */
    unur_distr_cont_set_domain(distr, -1., 1.); /* domain */

    /* Choose a method: TDR. */
    par = unur_tdr_new(distr);

    /* Set some parameters of the method TDR. */
    unur_tdr_set_variant_gw(par);
    unur_tdr_set_max_sqratio(par, 0.90);
    unur_tdr_set_c(par, -0.5);
    unur_tdr_set_max_intervals(par, 100);
    unur_tdr_set_cpoints(par, 10, NULL);

    /* Create the generator object. */
    gen = unur_init(par);

    /* Notice that this call has also destroyed the parameter */
    /* object 'par' as a side effect. */

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* It is possible to reuse the distribution object to create */
    /* another generator object. If you do not need it any more, */
    /* it should be destroyed to free memory. */
    unur_distr_free(distr);

    /* Now you can use the generator object 'gen' to sample from */
    /* the distribution. Eg.: */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n", x);
    }
}

```

```

/* When you do not need the generator object any more, you */
/* can destroy it. */
unur_free(gen);

exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */

```

Example (String API)

```

/* ----- */
/* File: example_cont_str.c */
/* ----- */
/* String API. */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* Example how to sample from a continuous univariate */
/* distribution. */

/* We use a generic distribution object and sample. */
/*
/* The PDF of our distribution is given by
/*
/*      / 1 - x*x  if |x| <= 1
/* f(x) = <
/*      \ 0        otherwise
/*
/* ----- */

int main()
{
    int i; /* loop variable */
    double x; /* will hold the random number */

    /* Declare UNURAN generator object. */
    UNUR_GEN *gen; /* generator object */

    /* Create the generator object. */
    /* Use a generic continuous distribution. */
    /* Choose a method: TDR. */
    gen = unur_str2gen("distr = cont; pdf=\"1-x*x\"; domain=(-1,1); mode=0. & \
method=tdr; variant_gw; max_sqratio=0.90; c=-0.5; \
max_intervals=100; cpoints=10");

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Now you can use the generator object 'gen' to sample from */
    /* the distribution. Eg.: */
    for (i=0; i<10; i++) {
        x = unur_sample_cont(gen);
        printf("%f\n",x);
    }
}

```



```
    }

    /* When you do not need the generator object any more, you    */
    /* can destroy it.                                           */
    unur_free(gen);

    exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */
```

5.3.1 AROU – Automatic Ratio-Of-Uniforms method

Required: T-concave PDF, dPDF

Optional: mode

Speed: Set-up: slow, Sampling: fast

Reference: [LJa00]

AROU is a variant of the ratio-of-uniforms method that uses the fact that the transformed region is convex for many distributions. It works for all T-concave distributions with $T(x) = -1/\sqrt{x}$.

It is possible to use this method for correlation induction by setting an auxiliary uniform random number generator via the `unur_set_urng_aux` call. (Notice that this must be done after a possible `unur_set_urng` call.) When an auxiliary generator is used then the number of used uniform random numbers that is used up for one generated random variate is constant and equal to 1.

There exists a test mode that verifies whether the conditions for the method are satisfied or not while sampling. It can be switched on by calling `unur_arou_set_verify` and `unur_arou_chg_verify`, respectively. Notice however that sampling is (much) slower then.

For densities with modes not close to 0 it is suggested either to set the mode of the distribution or to use the `unur_arou_set_center` call to provide some information about the main part of the PDF to avoid numerical problems.

Function reference

`UNUR_PAR* unir_arou_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unir_arou_set_usedars (UNUR_PAR* parameters, int usedars)`

If *usedars* is set to `TRUE`, “derandomized adaptive rejection sampling” (DARS) is used in setup. Segments where the area between hat and squeeze is too large compared to the average area between hat and squeeze over all intervals are split. This procedure is repeated until the ratio between area below squeeze and area below hat exceeds the bound given by `unur_arou_set_max_sqratio` call or the maximum number of segments is reached. Moreover, it also aborts when no more segments can be found for splitting.

Segments are split such that the angle of the segments are halved (corresponds to arc-mean rule of method TDR (see [Section 5.3.12 \[TDR\]](#), page 120)).

Default is `FALSE`.

`int unir_arou_set_darsfactor (UNUR_PAR* parameters, double factor)`

Set factor for “derandomized adaptive rejection sampling”. This factor is used to determine the segments that are “too large”, that is, all segments where the area between squeeze and hat is larger than *factor* times the average area over all intervals between squeeze and hat. Notice that all segments are split when *factor* is set to 0., and that there is no splitting at all when *factor* is set to `UNUR_INFINITY`.

Default is 0.99. There is no need to change this parameter.

`int unir_arou_set_max_sqratio (UNUR_PAR* parameters, double max_ratio)`

Set upper bound for the ratio (area inside squeeze) / (area inside envelope). It must be a number between 0 and 1. When the ratio exceeds the given number no further construction

points are inserted via adaptive rejection sampling. Use 0 if no construction points should be added after the setup. Use 1 if adding new construction points should not be stopped until the maximum number of construction points is reached.

Default is 0.99.

`double unur_arou_get_sqhratio (const UNUR_GEN* generator)`

Get the current ratio (area inside squeeze) / (area inside envelope) for the generator. (In case of an error UNUR_INFINITY is returned.)

`double unur_arou_get_hatarea (const UNUR_GEN* generator)`

Get the area below the hat for the generator. (In case of an error UNUR_INFINITY is returned.)

`double unur_arou_get_squeezearea (const UNUR_GEN* generator)`

Get the area below the squeeze for the generator. (In case of an error UNUR_INFINITY is returned.)

`int unur_arou_set_max_segments (UNUR_PAR* parameters, int max_segs)`

Set maximum number of segments. No construction points are added *after* the setup when the number of segments succeeds *max_segs*.

Default is 100.

`int unur_arou_set_cpoints (UNUR_PAR* parameters, int n_stp, const double* stp)`

Set construction points for enveloping polygon. If *stp* is NULL, then a heuristical rule of thumb is used to get *n_stp* construction points. This is the default behavior when this routine is not called. The (default) number of construction points is 30, then.

`int unur_arou_set_center (UNUR_PAR* parameters, double center)`

Set the center (approximate mode) of the PDF. It is used to find construction points by means of a heuristical rule of thumb. If the mode is given the center is set equal to the mode.

It is suggested to use this call to provide some information about the main part of the PDF to avoid numerical problems, when the most important part of the PDF is not close to 0.

By default the mode is used as center if available. Otherwise 0 is used.

`int unur_arou_set_usecenter (UNUR_PAR* parameters, int usecenter)`

Use the center as construction point. Default is TRUE.

`int unur_arou_set_guidefactor (UNUR_PAR* parameters, double factor)`

Set factor for relative size of the guide table for indexed search (see also method DGT [Section 5.8.3 \[DGT\], page 166](#)). It must be greater than or equal to 0. When set to 0, then sequential search is used.

Default is 2.

`int unur_arou_set_verify (UNUR_PAR* parameters, int verify)`

`int unur_arou_chg_verify (UNUR_GEN* generator, int verify)`

Turn verifying of algorithm while sampling on/off. If the condition $\text{squeeze}(x) \leq \text{PDF}(x) \leq \text{hat}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

```
int unur_arou_set_pedantic (UNUR_PAR* parameters, int pedantic)
```

Sometimes it might happen that `unur_init` has been executed successfully. But when additional construction points are added by adaptive rejection sampling, the algorithm detects that the PDF is not T-concave.

With *pedantic* being `TRUE`, the sampling routine is then exchanged by a routine that simply returns `UNUR_INFINITY`. Otherwise the new point is not added to the list of construction points. At least the hat function remains T-concave.

Setting *pedantic* to `FALSE` allows sampling from a distribution which is “almost” T-concave and small errors are tolerated. However it might happen that the hat function cannot be improved significantly. When the hat function that has been constructed by the `unur_init` call is extremely large then it might happen that the generation times are extremely high (even hours are possible in extremely rare cases).

Default is `FALSE`.

5.3.2 CSTD – Continuous STandard distributions

Required: standard distribution from UNURAN library (see [Chapter 7 \[Standard distributions\]](#), page 187).

Speed: Set-up: fast, Sampling: depends on distribution and generator

CSTD is a wrapper for special generators for continuous univariate standard distributions. It only works for distributions in the UNURAN library of standard distributions (see [Chapter 7 \[Standard distributions\]](#), page 187). If a distribution object is provided that is build from scratch, or if no special generator for the given standard distribution is provided, the NULL pointer is returned.

For some distributions more than one special generator is possible.

How To Use

Create a distribution object for a standard distribution from the UNURAN library (see [Chapter 7 \[Standard distributions\]](#), page 187). For some distributions more than one special generator (*variants*) is possible. These can be choosen by a `unur_cstd_set_variant` call. For possible variants see [Chapter 7 \[Standard distributions\]](#), page 187. However the following are common to all distributions:

`UNUR_STDGEN_DEFAULT`

the default generator.

`UNUR_STDGEN_FAST`

the fastest available special generator.

`UNUR_STDGEN_INVERSION`

the inversion method (if available).

Notice that the variant `UNUR_STDGEN_FAST` for a special generator may be slower than one of the universal algorithms! Additional variants may exist for particular distributions.

Sampling from truncated distributions (which can be constructed by changing the default domain of a distribution by means of `unur_distr_cont_set_domain` or `unur_cstd_chg_truncated` calls) is possible but requires the inversion method.

It is possible to change the parameters and the domain of the chosen distribution without building a new generator object by means of `unur_cstd_chg_pdfparams`.

Function reference

`UNUR_PAR* unir_cstd_new (const UNUR_DISTR* distribution)`

Get default parameters for new generator. It requires a distribution object for a continuous univariant distribution from the UNURAN library of standard distributions (see [Chapter 7 \[Standard distributions\]](#), page 187).

Using a truncated distribution is allowed only if the inversion method is available and selected by the `unur_cstd_set_variant` call immediately after creating the parameter object. Use a `unur_distr_cont_set_domain` call to get a truncated distribution. To change the domain of a (truncated) distribution of a generator use the `unur_cstd_chg_truncated` call.

`int unir_cstd_set_variant (UNUR_PAR* parameters, unsigned variant)`

Set variant (special generator) for sampling from a given distribution. For possible variants see [Chapter 7 \[Standard distributions\]](#), page 187.

Common variants are `UNUR_STDGEN_DEFAULT` for the default generator, `UNUR_STDGEN_FAST` for (one of the) fastest implemented special generators, and `UNUR_STDGEN_INVERSION` for the

inversion method (if available). If the selected variant number is not implemented, then an error code is returned and the variant is not changed.

```
int unur_cstd_chg_pdfparams (UNUR_GEN* generator, double* params, int
                             n_params)
```

Change array of parameters of the distribution in a given generator object. If the given parameters are invalid for the distribution, no parameters are set. Notice that optional parameters are (re-)set to their default values if not given for UNURAN standard distributions.

```
int unur_cstd_chg_truncated (UNUR_GEN* generator, double left, double
                             right)
```

Change left and right border of the domain of the (truncated) distribution. This is only possible if the inversion method is used. Otherwise this call has no effect and an error code is returned.

Notice that the given truncated domain must be a subset of the domain of the given distribution. The generator always uses the intersection of the domain of the distribution and the truncated domain given by this call.

Important: If the CDF is (almost) the same for *left* and *right* and (almost) equal to 0 or 1, then the truncated domain is not changed and the call returns an error code.

Notice: If the parameters of the distribution has been changed by a `unur_cstd_chg_pdfparams` call it is recommended to set the truncated domain again, since the former call might change the domain of the distribution but not update the values for the boundaries of the truncated distribution.

5.3.3 HINV – Hermite interpolation based INVersion of CDF

Required: CDF

Optional: PDF, dPDF

Speed: Set-up: (very) slow, Sampling: (very) fast

Reference: [HLa03] [HLD04: Sect.7.2; Alg.7.1]

HINV is a variant of numerical inversion, where the inverse CDF is approximated using Hermite interpolation, i.e., the interval $[0,1]$ is split into several intervals and in each interval the inverse CDF is approximated by polynomials constructed by means of values of the CDF and PDF at interval boundaries. This makes it possible to improve the accuracy by splitting a particular interval without recomputations in unaffected intervals. Three types of splines are implemented: linear, cubic, and quintic interpolation. For linear interpolation only the CDF is required. Cubic interpolation also requires PDF and quintic interpolation PDF and its derivative.

These splines have to be computed in a setup step. However, it only works for distributions with bounded domain; for distributions with unbounded domain the tails are chopped off such that the probability for the tail regions is small compared to the given u-resolution.

The method is not exact, as it only produces random variates of the approximated distribution. Nevertheless, the maximal numerical error in "u-direction" (i.e. $|U-CDF(X)|$, for $X = \text{"approximate inverse CDF"}(U) + |U-CDF(X)|$) can be set to the required resolution (within machine precision). Notice that very small values of the u-resolution are possible but may increase the cost for the setup step.

As the possible maximal error is only estimated in the setup it may be necessary to set some special design points for computing the Hermite interpolation to guarantee that the maximal u-error can not be bigger than desired. Such points are points where the density is not differentiable or has a local extremum. Notice that there is no necessity to do so. However, if you do not provide these points to the algorithm there might be a small chance that the approximation error is larger than the given u-resolution, or that the required number of intervals is larger than necessary.

How To Use

HINV works for continuous univariate distribution objects with given CDF and (optional) PDF. It uses Hermite interpolation of order 1, 3 [default] or 5. The order can be set by means of `unur_hinv_set_order`. For distributions with unbounded domains the tails are chopped off such that the probability for the tail regions is small compared to the given u-resolution. For finding these cut points the algorithm starts with the region $[-1.e20, 1.e20]$. For the exceptional case where this might be too small (or one knows this region and wants to avoid this search heuristics) it can be directly set via a `unur_hinv_set_boundary` call.

It is possible to use this method for generating from truncated distributions. It even can be changed for an existing generator object by an `unur_hinv_chg_truncated` call.

This method is not exact, as it only produces random variates of the approximated distribution. Nevertheless, the numerical error in "u-direction" (i.e. $|U-CDF(X)|$, for $X = \text{"approximate inverse CDF"}(U) + |U-CDF(X)|$) can be controlled by means of `unur_hinv_set_u_resolution`. The possible maximal error is only estimated in the setup. Thus it might be necessary to set some special design points for computing the Hermite interpolation to guarantee that the maximal u-error can not be bigger than desired. Such points (e.g. extremal points of the PDF, points with infinite derivative) can be set using the `unur_hinv_set_cpoints` call. If the mode for a unimodal distribution is set in the distribution object this mode is automatically used as design-point if the `unur_hinv_set_cpoints` call is not used.

As already mentioned the maximal error of this approximation is only estimated. If this error is crucial for an application we recommend to compute this error using `unur_hinv_estimate_error` which runs a small Monte Carlo simulation.

Function reference

`UNUR_PAR* unur_hinv_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unur_hinv_set_order (UNUR_PAR* parameters, int order)`

Set order of Hermite interpolation. Valid orders are 1, 3, and 5. Notice that *order* greater than 1 requires the density of the distribution, and *order* greater than 3 even requires the derivative of the density. Using *order* 1 results for most distributions in a huge number of intervals and is therefore not recommended. If the maximal error in u-direction is very small (say smaller than $1.e-10$), *order* 5 is recommended as it leads to considerably fewer design points.

Default is 3 if the density is given and 1 otherwise.

`int unur_hinv_set_u_resolution (UNUR_PAR* parameters, double u_resolution)`

Set maximal error in u-direction. However, the given u-error must not be smaller than machine epsilon (`DBL_EPSILON`) and should not be too close to this value. As the resolution of most uniform random number sources is $2^{-(32)} = 2.3e-10$, a value of $1.e-10$ leads to an inversion algorithm that could be called exact. For most simulations slightly bigger values for the maximal error are enough as well.

Default is 10^{-8} .

`int unur_hinv_set_cpoints (UNUR_PAR* parameters, const double* stp, int n_stp)`

Set starting construction points (nodes) for Hermite interpolation.

As the possible maximal error is only estimated in the setup it may be necessary to set some special design points for computing the Hermite interpolation to guarantee that the maximal u-error can not be bigger than desired. We suggest to include as special design points all local extrema of the density, all points where the density is not differentiable, and isolated points inside of the domain with density 0. If there is an interval with density constant equal to 0 inside of the given domain of the density, both endpoints of this interval should be included as special design points. Notice that there is no necessity to do so. However, if these points are not provided to the algorithm the approximation error might be larger than the given u-resolution, or the required number of intervals could be larger than necessary.

Important: Notice that the given points must be in increasing order and they must be disjoint.

Important: The boundary point of the computational region must not be given in this list! Points outside the boundary of the computational region are ignored.

Default is for unimodal densities - if known - the mode of the density, if it is not equal to the border of the domain.

`int unur_hinv_set_boundary (UNUR_PAR* parameters, double left, double right)`

Set the left and right boundary of the computation interval. The approximate CDF is only constructed inside this interval. The probability outside of this region must not be of computational relevance. Of course $\pm UNUR_INFINITY$ is not allowed.

Important: This call does not change the domain of the given distribution itself. But it restricts the domain for the resulting random variates.

Default is 1.e20.

int unur_hinv_set_guidedefactor (*UNUR_PAR* parameters, double factor*)

Set factor for relative size of the guide table for indexed search (see also method DGT Section 5.8.3 [DGT], page 166). It must be greater than or equal to 0. When set to 0, then sequential search is used.

Default is 1.

int unur_hinv_set_max_intervals (*UNUR_PAR* parameters, int max_ivs*)

Set maximum number of intervals. No generator object is created if the necessary number of intervals for the Hermite interpolation exceeds *max_ivs*. It is used to prevent the algorithm to eat up all memory for very badly shaped CDFs.

Default is 1000000 (1.e6).

int unur_hinv_get_n_intervals (*const UNUR_GEN* generator*)

Get number of nodes (design points) used for Hermite interpolation in the generator object. The number of intervals is the number of nodes minus 1. It returns an error code in case of an error.

double unur_hinv_eval_approxinvcdf (*const UNUR_GEN* generator, double u*)

Evaluate Hermite interpolation of inverse CDF at *u*. If *u* is out of the domain [0,1] then *unur_errno* is set to *UNUR_ERR_DOMAIN* and the respective bound of the domain of the distribution are returned (which is *-UNUR_INFINITY* or *UNUR_INFINITY* in the case of unbounded domains).

Notice: This function always evaluates the inverse CDF of the given distribution. A call to *unur_hinv_chg_truncated* call has no effect.

int unur_hinv_chg_truncated (*UNUR_GEN* generator, double left, double right*)

Changes the borders of the domain of the (truncated) distribution.

Notice that the given truncated domain must be a subset of the domain of the given distribution. The generator always uses the intersection of the domain of the distribution and the truncated domain given by this call. The tables of splines are not recomputed. Thus it might happen that the relative error for the generated variates from the truncated distribution is greater than the bound for the non-truncated distribution. This call also fails when the CDF values of the boundary points are too close, i.e. when only a few different floating point numbers would be computed due to round-off errors with floating point arithmetic.

When failed an error code is returned.

Important: Always check the return code since the domain is not changed in case of an error.

int unur_hinv_estimate_error (*const UNUR_GEN* generator, int samplesize, double* max_error, double* MAE*)

Estimate maximal u-error and mean absolute error (MAE) for *generator* by means of Monte-Carlo simulation with sample size *samplesize*. The results are stored in *max_error* and *MAE*, respectively.

It returns *UNUR_SUCCESS* if successful.

5.3.4 HRB – Hazard Rate Bounded

Required: bounded hazard rate
Optional: upper bound for hazard rate
Speed: Set-up: fast, Sampling: slow
Reference: [HLD04: Sect.9.1.4; Alg.9.4]

Generates random variate with given hazard rate which must be bounded from above. It uses the thinning method with a constant dominating hazard function.

How To Use

HRB requires a hazard function for a continuous distribution together with an upper bound. The latter has to be set using the `unur_hrb_set_upperbound` call. If no such upper bound is given it is assumed that the upper bound can be achieved by evaluating the hazard rate at the left hand boundary of the domain of the distribution. Notice, however, that for decreasing hazard rate the method HRD (see [Section 5.3.5 \[Hazard Rate Decreasing\]](#), page 101) is much faster and thus the preferred method.

It is important to note that the domain of the distribution can be set via a `unur_distr_cont_set_domain` call. However, the left border must not be negative. Otherwise it is set to 0. This is also the default if no domain is given at all. For computational reasons the right border is always set to `UNUR_INFINITY` independently of the given domain. Thus for domains bounded from right the function for computing the hazard rate should return `UNUR_INFINITY` right of this domain.

For distributions with increasing hazard rate method HRI (see [Section 5.3.6 \[Hazard Rate Increasing\]](#), page 102) is required.

Function reference

`UNUR_PAR*` `unur_hrb_new` (*const UNUR_DISTR** *distribution*)

Get default parameters for generator.

`int` `unur_hrb_set_upperbound` (*UNUR_PAR** *parameters*, *double* *upperbound*)

Set upper bound for hazard rate. If this call is not used it is assumed that the the maximum of the hazard rate is achieved at the left hand boundary of the domain of the distribution.

`int` `unur_hrb_set_verify` (*UNUR_PAR** *parameters*, *int* *verify*)

`int` `unur_hrb_chg_verify` (*UNUR_GEN** *generator*, *int* *verify*)

Turn verifying of algorithm while sampling on/off. If the hazard rate is not bounded by the given bound, then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`.

Default is `FALSE`.

5.3.5 HRD – Hazard Rate Decreasing

Required: decreasing (non-increasing) hazard rate

Speed: Set-up: fast, Sampling: slow

Reference: [HLD04: Sect.9.1.5; Alg.9.5]

Generates random variate with given non-increasing hazard rate. It is necessary that the distribution object contains this hazard rate. Decreasing hazard rate implies that the corresponding PDF of the distribution has heavier tails than the exponential distribution (which has constant hazard rate).

How To Use

HRD requires a hazard function for a continuous distribution with non-increasing hazard rate. There are no parameters for this method.

It is important to note that the domain of the distribution can be set via a `unur_distr_cont_set_domain` call. However, only the left hand boundary is used. For computational reasons the right hand boundary is always reset to `UNUR_INFINITY`. If no domain is given by the user then the left hand boundary is set to 0.

For distributions which do not have decreasing hazard rates but are bounded from above use method HRB (see [Section 5.3.4 \[Hazard Rate Bounded\]](#), page 100). For distributions with increasing hazard rate method HRI (see [Section 5.3.6 \[Hazard Rate Increasing\]](#), page 102) is required.

Function reference

`UNUR_PAR* unsur_hrd_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unsur_hrd_set_verify (UNUR_PAR* parameters, int verify)`

`int unsur_hrd_chg_verify (UNUR_GEN* generator, int verify)`

Turn verifying of algorithm while sampling on/off. If the hazard rate is not bounded by the given bound, then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`.

Default is FALSE.

5.3.6 HRI – Hazard Rate Increasing

Required: increasing (non-decreasing) hazard rate

Speed: Set-up: fast, Sampling: slow

Reference: [HLD04: Sect.9.1.6; Alg.9.6]

Generates random variate with given non-increasing hazard rate. It is necessary that the distribution object contains this hazard rate. Increasing hazard rate implies that the corresponding PDF of the distribution has heavier tails than the exponential distribution (which has constant hazard rate).

The method uses a decomposition of the hazard rate into a main part which is constant for all x beyond some point $p0$ and a remaining part. From both of these parts points are sampled using the thinning method and the minimum of both is returned. Sampling from the first part is easier as we have a constant dominating hazard rate. Thus $p0$ should be large. On the other hand, if $p0$ is large than the thinning algorithm needs many iteration. Thus the performance of the the algorithm depends on the choice of $p0$. We found that values close to the expectation of the generated distribution result in good performance.

How To Use

HRI requires a hazard function for a continuous distribution with non-decreasing hazard rate. The parameter $p0$ should be set to a value close to the expectation of the required distribution using `unur_hri_set_p0`. If performance is crucial one may try other values as well.

It is important to note that the domain of the distribution can be set via a `unur_distr_cont_set_domain` call. However, only the left hand boundary is used. For computational reasons the right hand boundary is always reset to `UNUR_INFINITY`. If no domain is given by the user then the left hand boundary is set to 0.

For distributions with decreasing hazard rate method HRD (see [Section 5.3.6 \[Hazard Rate Decreasing\]](#), page 102) is required. For distributions which do not have increasing or decreasing hazard rates but are bounded from above use method HRB (see [Section 5.3.4 \[Hazard Rate Bounded\]](#), page 100).

Function reference

`UNUR_PAR* unsur_hri_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unsur_hri_set_p0 (UNUR_PAR* parameters, double p0)`

Set design point for algorithm. It is used to split the domain of the distribution. Values for $p0$ close to the expectation of the distribution results in a relatively good performance of the algorithm. It is important that the hazard rate at this point must be greater than 0 and less than `UNUR_INFINITY`.

Default: left boundary of domain + 1.

`int unsur_hri_set_verify (UNUR_PAR* parameters, int verify)`

`int unsur_hri_chg_verify (UNUR_GEN* generator, int verify)`

Turn verifying of algorithm while sampling on/off. If the hazard rate is not bounded by the given bound, then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`.

Default is FALSE.

5.3.7 NINV – Numerical INVersion

Required: CDF

Optional: PDF

Speed: Set-up: optional, Sampling: (very) slow

NINV is the implementation of numerical inversion. For finding the root it is possible to choose between Newton's method and the regula falsi (combined with interval bisectioning). The regula falsi requires only the CDF while Newton's method also requires the PDF. To speed up the marginal generation time a table with suitable starting points can be computed in the setup. The performance of the algorithm can be adjusted by desired accuracy of the method. It is possible to use this method for generating from truncated distributions. The truncated domain can be changed for an existing generator object.

How To Use

The method works generates random variates by numerical inversion and requires a continuous univariate distribution objects with given CDF. Two methods are available:

- Regula falsi [default]
- Newton's method

Newton's method additionally requires the PDF of the distribution and cannot be used otherwise (NINV automatically switches to regula falsi then. Default algorithm is regula falsi. It is slightly slower but numerically much more stable than Newton's algorithm.

It is possible to use this method for generating from truncated distributions. It even can be changed for an existing generator object by an `unur_ninv_chg_truncated` call.

To speed up the marginal generation time a table with suitable starting points can be computed in the setup. Using such a table can be switched on by means of a `unur_ninv_set_table` call where the table size is given as a parameter. The table is still useful when the (truncated) domain is changed often, since it is computed for the domain of the given distribution. (It is not possible to enlarge this domain.) If it is necessary to recalculate the table during sampling, the command `unur_ninv_chg_table` can be used. As a rule of thumb using such a table is appropriate when the number of generated points exceeds the table size by a factor of 100.

The default number of iterations of NINV should be enough for all reasonable cases. Nevertheless, it is possible to adjust the maximal number of iterations with the commands `unur_ninv_set_max_iter` and `unur_ninv_chg_max_iter`. In particular this might be necessary when the PDF has a pole (where it is not bounded from below).

It is also possible to set/change the accuracy of the method (which also heavily influences the generation time). For this it is possible to change the maximum error allowed in x with `unur_ninv_set_x_resolution` and `unur_ninv_chg_x_resolution`, respectively.

NINV tries to use proper starting values for both the regula falsi and Newton's method. Of course the user might have more knowledge about the properties of the underlying distribution and is able to share his wisdom with NINV using the respective commands `unur_ninv_set_start` and `unur_ninv_chg_start`

It is also possible to change the parameters of the given distribution by a `unur_ninv_chg_pdfparams` call. If a table exists, it will be recomputed immediately.

It might happen that NINV aborts `unur_sample_cont` without computing the correct value (because the maximal number iterations has been exceeded). Then the last approximate value for x is returned (with might be fairly false) and `unur_error` is set to `UNUR_ERR_GEN_SAMPLING`.

Function reference

UNUR_PAR* `unur_ninv_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

int `unur_ninv_set_useregula (UNUR_PAR* parameters)`

Switch to regula falsi combined with interval bisectioning. (This the default.)

int `unur_ninv_set_usenewton (UNUR_PAR* parameters)`

Switch to Newton's method. Notice that it is numerically less stable than regula falsi. If it is not possible to invert the CDF for a particular uniform random number U when calling `unur_sample_cont`, `unur_error` is set to `UNUR_ERR_GEN_SAMPLING`. Thus it is recommended to check `unur_error` before using the result of the sampling routine.

int `unur_ninv_set_max_iter (UNUR_PAR* parameters, int max_iter)`

int `unur_ninv_chg_max_iter (UNUR_GEN* generator, int max_iter)`

Set and change number of maximal iterations. Default is 40.

int `unur_ninv_set_x_resolution (UNUR_PAR* parameters, double x_resolution)`

int `unur_ninv_chg_x_resolution (UNUR_GEN* generator, double x_resolution)`

Set and change the maximal relative error in x. Default is 1.e-8.

int `unur_ninv_set_start (UNUR_PAR* parameters, double left, double right)`

Set starting points. If not set, suitable values are chosen automatically.

Newton: *left*: starting point

Regula falsi: *left, right*: boundary of starting interval

If the starting points are not set then the following points are used by default:

Newton: *left*: CDF(*left*) = 0.5

Regula falsi: *left*: CDF(*left*) = 0.1

right: CDF(*right*) = 0.9

If *left* == *right*, then UNURAN always uses the default starting points!

int `unur_ninv_chg_start (UNUR_GEN* gen, double left, double right)`

Change the starting points for numerical inversion. If *left*==*right*, then UNURAN uses the default starting points (see `unur_ninv_set_start`).

int `unur_ninv_set_table (UNUR_PAR* parameters, int no_of_points)`

Generates a table with *no_of_points* points containing suitable starting values for the iteration. The value of *no_of_points* must be at least 10 (otherwise it will be set to 10 automatically).

The table points are chosen such that the CDF at these points form an equidistance sequence in the interval (0,1).

If a table is used, then the starting points given by `unur_ninv_set_start` are ignored.

No table is used by default.

int `unur_ninv_chg_table (UNUR_GEN* gen, int no_of_points)`

Recomputes a table as described in `unur_ninv_set_table`.

```
int unur_ninv_chg_truncated (UNUR_GEN* gen, double left, double right)
```

Changes the borders of the domain of the (truncated) distribution.

Notice that the given truncated domain must be a subset of the domain of the given distribution. The generator always uses the intersection of the domain of the distribution and the truncated domain given by this call. Moreover the starting point(s) will not be changed.

Important: If the CDF is (almost) the same for *left* and *right* and (almost) equal to 0 or 1, then the truncated domain is *not* changed and the call returns an error code.

Notice: If the parameters of the distribution has been changed by a `unur_ninv_chg_pdfparams` call it is recommended to set the truncated domain again, since the former call might change the domain of the distribution but not update the values for the boundaries of the truncated distribution.

```
int unur_ninv_chg_pdfparams (UNUR_GEN* generator, double* params, int  
                             n_params)
```

Change array of parameters of the distribution in a given generator object.

For standard distributions from the UNURAN library the parameters are checked. If these are invalid, then an error code is returned. Moreover the domain is updated automatically unless it has been changed before by a `unur_distr_discr_set_domain` call. Notice that optional parameters are (re-)set to their default values if not given for UNURAN standard distributions.

For other distributions *params* is simply copied into to distribution object. It is only checked that *n_params* does not exceed the maximum number of parameters allowed. Then an error code is returned and `unur_errno` is set to `UNUR_ERR_DISTR_NPARAMS`.

```
double unur_ninv_eval_approxinvcdf (UNUR_GEN* generator, double u)
```

Get approximate value of inverse CDF at *u*. If *u* is out of the domain [0,1] then `unur_errno` is set to `UNUR_ERR_DOMAIN` and the respective bound of the domain of the distribution are returned (which is `-UNUR_INFINITY` or `UNUR_INFINITY` in the case of unbounded domains).

Notice: This function always evaluates the inverse CDF of the given distribution. A call to `unur_ninv_chg_truncated` call has no effect.

5.3.8 NROU – Naive Ratio-Of-Uniforms method

Required: PDF

Optional: mode, center, bounding rectangle for acceptance region

Speed: Set-up: slow or fast, Sampling: moderate

Reference: [HLD04: Sect.2.4 and Sect.6.4]

NROU is an implementation of the (generalized) ratio-of-uniforms method which uses (minimal) bounding rectangles, see [Section A.4 \[Ratio-of-Uniforms\], page 216](#). It uses a positive control parameter r for adjusting the algorithm to the given distribution to improve performance and/or to make this method applicable. Larger values of r increase the class of distributions for which the method works at the expense of a higher rejection constant. For computational reasons $r=1$ should be used if possible (this is the default). Moreover, this implementation uses the center μ of the distribution (see `unur_distr_cont_get_center` for details of its default values).

For the special case with $r = 1$ the coordinates of the minimal bounding rectangles are given by

$$\begin{aligned} v^+ &= \sup_x \sqrt{PDF(x)}, \\ u^- &= \inf_x (x - \mu) \sqrt{PDF(x)}, \\ u^+ &= \sup_x (x - \mu) \sqrt{PDF(x)}, \end{aligned}$$

where μ is the center of the distribution. For other values of r we have

$$\begin{aligned} v^+ &= \sup_x (PDF(x))^{1/(r+1)}, \\ u^- &= \inf_x (x - \mu) (PDF(x))^{r/(r+1)}, \\ u^+ &= \sup_x (x - \mu) (PDF(x))^{r/(r+1)}. \end{aligned}$$

These bounds can be given directly. Otherwise they are computed automatically by means of a (slow) numerical routine. Of course this routine can fail, especially when this rectangle is not bounded.

It is important to note that the algorithm works with $PDF(x - \mu)$ instead of $PDF(x)$. This is important as otherwise the acceptance region can become a very long and skinny ellipsoid along a diagonal of the (huge) bounding rectangle.

How To Use

For using the NROU method UNURAN needs the PDF of the distribution. Additionally, the parameter r can be set via a `unur_vnrou_set_r` call. Notice that the acceptance probability decreases when r is increased. On the other hand is is more unlikely that the bounding rectangle does not exist if r is small.

A bounding rectangle can be given by the `unur_vnrou_set_u` and `unur_vnrou_set_v` calls.

Important: The bounding rectangle has to be provided for the function $PDF(x - center)$! Notice that `center` is the center of the given distribution, see `unur_distr_cont_set_center`. If in doubt or if this value is not optimal, it can be changed (overridden) by a `unur_nrou_set_center` call.

If the coordinates of the bounding rectangle are not provided by the user then the minimal bounding rectangle is computed automatically.

By means of `unur_vnrou_set_verify` and `unur_vnrou_chg_verify` one can run the sampling algorithm in a checking mode, i.e., in every cycle of the rejection loop it is checked whether the used rectangle indeed enclosed the acceptance region of the distribution. When in doubt

(e.g., when it is not clear whether the numerical routine has worked correctly) this can be used to run a small Monte Carlo study.

Function reference

`UNUR_PAR* unur_nrou_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unur_nrou_set_u (UNUR_PAR* parameters, double umin, double umax)`

Sets left and right boundary of bounding rectangle. If no values are given, the boundary of the minimal bounding rectangle is computed numerically.

Notice: Computing the minimal bounding rectangle may fail under some circumstances. Moreover, for multimodal distributions the bounds might be too small as only local extrema are computed. Nevertheless, for T_c -concave distributions with $c = -1/2$ it should work.

Important: The bounding rectangle that has to be provided is for the function $PDF(x - center)$!

Default: not set.

`int unur_nrou_set_v (UNUR_PAR* parameters, double vmax)`

Set upper boundary for bounding rectangle. If this value is not given then $\sqrt{PDF(mode)}$ is used instead.

Notice: When the mode is not given for the distribution object, then it will be computed numerically.

Default: not set.

`int unur_nrou_set_r (UNUR_PAR* parameters, double r)`

Sets the parameter r of the generalized ratio-of-uniforms method.

Notice: This parameter must satisfy $r > 0$.

Default: 1.

`int unur_nrou_set_center (UNUR_PAR* parameters, double center)`

Set the center (μ) of the PDF. If not set the center of the given distribution object is used.

Default: see `unur_distr_cont_set_center`.

`int unur_nrou_set_verify (UNUR_PAR* parameters, int verify)`

Turn verifying of algorithm while sampling on/off.

If the condition $PDF(x) \leq \hat{h}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However, notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

`int unur_nrou_chg_verify (UNUR_GEN* generator, int verify)`

Change the verifying of algorithm while sampling on/off.

5.3.9 SROU – Simple Ratio-Of-Uniforms method

Required: T-concave PDF, mode, area

Speed: Set-up: fast, Sampling: slow

Reference: [LJa01] [LJa02] [HLD04: Sect.6.3.1; Sect.6.3.2; Sect.6.4.1; Alg.6.4; Alg.6.5; Alg.6.7]

SROU is based on the ratio-of-uniforms method (see [Section A.4 \[Ratio-of-Uniforms\]](#), [page 216](#)) that uses universal inequalities for constructing a (universal) bounding rectangle. It works for all T -concave distributions, including log-concave and T -concave distributions with $T(x) = -1/\sqrt{x}$.

Moreover an (optional) parameter r can be given, to adjust the generator to the given distribution. This parameter is strongly related to the parameter c for transformed density rejection (see [Section 5.3.12 \[TDR\]](#), [page 120](#)) via the formula $c = -r/(r+1)$. The rejection constant increases with higher values for r . On the other hand, the given density must be T_c -concave for the corresponding c . The default setting for r is 1 which results in a very simple code. (For other settings, sampling uniformly from the acceptance region is more complicated.)

Optionally the CDF at the mode can be given to increase the performance of the algorithm. Then the rejection constant is reduced by 1/2 and (if $r=1$) even a universal squeeze can (but need not be) used. A way to increase the performance of the algorithm when the CDF at the mode is not provided is the usage of the mirror principle (only if $r=1$). However using squeezes and using the mirror principle is only recommended when the PDF is expensive to compute.

The exact location of the mode and/or the area below the PDF can be replaced by appropriate bounds. Then the algorithm still works but has larger rejection constants.

How To Use

SROU works for any continuous univariate distribution object with given T_c -concave PDF with $c < 1$, mode and area below PDF. Optional the CDF at the mode can be given to increase the performance of the algorithm by means of the `unur_srou_set_cdfatmode` call. Additionally squeezes can be used and switched on via `unur_srou_set_usesqueeze` (only if $r=1$). A way to increase the performance of the algorithm when the CDF at the mode is not provided is the usage of the mirror principle which can be switched on by means of a `unur_srou_set_usemirror` call (only if $r=1$). However using squeezes and using the mirror principle is only recommended when the PDF is expensive to compute.

The parameter r can be given, to adjust the generator to the given distribution. This parameter is strongly related parameter c for transformed density rejection via the formula $c = -r/(r+1)$. The parameter r can be any value larger than or equal to 1. Values less than 1 are automatically set to 1. The rejection constant depends on the chosen parameter r but not on the particular distribution. It is 4 for r equal to 1 and higher for higher values of r . It is important to note that different algorithms for different values of r : If r equal to 1 this is much faster than the algorithm for r greater than 1. The default setting for r is 1.

If the (exact) area below the PDF is not known, then an upper bound can be used instead (which of course increases the rejection constant). But then the squeeze flag must not be set and `unur_srou_set_cdfatmode` must not be used.

If the exact location of the mode is not known, then use the approximate location and provide the (exact) value of the PDF at the mode by means of the `unur_srou_set_pdfatmode` call. But then `unur_srou_set_cdfatmode` must not be used. Notice, that a (slow) numerical mode finder will be used if no mode is given at all. It is even possible to give an upper bound for the PDF only. However, then the (upper bound for the) area below the PDF has to be multiplied by the ratio between the upper bound and the lower bound of the PDF at the mode. Again setting the squeeze flag and using `unur_srou_set_cdfatmode` is not allowed.

It is possible to change the parameters and the domain of the chosen distribution without building a new generator object using the `unur_srou_chg_pdfparams` and `unur_srou_chg_domain` call, respectively. But then `unur_srou_chg_pdfarea`, `unur_srou_chg_mode` and `unur_srou_chg_cdfatmode` have to be used to reset the corresponding figures whenever they have changed. If the PDF at the mode has been provided by a `unur_srou_set_pdfatmode` call, additionally `unur_srou_chg_pdfatmode` must be used (otherwise this call is not necessary since then this figure is computed directly from the PDF).

Important: If any of mode, PDF or CDF at the mode, or the area below the mode has been changed, then `unur_srou_reinit` must be executed. (Otherwise the generator produces garbage).

There exists a test mode that verifies whether the conditions for the method are satisfied or not while sampling. It can be switched on by calling `unur_srou_set_verify` and `unur_srou_chg_verify`, respectively. Notice however that sampling is (a little bit) slower then.

Function reference

`UNUR_PAR* unsur_srou_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unsur_srou_reinit (UNUR_GEN* generator)`

Update an existing generator object after the distribution has been modified. It must be executed whenever the parameters or the domain of the distributions have been changed (see below). It is faster than destroying the existing object and building a new one from scratch. If reinitialization has been successful `UNUR_SUCCESS` is returned, in case of a failure an error code is returned.

`int unsur_srou_set_r (UNUR_PAR* parameters, double r)`

Set parameter r for transformation. Only values greater than or equal to 1 are allowed. The performance of the generator decreases when r is increased. On the other hand r must not be set to small, since the given density must be T_c -concave for $c = -r/(r+1)$.

Notice: If r is set to 1 a simpler and much faster algorithm is used then for r greater than one.

For computational reasons values of r that are greater than 1 but less than 1.01 are always set to 1.01.

Default is 1.

`int unsur_srou_set_cdfatmode (UNUR_PAR* parameters, double Fmode)`

Set CDF at mode. When set, the performance of the algorithm is increased by factor 2. However, when the parameters of the distribution are changed `unur_srou_chg_cdfatmode` has to be used to update this value.

Default: not set.

`int unsur_srou_set_pdfatmode (UNUR_PAR* parameters, double fmode)`

Set pdf at mode. When set, the PDF at the mode is never changed. This is to avoid additional computations, when the PDF does not change when parameters of the distributions vary. It is only useful when the PDF at the mode does not change with changing parameters of the distribution.

IMPORTANT: This call has to be executed after a possible call of `unur_srou_set_r`. Default: not set.

```
int unur_srou_set_usesqueeze (UNUR_PAR* parameters, int usesqueeze)
```

Set flag for using universal squeeze (default: off). Using squeezes is only useful when the evaluation of the PDF is (extremely) expensive. Using squeezes is automatically disabled when the CDF at the mode is not given (then no universal squeezes exist).

Default is FALSE.

```
int unur_srou_set_usemirror (UNUR_PAR* parameters, int usemirror)
```

Set flag for using mirror principle (default: off). Using the mirror principle is only useful when the CDF at the mode is not known and the evaluation of the PDF is rather cheap compared to the marginal generation time of the underlying uniform random number generator. It is automatically disabled when the CDF at the mode is given. (Then there is no necessity to use the mirror principle. However disabling is only done during the initialization step but not at a re-initialization step.)

Default is FALSE.

```
int unur_srou_set_verify (UNUR_PAR* parameters, int verify)
```

```
int unur_srou_chg_verify (UNUR_GEN* generator, int verify)
```

Turn verifying of algorithm while sampling on/off. If the condition $\text{squeeze}(x) \leq \text{PDF}(x) \leq \hat{\text{PDF}}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

```
int unur_srou_chg_pdfparams (UNUR_GEN* generator, double* params, int  
                             n_params)
```

Change array of parameters of the distribution in a given generator object.

For standard distributions from the UNURAN library the parameters are checked. If these are invalid, then an error code is returned. Moreover the domain is updated automatically unless it has been changed before by a `unur_distr_discr_set_domain` call. Notice that optional parameters are (re-)set to their default values if not given for UNURAN standard distributions.

For other distributions *params* is simply copied into to distribution object. It is only checked that *n_params* does not exceed the maximum number of parameters allowed. Then an error code is returned and `unur_errno` is set to `UNUR_ERR_DIST_NPARAMS`.

```
int unur_srou_chg_domain (UNUR_GEN* generator, double left, double right)
```

Change left and right border of the domain of the (truncated) distribution. If the mode changes when the domain of the (truncated) distribution is changed, then a corresponding `unur_srou_chg_mode` is required. (There is no checking whether the domain is set or not as in the `unur_init` call.)

```
int unur_srou_chg_mode (UNUR_GEN* generator, double mode)
```

Change mode of distribution. `unur_srou_reinit` must be executed before sampling from the generator again.

```
int unur_srou_upd_mode (UNUR_GEN* generator)
```

Recompute the mode of the distribution. See `unur_distr_cont_upd_mode` for more details. `unur_srou_reinit` must be executed before sampling from the generator again.

```
int unur_srou_chg_cdfatmode (UNUR_GEN* generator, double Fmode)
```

Change CDF at mode of distribution. `unur_srou_reinit` must be executed before sampling from the generator again.

```
int unur_srou_chg_pdfatmode (UNUR_GEN* generator, double fmode)
```

Change PDF at mode of distribution. `unur_srou_reinit` must be executed before sampling from the generator again.

```
int unur_srou_chg_pdfarea (UNUR_GEN* generator, double area)
```

Change area below PDF of distribution. `unur_srou_reinit` must be executed before sampling from the generator again.

```
int unur_srou_upd_pdfarea (UNUR_GEN* generator)
```

Recompute the area below the PDF of the distribution. It only works when a distribution objects from the UNURAN library of standard distributions is used (see [Chapter 7 \[Standard distributions\]](#), page 187). Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_srou_reinit` must be executed before sampling from the generator again.

5.3.10 SSR – Simple Setup Rejection

Required: T-concave PDF, mode, area

Speed: Set-up: fast, Sampling: slow

Reference: [LJa01] [HLD04: Sect.6.3.3; Alg.6.6]

SSR is an acceptance/rejection method that uses universal inequalities for constructing (universal) hats and squeezes (see [Section A.2 \[Rejection\]](#), page 214). It works for all T -concave distributions with $T(x) = -1/\sqrt{x}$.

It requires the PDF, the (exact) location of the mode and the area below the given PDF. The rejection constant is 4 for all T -concave distributions with unbounded domain and is less than 4 when the domain is bounded. Optionally the CDF at the mode can be given to increase the performance of the algorithm. Then the rejection constant is at most 2 and a universal squeeze can (but need not be) used. However, using squeezes is not recommended unless the evaluation of the PDF is expensive.

The exact location of the mode and/or the area below the PDF can be replaced by appropriate bounds. Then the algorithm still works but has larger rejection constants.

How To Use

SSR works for any continuous univariate distribution object with given T -concave PDF (with $T(x) = -1/\sqrt{x}$), mode and area below PDF. Optional the CDF at the mode can be given to increase the performance of the algorithm by means of the `unur_ssr_set_cdfatmode` call. Additionally squeezes can be used and switched on via `unur_ssr_set_usesqueeze`. If the (exact) area below the PDF is not known, then an upper bound can be used instead (which of course increases the rejection constant). But then the squeeze flag must not be set and `unur_ssr_set_cdfatmode` must not be used.

If the exact location of the mode is not known, then use the approximate location and provide the (exact) value of the PDF at the mode by means of the `unur_ssr_set_pdfatmode` call. But then `unur_ssr_set_cdfatmode` must not be used. Notice, that a (slow) numerical mode finder will be used if no mode is given at all. It is even possible to give an upper bound for the PDF only. However, then the (upper bound for the) area below the PDF has to be multiplied by the ratio between the upper bound and the lower bound of the PDF at the mode. Again setting the squeeze flag and using `unur_ssr_set_cdfatmode` is not allowed.

It is possible to change the parameters and the domain of the chosen distribution without building a new generator object using the `unur_ssr_chg_pdfparams` and `unur_ssr_chg_domain` call, respectively. Then `unur_ssr_chg_pdfarea`, `unur_ssr_chg_mode` and `unur_ssr_chg_cdfatmode` have to be used to reset the corresponding figures whenever they have changed. If the PDF at the mode has been provided by a `unur_ssr_set_pdfatmode` call, additionally `unur_ssr_chg_pdfatmode` must be used (otherwise this call is not necessary since then this figure is computed directly from the PDF).

Important: If any of mode, PDF or CDF at the mode, or the area below the mode has been changed, then `unur_ssr_reinit` must be executed. (Otherwise the generator produces garbage).

There exists a test mode that verifies whether the conditions for the method are satisfied or not while sampling. It can be switched on/off by calling `unur_ssr_set_verify` and `unur_ssr_chg_verify`, respectively. Notice, however, that sampling is (a little bit) slower then.

Function reference

`UNUR_PAR* unur_ssr_new (const UNUR_DIST* distribution)`

Get default parameters for generator.

int unur_ssr_reinit (*UNUR_GEN* generator*)

Update an existing generator object after the distribution has been modified. It must be executed whenever the parameters or the domain of the distribution has been changed (see below). It is faster than destroying the existing object and build a new one from scratch. If reinitialization has been successful **UNUR_SUCCESS** is returned, in case of a failure an error code is returned.

int unur_ssr_set_cdfatmode (*UNUR_PAR* parameters, double Fmode*)

Set CDF at mode. When set, the performance of the algorithm is increased by factor 2. However, when the parameters of the distribution are changed **unur_ssr_chg_cdfatmode** has to be used to update this value.

Default: not set.

int unur_ssr_set_pdfatmode (*UNUR_PAR* parameters, double fmode*)

Set pdf at mode. When set, the PDF at the mode is never changed. This is to avoid additional computations, when the PDF does not change when parameters of the distributions vary. It is only useful when the PDF at the mode does not change with changing parameters for the distribution.

Default: not set.

int unur_ssr_set_usesqueeze (*UNUR_PAR* parameters, int usesqueeze*)

Set flag for using universal squeeze (default: off). Using squeezes is only useful when the evaluation of the PDF is (extremely) expensive. Using squeezes is automatically disabled when the CDF at the mode is not given (then no universal squeezes exist).

Default is FALSE.

int unur_ssr_set_verify (*UNUR_PAR* parameters, int verify*)

int unur_ssr_chg_verify (*UNUR_GEN* generator, int verify*)

Turn verifying of algorithm while sampling on/off. If the condition $\text{squeeze}(x) \leq \text{PDF}(x) \leq \hat{\text{hat}}(x)$ is violated for some x then **unur_errno** is set to **UNUR_ERR_GEN_CONDITION**. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

int unur_ssr_chg_pdfparams (*UNUR_GEN* generator, double* params, int n_params*)

Change array of parameters of the distribution in a given generator object.

For standard distributions from the UNURAN library the parameters are checked. If these are invalid, then an error code is returned. Moreover the domain is updated automatically unless it has been changed before by a **unur_distr_discr_set_domain** call. Notice that optional parameters are (re-)set to their default values if not given for UNURAN standard distributions.

For other distributions *params* is simply copied into to distribution object. It is only checked that *n_params* does not exceed the maximum number of parameters allowed. Then an error code is returned and **unur_errno** is set to **UNUR_ERR_DISTR_NPARAMS**.

int unur_ssr_chg_domain (*UNUR_GEN* generator, double left, double right*)

Change left and right border of the domain of the distribution. If the mode changes when the domain of the distribution is changed, then a correspondig **unur_ssr_chg_mode** is required. (There is no domain checking as in the **unur_init** call.)

```
int unur_ssr_chg_mode (UNUR_GEN* generator, double mode)
```

Change mode of distribution. `unur_ssr_reinit` must be executed before sampling from the generator again.

```
int unur_ssr_upd_mode (UNUR_GEN* generator)
```

Recompute the mode of the distribution. See `unur_distr_cont_upd_mode` for more details. `unur_ssr_reinit` must be executed before sampling from the generator again.

```
int unur_ssr_chg_cdfatmode (UNUR_GEN* generator, double Fmode)
```

Change CDF at mode of distribution. `unur_ssr_reinit` must be executed before sampling from the generator again.

```
int unur_ssr_chg_pdfatmode (UNUR_GEN* generator, double fmode)
```

Change PDF at mode of distribution. `unur_ssr_reinit` must be executed before sampling from the generator again.

```
int unur_ssr_chg_pdfarea (UNUR_GEN* generator, double area)
```

Change area below PDF of distribution. `unur_ssr_reinit` must be executed before sampling from the generator again.

```
int unur_ssr_upd_pdfarea (UNUR_GEN* generator)
```

Recompute the area below the PDF of the distribution. It only works when a distribution objects from the UNURAN library of standard distributions is used (see [Chapter 7 \[Standard distributions\]](#), page 187). Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_ssr_reinit` must be executed before sampling from the generator again.

5.3.11 TABL – a TABLE method with piecewise constant hats

Required: PDF, all local extrema, cut-off values for the tails

Optional: approximate area

Speed: Set-up: (very) slow, Sampling: fast

Reference: [AJa93] [AJa95] [HLD04: Cha.5.1]

TABL (called Ahrens method in [HLD04]) is an acceptance/rejection method (see [Section A.2 \[Rejection\]](#), page 214) that uses a decomposition of the domain of the distribution into many short subintervals. Inside of these subintervals constant hat and squeeze functions are utilized. Thus it is easy to use the idea of immediate acceptance for points below the squeeze. This reduces the expected number of uniform random numbers per generated random variate to less than two. Using a large number of subintervals only little more than one random number is necessary on average. Thus this method becomes very fast.

Due to the constant hat function this method only works for distributions with bounded domains. Thus for unbounded domains the left and right tails have to be cut off. This is no problem when the probability of falling into these tail regions is beyond computational relevance (e.g. smaller than $1.e-12$).

For easy construction of hat and squeeze functions it is necessary to know the regions of monotonicity (called *slopes*) or equivalently all local maxima and minima of the density. The main problem for this method in the setup is the choice of the subintervals. A simple and close to optimal approach is the "equal area rule" [HLD04: Cha.5.1]. There the subintervals are selected such that the area below the hat is the same for each subinterval which can be realized with a simple recursion. If more subintervals are necessary it is possible to split either randomly chosen intervals (adaptive rejection sampling, ARS) or those intervals, where the ratio between squeeze and hat is smallest. This version of the setup is called derandomized ARS (DARS). With the default settings TABL is first calculating approximately 30 subintervals with the equal area rule. Then DARS is used till the desired fit of the hat is reached.

A convenient measure to control the quality of the fit of hat and squeeze is the ratio (area below squeeze)/(area below hat) called `sqratio` which must be smaller or equal to one. The expected number of iterations in the rejection algorithm is known to be smaller than $1/\text{sqratio}$ and the expected number of evaluations of the density is bounded by $1/\text{sqratio} - 1$. So values of the `sqratio` close to one (e.g. 0.95 or 0.99) lead to many subintervals. Thus a better fitting hat is constructed and the sampling algorithm becomes fast; on the other hand large tables are needed and the setup is very slow. For moderate values of `sqratio` (e.g. 0.9 or 0.8) the sampling is slower but the required tables are smaller and the setup is not so slow.

It follows from the above explanations that TABL is always requiring a slow setup and that it is not very well suited for heavy-tailed distributions.

How To Use

For using the TABL method UNURAN needs a bounded interval to which the generated variates can be restricted and information about all local extrema of the distribution. For unimodal densities it is sufficient to provide the mode of the distribution. For the case of a built-in unimodal distribution with bounded domain all these information is present in the distribution object and thus no extra input is necessary (see `example.TABL1` below).

For a built-in unimodal distribution with unbounded domain we should specify the cut-off values for the tails. This can be done with the `unur_tabl_set_boundary` call (see `example.TABL2` below). For the case that we do not set these boundaries the default values of $\pm 1.e20$ are used. We can see in `example.TABL1` that this still works fine for many standard distributions.

For the case of a multimodal distribution we have to set the regions of monotonicity (called slopes) explicitly using the `unur_tabl_set_slopes` command (see example_TABL3 below).

To controll the fit of the hat and the size of the tables and thus the speed of the setup and the sampling it is most convenient to use the `unur_tabl_set_max_sqhratio` call. The default is 0.9 which is a sensible value for most distributions and applications. If very large samples of a distribution are required or the evaluation of a density is very slow it may be useful to increase the sqhratio to eg. 0.95 or even 0.99. With the `unur_tabl_get_sqhratio` call we can check which sqhratio was really reached. If that value is below the desired value it is necessary to increase the maximal number of subintervals, which defaults to 1000, using the `unur_tabl_set_max_intervals` call. The `unur_tabl_get_n_intervals` call can be used to find out the number of subintervals the setup calculated.

It is also possible to set the number of intervals and their respective boundaries by means of the `unur_tabl_set_cpoints` call.

It is also possible to use method TABL for correlation induction (variance reduction) by setting of an auxiliary uniform random number generator via the `unur_set_urng_aux` call. (Notice that this must be done after a possible `unur_set_urng` call.) However, this only works when immediate acceptance is switched of by a `unur_tabl_set_variant_ia` call.

Function reference

`UNUR_PAR* unsur_tabl_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unsur_tabl_set_variant_ia (UNUR_PAR* parameters, int use_ia)`

Use immediate acceptance when *use_ia* is set to TRUE. This technique requires less uniform. If it is set to FALSE, “classical” acceptance/rejection from hat distribution is used.

Notice: Auxiliary uniform random number generators for correlation induction (variance reduction) can only be used when “classical” acceptance/rejection is used.

Default: TRUE.

`int unsur_tabl_set_cpoints (UNUR_PAR* parameters, int n_cpoints, const double* cpoints)`

Set construction points for the hat function. If *stp* is NULL than a heuristic rule of thumb is used to get *n_stp* construction points. This is the default behavior.

The default number of construction points is 30.

`int unsur_tabl_set_nstp (UNUR_PAR* parameters, int n_stp)`

Set number of construction points for the hat function. *n_stp* must be greater than zero. After the setup there are about *n_stp* construction points. However it might be larger when a small fraction is given by the `unur_tabl_set_areafraction` call. It also might be smaller for some variants.

Default is 30.

`int unsur_tabl_set_useear (UNUR_PAR* parameters, int useear)`

If *useear* is set to TRUE, the “equal area rule” is used, the given slopes are partitioned in such a way that the area below the hat function in each subinterval (“stripe”) has the same area (except the last the last interval which can be smaller). The area can be set by means of the `unur_tabl_set_areafraction` call.

Default is TRUE.

int unur_tabl_set_areafraction (*UNUR_PAR* parameters, double fraction*)

Set parameter for the equal area rule. During the setup a piecewise constant hat is constructed, such that the area below each of these pieces (strips) is the same and equal to the (given) area below the PDF times *fraction* (which must be greater than zero).

Important: If the area below the PDF is not set in the distribution object, then 1 is assumed. Default is 0.1.

int unur_tabl_set_usedars (*UNUR_PAR* parameters, int usedars*)

If *usedars* is set to **TRUE**, “derandomized adaptive rejection sampling” (DARS) is used in the setup. Intervals, where the area between hat and squeeze is too large compared to the average area between hat and squeeze over all intervals, are split. This procedure is repeated until the ratio between squeeze and hat exceeds the bound given by `unur_tabl_set_max_sqratio` call or the maximum number of intervals is reached. Moreover, it also aborts when no more intervals can be found for splitting.

For finding splitting points the arc-mean rule (a mixture of arithmetic mean and harmonic mean) is used.

Default is **TRUE**.

int unur_tabl_set_darsfactor (*UNUR_PAR* parameters, double factor*)

Set factor for “derandomized adaptive rejection sampling”. This factor is used to determine the segments that are “too large”, that is, all segments where the area between squeeze and hat is larger than *factor* times the average area over all intervals between squeeze and hat. Notice that all segments are split when *factor* is set to 0., and that there is no splitting at all when *factor* is set to **UNUR_INFINITY**.

Default is 0.99. There is no need to change this parameter.

int unur_tabl_set_variant_splitmode (*UNUR_PAR* parameters, unsigned splitmode*)

There are three variants for adaptive rejection sampling. These differ in the way how an interval is split:

splitmode 1

use the generated point to split the interval.

splitmode 2

use the mean point of the interval.

splitmode 3

use the arcmean point; suggested for distributions with heavy tails.

Default is splitmode 2.

int unur_tabl_set_max_sqratio (*UNUR_PAR* parameters, double max_ratio*)

Set upper bound for the ratio (area below squeeze) / (area below hat). It must be a number between 0 and 1. When the ratio exceeds the given number no further construction points are inserted via DARS in the setup.

For the case of ARS (`unur_tabl_set_usedars()` must be set to **FALSE**): Use 0 if no construction points should be added after the setup. Use 1 if added new construction points should not be stopped until the maximum number of construction points is reached. If *max_ratio* is close to one, many construction points are used.

Default is 0.9.

`double unur_tabl_get_sqhratio (const UNUR_GEN* generator)`

Get the current ratio (area below squeeze) / (area below hat) for the generator. (In case of an error UNUR_INFINITY is returned.)

`double unur_tabl_get_hatarea (const UNUR_GEN* generator)`

Get the area below the hat for the generator. (In case of an error UNUR_INFINITY is returned.)

`double unur_tabl_get_squeezearea (const UNUR_GEN* generator)`

Get the area below the squeeze for the generator. (In case of an error UNUR_INFINITY is returned.)

`int unur_tabl_set_max_intervals (UNUR_PAR* parameters, int max_ivs)`

Set maximum number of intervals. No construction points are added in or after the setup when the number of intervals succeeds *max_ivs*.

Default is 1000.

`int unur_tabl_get_n_intervals (const UNUR_GEN* generator)`

Get the current number of intervals. (In case of an error 0 is returned.)

`int unur_tabl_set_slopes (UNUR_PAR* parameters, const double* slopes, int n_slopes)`

Set slopes for the PDF. A slope $\langle a, b \rangle$ is an interval $[a, b]$ or $[b, a]$ where the PDF is monotone and $\text{PDF}(a) \geq \text{PDF}(b)$. The list of slopes is given by an array *slopes* where each consecutive tuple (i.e. $(\text{slopes}[0], \text{slopes}[1])$, $(\text{slopes}[2], \text{slopes}[3])$, etc.) defines one slope. Slopes must be sorted (i.e. both $\text{slopes}[0]$ and $\text{slopes}[1]$ must not be greater than any entry of the slope $(\text{slopes}[2], \text{slopes}[3])$, etc.) and must not be overlapping. Otherwise no slopes are set and *unur_errno* is set to UNUR_ERR_PAR_SET.

Notice: *n_slopes* is the number of slopes (and not the length of the array *slopes*).

Notice that setting slopes resets the given domain for the distribution. However, in case of a standard distribution the area below the PDF is not updated.

`int unur_tabl_set_guidefactor (UNUR_PAR* parameters, double factor)`

Set factor for relative size of the guide table for indexed search (see also method DGT [Section 5.8.3 \[DGT\], page 166](#)). It must be greater than or equal to 0. When set to 0, then sequential search is used.

Default is 1.

`int unur_tabl_set_boundary (UNUR_PAR* parameters, double left, double right)`

Set the left and right boundary of the computation interval. The piecewise hat is only constructed inside this interval. The probability outside of this region must not be of computational relevance. Of course $\pm \text{UNUR_INFINITY}$ is not allowed.

Default is $-1.e20, 1.e20$.

`int unur_tabl_chg_truncated (UNUR_GEN* gen, double left, double right)`

Change the borders of the domain of the (truncated) distribution.

Notice that the given truncated domain must be a subset of the domain of the given distribution. The generator always uses the intersection of the domain of the distribution and the truncated domain given by this call. The hat function will not be changed.

Important: The ratio between the area below the hat and the area below the squeeze changes when the sampling region is restricted. In particular it becomes (very) large when sampling from the (far) tail of the distribution. Then it is better to create a generator object for the tail of distribution only.

Important: This call does not work for variant IA (immediate acceptance). In this case UNURAN switches *automatically* to variant RH (use “classical” acceptance/rejection from hat distribution) and does revert to the variant originally set by the user.

Important: It is not a good idea to use adaptive rejection sampling while sampling from a domain that is a strict subset of the domain that has been used to construct the hat. For that reason adaptive adding of construction points is *automatically disabled* by this call.

Important: If the CDF of the hat is (almost) the same for *left* and *right* and (almost) equal to 0 or 1, then the truncated domain is not changed and the call returns an error code.

```
int unur_tabl_set_verify (UNUR_PAR* parameters, int verify)
```

```
int unur_tabl_chg_verify (UNUR_GEN* generator, int verify)
```

Turn verifying of algorithm while sampling on/off. If the condition $\text{squeeze}(x) \leq \text{PDF}(x) \leq \text{hat}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

```
int unur_tabl_set_pedantic (UNUR_PAR* parameters, int pedantic)
```

Sometimes it might happen that `unur_init` has been executed successfully. But when additional construction points are added by adaptive rejection sampling, the algorithm detects that the PDF is not monotone in the given slopes.

With *pedantic* being TRUE, the sampling routine is exchanged by a routine that simply returns `UNUR_INFINITY` indicating an error.

Default is FALSE.

5.3.12 TDR – Transformed Density Rejection

Required: T-concave PDF, dPDF

Optional: mode

Speed: Set-up: slow, Sampling: fast

Reference: [GWa92] [HWa95] [HLD04: Cha.4]

TDR is an acceptance/rejection method that uses the concavity of a transformed density to construct hat function and squeezes automatically. Such PDFs are called T-concave. Currently the following transformations are implemented and can be selected by setting their *c*-values by a `unur_tdr_set_c` call:

c = 0 $T(x) = \log(x)$

c = -0.5 $T(x) = -1/\sqrt{x}$ (Default)

In future releases the transformations $T(x) = -(x)^c$ will be available for any *c* with $0 > c > -1$. Notice that if a PDF is T-concave for a *c* then it also T-concave for every $c' < c$. However the performance decreases when *c'* is smaller than *c*. For computational reasons we suggest the usage of *c* = -0.5 (this is the default). For $c \leq -1$ the hat is not bounded any more if the domain of the PDF is unbounded. But in the case of a bounded domain using method TABL is preferred to a TDR with $c < -1$ (except in a few special cases).

We offer three variants of the algorithm.

GW squeezes between construction points

PS squeezes proportional to hat function (Default)

IA same as variant PS but uses a composition method with “immediate acceptance” in the region below the squeeze.

GW has a slightly faster setup but higher marginal generation times. PS is faster than GW. IA uses less uniform random numbers and is therefore faster than PS.

It is also possible to evaluate the inverse of the CDF of the hat distribution directly using the `unur_tdr_eval_invcdfhat` call.

There are lots of parameters for these methods, see below.

It is possible to use this method for correlation induction by setting an auxiliary uniform random number generator via the `unur_set_urng_aux` call. (Notice that this must be done after a possible `unur_set_urng` call.) When an auxiliary generator is used then the number of uniform random numbers from the first URNG that are used for one generated random variate is constant and given in the following table:

GW ... 2

PS ... 2

IA ... 1

There exists a test mode that verifies whether the conditions for the method are satisfied or not. It can be switched on by calling `unur_tdr_set_verify` and `unur_tdr_chg_verify`, respectively. Notice however that sampling is (much) slower then.

For densities with modes not close to 0 it is suggested either to set the mode of the distribution or to use the `unur_tdr_set_center` call for provide some information about the main part of the PDF to avoid numerical problems.

It is possible to use this method for generating from truncated distributions. It even can be changed for an existing generator object by an `unur_tdr_chg_truncated` call.

Important: The ratio between the area below the hat and the area below the squeeze changes when the sampling region is restricted. Especially it becomes (very) small when sampling from the (far) tail of the distribution. Then it is better to create a new generator object for the tail of the distribution only.

Function reference

UNUR_PAR* `unur_tdr_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

int `unur_tdr_reinit (UNUR_GEN* generator)`

Update an existing generator object after the distribution has been modified. It must be executed whenever the parameters of the distribution has been changed. It is faster than destroying the existing object and build a new one from scratch. If reinitialization has been successful UNUR_SUCCESS is returned. In case of a failure an error code is returned. Then *generator* cannot be used before another successful reinit (with proper parameters for the underlying distribution).

Notice that currently reinit only makes sense when the *generator* object does not use a private copy of the given distribution object. This behavior can be switched on by means of a `unur_set_use_distr_privatecopy` call. Then it is possible to manipulate the (external) distribution and (immediately after doing this) reinitialize the generator object. Obviously using this procedure must be done with extreme care.

int `unur_tdr_set_c (UNUR_PAR* parameters, double c)`

Set parameter *c* for transformation T. Currently only values between 0 and -0.5 are allowed. If *c* is between 0 and -0.5 it is set to -0.5.

Default is -0.5.

int `unur_tdr_set_variant_gw (UNUR_PAR* parameters)`

Use original version with squeezes between construction points as proposed by Gilks & Wild (1992).

int `unur_tdr_set_variant_ps (UNUR_PAR* parameters)`

Use squeezes proportional to the hat function. This is faster than the original version. This is the default.

int `unur_tdr_set_variant_ia (UNUR_PAR* parameters)`

Use squeezes proportional to the hat function together with a composition method that required less uniform random numbers.

int `unur_tdr_set_usedars (UNUR_PAR* parameters, int usedars)`

If *usedars* is set to TRUE, “derandomized adaptive rejection sampling” (DARS) is used in setup. Intervals where the area between hat and squeeze is too large compared to the average area between hat and squeeze over all intervals are split. This procedure is repeated until the ratio between area below squeeze and area below hat exceeds the bound given by `unur_tdr_set_max_sqratio` call or the maximum number of intervals is reached. Moreover, it also aborts when no more intervals can be found for splitting.

For finding splitting points the following rules are used (in this order, i.e., is if the first rule cannot be applied, the next one is used):

1. Use the expected value of adaptive rejection sampling.
2. Use the arc-mean rule (a mixture of arithmetic mean and harmonic mean).

3. Use the arithmetic mean of the interval boundaries.

Notice, however, that for unbounded intervals neither rule 1 nor rule 3 can be used.

As an additional feature, it is possible to choose among these rules. If *usedars* is set to 1 or **TRUE** the expected point (rule 1) is used (it switches to rule 2 for a particular interval if rule 1 cannot be applied). If it is set to 2 the arc-mean rule is used. If it is set to 3 the mean is used. Notice that rule 3 can only be used if the domain of the distribution is bounded. It is faster than the other two methods but for heavy-tailed distribution and large domain the hat converges extremely slowly.

The default depends on the given construction points. If the user has provided such points via a `unur_tdr_set_cpoints` call, then *usedars* is set to **FALSE** by default, i.e., there is no further splitting. If the user has only given the number of construction points (or only uses the default number), then *usedars* is set to **TRUE** (i.e., use rule 1).

`int unsur_tdr_set_darsfactor (UNUR_PAR* parameters, double factor)`

Set factor for “derandomized adaptive rejection sampling”. This factor is used to determine the intervals that are “too large”, that is, all intervals where the area between squeeze and hat is larger than *factor* times the average area over all intervals between squeeze and hat. Notice that all intervals are split when *factor* is set to 0., and that there is no splitting at all when *factor* is set to **UNUR_INFINITY**.

Default is 0.99. There is no need to change this parameter.

`int unsur_tdr_set_cpoints (UNUR_PAR* parameters, int n_stp, const double* stp)`

Set construction points for the hat function. If *stp* is **NULL** than a heuristic rule of thumb is used to get *n_stp* construction points. This is the default behavior.

The default number of construction points is 30.

`int unsur_tdr_set_reinit_percentiles (UNUR_PAR* parameters, int n_percentiles, const double* percentiles)`

`int unsur_tdr_chg_reinit_percentiles (UNUR_GEN* generator, int n_percentiles, const double* percentiles)`

By default, when the *generator* object is reinitialized, it used the same construction points as for the initialization procedure. Often the underlying distribution object has been changed only moderately. For example, the full conditional distribution of a multivariate distribution. In this case it might be more appropriate to use percentiles of the hat function for the last (unchanged) distribution. *percentiles* must then be a pointer to an ordered array of numbers between 0.01 and 0.99. If *percentiles* is **NULL**, then a heuristic rule of thumb is used to get *n_percentiles* values for these percentiles. Notice that *n_percentiles* must be at least 2, otherwise defaults are used. (Then the first and third quartiles are used by default.)

`int unsur_tdr_set_reinit_ncpoints (UNUR_PAR* parameters, int ncpoints)`

`int unsur_tdr_chg_reinit_ncpoints (UNUR_GEN* generator, int ncpoints)`

When *reinit* fails with the given construction points or the percentiles of the old hat function, another trial is undertaken with *ncpoints* construction points. *ncpoints* must be at least 10.

Default: 50

`int unsur_tdr_chg_truncated (UNUR_GEN* gen, double left, double right)`

Change the borders of the domain of the (truncated) distribution.

Notice that the given truncated domain must be a subset of the domain of the given distribution. The generator always uses the intersection of the domain of the distribution and the truncated domain given by this call. The hat function will not be changed.

Important: The ratio between the area below the hat and the area below the squeeze changes when the sampling region is restricted. In particular it becomes (very) large when sampling from the (far) tail of the distribution. Then it is better to create a generator object for the tail of distribution only.

Important: This call does not work for variant IA (immediate acceptance). In this case UNURAN switches *automatically* to variant PS.

Important: It is not a good idea to use adaptive rejection sampling while sampling from a domain that is a strict subset of the domain that has been used to construct the hat. For that reason adaptive adding of construction points is *automatically disabled* by this call.

Important: If the CDF of the hat is (almost) the same for *left* and *right* and (almost) equal to 0 or 1, then the truncated domain is not changed and the call returns an error code.

int unur_tdr_set_max_sqratio (*UNUR_PAR* parameters, double max_ratio*)

Set upper bound for the ratio (area below squeeze) / (area below hat). It must be a number between 0 and 1. When the ratio exceeds the given number no further construction points are inserted via adaptive rejection sampling. Use 0 if no construction points should be added after the setup. Use 1 if added new construction points should not be stopped until the maximum number of construction points is reached.

Default is 0.99.

double unur_tdr_get_sqratio (*const UNUR_GEN* generator*)

Get the current ratio (area below squeeze) / (area below hat) for the generator. (In case of an error UNUR_INFINITY is returned.)

double unur_tdr_get_hatarea (*const UNUR_GEN* generator*)

Get the area below the hat for the generator. (In case of an error UNUR_INFINITY is returned.)

double unur_tdr_get_squeezearea (*const UNUR_GEN* generator*)

Get the area below the squeeze for the generator. (In case of an error UNUR_INFINITY is returned.)

int unur_tdr_set_max_intervals (*UNUR_PAR* parameters, int max_ivs*)

Set maximum number of intervals. No construction points are added after the setup when the number of intervals succeeds *max_ivs*. It is increased automatically to twice the number of construction points if this is larger.

Default is 100.

int unur_tdr_set_center (*UNUR_PAR* parameters, double center*)

Set the center (approximate mode) of the PDF. It is used to find construction points by means of a heuristical rule of thumb. If the mode is given the center is set equal to the mode. It is suggested to use this call to provide some information about the main part of the PDF to avoid numerical problems.

By default the mode is used as center if available. Otherwise 0 is used.

int unur_tdr_set_usecenter (*UNUR_PAR* parameters, int usecenter*)

Use the center as construction point. Default is TRUE.

```
int unur_tdr_set_usemode (UNUR_PAR* parameters, int usemode)
```

Use the (exact!) mode as construction point. Notice that the behavior of the algorithm is different to simply adding the mode in the list of construction points via a `unur_tdr_set_cpounds` call. In the latter case the mode is treated just like any other point. However, when `usemode` is `TRUE`, the tangent in the mode is always set to 0. Then the hat of the transformed density can never cut the x-axis which must never happen if $c < 0$, since otherwise the hat would not be bounded.

Default is `TRUE`.

```
int unur_tdr_set_guidefactor (UNUR_PAR* parameters, double factor)
```

Set factor for relative size of the guide table for indexed search (see also method DGT [Section 5.8.3 \[DGT\], page 166](#)). It must be greater than or equal to 0. When set to 0, then sequential search is used.

Default is 2.

```
int unur_tdr_set_verify (UNUR_PAR* parameters, int verify)
```

```
int unur_tdr_chg_verify (UNUR_GEN* generator, int verify)
```

Turn verifying of algorithm while sampling on/off. If the condition $\text{squeeze}(x) \leq \text{PDF}(x) \leq \text{hat}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is `FALSE`.

```
int unur_tdr_set_pedantic (UNUR_PAR* parameters, int pedantic)
```

Sometimes it might happen that `unur_init` has been executed successfully. But when additional construction points are added by adaptive rejection sampling, the algorithm detects that the PDF is not T-concave.

With `pedantic` being `TRUE`, the sampling routine is exchanged by a routine that simply returns `UNUR_INFINITY`. Otherwise the new point is not added to the list of construction points. At least the hat function remains T-concave.

Setting `pedantic` to `FALSE` allows sampling from a distribution which is “almost” T-concave and small errors are tolerated. However it might happen that the hat function cannot be improved significantly. When the hat functions that has been constructed by the `unur_init` call is extremely large then it might happen that the generation times are extremely high (even hours are possible in extremely rare cases).

Default is `FALSE`.

```
double unur_tdr_eval_invcdfhat (const UNUR_GEN* generator, double u,
    double* hx, double* fx, double* sqx)
```

Evaluate the inverse of the CDF of the hat distribution at u . As a side effect the values of the hat, the density, and the squeeze at the computed point x are stored in hx , fx , and sqx , respectively. However, these computations are suppressed if the corresponding variable is set to `NULL`.

If u is out of the domain $[0,1]$ then `unur_errno` is set to `UNUR_ERR_DOMAIN` and the respective bound of the domain of the distribution are returned (which is `-UNUR_INFINITY` or `UNUR_INFINITY` in the case of unbounded domains).

Important: This call does not work for variant `IA` (immediate acceptance). In this case the hat CDF is evaluated as if variant `PS` is used.

Notice: This function always evaluates the inverse CDF of the hat distribution. A call to `unur_tdr_chg_truncated` call has no effect.

5.3.13 TDRGW – Transformed Density Rejection - Gilks&Wild variant

Required: concave logPDF, derivative of logPDF

Optional: mode

Speed: Set-up: fast, Sampling: slow

Reference: [GWa92] [HLD04: Cha.4]

TDRGW is an acceptance/rejection method that uses the concavity of the log-density function to construct hat function and squeezes automatically. It is very similar to method TDR (see [Section 5.3.12 \[TDR\], page 120](#)) with variant GW, parameter $c = 0$, and DARS switched off. Moreover, method TDRGW requires the logPDF and its derivative dlogPDF to run. On the other hand, it is designed to draw only a (very) small samples and it is much more robust against densities with very large or small areas below the PDF as it occurs, for example, in conditional distributions of (high dimensional) multivariate distributions. Additionally, it can be re-initialized when the underlying distribution has been modified. Thus it is well suited for Gibbs sampling.

Notice, that method TDRGW is a restricted version of TDR. If the full functionality of Transformed Density Rejection is needed use method [Section 5.3.12 \[TDR\], page 120](#).

How To Use

Method TDRGW is designed for distributions with log-concave densities. To use this method you need a distribution object with the logarithm of the PDF and its derivative given.

The number of construction points as well as a set of such points can be provided using `unur_tdr gw_set_cp oints`. Notice that addition construction points are added by means of adaptive rejection sampling until the maximal number of intervals given by `unur_tdr gw_set_max_intervals` is reached.

A generated distribution object can be re-initialized using the `unur_tdr gw_reinit` call. However, this only works if the generator does not use a private copy but a pointer to the distribution object. This feature can be switched on by means of a `unur_set_use_distr_privatecopy` call. Of course the distribution object given as argument to the `unur_tdr gw_new` must not be freed and `unur_tdr gw_reinit` must be called whenever this distribution object has been changed.

When `unur_tdr gw_reinit` is called construction points for the new generator are necessary. There are two options: Either the same construction points as for the initial generator (given by a `unur_tdr gw_set_cp oints` call) are used (this is the default), or percentiles of the old hat function can be used. This can be set or changed using `unur_tdr gw_set_reinit_percentiles` and `unur_tdr gw_chg_reinit_percentiles`. This feature is usefull when the underlying distribution object is only moderately changed. (An example is Gibbs sampling with small correlations.)

There exists a test mode that verifies whether the conditions for the method are satisfied or not. It can be switched on by calling `unur_tdr gw_set_verify` and `unur_tdr gw_chg_verify`, respectively. Notice however that sampling is (much) slower then.

Function reference

`UNUR_PAR* unur_tdr gw_new (const UNUR_DIST* distribution)`

Get default parameters for generator.

`int unur_tdr gw_reinit (UNUR_GEN* generator)`

Update an existing generator object after the distribution has been modified. It must be executed whenever the parameters of the distribution has been changed. It is faster than destroying the existing object and build a new one from scratch. If reinitialization has been

successful `UNUR_SUCCESS` is returned. In case of a failure an error code is returned. Then *generator* cannot be used before another successful `reinit` (with proper parameters for the underlying distribution).

Notice that currently `reinit` only makes sense when the *generator* object does not use a private copy of the given distribution object. This behavior can be switched on by means of a `unur_set_use_distr_privatecopy` call. Then it is possible to manipulate the (external) distribution and (immediately after doing this) reinitialize the generator object. Obviously using this procedure must be done with extreme care.

```
int unsur_tdrwg_set_max_intervals (UNUR_PAR* parameters, int max_ivs)
```

Set maximum number of intervals. No construction points are added after the setup when the number of intervals succeeds *max_ivs*. It is increased automatically to twice the number of construction points if this is larger.

Default is 200.

```
int unsur_tdrwg_set_cpnts (UNUR_PAR* parameters, int n_cpnts, const
                           double* cpnts)
```

Set construction points for the hat function. If *cpnts* is `NULL` then a heuristic rule of thumb is used to get *n_cpnts* construction points. This is the default behavior. *n_cpnts* should be at least 2, otherwise defaults are used.

The default number of construction points is 2.

```
int unsur_tdrwg_set_reinit_percentiles (UNUR_PAR* parameters, int
                                         n_percentiles, const double* percentiles)
```

```
int unsur_tdrwg_chg_reinit_percentiles (UNUR_GEN* generator, int
                                         n_percentiles, const double* percentiles)
```

By default, when the *generator* object is reinitialized, it used the same construction points as for the initialization procedure. Often the underlying distribution object has been changed only moderately. For example, the full conditional distribution of a multivariate distribution. In this case it might be more appropriate to use percentiles of the hat function for the last (unchanged) distribution. *percentiles* must then be a pointer to an ordered array of numbers between 0.01 and 0.99. If *percentiles* is `NULL`, then a heuristic rule of thumb is used to get *n_percentiles* values for these percentiles. Notice that *n_percentiles* must be at least 2, otherwise defaults are used. (Then the first and third quartiles are used by default.)

```
int unsur_tdrwg_set_reinit_ncpnts (UNUR_PAR* parameters, int ncpnts)
```

```
int unsur_tdrwg_chg_reinit_ncpnts (UNUR_GEN* generator, int ncpnts)
```

When `reinit` fails with the given construction points or the percentiles of the old hat function, another trial is undertaken with *ncpnts* construction points. *ncpnts* must be at least 10.

Default: 30

```
int unsur_tdrwg_set_verify (UNUR_PAR* parameters, int verify)
```

```
int unsur_tdrwg_chg_verify (UNUR_GEN* generator, int verify)
```

Turn verifying of algorithm while sampling on/off. If the condition $\text{squeeze}(x) \leq \text{PDF}(x) \leq \text{hat}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is `FALSE`.

```
int unur_tdrwg_set_pedantic (UNUR_PAR* parameters, int pedantic)
```

Sometimes it might happen that `unur_init` has been executed successfully. But when additional construction points are added by adaptive rejection sampling, the algorithm detects that the PDF is not log-concave.

With `pedantic` being `TRUE`, the sampling routine is exchanged by a routine that simply returns `UNUR_INFINITY`. Otherwise the new point is not added to the list of construction points. At least the hat function remains log-concave.

Setting `pedantic` to `FALSE` allows sampling from a distribution which is “almost” log-concave and small errors are tolerated. However it might happen that the hat function cannot be improved significantly. When the hat functions that has been constructed by the `unur_init` call is extremely large then it might happen that the generation times are extremely high (even hours are possible in extremely rare cases).

Default is `FALSE`.

```
double unur_tdrwg_get_loghatarea (const UNUR_GEN* generator)
```

Get the logarithm of area below the hat for the generator. (In case of an error `UNUR_INFINITY` is returned.)

```
double unur_tdrwg_eval_invcdfhat (const UNUR_GEN* generator, double u)
```

Evaluate the inverse of the CDF of the hat distribution at `u`.

If `u` is out of the domain `[0,1]` then `unur_errno` is set to `UNUR_ERR_DOMAIN` and the respective bound of the domain of the distribution are returned (which is `-UNUR_INFINITY` or `UNUR_INFINITY` in the case of unbounded domains).

5.3.14 UTDR – Universal Transformed Density Rejection

Required: T-concave PDF, mode, approximate area

Speed: Set-up: moderate, Sampling: Moderate

Reference: [HWa95] [HLD04: Sect.4.5.4; Alg.4.4]

UTDR is based on the transformed density rejection and uses three almost optimal points for constructing hat and squeezes. It works for all T -concave distributions with $T(x) = -1/\sqrt{x}$.

It requires the PDF and the (exact) location of the mode. Notice that if no mode is given at all, a (slow) numerical mode finder will be used. Moreover the approximate area below the given PDF is used. (If no area is given for the distribution the algorithm assumes that it is approximately 1.) The rejection constant is bounded from above by 4 for all T -concave distributions.

How To Use

UTDR works for any continuous univariate distribution object with given T -concave PDF (with $T(x) = -1/\sqrt{x}$), mode and approximate area below PDF.

It is possible to change the parameters and the domain of the chosen distribution without building a new generator object by using the `unur_utdr_chg_pdfparams` and `unur_utdr_chg_domain` call, respectively. But then `unur_utdr_chg_mode` and `unur_utdr_chg_pdfarea` have to be used to reset the corresponding figures whenever these have changed. Before sampling from the distribution again, `unur_utdr_reinit` must be executed. (Otherwise the generator produces garbage).

When the PDF does not change at the mode for varying parameters, then this value can be set with `unur_utdr_set_pdfatmode` to avoid some computations. Since this value will not be updated any more when the parameters of the distribution are changed, the `unur_utdr_chg_pdfatmode` call is necessary to do this manually.

There exists a test mode that verifies whether the conditions for the method are satisfied or not. It can be switched on by calling `unur_utdr_set_verify` and `unur_utdr_chg_verify`, respectively. Notice however that sampling is slower then.

Function reference

`UNUR_PAR* unsur_utdr_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unsur_utdr_reinit (UNUR_GEN* generator)`

Update an existing generator object after the distribution has been modified. It must be executed whenever the parameters or the domain of the distributions has been changed (see below). It is faster than destroying the existing object and building a new one from scratch. If reinitialization has been successful `UNUR_SUCCESS` is returned, in case of a failure an error code is returned.

Important: Do not use the *generator* object for sampling after a failed reinit, since otherwise it may produce garbage.

`int unsur_utdr_set_pdfatmode (UNUR_PAR* parameters, double fmode)`

Set pdf at mode. When set, the PDF at the mode is never changed. This is to avoid additional computations, when the PDF does not change when parameters of the distributions vary. It is only useful when the PDF at the mode does not change with changing parameters for the distribution.

Default: not set.


```
int unur_utdr_set_cpfactor (UNUR_PAR* parameters, double cp_factor)
```

Set factor for position of left and right construction point. The *cp_factor* is used to find almost optimal construction points for the hat function. There is no need to change this factor in almost all situations.

Default is 0.664.

```
int unur_utdr_set_deltafactor (UNUR_PAR* parameters, double delta)
```

Set factor for replacing tangents by secants. higher factors increase the rejection constant but reduces the risk of serious round-off errors. There is no need to change this factor it almost all situations.

Default is 1.e-5.

```
int unur_utdr_set_verify (UNUR_PAR* parameters, int verify)
```

```
int unur_utdr_chg_verify (UNUR_GEN* generator, int verify)
```

Turn verifying of algorithm while sampling on/off. If the condition $\text{squeeze}(x) \leq \text{PDF}(x) \leq \text{hat}(x)$ is violated for some x then *unur_errno* is set to *UNUR_ERR_GEN_CONDITION*. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

```
int unur_utdr_chg_pdfparams (UNUR_GEN* generator, double* params, int
                             n_params)
```

Change array of parameters of the distribution in a given generator object.

For standard distributions from the UNURAN library the parameters are checked. If these are invalid, then an error code is returned. Moreover the domain is updated automatically unless it has been changed before by a *unur_distr_discr_set_domain* call. Notice that optional parameters are (re-)set to their default values if not given for UNURAN standard distributions.

For other distributions *params* is simply copied into to distribution object. It is only checked that *n_params* does not exceed the maximum number of parameters allowed. Then an error code is returned and *unur_errno* is set to *UNUR_ERR_DISTR_NPARAMS*.

```
int unur_utdr_chg_domain (UNUR_GEN* generator, double left, double right)
```

Change left and right border of the domain of the (truncated) distribution. If the mode changes when the domain of the (truncated) distribution is changed, then a correspondig *unur_utdr_chg_mode* is required. (There is no domain checking as in the *unur_init* call.)

```
int unur_utdr_chg_mode (UNUR_GEN* generator, double mode)
```

Change mode of distribution. *unur_utdr_reinit* must be executed before sampling from the generator again.

```
int unur_utdr_upd_mode (UNUR_GEN* generator)
```

Recompute the mode of the distribution. See *unur_distr_cont_upd_mode* for more details. *unur_srou_reinit* must be executed before sampling from the generator again.

```
int unur_utdr_chg_pdfatmode (UNUR_GEN* generator, double fmode)
```

Change PDF at mode of distribution. *unur_utdr_reinit* must be executed before sampling from the generator again.

```
int unur_utdr_chg_pdfarea (UNUR_GEN* generator, double area)
```

Change area below PDF of distribution. *unur_utdr_reinit* must be executed before sampling from the generator again.

```
int unur_utdr_upd_pdfarea (UNUR_GEN* generator)
```

Recompute the area below the PDF of the distribution. It only works when a distribution objects from the UNURAN library of standard distributions is used (see [Chapter 7 \[Standard distributions\]](#), [page 187](#)). Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_srou_reinit` must be executed before sampling from the generator again.

5.4 Methods for continuous empirical univariate distributions

Overview of methods

Methods for **continuous empirical univariate distributions**

sample with `unur_sample_cont`

EMPK: Requires an observed sample. EMPL: Requires an observed sample.

Example

```
/* ----- */
/* File: example_emp.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* Example how to sample from an empirical continuous univariate */
/* distribution. */

/* ----- */

int main()
{
    int    i;
    double x;

    /* data points */
    double data[15] = { -0.1,  0.05, -0.5,  0.08,  0.13,\
                        -0.21,-0.44, -0.43, -0.33, -0.3, \
                        0.18, 0.2,  -0.37, -0.29, -0.9 };

    /* Declare the three UNURAN objects. */
    UNUR_DISTR *distr; /* distribution object */
    UNUR_PAR   *par;   /* parameter object */
    UNUR_GEN   *gen;   /* generator object */

    /* Create a distribution object and set empirical sample. */
    distr = unsur_distr_cemp_new();
    unsur_distr_cemp_set_data(distr, data, 15);

    /* Choose a method: EMPK. */
    par = unsur_empk_new(distr);

    /* Set smooting factor. */
    unsur_empk_set_smoothing(par, 0.8);

    /* Create the generator object. */
    gen = unsur_init(par);

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* It is possible to reuse the distribution object to create */
```

```

/* another generator object. If you do not need it any more, */
/* it should be destroyed to free memory. */
unur_distr_free(distr);

/* Now you can use the generator object 'gen' to sample from */
/* the distribution. Eg.: */
for (i=0; i<10; i++) {
    x = unur_sample_cont(gen);
    printf("%f\n",x);
}

/* When you do not need the generator object any more, you */
/* can destroy it. */
unur_free(gen);

exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */

```

Example (String API)

```

/* ----- */
/* File: example_emp_str.c */
/* ----- */
/* String API. */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* Example how to sample from an empirical continuous univariate */
/* distribution. */

/* ----- */

int main()
{
    int i;
    double x;

    /* Declare UNURAN generator object. */
    UNUR_GEN *gen; /* generator object */

    /* Create the generator object. */
    gen = unur_str2gen("distr = cemp; \
                        data=(-0.10, 0.05,-0.50, 0.08, 0.13, \
                          -0.21,-0.44,-0.43,-0.33,-0.30, \
                          0.18, 0.20,-0.37,-0.29,-0.90) & \
                        method=empk; smoothing=0.8");

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Now you can use the generator object 'gen' to sample from */
    /* the distribution. Eg.: */
    for (i=0; i<10; i++) {

```

```
    x = unur_sample_cont(gen);
    printf("%f\n",x);
}

/* When you do not need the generator object any more, you    */
/* can destroy it.                                           */
unur_free(gen);

exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */
```

5.4.1 EMPK – EMPirical distribution with Kernel smoothing

Required: observed sample

Speed: Set-up: slow (as sample is sorted), Sampling: fast (depends on kernel)

Reference: [HLa00] [HLD04: Sect.12.1.2]

EMPK generates random variates from an empirical distribution that is given by an observed sample. The idea is that simply choosing a random point from the sample and to return it with some added noise results in a method that has very nice properties, as it can be seen as sampling from a kernel density estimate. If the underlying distribution is continuous, especially the fine structure of the resulting empirical distribution is much better than using only resampling without noise.

Clearly we have to decide about the density of the noise (called kernel) and about the standard deviation of the noise. The mathematical theory of kernel density estimation shows us that we are comparatively free in choosing the kernel. It also supplies us with a simple formula to compute the optimal standard deviation of the noise, called bandwidth (or window width) of the kernel.

The variance of the estimated density is slightly larger than that of the observed sample. However, this can be easily corrected if required.

There is also a correction (mirroring technique) for distributions with non-negative support.

A simple robust reference method is implemented to find a good standard deviation of the noise (i.e. the bandwidth of kernel density estimation). For some cases (e.g. densities with two or more sharp distinct peaks) there kernel density estimation can be adjusted by changing the smoothness factor and the so called beta factor.

How To Use

EMPK uses empirical distributions. The main parameter is the choice if of kernel density. The most important kernels can be set by `unur_empk_set_kernel`. Additionally generators for other kernels can be used by using `unur_empk_set_kernel` instead. Additionally variance correction and a correction for non-negative variates can be switched on.

The two other parameters (smoothing factor and beta factor) are only useful for people knowing the theory of kernel density estimation. It is not necessary to change them if the true underlying distribution is somehow comparable with a bell-shaped curve, even skewed or with some not too sharp extra peaks. In all these cases the simple robust reference method implemented to find a good standard deviation of the noise (i.e. the bandwidth of kernel density estimation) should give sensible results. However, it might be necessary to overwrite this automatic method to find the bandwidth eg. when resampling from data with two or more sharp distinct peaks. Then the distribution has nearly discrete components as well and our automatic method may easily choose too large a bandwidth which results in an empirical distribution which is oversmoothed (i.e. it has lower peaks than the original distribution). Then it is recommended to decrease the bandwidth using the `unur_empk_set_smoothing` call. A smoothing factor of 1 is the default. A smoothing factor of 0 leads to naive resampling of the data. Thus an appropriate value between these extremes should be chosen. We recommend to consult a reference on kernel smoothing when doing so; but it is not a simple problem to determine an optimal bandwidth for distributions with sharp peaks.

In general, for most applications it is perfectly ok to use the default values offered. Unless you have some knowledge on density estimation we do not recommend to change anything. There are two exceptions:

- A. In the case that the unknown underlying distribution is not continuous but discrete you should "turn off" the adding of the noise by setting:

```
unur_empk_set_smoothing(par, 0.)
```

- B. In the case that you are especially interested in a fast sampling algorithm use the call

```
unur_empk_set_kernel(par, UNUR_DISTR_BOXCAR);
```

to change the used noise distribution from the default Gaussian distribution to the uniform distribution.

Function reference

UNUR_PAR* `unur_empk_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

int `unur_empk_set_kernel (UNUR_PAR* parameters, unsigned kernel)`

Select one of the supported kernel distributions. Currently the following kernels are supported:

UNUR_DISTR_GAUSSIAN

Gaussian (normal) kernel

UNUR_DISTR_EPANECHNIKOV

Epanechnikov kernel

UNUR_DISTR_BOXCAR

Boxcar (uniform, rectangular) kernel

UNUR_DISTR_STUDENT

t3 kernel (Student's distribution with 3 degrees of freedom)

UNUR_DISTR_LOGISTIC

logistic kernel

For other kernels (including kernels with Student's distribution with other than 3 degrees of freedom) use the `unur_empk_set_kernelgen` call.

It is not possible to call `unur_empk_set_kernel` twice.

Default is the Gaussian kernel.

int `unur_empk_set_kernelgen (UNUR_PAR* parameters, const UNUR_GEN* kernelgen, double alpha, double kernelvar)`

Set generator for the kernel used for density estimation.

alpha is used to compute the optimal bandwidth from the point of view of minimizing the mean integrated square error (MISE). It depends on the kernel *K* and is given by

$$\alpha(K) = \text{Var}(K)^{-2/5} \left\{ \int K(t)^2 dt \right\}^{1/5}$$

For standard kernels (see above) *alpha* is computed by the algorithm.

kernvar is the variance of the used kernel. It is only required for the variance corrected version of density estimation (which is used by default); otherwise it is ignored. If *kernelvar* is nonpositive, variance correction is disabled. For standard kernels (see above) *kernvar* is computed by the algorithm.

It is not possible to call `unur_empk_set_kernelgen` after a standard kernel has been selected by a `unur_empk_set_kernel` call.

Notice that the uniform random number generator of the kernel generator is overwritten during the `unur_init` call and at each `unur_chg_urng` call with the uniform generator used for the empirical distribution.

Default is the Gaussian kernel.

```
int unur_empk_set_beta (UNUR_PAR* parameters, double beta)
```

beta is used to compute the optimal bandwidth from the point of view of minimizing the mean integrated square error (MISE). *beta* depends on the (unknown) distribution of the sampled data points. By default Gaussian distribution is assumed for the sample (*beta* = 1.3637439). There is no requirement to change *beta*.

Default: 1.3637439

```
int unur_empk_set_smoothing (UNUR_PAR* parameters, double smoothing)
```

```
int unur_empk_chg_smoothing (UNUR_GEN* generator, double smoothing)
```

Set and change the smoothing factor. The smoothing factor controls how “smooth” the resulting density estimation will be. A smoothing factor equal to 0 results in naive resampling. A very large smoothing factor (together with the variance correction) results in a density which is approximately equal to the kernel. Default is 1 which results in a smoothing parameter minimising the MISE (mean integrated squared error) if the data are not too far away from normal. If a large smoothing factor is used, then variance correction must be switched on.

Default: 1

```
int unur_empk_set_varcor (UNUR_PAR* parameters, int varcor)
```

```
int unur_empk_chg_varcor (UNUR_GEN* generator, int varcor)
```

Switch variance correction in generator on/off. If *varcor* is TRUE then the variance of the used density estimation is the same as the sample variance. However this increases the MISE of the estimation a little bit.

Default is FALSE.

```
int unur_empk_set_positive (UNUR_PAR* parameters, int positive)
```

If *positive* is TRUE then only nonnegative random variates are generated. This is done by means of a mirroring technique.

Default is FALSE.

5.4.2 EMPL – EMPirical distribution with Linear interpolation

Required: observed sample

Speed: Set-up: slow (as sample is sorted), Sampling: very fast (inversion)

Reference: [HLa00] [HLD04: Sect.12.1.3]

EMPL generates random variates from an empirical distribution that is given by an observed sample. This is done by linear interpolation of the empirical CDF. Although this method is suggested in the books of Law and Kelton (2000) and Bratly, Fox, and Schrage (1987) we do not recommend this method at all since it has many theoretical drawbacks: The variance of empirical distribution function does not coincide with the variance of the given sample. Moreover, when the sample increases the empirical density function does not converge to the density of the underlying random variate. Notice that the range of the generated point set is always given by the range of the given sample.

This method is provided in UNURAN for the sake of completeness. We always recommend to use method EMPK (see [Section 5.4.1 \[EMPirical distribution with Kernel smoothing\]](#), page 134).

If the data seem to be far away from having a bell shaped histogram, then we think that naive resampling is still better than linear interpolation.

How To Use

EMPL creates and samples from an empirical distribution by linear interpolation of the empirical CDF. There are no parameters to set.

Important: We do not recommend to use this method! Use method EMPK (see [Section 5.4.1 \[EMPirical distribution with Kernel smoothing\]](#), page 134) instead.

Function reference

UNUR_PAR* `unur_empl_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

5.5 Methods for continuous multivariate distributions

Overview of methods

Methods for **continuous multivariate distributions**

sample with `unur_sample_vec`

VMT: Requires the mean vector and the covariance matrix.

5.5.1 BALL – ball-sampler

Required: PDF

Optional: mode, center, bounding rectangle for acceptance region

Speed: Set-up: fast, Sampling: fast

BALL is an implementation of a ball-sampler.

It is possible to sample directly from the area below the pdf or from the area inside the Ratio of Uniforms shape as chosen by the appropriate `set_variant` calls.

In both variants the ball is moving around in a $(\text{dim}+1)$ -dimensional space ($u[],v$) for the RoU-variant and $(x[],y)$ for the PDF variant)

The (initial) radius of the ball can be set using the `unur_ball_set_ball_radius` call. The radius of the ball is only changed when sampling using the adaptive method has been chosen. In this case, the ball-radius is increased/decreased by a constant factor whenever the candidate point (i.e. current point + random direction vector) fall inside/outside the shape : RoU-shape for the RoU-variant, Volume below PDF for the PDF variant.

In case of the ratio-of-uniform variant an additional parameter r that can be used for adjusting the algorithm to the given distribution to improve performance and/or to make this method applicable. Moreover, this implementation uses the center μ of the distribution (which is set to the mode or mean by default, see `unur_distr_cvec_get_center` for details of its default values).

How To Use

For using the BALL method UNURAN needs the PDF of the distribution. By a call to `unur_ball_set_variant_rou` or `unur_ball_set_variant_pdf` the choice can be made if sampling should be done inside the ratio-of-uniforms shape or below the pdf.

Additionally, for the ratio-of-uniforms variant, the parameter r can be set via a `unur_vnrou_set_r` call.

Function reference

`UNUR_PAR* unsur_ball_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unsur_ball_set_variant_pdf (UNUR_PAR* par)`

Sampler variant : Sampling from the area below pdf. (not yet implemented)

`int unsur_ball_set_variant_rou (UNUR_PAR* par)`

Sampler variant : Sampling from the area inside the RoU shape.

This is the default.

`int unsur_ball_set_r (UNUR_PAR* par, double r)`

Sets the parameter r of the generalized multivariate ratio-of-uniforms method.

Notice: This parameter must satisfy $r > 0$.

Default: 1.

`int unsur_ball_set_ball_radius (UNUR_PAR* par, double ball_radius)`

Sets (initial) radius of ball used for the ball-sampler.

Default: 1.0

int unur_ball_set_adaptive_ball (*UNUR_PAR** *par*, *int adaptive_flag*)

Increasing/Decreasing ball radius whenever candidate point is inside/outside the RoU shape.

Default: 0.

int unur_ball_set_adaptive_factor (*UNUR_PAR** *par*, *double adaptive_factor*)

Increasing/Decreasing ball radius by the factor given whenever candidate point (i.e. current point + random direction vector) is inside/outside the RoU shape.

Default: 1.01.

int unur_ball_set_thinning (*UNUR_PAR** *par*, *long thinning*)

Sets the parameter *thinning*. A thinning parameter of *n* means, that the interval size between two sampled points (returned random vectors) is *n*, whereas the internally sampled points are being a unit interval apart.

Notice: This parameter must satisfy *thinning* ≥ 1.

Default: 1.

5.5.2 NORTA – NORmal To Anything

Required: rank correlation matrix, marginal distributions

Speed: Set-up: slow, Sampling: depends on dimension

Reference: [HLD04: Sect.12.5.2; Alg.12.11.]

NORTA (NORmal to anything) is a model to get random vectors with given marginal distributions and rank correlation.

Important: Notice that marginal distribution and (rank) correlation structure do not uniquely define a multivariate distribution. Thus there are many other (more or less sensible) models.

In the NORTA model multinormal random variates with the given rank (Spearman's) correlations are generated. In a second step the (standard normal distributed) marginal variates are transformed by means of the CDF of the normal distribution to get uniform marginals. The resulting random vectors have uniform marginals and the desired rank correlation between its components. Such a random vector is called 'copula'.

By means of the inverse CDF the uniform marginals are then transformed into the target marginal distributions. This transformation does not change the rank correlation.

For the generation of the multinormal distribution the (Spearman's) rank correlation matrix is transformed into the corresponding (Pearson) correlation matrix and the VMT method (see [Section 5.5.4 \[VMT\], page 143](#)) is used to sample from the resulting multinormal distribution (by means of the Cholesky decomposition of the covariance matrix). It can happen that the desired rank correlation matrix is not feasible, i.e., it cannot occur as rank correlation matrix of a multinormal distribution. The resulting "covariance" matrix is not positive definite. In this an eigenvector correction method is used. Then all non-positive eigenvalues are set to a small positive value and hence the rank correlation matrix of the generated random vectors is "close" to the desired matrix.

How To Use

Create a multivariate generator object and set marginal distributions using `unur_distr_cvec_set_marginals`, `unur_distr_cvec_set_marginal_array`, or `unur_distr_cvec_set_marginal_list`. (Do not use the corresponding calls for the standard marginal distributions).

If copulae are required (i.e. multivariate distributions with uniform marginals) such a generator object can be created by means of `unur_distr_copula`.

There are no optional parameters for this method.

Function reference

`UNUR_PAR* unur_norta_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

5.5.3 VAROU – Multivariate Adaptive Ratio-Of-Uniforms method

Required: PDF, dPDF, ...

Optional: ...

Speed: Set-up: slow, Sampling: slow

Reference: [WGS91]

VAROU is an implementation of the multivariate ratio-of-uniforms method which uses a piecewise continuous hut consisting of a family of conii with tops at the origin of the $(U_1, U_2, \dots, U_d, V)$ coordinate system and spanning vectors pointing towards an initial triangulation of the upper half unit-sphere.

The base of such conii is given as the tangent-plane to the surface :

$$\{ (U_1, \dots, U_d, V): V = PDF(U_1/V, \dots, U_d/V)^{1/(d+1)} \}$$

through one of the pierce-points of the spanning vectors. In particular, we choose the point 'nearest' to $(0, \dots, 0, V_{max})$ as the construction point for the tangent-plane.

VAROU is based on the rejection method (see [Section A.2 \[Rejection\]](#), page 214). And it is important to note that the acceptance probability decreases exponentially with dimension. Thus even for moderately many dimensions (e.g. 5) the number of repetitions to get one random vector can be prohibitively large and the algorithm seems to stay in an infinite loop.

How To Use

For using the VAROU method UNURAN needs the PDF of the distribution.

Function reference

UNUR_PAR* unur_varou_new (*const UNUR_DISTR* distribution*)

Get default parameters for generator.

int unur_varou_set_cones (*UNUR_PAR* parameters, long ncones*)

Sets the number of cones to be used in the setup.

Default is 10000.

int unur_varou_set_verify (*UNUR_PAR* parameters, int verify*)

Turn verifying of algorithm while sampling on/off.

If the condition $PDF(x) \leq \hat{PDF}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

int unur_varou_chg_verify (*UNUR_GEN* generator, int verify*)

Change the verifying of algorithm while sampling on/off.

5.5.4 VMT – Vector Matrix Transformation

Required: mean vector, covariance matrix, standardized marginal distributions

Speed: Set-up: slow, Sampling: depends on dimension

Reference: [HLD04: Sect.11.1.6; Alg.11.3.]

VMT generates random vectors for distributions with given mean vector μ and covariance matrix Σ . It produces random vectors of the form $X = LY + \mu$, where L is the Cholesky factor of Σ , i.e. $LL^t = \Sigma$, and Y has independent components of the same distribution with mean 0 and standard deviation 1.

The method VMT has been implemented especially to sample from a multinormal distribution. Nevertheless, it can also be used (or abused) for other distributions. However, notice that the given standardized marginal distributions are not checked; i.e. if the given distributions do not have mean 0 and variance 1 then μ and Σ are not the mean vector and covariance matrix, respectively, of the resulting distribution.

Important: Notice that except for the multinormal distribution the given marginal distribution are distorted by the transformation using the Cholesky matrix. Thus for other (non-multinormal) distributions this method should only be used when everything else fails and some approximate results which might even be not entirely correct are better than no results. A much better method is the NORTA (NORmal To Anything) method, see [Section 5.5.2 \[NORTA\]](#), [page 141](#).

How To Use

Create a multivariate generator object, set mean vector and covariance matrix by means of the `unur_distr_cvec_set_mean` and `unur_distr_cvec_set_covar` call. Set standard marginal distributions using `unur_distr_cvec_set_stdmarginals`, `unur_distr_cvec_set_stdmarginal_array`, or `unur_distr_cvec_set_stdmarginal_list`. (Do not use the corresponding calls for the (non-standard) marginal distributions).

There are no optional parameters for this method.

Function reference

`UNUR_PAR* unur_vmt_new (const UNUR_DISTR* distribution)`

Get parameters for generator.

5.5.5 VNROU – Multivariate Naive Ratio-Of-Uniforms method

Required: PDF

Optional: mode, center, bounding rectangle for acceptance region

Speed: Set-up: fast or slow, Sampling: slow

Reference: [WGS91]

VNROU is an implementation of the multivariate ratio-of-uniforms method which uses a (minimal) bounding hyper-rectangle, see also [Section A.4 \[Ratio-of-Uniforms\]](#), page 216. It uses an additional parameter r that can be used for adjusting the algorithm to the given distribution to improve performance and/or to make this method applicable. Larger values of r increase the class of distributions for which the method works at the expense of higher rejection constants. Moreover, this implementation uses the center μ of the distribution (which is set to the mode or mean by default, see `unur_distr_cvec_get_center` for details of its default values).

The minimal bounding has then the coordinates

$$\begin{aligned} v^+ &= \sup_x (f(x))^{1/r d+1}, \\ u_i^- &= \inf_{x_i} (x_i - \mu_i) (f(x))^{r/r d+1}, \\ u_i^+ &= \sup_{x_i} (x_i - \mu_i) (f(x))^{r/r d+1}, \end{aligned}$$

where x_i is the i -th coordinate of point x ; μ_i is the i -th coordinate of the center μ . d denotes the dimension of the distribution. These bounds can either be given directly, or are computed automatically by means of a numerical routine by Hooke and Jeeves [HJa61] called direct search (see ‘`src/utls/hooke.c`’ for further references and details). Of course this algorithm can fail, especially when this rectangle is not bounded.

It is important to note that the algorithm works with $PDF(x - center)$ instead of $PDF(x)$, i.e. the bounding rectangle has to be provided for $PDF(x - center)$. This is important as otherwise the acceptance region can become a very long and skinny ellipsoid along a diagonal of the (huge) bounding rectangle.

VNROU is based on the rejection method (see [Section A.2 \[Rejection\]](#), page 214), and it is important to note that the acceptance probability decreases exponentially with dimension. Thus even for moderately many dimensions (e.g. 5) the number of repetitions to get one random vector can be prohibitively large and the algorithm seems to stay in an infinite loop.

How To Use

For using the VNROU method UNURAN needs the PDF of the distribution. Additionally, the parameter r can be set via a `unur_vnrou_set_r` call. Notice that the acceptance probability decreases when r is increased. On the other hand is more unlikely that the bounding rectangle does not exist if r is small.

A bounding rectangle can be given by the `unur_vnrou_set_u` and `unur_vnrou_set_v` calls.

Important: The bounding rectangle has to be provided for the function $PDF(x - center)$! Notice that `center` is the center of the given distribution, see `unur_distr_cvec_set_center`. If in doubt or if this value is not optimal, it can be changed (overridden) by a `unur_distr_cvec_set_center` call.

If the coordinates of the bounding rectangle are not provided by the user then the minimal bounding rectangle is computed automatically.

By means of `unur_vnrou_set_verify` and `unur_vnrou_chg_verify` one can run the sampling algorithm in a checking mode, i.e., in every cycle of the rejection loop it is checked whether the used rectangle indeed enclosed the acceptance region of the distribution. When in doubt

(e.g., when it is not clear whether the numerical routine has worked correctly) this can be used to run a small Monte Carlo study.

Important: The rejection constant (i.e. the expected number of iterations for generating one random vector) can be extremely high, in particular when the dimension is 4 or higher. Then the algorithm will perform almost infinite loops. Thus it is recommended to read the volume below the hat function by means of the `unur_vnrou_get_volumehat` call. The returned number divided by the volume below the PDF (which is 1 in case of a normalized PDF) gives the rejection constant.

Function reference

`UNUR_PAR* unsur_vnrou_new (const UNUR_DIST* distribution)`

Get default parameters for generator.

`int unsur_vnrou_set_u (UNUR_PAR* parameters, double* umin, double* umax)`

Sets left and right boundaries of bounding hyper-rectangle. If no values are given, the boundary of the minimal bounding hyper-rectangle is computed numerically.

Important: The boundaries are those of the density shifted by the center of the distribution, i.e., for the function $PDF(x - center)$!

Notice: Computing the minimal bounding rectangle may fail under some circumstances. Moreover, for multimodal distributions the bounds might be too small as only local extrema are computed. Nevertheless, for log-concave distributions it should work.

Default: not set (i.e. computed automatically)

`int unsur_vnrou_set_v (UNUR_PAR* parameters, double vmax)`

Set upper boundary for bounding hyper-rectangle. If no values are given, the density at the mode is evaluated. If no mode is given for the distribution it is computed numerically (and might fail).

Default: not set (i.e. computed automatically)

`int unsur_vnrou_set_r (UNUR_PAR* parameters, double r)`

Sets the parameter r of the generalized multivariate ratio-of-uniforms method.

Notice: This parameter must satisfy $r > 0$.

Default: 1.

`int unsur_vnrou_set_verify (UNUR_PAR* parameters, int verify)`

Turn verifying of algorithm while sampling on/off.

If the condition $PDF(x) \leq \hat{f}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

`int unsur_vnrou_chg_verify (UNUR_GEN* generator, int verify)`

Change the verifying of algorithm while sampling on/off.

`double unsur_vnrou_get_volumehat (const UNUR_GEN* generator)`

Get the volume of below the hat. For normalized densities, i.e. when the volume below PDF is 1, this value equals the rejection constant for the vnrou method.

In case of an error `UNUR_INFINITY` is returned.

5.5.6 WALK – Metropolis random-walk sampler

Required: PDF

Optional: mode, center, bounding rectangle for acceptance region

Speed: Set-up: fast, Sampling: fast

WALK is an implementation of Metropolis random walk sampler.

How To Use

For using the WALK method UNURAN needs the PDF of the distribution.

Function reference

`UNUR_PAR* unur_walk_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unur_walk_set_ball_radius (UNUR_PAR* par, double ball_radius)`

Sets radius of ball used for the random walk sampler.

Default: 1.0

`int unur_walk_set_thinning (UNUR_PAR* par, long thinning)`

Sets the parameter *thinning*. A thinning parameter of n means, that the interval size between two sampled points (returned random vectors) is n, whereas the internally sampled points are being a unit interval apart.

Notice: This parameter must satisfy *thinning* ≥ 1.

Default: 1.

5.6 Markov chain samplers for continuous multivariate distributions

Markov chain samplers generate sequences of random vectors which have the target distribution as stationary distribution. There generated vectors are (more or less) correlated and it might take a long time until the sequence has converged to the given target distribution.

Beware: MCMC sampling can be dangerous!

5.6.1 GIBBS – Markov Chain - GIBBS sampler

Required: T-concave logPDF, derivatives of logPDF

Speed: Set-up: fast, Sampling: moderate

Reference: [HLD04: Sect.14.1.2]

Method GIBBS implements a Gibbs sampler for a multivariate distribution with given joint density and its gradient. When running such a Markov chain all coordinates are updated cyclically using full conditional distributions. After each step the state of the chain is returned (i.e., a random point is returned whenever a single coordinate has been updated). It is also possible to return only points after all coordinates have been updated by "thinning" the chain. Moreover, to reduce autocorrelation this thinning factor can be any integer. Notice, however, that the sampling time for a chain of given length is increased by the same factor, too.

GIBBS also provides a variant of the Gibbs sampler where in each step a point from the full conditional distribution along some random direction is sampled. This direction is chosen uniformly from the sphere in each step. This method is also known as Hit-and-Run algorithm for non-uniform distributions.

Our experiences shows that the original Gibbs sampler with sampling along coordinate axes is superior to random direction sampling as long as the correlations between the components of the random vector are not too high.

For both variants transformed density rejection (see methods see [Section 5.3.12 \[TDR\]](#), [page 120](#) and see [Section 5.3.13 \[TDRGW\]](#), [page 125](#)) is used to sample from the full conditional distributions. In opposition to the univariate case, it is important that the factor `c` is as large as possible. I.e., for a log-concave density `c` must be set to 0., since otherwise numerical underflow might stop the algorithm.

Important: GIBBS does not generate independent random points. The starting point of the Gibbs chain must be in a "typical" region of the target distribution. If such a point is not known or would be too expensive, then the first part of the chain should be discarded (burn-in of the chain).

How To Use

For using the GIBBS method UNURAN needs the logarithm of the PDF of the multivariate joint distribution and its gradient or partial derivatives.

It provides two variants:

coordinate direction sampling (Gibbs sampling) [default]

The coordinates are updated cyclically. It requires the partial derivatives of the (logarithm of the) PDF of the target distribution, see `unur_distr_cvec_set_pdlogpdf`. Otherwise, the gradient of the logPDF (see `unur_distr_cvec_set_dlogpdf`) is used, which is more expensive.

This variant can be selected using `unur_gibbs_set_variant_coordinate`.

random direction sampling (nonuniform Hit-and-Run algorithm)

In each step is a direction is sampled uniformly from the sphere and the next point in the chain is sampled from the full conditional distribution along this direction.

It requires the gradient of the logPDF and thus each step is more expensive than each step for coordinate direction sampling.

This variant can be selected using `unur_gibbs_set_variant_random_direction`.

It is important that the `c` parameter for the TDR method is as large as possible. For logconcave distribution it must be set to 0, since otherwise numerical underflow can cause the algorithm to stop.

The starting point of the Gibbs chain must be "typical" for the target distribution. If such a point is not known or would be too expensive, then the first part of the chain should be discarded (burn-in of the chain). When using the `unur_gibbs_set_burnin` call this is done during the setup of the Gibbs sampler object.

In case of a fatal error in the generator for conditional distributions the methods generates points that contain `UNUR_INFINITY`.

Warning: The algorithm requires that all full conditionals for the given distribution object are T -concave. However, this property is not checked. If this property is not satisfied, then generation from the conditional distributions becomes (very) slow and might fail or (even worse) produces random vectors from an incorrect distribution. When using `unur_gibbs_set_burnin` then the setup already might fail. Thus when in doubt whether GIBBS can be used for the target distribution it is a good idea to use a burn-in for checking.

Warning: Be careful with debugging flags. If it contains flag `0x01000000u` it produces a lot of output for each step in the algorithm. (This flag is switched off in the default debugging flags).

Function reference

`UNUR_PAR* unsur_gibbs_new (const UNUR_DISTR* distribution)`

.....

`int unsur_gibbs_set_variant_coordinate (UNUR_PAR* parameters)`

Coordinate Direction Sampling: Sampling along the coordinate directions (cyclic).

This is the default.

`int unsur_gibbs_set_variant_random_direction (UNUR_PAR* parameters)`

Random Direction Sampling: Sampling along the random directions.

`int unsur_gibbs_set_c (UNUR_PAR* parameters, double c)`

Set parameter c for transformation T of the transformed density rejection method. Currently only values between 0 and -0.5 are allowed. If c is between 0 and -0.5 it is set to -0.5.

For $c = 0$ (for logconcave densities) method TDRGW (see [Section 5.3.13 \[TDRGW\]](#), page 125) is used which is very robust against badly normalized PDFs. For other values method TDR (see [Section 5.3.12 \[TDR\]](#), page 120) is used.

The value for c should be as large as possible to avoid fatal numerical underflows. Thus for log-concave distributions c must be set to 0.

Default is 0.

`int unsur_gibbs_set_startingpoint (UNUR_PAR* parameters, const double* x0)`

Sets the starting point of the Gibbs sampler. $x0$ must be a "typical" point of the given distribution. If such a "typical" point is not known and a starting point is merely guessed, the first part of the Gibbs chain should be discarded (*burn-in*), e.g. by mean of the `unur_gibbs_set_burnin` call.

Default is the result of `unur_distr_cvec_get_center` for the given distribution object.

`int unsur_gibbs_set_thinning (UNUR_PAR* parameters, int thinning)`

Sets the *thinning* parameter. When *thinning* is set to k then every k -th point from the iteration is returned by the sampling algorithm.

Notice: This parameter must satisfy $thinning \geq 1$.

Default: 1.

```
int unur_gibbs_set_burnin (UNUR_PAR* parameters, int burnin)
```

If a "typical" point for the target distribution is not known but merely guessed, the first part of the Gibbs chain should be discarded (*burn-in*). This can be done during the initialization of the generator object. The length of the burn-in can is then *burnin*.

When method GIBBS is not applicable for the target distribution then the initialization already might fail during the burn-in. Thus this reduces the risk of running a generator that returns UNUR_INFINITY caused by some fatal error during sampling.

The thinning factor set by a `unur_gibbs_set_thinning` call has no effect on the length of the burn-in, i.e., for the burn-in always a thinning factor 1 is used.

Notice: This parameter must satisfy *thinning* ≥ 0.

Default: 0.

```
const double* unur_gibbs_get_state (UNUR_GEN* generator)
```

```
int unur_gibbs_chg_state (UNUR_GEN* generator, const double* state)
```

Get and change the current state of the Gibbs chain.

```
int unur_gibbs_reset_state (UNUR_GEN* generator)
```

Reset state of chain to starting point.

Notice: Currently this function does not reset the generators for conditional distributions. Thus it is not possible to get the same Gibbs chain even when the underlying uniform random number generator is reset.

5.6.2 HITRO – Markov Chain - HIT-and-run sampler with Ratio-Of-uniforms

Required: PDF

Optional: mode, center, bounding rectangle for acceptance region

Speed: Set-up: fast, Sampling: fast

HITRO is an implementation of a hit-and-run sampler that runs on the acceptance region of the multivariate ratio-of-uniforms method, see [Section A.4 \[Ratio-of-Uniforms\]](#), page 216.

The Ratio-of-Uniforms transforms the region below the density into some region that we call "region of acceptance" in the following. The minimal bounding hyperrectangle of this region is given by

$$\begin{aligned} v^+ &= \sup_x (f(x))^{1/r \, d+1}, \\ u_i^- &= \inf_{x_i} (x_i - \mu_i) (f(x))^{r/r \, d+1}, \\ u_i^+ &= \sup_{x_i} (x_i - \mu_i) (f(x))^{r/r \, d+1}, \end{aligned}$$

where d denotes the dimension of the distribution; x_i is the i -th coordinate of point x ; μ_i is the i -th coordinate of the center μ of the distribution, i.e., a point in the "main region" of the distribution. Using the center is important, since otherwise the acceptance region can become a very long and skinny ellipsoid along a diagonal of the (huge) bounding rectangle.

For each step of the Hit-and-Run algorithm we have to choose some direction. This direction together with the current point of the chain determines a straight line. Then a point is sampled uniformly on intersection of this line and the region of acceptance. This is done by rejection from a uniform distribution on a line segment that covers it. Depending of the chosen variant the endpoints of this covering line are computed either by means of a (not necessary minimal) bounding hyper-rectangle, or just the "covering plate" of the bounding hyper-rectangle.

The required bounds of the hyper-rectangle can be given directly by the user. Otherwise, these are computed automatically by means of a numerical routine by Hooke and Jeeves [HJa61] called direct search (see '`src/utls/hooke.c`' for further references and details). However, this expensive computation can be avoided by determine these bounds "on the fly" by the following adaptive algorithm: Start with some (small) hyper-rectangle and enlarge it whenever the endpoints of the covering line segment are not contained in the acceptance region of the Ratio-of-Uniforms method. This approach works reliable as long as the region of acceptance is convex.

The performance of the uniform sampling from the line segment is much improved if the covering line is adjusted (shortened) whenever a point is rejected (adaptive sampling). This technique reduces the expected number of iterations enormously.

Method HITRO requires that the region of acceptance of the Ratio-of-Uniforms method is bounded. The shape of this region can be controlled by a parameter r . Higher values of r result in larger classes of distributions with bounded region of acceptance. (A distribution that has such a bounded region for some r also has a bounded region for every r' greater than r .) On the other hand the acceptance probability decreases with increasing r . Moreover, round-off errors are more likely and (for large values of r) might result in a chain with a stationary distribution different from the target distribution.

Method HITRO works optimal for distributions whose region of acceptance is convex. This is in particular the case for all log-concave distributions when we set $r = 1$. For bounded but non-convex regions of acceptance convergence is yet not guarenteed by mathematical theory.

How To Use

Method HITRO requires the PDF of the target distribution (derivatives are not necessary).

The acceptance region of the Ratio-of-Uniforms transformation must be bounded. Its shape is controlled by parameter r . By default this parameter is set to 1 as this guarantees a convex region of acceptance when the PDF of the given distribution is log-concave. It should only be set to a different (higher!) value using `unur_vnrou_set_r` if otherwise $x_i (f(x))^{r/r+1}$ were not bounded for each coordinate.

There are two variants of the HITRO sampler:

coordinate direction sampling. [default]

The coordinates are updated cyclically. This can be seen as a Gibbs sampler running on the acceptance region of the Ratio-of-Uniforms method. This variant can be selected using `unur_hitro_set_variant_coordinate`.

random direction sampling.

In each step a direction is sampled uniformly from the sphere.

This variant can be selected using `unur_hitro_set_variant_random_direction`.

Notice that each iteration of coordinate direction sampling is cheaper than an iteration of random direction sampling.

Sampling uniformly from the line segment can be adjusted in several ways:

Adaptive line sampling vs. simple rejection.

When adaptive line sampling is switched on the covering line is shortened whenever a point is rejected. However, when the region of acceptance is not convex the line segment from which we have to sample might not be connected. We found that the algorithm still works but at the time being there is no formal proof that the generated Markov chain has the required stationary distribution.

Adaptive line sampling can switch on/off by means of the `unur_hitro_set_use_adaptiveline` call.

Bounding hyper-rectangle vs. "covering plate".

For computing the covering line we can use the bounding hyper-rectangle or just its upper bound. The latter saves computing time during the setup and when computing the covering during at each iteration step at the expense of a longer covering line. When adaptive line sampling is used the total generation time for the entire chain is shorter when only the "covering plate" is used.

Notice: When coordinate sampling is used the entire bounding rectangle has is used. Using the entire bounding hyper-rectangle can be switched on/off by means of the `unur_hitro_set_use_boundingrectangle` call.

Deterministic vs. adaptive bounding hyper-rectangle.

A bounding rectangle can be given by the `unur_vnrou_set_u` and `unur_vnrou_set_v` calls. Otherwise, the minimal bounding rectangle is computed automatically during the setup by means of a numerical algorithm. However, this is (very) slow especially in higher dimensions.

Alternatively the bounding rectangle rectangle can be computed adaptively. In the latter case `unur_vnrou_set_u` and `unur_vnrou_set_v` can be used to provide a starting rectangle which must be sufficiently small. Then both endpoints are the covering line segment are always check whether they are outside the acceptance region of the Ratio-of-Uniforms method. If they are not then then line segment and the ("bounding") rectangle are enlarged using a factor that can be given using the `unur_hitro_set_adaptive_multiplier` call (default is the golden ratio). Notice,

that this running the method in the adaptive rectangle mode requires that the region of acceptance is convex when random directions are used, or the given PDF is unimodal when coordinate direction sampling is used. Moreover, it requires two additional calls to the PDF in each iteration step of the chain.

Using addaptive bounding rectangles can be switched on/off by means of the `unur_hitro_set_use_adaptiverectangle` call.

Important: The bounding rectangle has to be provided for the function $PDF(x - center)$! Notice that `center` is the center of the given distribution, see `unur_distr_cvec_set_center`. If in doubt or if this value is not optimal, it can be changed (overridden) by a `unur_distr_cvec_set_center` call.

Function reference

`UNUR_PAR* unur_hitro_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unur_hitro_set_variant_coordinate (UNUR_PAR* parameters)`

Coordinate Direction Sampling: Sampling along the coordinate directions (cyclic).

Notice: For this variant the entire bounding rectangle is always used independent of the `unur_hitro_set_use_boundingrectangle` call.

This is the default.

`int unur_hitro_set_variant_random_direction (UNUR_PAR* parameters)`

Random Direction Sampling: Sampling along the random directions.

`int unur_hitro_set_use_adaptiveline (UNUR_PAR* parameters, int adaptive)`

When *adaptive* is set to `TRUE` adaptive line sampling is applied, otherwise simple rejection is used.

Notice: When adaptive line sampling is switched off, the entire bounding rectangle must be used since otherwise the sampling time can be arbitrarily slow.

Default is `TRUE`.

`int unur_hitro_set_use_boundingrectangle (UNUR_PAR* parameters, int rectangle)`

When *rectangle* is set to `TRUE` the entire bounding rectangle is used for computing the covering line. Otherwise, only an upper bound for the acceptance region is used.

Notice: When coordinate sampling is used the entire bounding rectangle has is always used and this call has no effect.

Default: `FALSE` for random direction samplig, `TRUE` for coordinate direction sampling.

`int unur_hitro_set_use_adaptiverectangle (UNUR_PAR* parameters, int adaptive)`

When *adaptive* is set to `FALSE` the bounding rectangle is determined during the setup. Either, it is computed automatically by a (slow) numerical method, or it must be provided by `unur_vnrou_set_u` and `unur_vnrou_set_v` calls.

If *adaptive* is set to `TRUE` the bounding rectangle is computed adaptively. In this case the `unur_vnrou_set_u` and `unur_vnrou_set_v` calls can be used to provide a starting rectangle. This should be sufficiently small. If not given then we assume $v_{max} = 1$,

$u_{min} = (-0.001, -0.001, \dots, -0.001)$, and $u_{max} = (0.001, 0.001, \dots, 0.001)$. Adaptive enlargements of the bounding hyperrectangle can be controlled set setting an enlargement factor given by a `unur_hitro_set_adaptive_multiplier` call.

Using adaptive computation of the bounding rectangle reduces the setup time significantly (when it is not given by the user) at the expense of two additional PDF evaluations during each iteration step.

Important: Using adaptive bounding rectangles requires that the region of acceptance is convex when random directions are used, or a unimodal PDF when coordinate direction sampling is used.

Default: `FALSE` for random direction samplig, `TRUE` for coordinate direction sampling.

`int unur_hitro_set_r (UNUR_PAR* parameters, double r)`

Sets the parameter r of the generalized multivariate ratio-of-uniforms method.

Notice: This parameter must satisfy $r > 0$.

Default: 1.

`int unur_hitro_set_v (UNUR_PAR* parameters, double vmax)`

Set upper boundary for bounding hyper-rectangle. If not set not set the mode of the distribution is used.

If adaptive bounding rectangles the value is used for the starting rectangle. If not given (and the mode of the distribution is not known) then $vmax=1e-3$ is used.

If deterministic bounding rectangles these values are the given values are used for the rectangle. If no value is given (and the mode of the distribution is not known), the upper bound of the minimal bounding hyper-rectangle is computed numerically (slow).

Default: not set.

`int unur_hitro_set_u (UNUR_PAR* parameters, const double* umin, const double* umax)`

Sets left and right boundaries of bounding hyper-rectangle.

If adaptive bounding rectangles these values are used for the starting rectangle. If not given then $umin=\{-b, -b, \dots, -b\}$ and $umax=\{b, b, \dots, b\}$ with $b=1.e-3$ is used.

If deterministic bounding rectangles these values are the given values are used for the rectangle. If no values are given, the boundary of the minimal bounding hyper-rectangle is computed numerically (slow).

Important: The boundaries are those of the density shifted by the center of the distribution, i.e., for the function $PDF(x - center)$!

Notice: Computing the minimal bounding rectangle may fail under some circumstances. Moreover, for multimodal distributions the bounds might be too small as only local extrema are computed. Nevertheless, for log-concave distributions it should work.

Default: not set.

`int unur_hitro_set_adaptive_multiplier (UNUR_PAR* parameters, double factor)`

Adaptive enlargements of the bounding hyperrectangle can be controlled set setting the enlargement *factor*. This must be greater than 1. Values close to 1 result in small adaptive steps and thus reduce the risk of too large bounding rectangles. On the other hand many adaptive steps might be necessary.

Notice: For practical reasons this call does not accept values for *factor* less than 1.0001. If this value is `UNUR_INFINITY` this results in infinite loops.

Default: 1.1


```
int unur_hitro_set_startingpoint (UNUR_PAR* parameters, const double*
                                x0)
```

Sets the starting point of the HITRO sampler in the original scale. *x0* must be a "typical" point of the given distribution. If such a "typical" point is not known and a starting point is merely guessed, the first part of the HITRO chain should be discarded (*burn-in*), e.g. by mean of the `unur_hitro_set_burnin` call.

Important: The PDF of the distribution must not vanish at the given point *x0*.

Default is the result of `unur_distr_cvec_get_center` for the given distribution object.

```
int unur_hitro_set_thinning (UNUR_PAR* parameters, int thinning)
```

Sets the *thinning* parameter. When *thinning* is set to *k* then every *k*-th point from the iteration is returned by the sampling algorithm. If thinning has to be set such that each coordinate is updated when using coordinate direction sampling, then *thinning* should be `dim+1` (or any multiple of it) where `dim` is the dimension of the distribution object.

Notice: This parameter must satisfy *thinning* ≥ 1.

Default: 1.

```
int unur_hitro_set_burnin (UNUR_PAR* parameters, int burnin)
```

If a "typical" point for the target distribution is not known but merely guessed, the first part of the HITRO chain should be discarded (*burn-in*). This can be done during the initialization of the generator object. The length of the burn-in can is then *burnin*.

The thinning factor set by a `unur_hitro_set_thinning` call has no effect on the length of the burn-in, i.e., for the burn-in always a thinning factor 1 is used.

Notice: This parameter must satisfy *burnin* ≥ 0.

Default: 0.

```
const double* unur_hitro_get_state (UNUR_GEN* generator)
int unur_hitro_chg_state (UNUR_GEN* generator, const double* state)
```

Get and change the current state of the HITRO chain.

Notice: The state variable contains the point in the `dim+1` dimensional point in the (transformed) region of acceptance of the Ratio-of-Uniforms method. Its coordinate are stored in the following order: `state[] = {v, u1, u2, ..., udim}`.

If the state can only be changed if the given *state* is inside this region.

```
int unur_hitro_reset_state (UNUR_GEN* generator)
```

Reset state of chain to starting point.

Notice: Currently this function does not reset the generators for conditional distributions. Thus it is not possible to get the same HITRO chain even when the underlying uniform random number generator is reset.

5.7 Methods for continuous empirical multivariate distributions

Overview of methods

Methods for **continuous empirical multivariate distributions**
sample with `unur_sample_vec`

VEMPK: Requires an observed sample.

Example

```

/* ----- */
/* File: example_vemp.c                               */
/* ----- */

/* Include UNURAN header file.                         */
#include <unuran.h>

/* ----- */

/* Example how to sample from an empirical continuous   */
/* multivariate distribution.                           */
/* ----- */

int main()
{
    int    i;

    /* 4 data points of dimension 2                      */
    double data[] = { 1. ,1.,          /* 1st data point */
                     -1.,1.,          /* 2nd data point */
                      1.,-1.,          /* 3rd data point */
                     -1.,-1. };      /* 4th data point */

    double result[2];

    /* Declare the three UNURAN objects.                 */
    UNUR_DISTR *distr; /* distribution object */
    UNUR_PAR   *par;   /* parameter object */
    UNUR_GEN   *gen;   /* generator object */

    /* Create a distribution object with dimension 2.     */
    distr = unsur_distr_cvemp_new( 2 );

    /* Set empirical sample.                             */
    unsur_distr_cvemp_set_data(distr, data, 4);

    /* Choose a method: VEMPK.                           */
    par = unsur_vempk_new(distr);

    /* Use variance correction.                          */
    unsur_vempk_set_varcor( par, 1 );

    /* Create the generator object.                      */
    gen = unsur_init(par);

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
    }
}

```

```

    exit (EXIT_FAILURE);
}

/* It is possible to reuse the distribution object to create
/* another generator object. If you do not need it any more,
/* it should be destroyed to free memory.
unur_distr_free(distr);

/* Now you can use the generator object 'gen' to sample from
/* the distribution. Eg.:
for (i=0; i<10; i++) {
    unur_sample_vec(gen, result);
    printf("(f,f)\n", result[0], result[1]);
}

/* When you do not need the generator object any more, you
/* can destroy it.
unur_free(gen);

exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */

```

Example (String API)

(not implemented)

5.7.1 VEMPK – (Vector) EMPirical distribution with Kernel smoothing

Required: observed sample

Speed: Set-up: slow, Sampling: slow (depends on dimension)

Reference: [HLa00] [HLD04: Sect.12.2.1]

VEMPK generates random variates from a multivariate empirical distribution that is given by an observed sample. The idea is that simply choosing a random point from the sample and to return it with some added noise results in a method that has very nice properties, as it can be seen as sampling from a kernel density estimate. Clearly we have to decide about the density of the noise (called kernel) and about the covariance matrix of the noise. The mathematical theory of kernel density estimation shows us that we are comparatively free in choosing the kernel. It also supplies us with a simple formula to compute the optimal standard deviation of the noise, called bandwidth (or window width) of the kernel.

Currently only a Gaussian kernel with the same covariance matrix as the given sample is implemented. However it is possible to choose between a variance corrected version or those with optimal MISE. Additionally a smoothing factor can be set to adjust the estimated density to non-bell-shaped data densities.

How To Use

VEMPK uses empirical distributions. The main parameter would be the choice of kernel density. However, currently only Gaussian kernels are supported. The parameters for the density are computed by a simple but robust method. However, it is possible to control its behavior by changing the smoothing factor. Additionally, variance correction can be switched on (at the price of suboptimal MISE).

Function reference

`UNUR_PAR* unur_vempk_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unur_vempk_set_smoothing (UNUR_PAR* parameters, double smoothing)`

`int unur_vempk_chg_smoothing (UNUR_GEN* generator, double smoothing)`

Set and change the smoothing factor. The smoothing factor controls how “smooth” the resulting density estimation will be. A smoothing factor equal to 0 results in naive resampling. A very large smoothing factor (together with the variance correction) results in a density which is approximately equal to the kernel. Default is 1 which results in a smoothing parameter minimising the MISE (mean integrated squared error) if the data are not too far away from normal. If a large smoothing factor is used, then variance correction must be switched on.

Default: 1

`int unur_vempk_set_varcor (UNUR_PAR* parameters, int varcor)`

`int unur_vempk_chg_varcor (UNUR_GEN* generator, int varcor)`

Switch variance correction in generator on/off. If `varcor` is `TRUE` then the variance of the used density estimation is the same as the sample variance. However this increases the MISE of the estimation a little bit.

Default is `FALSE`.

5.8 Methods for discrete univariate distributions

Overview of methods

Methods for **discrete univariate distributions**

sample with `unur_sample_discr`

method	PMF	PV	mode	sum	other
DARI	x		x	~	T-concave
DAU	[x]	x			
DGT	[x]	x			
DSTD					build-in standard distribution
DSS	[x]	x		x	

Example

```

/* ----- */
/* File: example_discr.c */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* Example how to sample from a discrete univariate distribution.*/
/* ----- */

int main()
{
    int i;
    double param = 0.3;

    double probvec[10] = {1.0, 2.0, 3.0, 4.0, 5.0,\
                          6.0, 7.0, 8.0, 4.0, 3.0};

    /* Declare the three UNURAN objects. */
    UNUR_DISTR *distr1, *distr2; /* distribution objects */
    UNUR_PAR *par1, *par2; /* parameter objects */
    UNUR_GEN *gen1, *gen2; /* generator objects */

    /* First distribution: defined by PMF. */
    distr1 = unsur_distr_geometric(&param, 1);
    unsur_distr_discr_set_mode(distr1, 0);

    /* Choose a method: DARI. */
    par1 = unsur_dari_new(distr1);
    gen1 = unsur_init(par1);

    /* It is important to check if the creation of the generator
    /* object was successful. Otherwise 'gen' is the NULL pointer
    /* and would cause a segmentation fault if used for sampling.
    if (gen1 == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Second distribution: defined by (finite) PV.
    distr2 = unsur_distr_discr_new();
    unsur_distr_discr_set_pv(distr2, probvec, 10);

```

```

/* Choose a method: DGT. */
par2 = unur_dgt_new(distr2);
gen2 = unur_init(par2);
if (gen2 == NULL) {
    fprintf(stderr, "ERROR: cannot create generator object\n");
    exit (EXIT_FAILURE);
}

/* print some random integers */
for (i=0; i<10; i++){
    printf("number %d: %d\n", i*2, unur_sample_discr(gen1) );
    printf("number %d: %d\n", i*2+1, unur_sample_discr(gen2) );
}

/* Destroy all objects. */
unur_distr_free(distr1);
unur_distr_free(distr2);
unur_free(gen1);
unur_free(gen2);

exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */

```

Example (String API)

```

/* ----- */
/* File: example_discr_str.c */
/* ----- */
/* String API. */
/* ----- */

/* Include UNURAN header file. */
#include <unuran.h>

/* ----- */

/* Example how to sample from a discrete univariate distribution.*/
/* ----- */

int main()
{
    int    i;      /* loop variable */

    /* Declare UNURAN generator objects. */
    UNUR_GEN *gen1, *gen2;      /* generator objects */

    /* First distribution: defined by PMF. */
    gen1 = unur_str2gen("geometric(0.3); mode=0 & method=dari");

    /* It is important to check if the creation of the generator */
    /* object was successful. Otherwise 'gen' is the NULL pointer */
    /* and would cause a segmentation fault if used for sampling. */
    if (gen1 == NULL) {
        fprintf(stderr, "ERROR: cannot create generator object\n");
        exit (EXIT_FAILURE);
    }

    /* Second distribution: defined by (finite) PV. */
    gen2 = unur_str2gen("distr=discr; pv=(1,2,3,4,5,6,7,8,4,3) & method=dgt");
}

```

```
if (gen2 == NULL) {
    fprintf(stderr, "ERROR: cannot create generator object\n");
    exit (EXIT_FAILURE);
}

/* print some random integers */
for (i=0; i<10; i++){
    printf("number %d: %d\n", i*2,    unur_sample_discr(gen1) );
    printf("number %d: %d\n", i*2+1, unur_sample_discr(gen2) );
}

/* Destroy all objects. */
unur_free(gen1);
unur_free(gen2);

exit (EXIT_SUCCESS);

} /* end of main() */

/* ----- */
```

5.8.1 DARI – discrete automatic rejection inversion

Required: T-concave PMF, mode, approximate area

Speed: Set-up: moderate, Sampling: fast

Reference: [HDa96] [HLD04: Sect.10.2; Alg.10.4]

DARI is based on rejection inversion, which can be seen as an adaptation of transformed density rejection to discrete distributions. The used transformation is $-1/\sqrt{x}$.

DARI uses three almost optimal points for constructing the (continuous) hat. Rejection is then done in horizontal direction. Rejection inversion uses only one uniform random variate per trial.

DARI has moderate set-up times (the PMF is evaluated nine times), and good marginal speed, especially if an auxiliary array is used to store values during generation.

DARI works for all $T_{-1/2}$ -concave distributions. It requires the PMF and the location of the mode. Moreover the approximate sum over the PMF is used. (If no sum is given for the distribution the algorithm assumes that it is approximately 1.) The rejection constant is bounded from above by 4 for all T -concave distributions.

How To Use

DARI works for discrete distribution object with given PMF. The sum over probabilities should be approximately one. Otherwise it must be set by a `unur_distr_discr_set_pmfsum` call to its (approximate) value.

The size of an auxiliary table can be set by `unur_dari_set_tablesize`. The expected number of evaluations can be reduced by switching the use of squeezes by means of `unur_dari_set_squeeze`. It is possible to change the parameters and the domain of the chosen distribution without building a new generator object by using the `unur_dari_chg_pmfparams` and `unur_dari_chg_domain` call, respectively. But then `unur_dari_chg_mode` and `unur_dari_chg_pmfsum` have to be used to reset the corresponding figures whenever they were changed. Before sampling from the distribution again, `unur_dari_reinit` must be executed. (Otherwise the generator might produce garbage).

There exists a test mode that verifies whether the conditions for the method are satisfied or not. It can be switched on by calling `unur_dari_set_verify` and `unur_dari_chg_verify`, respectively. Notice however that sampling is (much) slower then.

Function reference

`UNUR_PAR* unsur_dari_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unsur_dari_reinit (UNUR_GEN* generator)`

Update an existing generator object after the distribution has been modified. It must be executed whenever the parameters or the domain of the distributions has been changed (see below). It is faster than destroying the existing object and building a new one from scratch. If reinitialization has been successful `UNUR_SUCCESS` is returned, in case of a failure an error code is returned.

`int unsur_dari_set_squeeze (UNUR_PAR* parameters, int squeeze)`

Turn utilization of the squeeze of the algorithm on/off. This squeeze does not resample the squeeze of the continuous TDR method. It was especially designed for rejection inversion.

The squeeze is not necessary if the size of the auxiliary table is big enough (for the given distribution). Using a squeeze is suggested to speed up the algorithm if the domain of the distribution is very big or if only small samples are produced.

Default: no squeeze.

```
int unur_dari_set_tablesize (UNUR_PAR* parameters, int size)
```

Set the size for the auxiliary table, that stores constants computed during generation. If *size* is set to 0 no table is used. The speed-up can be impressive if the PMF is expensive to evaluate and the “main part of the distribution” is concentrated in an interval shorter than the size of the table.

Default is 100.

```
int unur_dari_set_cpfactor (UNUR_PAR* parameters, double cp_factor)
```

Set factor for position of the left and right construction point, resp. The *cp_factor* is used to find almost optimal construction points for the hat function. The *cp_factor* must be positive and should not exceed 2. There is no need to change this factor in almost all situations.

Default is 0.664.

```
int unur_dari_set_verify (UNUR_PAR* parameters, int verify)
```

```
int unur_dari_chg_verify (UNUR_GEN* generator, int verify)
```

Turn verifying of algorithm while sampling on/off. If the condition is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

```
int unur_dari_chg_pmfparams (UNUR_GEN* generator, double* params, int  
                             n_params)
```

Change array of parameters of the distribution in a given generator object. Notice that this call simply copies the parameters into the generator object. Thus if fewer parameters are provided then the remaining parameters are left unchanged. `unur_dari_reinit` must be executed before sampling from the generator again.

Important: The given parameters are not checked against domain errors; in opposition to the `unur_<distr>_new` calls.

```
int unur_dari_chg_domain (UNUR_GEN* generator, int left, int right)
```

Change the left and right border of the domain of the (truncated) distribution. If the mode changes when the domain of the (truncated) distribution is changed, then a corresponding `unur_dari_chg_mode` call is required. (There is no domain checking as in the `unur_init` call.) Use `INT_MIN` and `INT_MAX` for (minus) infinity. `unur_dari_reinit` must be executed before sampling from the generator again.

```
int unur_dari_chg_mode (UNUR_GEN* generator, int mode)
```

Change mode of distribution. `unur_dari_reinit` must be executed before sampling from the generator again.

```
int unur_dari_upd_mode (UNUR_GEN* generator)
```

Recompute the mode of the distribution. This call only works well when a distribution object from the UNURAN library of standard distributions is used (see [Chapter 7 \[Standard distributions\]](#), page 187). Otherwise a (slow) numerical mode finder is called. If no mode can be found, then an error code is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_dari_reinit` must be executed before sampling from the generator again.

```
int unur_dari_chg_pmfsum (UNUR_GEN* generator, double sum)
```

Change sum over the PMF of distribution. `unur_dari_reinit` must be executed before sampling from the generator again.

```
int unur_dari_upd_pmfsum (UNUR_GEN* generator)
```

Recompute sum over the PMF of the distribution. It only works when a distribution objects from the UNURAN library of standard distributions is used (see [Chapter 7 \[Standard distributions\]](#), [page 187](#)). Otherwise an error code is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_dari_reinit` must be executed before sampling from the generator again.

5.8.2 DAU – (Discrete) Alias-Urn method

Required: probability vector (PV)

Speed: Set-up: slow (linear with the vector-length), Sampling: very fast

Reference: [WAa77] [HLD04: Sect.3.2]

DAU samples from distributions with arbitrary but finite probability vectors (PV) of length N . The algorithm is based on an ingenious method by A.J. Walker and requires a table of size (at least) N . It needs one random numbers and only one comparison for each generated random variate. The setup time for constructing the tables is $O(N)$.

By default the probability vector is indexed starting at 0. However this can be changed in the distribution object by a `unur_distr_discr_set_domain` call.

The method also works when no probability vector but a PMF is given. However then additionally a bounded (not too large) domain must be given or the sum over the PMF (see `unur_distr_discr_make_pv` for details).

How To Use

Create an object for a discrete distribution either by setting a probability vector or a PMF. The performance can be slightly influenced by setting the size of the used table which can be changed by `unur_dau_set_urnfactor`.

Function reference

`UNUR_PAR*` `unur_dau_new` (*const UNUR_DISTR** *distribution*)

Get default parameters for generator.

`int` `unur_dau_set_urnfactor` (*UNUR_PAR** *parameters*, *double factor*)

Set size of urn table relative to length of the probability vector. It must not be less than 1. Larger tables result in (slightly) faster generation times but require a more expensive setup. However sizes larger than 2 are not recommended.

Default is 1.

5.8.3 DGT – (Discrete) Guide Table method (indexed search)

Required: probability vector (PV)

Speed: Set-up: slow (linear with the vector-length), Sampling: very fast

Reference: [CAa74] [HLD04: Sect.3.1.2]

DGT samples from arbitrary but finite probability vectors. Random numbers are generated by the inversion method, i.e.,

1. Generate a random number $U \sim U(0,1)$.
2. Find largest integer I such that $F(I) = P(X \leq I) \leq U$.

Step (2) is the crucial step. Using sequential search requires $O(E(X))$ comparisons, where $E(X)$ is the expectation of the distribution. Indexed search, however, uses a guide table to jump to some $I' \leq I$ near I to find X in constant time. Indeed the expected number of comparisons is reduced to 2, when the guide table has the same size as the probability vector (this is the default). For larger guide tables this number becomes smaller (but is always larger than 1), for smaller tables it becomes larger. For the limit case of table size 1 the algorithm simply does sequential search (but uses a more expensive setup then method DSS (see [Section 5.8.5 \[DSS\], page 170](#)). On the other hand the setup time for guide table is $O(N)$, where N denotes the length of the probability vector (for size 1 no preprocessing is required). Moreover, for very large guide tables memory effects might even reduce the speed of the algorithm. So we do not recommend to use guide tables that are more than three times larger than the given probability vector. If only a few random numbers have to be generated, (much) smaller table sizes are better. The size of the guide table relative to the length of the given probability vector can be set by a `unur_dgt_set_guidefactor` call.

There exist two variants for the setup step which can be set by a `unur_dgt_set_variant` call: Variants 1 and 2. Variant 2 is faster but more sensitive to roundoff errors when the guide table is large. By default variant 2 is used for short probability vectors ($N < 1000$) and variant 1 otherwise.

By default the probability vector is indexed starting at 0. However this can be changed in the distribution object by a `unur_distr_discr_set_domain` call.

The method also works when no probability vector but a PMF is given. However, then additionally a bounded (not too large) domain must be given or the sum over the PMF. In the latter case the domain of the distribution is truncated (see `unur_distr_discr_make_pv` for details).

How To Use

Create an object for a discrete distribution either by setting a probability vector or a PMF. The performance can be slightly influenced by setting the size of the used table which can be changed by `unur_dgt_set_guidefactor`.

Function reference

UNUR_PAR* `unur_dgt_new` (*const UNUR_DISTR** *distribution*)

Get default parameters for generator.

int `unur_dgt_set_guidefactor` (*UNUR_PAR** *parameters*, *double factor*)

Set size of guide table relative to length of PV. Larger guide tables result in faster generation time but require a more expensive setup. Sizes larger than 3 are not recommended. If the relative size is set to 0, sequential search is used. However, this is not recommended, except

in exceptional cases, since method DSS (see [Section 5.8.5 \[DSS\], page 170](#)) is has almost no setup and is thus faster (but requires the sum over the PV as input parameter).

Default is 1.

int `unur_dgt_set_variant` (*UNUR_PAR** *parameters*, *unsigned variant*)

Set variant for setup step. Possible values are 1 or 2. Variant 2 is faster but more sensitive to roundoff errors when the guide table is large. By default variant 2 is used for short probability vectors ($N < 1000$) and variant 1 otherwise.

5.8.4 DSROU – Discrete Simple Ratio-Of-Uniforms method

Required: T-concave PMF, mode, sum over PMF

Speed: Set-up: fast, Sampling: slow

Reference: [LJa01] [HLD04: Sect.10.3.2; Alg.10.6]

DSROU is based on the ratio-of-uniforms method (see [Section A.4 \[Ratio-of-Uniforms\]](#), [page 216](#)) but uses universal inequalities for constructing a (universal) bounding rectangle. It works for all T -concave distributions with $T(x) = -1/\sqrt{x}$.

The method requires the PMF, the (exact) location of the mode and the sum over the given PDF. The rejection constant is 4 for all T -concave distributions. Optionally the CDF at the mode can be given to increase the performance of the algorithm. Then the rejection constant is reduced to 2.

How To Use

The method works for T -concave discrete distributions with given PMF. The sum over of the PMF or an upper bound of this sum must be known.

Optionally the CDF at the mode can be given to increase the performance using `unur_dsrou_set_cdfatmode`. However, this **must not** be called if the sum over the PMF is replaced by an upper bound.

It is possible to change the parameters and the domain of the chosen distribution without building a new generator object using the `unur_dsrou_chg_pmparams` and `unur_dsrou_chg_domain` call, respectively. But then `unur_dsrou_chg_pmsum`, `unur_dsrou_chg_mode` and `unur_dsrou_chg_cdfatmode` have to be used to reset the corresponding figures whenever they have changed.

If any of mode, CDF at mode, or the sum over the PMF has been changed, then `unur_dsrou_reinit` must be executed. (Otherwise the generator produces garbage).

There exists a test mode that verifies whether the conditions for the method are satisfied or not while sampling. It can be switched on or off by calling `unur_dsrou_set_verify` and `unur_dsrou_chg_verify`, respectively. Notice however that sampling is (a little bit) slower then.

Function reference

`UNUR_PAR* unsur_dsrou_new (const UNUR_DIST* distribution)`

Get default parameters for generator.

`int unsur_dsrou_reinit (UNUR_GEN* generator)`

Update an existing generator object after the distribution has been modified. It must be executed whenever the parameters or the domain of the distribution have been changed (see below). It is faster than destroying the existing object and building a new one from scratch. If reinitialization has been successful 1 is returned, in case of a failure an error code is returned.

`int unsur_dsrou_set_cdfatmode (UNUR_PAR* parameters, double Fmode)`

Set CDF at mode. When set, the performance of the algorithm is increased by factor 2. However, when the parameters of the distribution are changed `unur_dsrou_chg_cdfatmode` has to be used to update this value. Notice that the algorithm detects a mode at the left boundary of the domain automatically and it is not necessary to use this call for a monotonically decreasing PMF.

Default: not set.

```
int unur_dsrou_set_verify (UNUR_PAR* parameters, int verify)
```

```
int unur_dsrou_chg_verify (UNUR_GEN* generator, int verify)
```

Turn verifying of algorithm while sampling on/off. If the condition $\text{squeeze}(x) \leq \text{PMF}(x) \leq \text{hat}(x)$ is violated for some x then `unur_errno` is set to `UNUR_ERR_GEN_CONDITION`. However notice that this might happen due to round-off errors for a few values of x (less than 1%).

Default is FALSE.

```
int unur_dsrou_chg_pmparams (UNUR_GEN* generator, double* params, int  
    n_params)
```

Change array of parameters of the distribution in a given generator object.

For standard distributions from the UNURAN library the parameters are checked. If these are invalid, then an error code is returned. Moreover the domain is updated automatically unless it has been changed before by a `unur_distr_discr_set_domain` call. Notice that optional parameters are (re-)set to their default values if not given for UNURAN standard distributions.

For other distributions *params* is simply copied into to distribution object. It is only checked that *n_params* does not exceed the maximum number of parameters allowed. Then an error code is returned and `unur_errno` is set to `UNUR_ERR_DISTR_NPARAMS`.

```
int unur_dsrou_chg_domain (UNUR_GEN* generator, int left, int right)
```

Change left and right border of the domain of the (truncated) distribution. If the mode changes when the domain of the (truncated) distribution is changed, then a corresponding `unur_dsrou_chg_mode` is required. (There is no checking whether the domain is set or not as in the `unur_init` call.)

```
int unur_dsrou_chg_mode (UNUR_GEN* generator, int mode)
```

Change mode of distribution. `unur_dsrou_reinit` must be executed before sampling from the generator again.

```
int unur_dsrou_upd_mode (UNUR_GEN* generator)
```

Recompute the mode of the distribution. See `unur_distr_cont_upd_mode` for more details. `unur_dsrou_reinit` must be executed before sampling from the generator again.

```
int unur_dsrou_chg_cdfatmode (UNUR_GEN* generator, double Fmode)
```

Change CDF at mode of distribution. `unur_dsrou_reinit` must be executed before sampling from the generator again.

```
int unur_dsrou_chg_pmfsun (UNUR_GEN* generator, double sum)
```

Change sum over PMF of distribution. `unur_dsrou_reinit` must be executed before sampling from the generator again.

```
int unur_dsrou_upd_pmfsun (UNUR_GEN* generator)
```

Recompute the sum over the the PMF of the distribution. It only works when a distribution objects from the UNURAN library of standard distributions is used (see [Chapter 7 \[Standard distributions\]](#), [page 187](#)). Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_dsrou_reinit` must be executed before sampling from the generator again.

5.8.5 DSS – (Discrete) Sequential Search method

Required: probability vector (PV) and sum over PV; or probability mass function (PMF), sum over PV and domain; or or cumulative distribution function (CDF)

Speed: Set-up: fast, Sampling: very slow (linear in expectation)

Reference: [HLD04: Sect.3.1.1; Alg.3.1]

DSS samples from arbitrary discrete distributions. Random numbers are generated by the inversion method, i.e.,

1. Generate a random number $U \sim U(0,1)$.
2. Find largest integer I such that $F(I) = P(X \leq I) \leq U$.

Step (2) is the crucial step. Using sequential search requires $O(E(X))$ comparisons, where $E(X)$ is the expectation of the distribution. Thus this method is only recommended when only a few random variates from the given distribution are required. Otherwise, table methods like DGT (see [Section 5.8.3 \[DGT\], page 166](#)) or DAU (see [Section 5.8.2 \[DAU\], page 165](#)) are much faster. These methods also need not the sum over the PMF (or PV) as input. On the other hand, however, these methods always compute a table.

DSS runs with the PV, the PMF, or the CDF of the distribution. It uses actually uses the first one in this list (in this ordering) that could be found.

How To Use

It works with a discrete distribution object with contains at least the PV, the PMF, or the CDF.

Function reference

`UNUR_PAR* unur_dss_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

5.8.6 DSTD – Discrete STandarD distributions

Required: standard distribution from UNURAN library (see [Chapter 7 \[Standard distributions\]](#), page 187).

Speed: Set-up: fast, Sampling: depends on distribution and generator

DSTD is a wrapper for special generators for discrete univariate standard distributions. It only works for distributions in the UNURAN library of standard distributions (see [Chapter 7 \[Standard distributions\]](#), page 187). If a distribution object is provided that is build from scratch, or no special generator for the given standard distribution is provided, the NULL pointer is returned.

For some distributions more than one special generator is possible.

How To Use

Create a distribution object for a standard distribution from the UNURAN library (see [Chapter 7 \[Standard distributions\]](#), page 187). For some distributions more than one special generator (*variants*) is possible. These can be chosen by a `unur_dstd_set_variant` call. For possible variants See [Chapter 7 \[Standard distributions\]](#), page 187. However the following are common to all distributions:

`UNUR_STDGEN_DEFAULT`

the default generator.

`UNUR_STDGEN_FAST`

the fastest available special generator.

`UNUR_STDGEN_INVERSION`

the inversion method (if available).

Notice that the variant `UNUR_STDGEN_FAST` for a special generator might be slower than one of the universal algorithms! Additional variants may exist for particular distributions.

Sampling from truncated distributions (which can be constructed by changing the default domain of a distribution by means of `unur_distr_discr_set_domain` call) is possible but requires the inversion method.

It is possible to change the parameters and the domain of the chosen distribution without building a new generator object by means of `unur_dstd_chg_pmfparams`.

Function reference

`UNUR_PAR* unur_dstd_new (const UNUR_DISTR* distribution)`

Get default parameters for new generator. It requires a distribution object for a discrete univariate distribution from the UNURAN library of standard distributions (see [Chapter 7 \[Standard distributions\]](#), page 187).

Using a truncated distribution is allowed only if the inversion method is available and selected by the `unur_dstd_set_variant` call immediately after creating the parameter object. Use a `unur_distr_discr_set_domain` call to get a truncated distribution.

`int unur_dstd_set_variant (UNUR_PAR* parameters, unsigned variant)`

Set variant (special generator) for sampling from a given distribution. For possible variants see [Chapter 7 \[Standard distributions\]](#), page 187.

Common variants are `UNUR_STDGEN_DEFAULT` for the default generator, `UNUR_STDGEN_FAST` for (one of the) fastest implemented special generators, and `UNUR_STDGEN_INVERSION` for the inversion method (if available). If the selected variant number is not implemented, this call has no effect.

```
int unur_dstd_chg_pmfparams (UNUR_GEN* gen, double* params, int n_params)
```

Change array of parameters of the distribution in a given generator object. If the given parameters are invalid for the distribution, no parameters are set. Notice that optional parameters are (re-)set to their default values if not given for UNURAN standard distributions.

Important: Integer parameter must be given as doubles.

5.9 Methods for random matrices

5.9.1 MCORR – Random CORRelation matrix

Required: Distribution object for random correlation matrix

Speed: Set-up: fast, Sampling: depends on dimension

Reference: [DLa86: Sect.6.1; p.605] [MOa84]

MCORR generates a random correlation matrix (Pearson's correlation). Two methods are used:

1. When a random correlation matrix having given eigenvalues is sought, the method of Marsaglia and Olkin [MOa84] is used. In this case, the correlation matrix R is given as $R = PDP'$ where D is a diagonal matrix containing the eigenvalues and P is a random orthonormal matrix. In higher dimensions, the rounding-errors introduced in the previous matrix multiplications could lead to a non-symmetric correlation matrix. Therefore the symmetric correlation matrix is computed as $R = (PDP' + P'DP)/2$.
2. A matrix H is generated where all rows are independent random vectors of unit length uniformly on a sphere. Then HH' is a correlation matrix (and vice versa if HH' is a correlation matrix then the rows of H are random vectors on a sphere).

Notice that due to round-off errors the generated matrices might not be positive definite in extremely rare cases (especially when the given eigenvalues are almost 0).

There are many other possibilities (distributions) of sampling the random rows from a sphere. The chosen methods are simple but does not result in a uniform distribution of the random correlation matrices.

It only works with distribution objects of random correlation matrices (see [Section 7.4.1 \[Random Correlation Matrix\]](#), page 199).

How To Use

Create a distribution object for random correlation matrices by a `unur_distr_correlation` call (see [Section 7.4.1 \[Random Correlation Matrix\]](#), page 199).

When a correlation matrix with given eigenvalues should be generated, these eigenvalues can be set by a `unur_mcorr_set_eigenvalues` call.

Otherwise, a faster algorithm is used that generates correlation matrices with random eigenstructure.

Notice that due to round-off errors, there is a (small) chance that the resulting matrix is not positive definite for a Cholesky decomposition algorithm, especially when the dimension of the distribution is high.

Function reference

`UNUR_PAR* unur_mcorr_new (const UNUR_DISTR* distribution)`

Get default parameters for generator.

`int unur_mcorr_set_eigenvalues (UNUR_PAR* par, const double* eigenvalues)`

Sets the (optional) eigenvalues of the correlation matrix. If set, then the Marsaglia and Olkin algorithm will be used to generate random correlation matrices with given eigenvalues.

Important: the given eigenvalues of the correlation matrix must be strictly positive and sum to the dimension of the matrix. If non-positive eigenvalues are attempted, no eigenvalues are set and an error code is returned. In case, that their sum is different from the dimension, an implicit scaling to give the correct sum is performed.

5.10 Methods for uniform univariate distributions

5.10.1 UNIF – wrapper for UNIFORM random number generator

UNIF is a simple wrapper that makes it possible to use a uniform random number generator as a UNURAN generator. There are no parameters for this method.

How To Use

Create a generator object with `NULL` as argument. The created generator object returns raw random numbers from the underlying uniform random number generator.

Function reference

`UNUR_PAR* unur_unif_new (const UNUR_DISTR* dummy)`

Get default parameters for generator. UNIF does not need a distribution object. *dummy* is not used and can (should) be set to `NULL`. It is used to keep the API consistent.

6 Using uniform random number generators

UNURAN is designed to work with many sources of (pseudo-) random numbers or low discrepancy numbers (so called quasi-random numbers) for almost all tasks in discrete event simulation, (quasi-) Monte Carlo integration or any other stochastic methods. Hence UNURAN uses pointers to access uniform (pseudo-) random number generators (URNG).

Each UNURAN (non-uniform random variate) generator object has a pointer to a URNG object. Thus each UNURAN generator object may have its own (independent) URNG or several generator objects can share the same URNG.

If no URNG is provided for a parameter or generator object a default generator is used which is the same for all generators. This URNG is defined in ‘`unuran_config.h`’ at compile time and can be changed at runtime.

In earlier versions of UNURAN the type of the source of the random numbers (i.e., the library which is used for generating streams of uniform random numbers) has to be set at compile time which restricts the flexibility of this approach. With the current UNURAN version a unified interface is used for all sources of random numbers. Unfortunately, the API for random number generators, like the ‘GSL’ (GNU Scientific Library), Otmar Lendl’s ‘`prng`’ (Pseudo random number generators), or a single function implemented by the user herself, are quite different. Hence an object of type `UNUR_URNG` is introduced to store the URNG. Now it is possible to use different types of uniform (pseudo- or quasi-) random numbers in one simulation study. Moreover, it is possible to handle different sources of such URNGs with a unified API. This programming interface is inspired from and similar to Pierre L’Ecuyers ‘`RngStreams`’ library:

- seed the random number generator;
- get a uniform random number;
- reset the URNG;
- skip to the beginning next substream;
- sample antithetic numbers;
- delete the URNG object.

The routine to create a URNG depends on the chosen random number generator (i.e. library). Nevertheless, there exist wrapper functions to simplify this task.

Currently the following sources of uniform random numbers are directly supported (i.e., there exist wrapper functions). Of course other random number generation libraries can be used.

1. `UNUR_URNG_FVOID`

URNGs of type `double uniform(void)`. If independent versions of the same URNG should be used, a copy of the subroutine has to be implemented in the program code (with different names, of course). UNURAN contains some build-in URNGs of this type in directory ‘`src/uniform/`’.

2. `UNUR_URNG_PRNG`

URNGs from the `prng` library. It provides a very flexible way to sample from arbitrary URNGs by means of an object oriented programming paradigm. Similarly to the UNURAN library independent generator objects can be build and used.

This library has been developed by the pLab group at the university of Salzburg (Austria, EU) and implemented by Otmar Lendl. It is available via anonymous ftp from <http://statistik.wu-wien.ac.at/prng/> or from the pLab site at <http://random.mat.sbg.ac.at/>.

3. `UNUR_URNG_RNGSTREAM`

Pierre L’Ecuyer’s `RngStream` library for multiple independent streams of pseudo-random numbers. A GNU-style package is available from <http://statistik.wu-wien.ac.at/software/RngStreams/>.

4. UNUR_URNG_GSL

URNG from the GNU Scientific Library (GSL). It is available from <http://www.gnu.org/software/gsl/>.

How To Use

Each UNURAN generator object has a pointer to a uniform (pseudo-) random number generator (URNG). It can be set via the `unur_set_urng` call. It is also possible to read this pointer via `unur_get_urng` or change the URNG for an existing generator object by means of `unur_chg_urng`. It is important to note that these calls only copy the pointer to the URNG object into the generator object.

If no URNG is provided for a parameter or generator object a default URNG is used which is the same for all generators. This URNG is defined in ‘`unuran_config.h`’ at compile time. A pointer to this default URNG can be obtained via `unur_get_default_urng`. Nevertheless, it is also possible to change this default URNG by another one at runtime by means of the `unur_set_default_urng` call. However, this only takes effect for new parameter objects.

Some generating methods provide the possibility of correlation induction. For this feature a second auxiliary URNG is required. It can be set and changed by `unur_set_urng_aux` and `unur_chg_urng_aux` calls, respectively. Since the auxiliary URNG is by default the same as the main URNG, the auxiliary URNG must be set after any `unur_set_urng` or `unur_chg_urng` call! Since in special cases mixing of two URNG might cause problems, we supply a default auxiliary generator that can be used by a `unur_use_urng_aux_default` call (after the main URNG has been set). This default auxiliary generator can be changed with analogous calls as the (main) default uniform generator.

Uniform random number generators from different sources have different programming interfaces. Thus UNURAN stores all information about a particular uniform random number generator in a structure of type `UNUR_URNG`. Before a URNG can be used with UNURAN an appropriate object has to be created by a `unur_urng_new` call. This call takes two arguments: the pointer to the sampling routine of the generator and a pointer to a possible argument that stores the state of the generator. The function must be of type `double (*sampleunif)(void *params)`, but functions without any argument also work. Additionally one can set pointers to functions for resetting or jumping the streams generated by the URNG by the corresponding `set` calls.

UNURAN provides a unified API to all sources of random numbers. Notice, however, that not all functions work for all random number generators (as the respective library has not implemented the corresponding feature).

There are wrapper functions for some libraries of uniform random number generators to simplify the task of creating a UNURAN object for URNGs.

Function reference

Set and get default uniform RNGs

`UNUR_URNG* unir_get_default_urng (void)`

Get the pointer to the default URNG. The default URNG is used by all generators where no URNG was set explicitly by a `unur_set_urng` call.

`UNUR_URNG* unir_set_default_urng (UNUR_URNG* urng_new)`

Change the default URNG that is used for new parameter objects. It returns the pointer to the old default URNG that has been used.


```
UNUR_URNG* unur_set_default_urng_aux (UNUR_URNG* urng_new)
UNUR_URNG* unur_get_default_urng_aux (void)
```

Analogous calls for default auxiliary generator.

Set, change and get uniform RNGs in generator objects

```
int unur_set_urng (UNUR_PAR* parameters, UNUR_URNG* urng)
```

Use the URNG `urng` for the new generator. This overrides the default URNG. It also sets the auxiliary URNG to `urng`.

Important: For multivariate distributions that use marginal distributions this call does not work properly. It is then better first to create the generator object (by a `unur_init` call) and then change the URNG by means of `unur_chg_urng`.

```
UNUR_URNG* unur_chg_urng (UNUR_GEN* generator, UNUR_URNG* urng)
```

Change the URNG for the given generator. It returns the pointer to the old URNG that has been used by the generator. It also changes the auxiliary URNG to `urng` and thus it overrides the last `unur_chg_urng_aux` call.

```
UNUR_URNG* unur_get_urng (UNUR_GEN* generator)
```

Get the pointer to the URNG that is used by the *generator*. This is usefull if two generators should share the same URNG.

```
int unur_set_urng_aux (UNUR_PAR* parameters, UNUR_URNG* urng_aux)
```

Use the auxiliary URNG `urng_aux` for the new generator. (Default is the default URNG or the URNG from the last `unur_set_urng` call. Thus if the auxiliary generator should be different to the main URNG, `unur_set_urng_aux` must be called after `unur_set_urng`. The auxiliary URNG is used as second stream of uniform random number for correlation induction. It is not possible to set an auxiliary URNG for a method that does not need one. In this case an error code is returned.

```
int unur_use_urng_aux_default (UNUR_PAR* parameters)
```

Use the default auxiliary URNG. (It must be set after `unur_get_urng`.) It is not possible to set an auxiliary URNG for a method that does not use one (i.e. the call returns an error code).

```
int unur_chgto_urng_aux_default (UNUR_GEN* generator)
```

Switch to default auxiliary URNG. (It must be set after `unur_get_urng`.) It is not possible to set an auxiliary URNG for a method that does not use one (i.e. the call returns an error code).

```
UNUR_URNG* unur_chg_urng_aux (UNUR_GEN* generator, UNUR_URNG*
    urng_aux)
```

Change the auxiliary URNG for the given *generator*. It returns the pointer to the old auxiliary URNG that has been used by the generator. It has to be called after each `unur_chg_urng` when the auxiliary URNG should be different from the main URNG. It is not possible to change the auxiliary URNG for a method that does not use one (i.e. the call NULL).

```
UNUR_URNG* unur_get_urng_aux (UNUR_GEN* generator)
```

Get the pointer to the auxiliary URNG that is used by the *generator*. This is usefull if two generators should share the same URNG.

Handle uniform RNGs

Notice: Most of these calls are only available for the new generic API to uniform URNGs (that is, `UNUR_URNG_TYPE` must be set to `UNUR_URNG_GENERIC` in ‘`unuran_config.h`’)!

Notice: Some of the below function calls do not work for every source of random numbers since not every library has implemented these features.

double unur_urng_sample (*UNUR_URNG* urng*)

Get a uniform random number from *urng*. If the NULL pointer is given, the default uniform generator is used.

unsigned int unur_urng_sample_array (*UNUR_URNG* urng, double* X, unsigned int dim*)

Set array *X* of length *dim* with uniform random numbers sampled from generator *urng*. If *urng* is the NULL pointer, the default uniform generator is used.

Important: If *urng* is based on a point set generator (this is the case for generators of low discrepancy point sets as used in quasi-Monte Carlo methods) it has a “natural dimension” *s*. In this case either only the first *s* entries of *X* are filled (if *s* < *dim*), or the first *dim* coordinates of the generated point are filled.

The called returns the actual number of entries filled. In case of an error 0 is returned.

int unur_urng_reset (*UNUR_URNG* urng*)

Reset *urng* object. The routine tries two ways to reset the generator (in this order):

1. It uses the reset function given by an `unur_urng_set_reset` call.
2. It uses the seed given by the last `unur_urng_seed` call (which requires a seeding function given by a `unur_urng_set_seed` call).

If neither of the two methods work resetting of the generator is not possible and an error code is returned.

If the NULL pointer is given, the default uniform generator is reset.

int unur_urng_sync (*UNUR_URNG* urng*)

Jump into defined state ("sync") of the generator. This is useful when point generators are used where the coordinates are sampled via `unur_urng_sample`. Then this call can be used to jump to the first coordinate of the next generated point.

int unur_urng_seed (*UNUR_URNG* urng, unsigned long seed*)

Set *seed* for generator *urng*. It returns an error code if this is not possible for the given URNG. If the NULL pointer is given, the default uniform generator is seeded (if possible).

Notice: Seeding should be done only once for a particular generator (except for resetting it to the initial state). Expertise is required when multiple seeds are used to get independent streams. Thus we recommend appropriate libraries for this task, e.g. Pierre L’Ecuyer’s ‘RngStreams’ package. For this library only a package seed can be set and thus the `unur_urng_seed` call will not have any effect to generators of this type. Use `unur_urng_reset` or `unur_urng_rngstream_new` instead, depending whether one wants to reset the stream or get a new stream that is independent from the previous ones.

int unur_urng_anti (*UNUR_URNG* urng, int anti*)

Switch to antithetic random numbers in *urng*. It returns an error code if this is not possible for the given URNG.

If the NULL pointer is given, the antithetic flag of the default uniform generator is switched (if possible).

```
int unur_urng_nextsub (UNUR_URNG* urng)
```

Jump to start of the next substream of *urng*. It returns an error code if this is not possible for the given URNG.

If the NULL pointer is given, the default uniform generator is set to the start of the next substream (if possible).

```
int unur_urng_resetsub (UNUR_URNG* urng)
```

Jump to start of the current substream of *urng*. It returns an error code if this is not possible for the given URNG.

If the NULL pointer is given, the default uniform generator is set to the start of the current substream (if possible).

```
int unur_gen_sync (UNUR_GEN* generator)
```

```
int unur_gen_seed (UNUR_GEN* generator, unsigned long seed)
```

```
int unur_gen_anti (UNUR_GEN* generator, int anti)
```

```
int unur_gen_reset (UNUR_GEN* generator)
```

```
int unur_gen_nextsub (UNUR_GEN* generator)
```

```
int unur_gen_resetsub (UNUR_GEN* generator)
```

Analogous to *unur_urng_sync*, *unur_urng_seed*, *unur_urng_anti*, *unur_urng_reset*, *unur_urng_nextsub*, and *unur_urng_resetsub*, but act on the URNG object used by the *generator* object.

Warning: These calls should be used with care as it influences all generator objects that share the same URNG object!

Generic API to create a new URNG object

Notice: These calls are only available for the new generic API to uniform URNGs (that is, *UNUR_URNG_TYPE* must be set to *UNUR_URNG_GENERIC* in ‘*unuran_config.h*’)

Notice: These functions are provided to build a *UNUR_URNG* object for a particular external random number generator from scratch. For some libraries that contain random number generators (like the GSL) there are special calls, e.g. *unur_urng_gsl_new*, to get such an object. Then there is no need to change the *UNUR_URNG* object as it already contains all available features.

If you have a particular library for random number generators you can either write wrapper function like those in ‘*src/uniform/urng_gsl.c*’ or write an email to the authors of UNURAN to write it for you.

```
UNUR_URNG* unur_urng_new (double (* sampleunif)(void* state), void* state)
```

Get a new URNG object. *sampleunif* is a function to the uniform sampling routine, *state* a pointer to its arguments which usually contains the state variables of the generator.

Functions *sampleunif* with a different type for *p* or without an argument at all also work. A typecast might be necessary to avoid compiler warnings or error messages.

For functions *sampleunif* that does not have any argument should use NULL for *state*.

Important: *sampleunif* must not be the NULL pointer.

There are appropriate calls that simplifies the task of creating URNG objects for some libraries with uniform random number generators, see below.

```
int unur_urng_free (UNUR_URNG* urng)
```

Destroy *urng* object. It returns an error code if this is not possible.

If the NULL is given, this function does nothing.

Warning: This call must be used with care. The *urng* object must not be used by any existing generator object! It is designed to work in conjunction with the wrapper functions to create URNG objects for generators of a particular library. Thus an object created by an *unur_urng_prng_new* call can be simply destroyed by an *unur_urng_free* call.

```
int unir_urng_set_sample_array (UNUR_URNG* urng, unsigned int(*
    samplearray)(void* state, double* X, unsigned int dim ))
    Set function to fill array X of length dim with random numbers generated by generator urng
    (if available).
```

```
int unir_urng_set_sync (UNUR_URNG* urng, void (* reset)(void* sync ))
    Set function for jumping into a defined state ("sync").
```

```
int unir_urng_set_seed (UNUR_URNG* urng, void (* setseed)(void* state,
    unsigned long seed ))
    Set function to seed generator urng (if available).
```

```
int unir_urng_set_anti (UNUR_URNG* urng, void (* setanti)(void* state, int
    anti ))
    Set function to switch the antithetic flag of generator urng (if available).
```

```
int unir_urng_set_reset (UNUR_URNG* urng, void (* reset)(void* state ))
    Set function for resetting the uniform random number generator urng (if available).
```

```
int unir_urng_set_nextsub (UNUR_URNG* urng, void (* nextsub)(void* state ))
    Set function that allows jumping to start of the next substream of urng (if available).
```

```
int unir_urng_set_resetsub (UNUR_URNG* urng, void (* resetsub)(void* state
    ))
    Set function that allows jumping to start of the current substream of urng (if available).
```

```
int unir_urng_set_delete (UNUR_URNG* urng, void (* delete)(void* state ))
    Set function for destroying urng (if available).
```

6.1 Simple interface for uniform random number generators

Simple interface for URNGs of type FVOID, i.e., routines without an argument: `double uniform(void)`.

If independent versions of the same URNG should be used, a copy of the subroutine has to be implemented in the program code (with different names, of course). UNURAN contains some build-in URNGs of this type in directory 'src/uniform/

How To Use

Create an URNG object using *unur_urng_fvoid_new*. By this call a pointer to the sampling routine and (optional) a pointer to a reset routine are copied into the URNG object. Other functions, like seeding the URNG, switching to antithetic random number, or jumping to next substream, can be added to the URNG object by the respective calls, e.g. by *unur_urng_set_seed*. The following routines are supported for URNG objects of type FVOID:

- *unur_urng_sample*
- *unur_urng_sample_array*
- *unur_urng_seed* [optional]
- *unur_urng_reset* [optional]
- *unur_urng_free*

Function reference

UNUR_URNG* `unur_urng_fvoid_new (double (* random)(void), int (* reset)(void))`

Make a URNG object for a genertor that consists of a single function call with a global state variable.

Notice: If independent versions of the same URNG should be used, copies of the subroutine with different names has to be implement in the program code.

If there is no reset function use NULL for the second argument.

UNURAN contains some build-in URNGs of this type in directory ‘src/uniform/’.

6.2 Interface to GSL uniform random number generators

Interface to the uniform random number generators from the GNU Scientific Library (GSL). Documentation and source code of this library is available from <http://www.gnu.org/software/gsl/>.

How To Use

This library has to be installed before compiling UNURAN and UNURAN_HAS_GSL has to be defined in ‘src/unuran_config.h’. Do not forget to link your executables against ‘libgsl’.

The following routines are supported for URNG objects of type GSL:

- `unur_urng_sample`
- `unur_urng_sample_array`
- `unur_urng_seed`
- `unur_urng_reset`
- `unur_urng_free`

Function reference

UNUR_URNG* `unur_urng_gsl_new (const gsl_rng_type* urngtype)`

Make object for URNGs from the ‘GSL’ (GNU Scientific Library). *urngtype* is the type of the chosen generator as described in the GSL manual (see Section Random Number Generation).

This library is available from <http://www.gnu.org/software/gsl/>.

UNUR_URNG* `unur_urng_gslptr_new (gsl_rng* urng)`

Similar to `unur_urng_gsl_new` but it uses a pointer to a generator object as returned by `gsl_rng_alloc(rng_type)`; see ‘GSL’ manual for details.

Notice: There is a subtle but important difference between these two calls. When a generator object is created by a `unur_urng_gsl_new` call, then resetting of the generator works. When a generator object is created by a `unur_urng_gslptr_new` call, then resetting only works after a `unur_urng_seed(urng,myseed)` call.

6.3 Interface to GSL generators for quasi-random points

Interface to the generators for quasi-random points (also called low discrepancy point sets) from the GNU Scientific Library (GSL). Documentation and source code of this library is available from <http://www.gnu.org/software/gsl/>.

How To Use

This library has to be installed before compiling UNURAN and UNURAN_HAS_GSL has to be defined in ‘src/unuran_config.h’. Do not forget to link your executables against ‘libgsl’.

The following routines are supported for URNG objects of type GSL-QRNG:

- `unur_urng_sample`
- `unur_urng_sample_array`
- `unur_urng_reset`
- `unur_urng_sync`
- `unur_urng_free`

`unur_urng_sync` is used to jump to the first coordinate of the next point generated by the generator.

Function reference

`UNUR_URNG* unsur_urng_gslqrng_new (const gsl_qrng_type* qrngtype, unsigned int dim)`

Make object for quasi-random point generators for dimension *dim* from the ‘GSL’ (GNU Scientific Library). *qrngtype* is the type of the chosen generator as described in the GSL manual (see section Quasi-Random Sequences). This library is available from <http://www.gnu.org/software/gsl/>.

6.4 Interface to Otmar Lendl’s pseudo-random number generators

URNGs from the `prng` library. It provides a very flexible way to sample from arbitrary URNGs by means of an object oriented programming paradigm. Similarly to the UNURAN library independent generator objects can be build and used.

This library has been developed by the pLab group at the university of Salzburg (Austria, EU) and implemented by Otmar Lendl. It is available from <http://statistik.wu-wien.ac.at/prng/> or from the pLab site at <http://random.mat.sbg.ac.at/>.

How To Use

This library has to be installed before compiling UNURAN and `UNURAN_HAS_PRNG` has to be defined in ‘`src/unuran_config.h`’. Do not forget to link your executables against ‘`libprng`’.

The following routines are supported for URNG objects of type `PRNG`:

- `unur_urng_sample`
- `unur_urng_sample_array`
- `unur_urng_reset`
- `unur_urng_free`

Function reference

`UNUR_URNG* unsur_urng_prng_new (const char* prngstr)`

Make object for URNGs from Otmar Lendl’s ‘`prng`’ package. *prngstr* is a string that contains the necessary information to create a uniform random number generator. For the format of this string see the ‘`prng`’ user manual.

The ‘`prng`’ library provides a very flexible way to sample from arbitrary URNGs by means of an object oriented programming paradigm. Similarly to the UNURAN library independent generator objects can be build and used. The library has been developed and implemented by Otmar Lendl as member of the pLab group at the university of Salzburg (Austria, EU).

It is available via anonymous ftp from <http://statistik.wu-wien.ac.at/prng/> or from the pLab site at <http://random.mat.sbg.ac.at/>.

UNUR_URNG* `unur_urng_prngptr_new` (*struct prng* urng*)

Similar to `unur_urng_prng_new` but it uses a pointer to a generator object as returned by `prng_new(prngstr)`; see ‘prng’ manual for details.

6.5 Interface to L’Ecuyer’s RNGSTREAMS random number generators

URNGs from Pierre L’Ecuyer’s `RngStream` library for multiple independent streams of pseudo-random numbers. It allows to split a random stream into many substreams. It is available from A GNU-style package is available from <http://statistik.wu-wien.ac.at/software/RngStreams/>.

How To Use

This library has to be installed before compiling UNURAN and UN-URAN_HAS_RNGSTREAMS has to be defined in ‘src/unuran_config.h’. Do not forget to link your executables against this library.

The following routines are supported for URNG objects of type RANDOMSTREAMS:

- `unur_urng_sample`
- `unur_urng_sample_array`
- `unur_urng_reset`
- `unur_urng_nextsub`
- `unur_urng_resetsub`
- `unur_urng_anti`
- `unur_urng_free`

Function reference

UNUR_URNG* `unur_urng_rngstream_new` (*const char* urngstr*)

Make object for URNGs from Pierre L’Ecuyer’s ‘RngStream’ library. This library provides multiple independent streams of pseudo-random numbers and is available from <http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/c/>. `urngstr` is an arbitrary string to label a stream. It need not be unique.

UNUR_URNG* `unur_urng_rngstreamptr_new` (*RngStream rngstream*)

Similar to `unur_urng_rngstream_new` but it uses a pointer to a generator object as returned by `RngStream_CreateStream()`.

6.6 Combine point set generator with random shifts

Generators of type RANDOMSHIFT combine a point set generators with generators to apply random shifts as proposed in [CPa76] :

1. Sample and store a random vector *S*.
2. Run a QMC simulation where *S* is added to each point of the generated quasi-random point (mod 1).
3. Repeat steps 1 and 2.

How To Use

Create a URNG object for a point set generator and a URNG object for a generator to create shift vectors at random. The meta URNG object can then be created using `unur_urng_randomshift_new`. Notice that only pointers to the two underlying URNG generator objects are copied into the

newly created meta generator. Thus manipulating the meta URNG also changes the underlying URNGs and vice versa.

The following routines are supported for URNG objects of type RANDOMSHIFT:

- `unur_urng_sample`
- `unur_urng_sample_array`
- `unur_urng_reset`
- `unur_urng_sync`
- `unur_urng_randomshift_nextshift`
- `unur_urng_free`

`unur_urng_sync` is used to jump to the first coordinate of the next point generated by the generator. `unur_urng_randomshift_nextshift` allows to replace the shift vector by another randomly chosen shift vector.

Important: `unur_urng_sync` is only available if it is implemented for the underlying point set generator.

Important: `unur_urng_reset` is only available if it is available for both underlying generators.

Function reference

`UNUR_URNG* unsur_urng_randomshift_new (UNUR_URNG* qrng, UNUR_URNG* srng, unsigned int dim)`

Make object for URNG with randomly shifted point sets. *qrng* is a generated that generates point sets of dimension *dim*. *srng* is a generated that generates random numbers or vectors.

Notice: Only pointers to the respective objects *qrng* and *srng* are copied into the created meta generator. Thus manipulating the meta URNG also changes the underlying URNGs and vice versa.

`int unsur_urng_randomshift_nextshift (UNUR_URNG* urng)`

Get the next (randomly chosen) vector for shifting the points set, and the underlying point generator *qrng* is reset.

7 UNURAN Library of standard distributions

Although it is not its primary target, many distributions are already implemented in UNURAN. This section presents these available distributions and their parameters.

The syntax to get a distribuion object for distributions `<dname>` is:

```
UNUR_DISTR* unur_distr_<dname> (double* params, int n_params)
```

[-]

params is an array of doubles of size *n_params* holding the parameters.

E.g. to get an object for the gamma distribution (with shape parameter) use

```
unur_distr_gamma( params, 1 );
```

Distributions may have default parameters with need not be given explicitly. E.g. The gamma distribution has three parameters: the shape, scale and location parameter. Only the (first) shape parameter is required. The others can be omitted and are then set by default values.

```
/* alpha = 5; default: beta = 1, gamma = 0 */
double fpar[] = {5.};
unur_distr_gamma( fpar, 1 );

/* alpha = 5, beta = 3; default: gamma = 0 */
double fpar[] = {5., 3.};
unur_distr_gamma( fpar, 2 );

/* alpha = 5, beta = 3, gamma = -2
double fpar[] = {5., 3., -2.};
unur_distr_gamma( fpar, 3 );
```

Important: Naturally the computational accuracy limits the possible parameters. There shouldn't be problems when the parameters of a distribution are in a "reasonable" range but e.g. the normal distribution $N(10^{15}, 1)$ won't yield the desired results. (In this case it would be better generating $N(0, 1)$ and *then* transform the results.)

Of course computational inaccuracy is not specific to UNURAN and should always be kept in mind when working with computers.

Important: The routines of the standard library are included for non-uniform random variate generation and not to provide special functions for statistical computations.

Remark

The following keywords are used in the tables:

<i>PDF</i>	probability density function, with variable x .
<i>PMF</i>	probability mass function, with variable k .
<i>constant</i>	normalization constant for given PDF and PMF, resp. They must be multiplied by <i>constant</i> to get the "real" PDF and PMF.
<i>CDF</i>	gives information whether the CDF is implemented in UNURAN.
<i>domain</i>	domain PDF and PMF, resp.
<i>parameters</i>	<i>n_std</i> (<i>n_total</i>): list list of parameters for distribution, where <i>n_std</i> is the number of parameters for the standard form of the distribution and <i>n_total</i> the total number for the (non-standard form of the) distribution. <i>list</i> is the list of parameters in the order as they are stored in the array of parameters. Optional parameter that can be omitted are enclosed in square brackets [...].

A detailed list of these parameters gives then the range of valid parameters and defaults for optional parameters that are used when these are omitted.

reference gives reference for distribution (see [Appendix C \[Bibliography\]](#), page 221).

special generators

lists available special generators for the distribution. The first number is the variant that to be set by `unur_cstd_set_variant` and `unur_dstd_set_variant` call, respectively. If no variant is set the default variant DEF is used. In the table the respective abbreviations DEF and INV are used for UNUR_STDGEN_DEFAULT and UNUR_STDGEN_INVERSION. Also the references for these methods are given (see [Appendix C \[Bibliography\]](#), page 221).

Notice that these generators might be slower than universal methods.

If DEF is omitted, the first entry is the default generator.

7.1 UNURAN Library of continuous univariate distributions

7.1.1 beta – Beta distribution

PDF: $(x-a)^{p-1} (b-x)^{q-1}$

constant: $1/(Beta(p, q) (b-a)^{p+q-1})$

domain: $a < x < b$

parameters 2 (4): p, q [, a, b]

No.	name		default	
[0]	p	> 0		(<i>scale</i>)
[1]	q	> 0		(<i>scale</i>)
[2]	a		0	(<i>location, scale</i>)
[3]	b	$> a$	1	(<i>location, scale</i>)

reference: [JKBc95: Ch.25; p.210]

7.1.2 cauchy – Cauchy distribution

PDF: $\frac{1}{1+((x-\theta)/\lambda)^2}$

constant: $\frac{1}{\pi\lambda}$

domain: $-\infty < x < \infty$

parameters 0 (2): [theta [, lambda]]

No.	name		default	
[0]	θ		0	(<i>location</i>)
[1]	λ	> 0	1	(<i>scale</i>)

reference: [JKBb94: Ch.16; p.299]

special generators:

INV Inversion method

7.1.3 chi – Chi distribution

PDF: $x^{\nu-1} \exp(-x^2/2)$

constant: $1/(2^{(\nu/2)-1} \Gamma(\nu/2))$

domain: $0 \leq x < \infty$

parameters 1 (1): nu

No.	name		default	
[0]	ν	> 0		(<i>shape</i>)

reference: [JKBb94: Ch.18; p.417]

special generators:

DEF Ratio of Uniforms with shift (only for $\nu \geq 1$) [MJ87]

7.1.4 chisquare – Chisquare distribution

PDF: $x^{(\nu/2)-1} \exp(-x/2)$

constant: $1/(2^{\nu/2} \Gamma(\nu/2))$

domain: $0 \leq x < \infty$

parameters 1 (1): nu

No.	name	default	
[0]	ν	> 0	(shape (degrees of freedom))

reference: [JKBb94: Ch.18; p.416]

7.1.5 exponential – Exponential distribution

PDF: $\exp(-\frac{x-\theta}{\sigma})$

constant: $\frac{1}{\sigma}$

domain: $\theta \leq x < \infty$

parameters 0 (2): [sigma [, theta]]

No.	name	default	
[0]	σ	> 0	1 (scale)
[1]	θ		0 (location)

reference: [JKBb94: Ch.19; p.494]

special generators:

INV	Inversion method
-----	------------------

7.1.6 extremeI – Extreme value type I (Gumbel-type) distribution

PDF: $\exp(-\exp(-\frac{x-\zeta}{\theta}) - \frac{x-\zeta}{\theta})$

constant: $\frac{1}{\theta}$

domain: $-\infty < x < \infty$

parameters 0 (2): [zeta [, theta]]

No.	name	default	
[0]	ζ	0	(location)
[1]	θ	> 0	1 (scale)

reference: [JKBc95: Ch.22; p.2]

special generators:

INV	Inversion method
-----	------------------

7.1.7 extremeII – Extreme value type II (Frechet-type) distribution

PDF: $\exp(-(\frac{x-\zeta}{\theta})^{-k}) (\frac{x-\zeta}{\theta})^{-k-1}$

constant: $\frac{k}{\theta}$

domain: $\zeta < x < \infty$

parameters 1 (3): k [, zeta [, theta]]

No.	name		default	
[0]	k	> 0		(<i>shape</i>)
[1]	ζ		0	(<i>location</i>)
[2]	θ	> 0	1	(<i>scale</i>)

reference: [JKBc95: Ch.22; p.2]

special generators:

INV Inversion method

7.1.8 gamma – Gamma distribution

PDF: $(\frac{x-\gamma}{\beta})^{\alpha-1} \exp(-\frac{x-\gamma}{\beta})$

constant: $1/(\beta \Gamma(\alpha))$

domain: $\gamma < x < \infty$

parameters 1 (3): alpha [, beta [, gamma]]

No.	name		default	
[0]	α	> 0		(<i>shape</i>)
[1]	β	> 0	1	(<i>scale</i>)
[2]	γ		0	(<i>location</i>)

reference: [JKBb94: Ch.17; p.337]

special generators:

DEF Acceptance Rejection combined with Acceptance Complement [ADa74]
[ADa82]

2 Rejection from log-logistic envelopes [CHa77]

7.1.9 laplace – Laplace distribution

PDF: $\exp(-\frac{|x-\theta|}{\phi})$

constant: $\frac{1}{2\phi}$

domain: $-\infty < x < \infty$

parameters 0 (2): [theta [, phi]]

No.	name		default	
[0]	θ		0	(<i>location</i>)
[1]	ϕ	> 0	1	(<i>scale</i>)

reference: [JKBc95: Ch.24; p.164]

special generators:

INV Inversion method

7.1.10 logistic – Logistic distribution

PDF: $\exp(-\frac{x-\alpha}{\beta}) (1 + \exp(-\frac{x-\alpha}{\beta}))^{-2}$

constant: $\frac{1}{\beta}$

domain: $-\infty < x < \infty$

parameters 0 (2): [alpha [, beta]]

No.	name	default
[0]	α	0 <i>(location)</i>
[1]	β > 0	1 <i>(scale)</i>

reference: [JKBc95: Ch.23; p.115]

special generators:

INV Inversion method

7.1.11 lomax – Lomax distribution (Pareto distribution of second kind)

PDF: $(x + C)^{-(a+1)}$

constant: $a C^a$

domain: $0 \leq x < \infty$

parameters 1 (2): a [, C]

No.	name	default
[0]	a > 0	<i>(shape)</i>
[1]	C > 0	1 <i>(scale)</i>

reference: [JKBb94: Ch.20; p.575]

special generators:

INV Inversion method

7.1.12 normal – Normal distribution

PDF: $\exp(-\frac{1}{2}(\frac{x-\mu}{\sigma})^2)$

constant: $\frac{1}{\sigma \sqrt{2\pi}}$

domain: $-\infty < x < \infty$

parameters 0 (2): [mu [, sigma]]

No.	name	default
[0]	μ	0 <i>(location)</i>
[1]	σ > 0	1 <i>(scale)</i>

reference: [JKBb94: Ch.13; p.80]

special generators:

DEF	ACR method (Acceptance-Complement Ratio) [HDa90]
1	Box-Muller method [BMa58]
2	Polar method with rejection [MGa62]
3	Kindermann-Ramage method [KRa76]
INV	Inversion method (slow)

7.1.13 pareto – Pareto distribution (of first kind)*PDF:* $x^{-(a+1)}$ *constant:* $a k^a$ *domain:* $k < x < \infty$ *parameters 2 (2):* k, a

No.	name	default	
[0]	k	> 0	$(shape, location)$
[1]	a	> 0	$(shape)$

reference: [JKBb94: Ch.20; p.574]*special generators:*

INV Inversion method

7.1.14 powerexponential – Powerexponential (Subbotin) distribution*PDF:* $\exp(-|x|^\tau)$ *constant:* $1/(2\Gamma(1 + 1/\tau))$ *domain:* $-\infty < x < \infty$ *parameters 1 (1):* tau

No.	name	default	
[0]	τ	> 0	$(shape)$

reference: [JKBc95: Ch.24; p.195]*special generators:*DEF Transformed density rejection (only for $\tau \geq 1$) [DLa86]**7.1.15 rayleigh – Rayleigh distribution***PDF:* $x \exp(-1/2 (\frac{x}{\sigma})^2)$ *constant:* $\frac{1}{\sigma^2}$ *domain:* $0 \leq x < \infty$ *parameters 1 (1):* sigma

No.	name	default	
[0]	σ	> 0	$(scale)$

reference: [JKBb94: Ch.18; p.456]**7.1.16 student – Student's t distribution***PDF:* $(1 + \frac{t^2}{\nu})^{-(\nu+1)/2}$ *constant:* $\frac{1}{\sqrt{\nu} B(1/2, \nu/2)}$ *CDF:* not implemented!*domain:* $-\infty < x < \infty$

parameters 1 (1): nu

No.	name	default	
[0]	ν	> 0	(<i>shape</i>)

reference: [JKBc95: Ch.28; p.362]

7.1.17 triangular – Triangular distribution

PDF: $2x/H$, for $0 \leq x \leq H$
 $2(1-x)/(1-H)$, for $H \leq x \leq 1$

constant: 1

domain: $0 \leq x \leq 1$

parameters 0 (1): [H]

No.	name	default	
[0]	H	$0 \leq H \leq 1$	1/2 (<i>shape</i>)

reference: [JKBc95: Ch.26; p.297]

special generators:

INV Inversion method

7.1.18 uniform – Uniform distribution

PDF: $\frac{1}{b-a}$

constant: 1

domain: $a < x < b$

parameters 0 (2): [a, b]

No.	name	default	
[0]	a	0	(<i>location</i>)
[1]	b	$> a$	1 (<i>location</i>)

reference: [JKBc95: Ch.26; p.276]

special generators:

INV Inversion method

7.1.19 weibull – Weibull distribution

PDF: $(\frac{x-\zeta}{\alpha})^{c-1} \exp(-(\frac{x-\zeta}{\alpha})^c)$

constant: $\frac{c}{\alpha}$

domain: $\zeta < x < \infty$

parameters 1 (3): c [, alpha [, zeta]]

No.	name	default	
[0]	c	> 0	(<i>shape</i>)
[1]	α	> 0	1 (<i>scale</i>)
[2]	ζ	0	(<i>location</i>)

reference: [JKBb94: Ch.21; p.628]

special generators:

INV Inversion method

7.2 UNURAN Library of continuous multivariate distributions

7.2.1 copula – Copula (distribution with uniform marginals)

UNUR_DISTR *unur_distr_copula(int dim, const double *rankcorr) creates a distribution object for a copula with *dim* components. *rankcorr* is an array of size *dim* × *dim* and holds the rank correlation matrix (Spearman's correlation), where the rows of the matrix are stored consecutively in this array. The NULL pointer can be used instead the identity matrix.

If covar is not a valid rank correlation matrix (i.e., not positive definite) then no distribution object is created and NULL is returned.

7.2.2 multicauchy – Multicauchy distribution

PDF: $f(x) = 1/(1 + (x - \mu)^t \cdot \Sigma^{-1} \cdot (x - \mu))^{(dim+1)/2}$

constant: $\Gamma((dim + 1)/2)/(\pi^{(dim+1)/2} \sqrt{\det(\Sigma)})$

domain: $-\infty^{dim} < x < \infty^{dim}$

parameters 0 (2): [mu, Sigma]

No.	name	default
[0]	μ	(0, . . . , 0) (<i>location</i>)
[1]	<i>Sigma</i>	<i>Symm, Pos.def.</i> I (<i>shape</i>)

7.2.3 multiexponential – Multiexponential distribution

PDF: $f(x) = \text{Prod}_{i=0}^{dim-1} \exp(-(dim-i)(x_i - x_{i-1} - (\theta_i - \theta_{i-1}))/\sigma_i); \text{with } x_{-1} = 0 \text{ and } \theta_{i-1} = 0$

constant: $\text{Prod}_{i=0}^{dim-1} 1/\sigma_i$

domain: $0^{dim} \leq x < \infty^{dim}$

parameters 0 (2): [sigma, theta]

No.	name	default
[0]	σ	(1, . . . , 1) (<i>shape</i>)
[1]	θ	(0, . . . , 0) (<i>location</i>)

7.2.4 multinormal – Multinormal distribution

PDF: $f(x) = \exp(-1/2 (x - \mu)^t \cdot \Sigma^{-1} \cdot (x - \mu))$

constant: $1/((2\pi)^{dim/2} \sqrt{\det(\Sigma)})$

domain: $-\infty^{dim} < x < \infty^{dim}$

parameters 0 (2): [mu, Sigma]

No.	name	default
[0]	μ	(0, . . . , 0) (<i>location</i>)
[1]	<i>Sigma</i>	<i>Symm, Pos.def.</i> I (<i>shape</i>)

reference: [KBJe00: Ch.45; p.105]

7.2.5 multistudent – Multistudent distribution

PDF: $f(x) = 1/(1 + (x - \mu)^t \cdot \Sigma^{-1} \cdot (x - \mu)/m)^{(dim+m)/2}$

constant: $\Gamma((dim + m)/2)/(\Gamma(m/2)(m\pi)^{dim/2} \sqrt{\det(\Sigma)})$

domain: $-\infty^{dim} < x < \infty^{dim}$

parameters 0 (3): [m, mu, Sigma]

No.	name		default	
[0]	m	$m > 0$	1	(location)
[1]	μ		(0, . . . , 0)	(location)
[2]	Σ	$Symm, Pos.def.$	I	(shape)

7.3 UNURAN Library of discrete univariate distributions

At the moment there are no CDFs implemented for discrete distribution. Thus `unur_distr_discr_upd_pmfsum` does not work properly for truncated distribution.

7.3.1 binomial – Binomial distribution

PMF: $\binom{n}{k} p^k (1-p)^{n-k}$

constant: 1

domain: $0 \leq k \leq n$

parameters 2 (2): n, p

No.	name	default	
[0]	n	≥ 1	(no. of elements)
[1]	p	$0 < p < 1$	(shape)

reference: [JKKa92: Ch.3; p.105]

special generators:

DEF Ratio of Uniforms/Inversion [STa89]

7.3.2 geometric – Geometric distribution

PMF: $p(1-p)^k$

constant: 1

domain: $0 \leq k < \infty$

parameters 1 (1): p

No.	name	default	
[0]	p	$0 < p < 1$	(shape)

reference: [JKKa92: Ch.5.2; p.201]

special generators:

INV Inversion method

7.3.3 hypergeometric – Hypergeometric distribution

PMF: $\binom{M}{k} \binom{N-M}{n-k} / \binom{N}{n}$

constant: 1

domain: $\max(0, n - N + M) \leq k \leq \min(n, M)$

parameters 3 (3): N, M, n

No.	name	default	
[0]	N	≥ 1	(no. of elements)
[1]	M	$1 \leq M \leq N$	(shape)
[2]	n	$1 \leq n \leq N$	(shape)

reference: [JKKa92: Ch.6; p.237]

special generators:

DEF Ratio of Uniforms/Inversion [STa89]

7.3.4 logarithmic – Logarithmic distribution

PMF: θ^k/k

constant: $-\log(1 - \theta);$

domain: $1 \leq k < \infty$

parameters 1 (1): theta

No.	name	default	
[0]	θ	$0 < \theta < 1$	(<i>shape</i>)

reference: [JKKa92: Ch.7; p.285]

special generators:

DEF	Inversion/Transformation [KAa81]
-----	----------------------------------

7.3.5 negativebinomial – Negative Binomial distribution

PMF: $\binom{k+r-1}{r-1} p^r (1-p)^k$

constant: 1

domain: $0 \leq k < \infty$

parameters 2 (2): p, r

No.	name	default	
[0]	p	$0 < p < 1$	(<i>shape</i>)
[1]	r	> 0	(<i>shape</i>)

reference: [JKKa92: Ch.5.1; p.200]

7.3.6 poisson – Poisson distribution

PMF: $\theta^k/k!$

constant: $\exp(\theta)$

domain: $0 \leq k < \infty$

parameters 1 (1): theta

No.	name	default	
[0]	θ	> 0	(<i>shape</i>)

reference: [JKKa92: Ch.4; p.151]

special generators:

DEF	Tabulated Inversion combined with Acceptance Complement [ADb82]
2	Tabulated Inversion combined with Patchwork Rejection [ZHa94]

7.4 UNURAN Library of random matrices

7.4.1 correlation – Random correlation matrix

UNUR_DISTR *unur_distr_correlation(int n) creates a distribution object for a random correlation matrix of n rows and columns. It can be used with method MCORR (see [Section 5.9.1 \[Random Correlation Matrix\]](#), page 174) to generate random correlation matrices of the given size.

8 Error handling

This chapter describes the way that UNURAN routines report errors.

8.1 Error reporting

UNURAN routines report an error whenever they cannot perform the task requested of them. For example, apply transformed density rejection to a distribution that violates the T-concavity condition, or trying to set a parameter that is out of range. It might also happen that the setup fails for transformed density rejection for a T-concave distribution with some extreme density function simply because of round-off errors that makes the generation of a hat function numerically impossible. Situations like this may happen when using black box algorithms and you should check the return values of all routines.

All `..._set_...`, and `..._chg_...` calls return `UNUR_SUCCESS` if they could be executed successfully. Otherwise some error codes are returned if it was not possible to set or change the desired parameters, e.g. because the given values are out of range, or simply because you have changed the method but not the corresponding set call and thus an invalid parameter or generator object is used.

All routines that return a pointer to the requested object will return a `NULL` pointer in case of error. (Thus you should always check the pointer to avoid possible segmentation faults. Sampling routines usually do not check the given pointer to the generator object. However you can switch on checking for `NULL` pointer defining the compiler switch `UNUR_ENABLE_CHECKNULL` in `'unuran_config.h'` to avoid nasty segmentation faults.)

The library distinguishes between two major classes of error:

(fatal) errors:

The library was not able to construct the requested object.

warnings: Some problems encounters while constructing a generator object. The routine has tried to solve the problem but the resulting object might not be what you want. For example, choosing a special variant of a method does not work and the initialization routine might switch to another variant. Then the generator produces random variates of the requested distribution but correlation induction is not possible. However it also might happen that changing the domain of a distribution has failed. Then the generator produced random variates with too large/too small range, i.e. their distribution is not correct

It is obvious from the example that this distinction between errors and warning is rather crude and sometimes arbitrary.

UNURAN routines use the global variable `unur_errno` to report errors, completely analogously to C library's `errno`. (However this approach is not thread-safe. There can be only one instance of a global variable per program. Different threads of execution may overwrite `unur_errno` simultaneously). Thus when an error occurs the caller of the routine can examine the error code in `unur_errno` to get more details about the reason why a routine failed. You get a short description of the error by a `unur_get_strerror` call. All the error code numbers have prefix `UNUR_ERR_` and expand to non-zero constant unsigned integer values. Error codes are divided into six main groups.

List of error codes

- Procedure executed successfully (no error)

`UNUR_SUCCESS (0x0u)`
success (no error)

- Errors that occurred while handling distribution objects.

UNUR_ERR_DISTR_SET

set failed (invalid parameter).

UNUR_ERR_DISTR_GET

get failed (parameter not set).

UNUR_ERR_DISTR_NPARAMS

invalid number of parameters.

UNUR_ERR_DISTR_DOMAIN

parameter(s) out of domain.

UNUR_ERR_DISTR_GEN

invalid variant for special generator.

UNUR_ERR_DISTR_REQUIRED

incomplete distribution object, entry missing.

UNUR_ERR_DISTR_UNKNOWN

unknown distribution, cannot handle.

UNUR_ERR_DISTR_INVALID

invalid distribution object.

UNUR_ERR_DISTR_DATA

data are missing.

UNUR_ERR_DISTR_PROP

desired property does not exist

- Errors that occurred while handling parameter objects.

UNUR_ERR_PAR_SET

set failed (invalid parameter)

UNUR_ERR_PAR_VARIANT

invalid variant -> using default

UNUR_ERR_PAR_INVALID

invalid parameter object

- Errors that occurred while handling generator objects.

UNUR_ERR_GEN

error with generator object.

UNUR_ERR_GEN_DATA

(possibly) invalid data.

UNUR_ERR_GEN_CONDITION

condition for method violated.

UNUR_ERR_GEN_INVALID

invalid generator object.

UNUR_ERR_GEN_SAMPLING

sampling error.

- Errors that occurred while handling URNG objects.

UNUR_ERR_URNG

generic error with URNG object.

- UNUR_ERR_URNG_MISS
 missing functionality.
- Errors that occurred while parsing strings.
 - UNUR_ERR_STR
 error in string.
 - UNUR_ERR_STR_UNKNOWN
 unknown keyword.
 - UNUR_ERR_STR_SYNTAX
 syntax error.
 - UNUR_ERR_STR_INVALID
 invalid parameter.
 - UNUR_ERR_FSTR_SYNTAX
 syntax error in function string.
 - UNUR_ERR_FSTR_DERIV
 cannot derivate function.
- Other run time errors.
 - UNUR_ERR_DOMAIN
 argument out of domain.
 - UNUR_ERR_ROUNDOFF
 (serious) round-off error.
 - UNUR_ERR_MALLOC
 virtual memory exhausted.
 - UNUR_ERR_NULL
 invalid NULL pointer.
 - UNUR_ERR_COOKIE
 invalid cookie.
 - UNUR_ERR_GENERIC
 generic error.
 - UNUR_ERR_SILENT
 silent error (no error message).
 - UNUR_ERR_INF
 infinity occurred.
 - UNUR_ERR_NAN
 NaN occurred.
 - UNUR_ERR_COMPILE
 Requested routine requires different compilation switches. Recompilation of library necessary.
 - UNUR_ERR_SHOULD_NOT_HAPPEN
 Internal error, that should not happen. Please report this bug!

Function reference

`extern int unur_errno` [Variable]
 Global variable for reporting diagnostics of error.

8.2 Output streams

In addition to reporting error via the `unur_errno` mechanism the library also provides an (optional) error handler. The error handler is called by the library functions when they are about to report an error. Then a short error diagnostics is written via two output streams. Both can be switched on/off by compiler flag `UNUR_WARNINGS_ON` in `'unuran_config.h'`.

The first stream is `stderr`. It can be enabled by defining the macro `UNUR_ENABLE_STDERR` in `'unuran_config.h'`.

The second stream can be set arbitrarily by the `unur_set_stream` call. If no such stream is given by the user a default stream is used by the library: all warnings and error messages are written into the file `unuran.log` in the current working directory. The name of this file defined by the macro `UNUR_LOG_FILE` in `'unuran_config.h'`. If the `stdout` should be used, define this macro by `"stdout"`.

This output stream is also used to log descriptions of build generator objects and for writing debugging information. If you want to use this output stream for your own programs use `unur_get_stream` to get its file handler. This stream is enabled by the compiler switch `UNUR_ENABLE_LOGFILE` in `'unuran_config.h'`.

All warnings, error messages and all debugging information are written onto the same output stream. To distinguish between the messages for different generators define the macro `UNUR_ENABLE_GENID` in `'unuran_config.h'`. Then every generator object has a unique identifier that is used for every message.

Function reference

`const char* unur_get_strerror (const int unur_errno)`

Get a short description for error code value.

`FILE* unur_set_stream (FILE* new_stream)`

Set new file handle for output stream; the old file handle is returned. The `NULL` pointer is not allowed. (If you want to disable logging of debugging information use `unur_set_default_debug(UNUR_DEBUG_OFF)` instead.)

The output stream is used to report errors and warning, and debugging information. It is also used to log descriptions of build generator objects (when this feature is switched on; see also ?).

`FILE* unur_get_stream (void)`

Get the file handle for the current output stream.

9 Debugging

The UNURAN library has several debugging levels which can be switched on/off by debugging flags. This debugging feature can be enabled by defining the macro `UNUR_ENABLE_LOGGING` in ‘`unuran_config.h`’. The debugging levels range from print a short description of the build generator object to a detailed description of hat functions and tracing the sampling routines. The output is print onto the output stream obtained by `unur_get_stream` (see also ?). These flags can be set or changed by the respective calls `unur_set_debug` and `unur_chg_debug` independently for each generator. The default debugging flags are given by the macro `UNUR_DEBUGFLAG_DEFAULT` in ‘`unuran_config.h`’. This default can be overwritten at run time by a `unur_set_default_debug` call.

Off course these debugging flags depend on the chosen method. Since most of these are merely for debugging the library itself, a description of the flags are given in the corresponding source files of the method. Nevertheless, the following flags can be used with all methods.

Common debug flags:

```
UNUR_DEBUG_OFF
    switch off all debuging information

UNUR_DEBUG_ALL
    all avaivable information

UNUR_DEBUG_INIT
    parameters of generator object after initialization

UNUR_DEBUG_SETUP
    data created at setup

UNUR_DEBUG_ADAPT
    data created during adaptive steps

UNUR_DEBUG_SAMPLE
    trace sampling
```

Almost all routines check a given pointer before they read from or write to the given address. This does not hold for time-critical routines like all sampling routines. Then you are responsible for checking a pointer that is returned from a `unur_init` call. However it is possible to turn on checking for invalid NULL pointers even in such time-critical routines by defining `UNUR_ENABLE_CHECKNULL` in ‘`unuran_config.h`’.

Another debugging tool used in the library are magic cookies that validate a given pointer. It produces an error whenever a given pointer points to an object that is invalid in the context. The usage of magic cookies can be switched on by defining `UNUR_COOKIES` in ‘`unuran_config.h`’.

Function reference

```
int unur_set_debug (UNUR_PAR* parameters, unsigned debug)
```

Set debugging flags for generator.

```
int unur_chg_debug (UNUR_GEN* generator, unsigned debug)
```

Change debugging flags for generator.

```
int unur_set_default_debug (unsigned debug)
```

Overwrite the default debugging flag.

10 Testing

The following routines can be used to test the performance of the implemented generators and can be used to verify the implementations. They are declared in ‘`unuran_tests.h`’ which has to be included.

Function reference

void unur_run_tests (UNUR_PAR* *parameters*, unsigned *tests*)

Run a battery of tests. The following tests are available (use | to combine these tests):

UNUR_TEST_ALL

run all possible tests.

UNUR_TEST_TIME

estimate generation times.

UNUR_TEST_N_URNG

count number of uniform random numbers

UNUR_TEST_N_PDF

count number of PDF calls

UNUR_TEST_CHI2

run χ^2 test for goodness of fit

UNUR_TEST_SAMPLE

print a small sample.

All these tests can be started individually (see below).

void unur_test_printsample (UNUR_GEN* *generator*, int *n_rows*, int *n_cols*, FILE* *out*)

Print a small sample with *n_rows* rows and *n_cols* columns. *out* is the output stream to which all results are written.

UNUR_GEN* unur_test_timing (UNUR_PAR* *parameters*, int *log_samplesize*, double* *time_setup*, double* *time_sample*, int *verbosity*, FILE* *out*)

Timing. *parameters* is an parameter object for which setup time and marginal generation times have to be measured. The results are written into *time_setup* and *time_sample*, respectively. *log_samplesize* is the common logarithm of the sample size that is used for timing.

If *verbosity* is TRUE then a small table is printed to the `stdout` with setup time, marginal generation time and average generation times for generating 10, 100, ... random variates. All times are given in micro seconds and relative to the generation times for the underlying uniform random number (using the UNIF interface) and an exponential distributed random variate using the inversion method.

The created generator object is returned. If a generator object could not be created successfully, then NULL is returned.

If *verbosity* is TRUE the result is written to the output stream *out*.

Notice: All timing results are subject to heavy changes. Reruning timings usually results in different results. Minor changes in the source code can cause changes in such timings up to 25 percent.

```
double unur_test_timing_uniform (const UNUR_PAR* parameters, int
                                log_samplesize)
double unur_test_timing_exponential (const UNUR_PAR* parameters, int
                                     log_samplesize)
```

Marginal generation times for the underlying uniform random number (using the UNIF interface) and an exponential distributed random variate using the inversion method. These times are used in `unur_test_timing` to compute the relative timings results.

```
double unur_test_timing_total (const UNUR_PAR* parameters, int samplesize,
                              double avg_duration)
```

Timing. *parameters* is an parameter object for which average times a sample of size *samplesize* (including setup) are estimated. Thus sampling is repeated and the median of these timings is returned (in micro seconds). The number of iterations is computed automatically such that the total amount of time necessary for the test ist approximately *avg_duration* (given in seconds). However, for very slow generator with expensive setup time the time necessary for this test may be (much) larger.

If an error occurs then -1 is returned.

Notice: All timing results are subject to heavy changes. Reruning timings usually results in different results. Minor changes in the source code can cause changes in such timings up to 25 percent.

```
int unur_test_count_urn (UNUR_GEN* generator, int samplesize, int
                        verbosity, FILE* out)
```

Count used uniform random numbers. It returns the total number of uniform random numbers required for a sample of non-uniform random variates of size *samplesize*. Counting uniform random numbers might not work for the chosen `UNUR_URNG_TYPE` in 'unuran_config.h'. In this case -1 is returned.

If *verbosity* is TRUE the result is written to the output stream *out*.

Notice: This test uses global variables to store counters. Thus it is not thread save.

```
int unur_test_count_pdf (UNUR_GEN* generator, int samplesize, int
                        verbosity, FILE* out)
```

Count evaluations of PDF and similar functions. It returns the total number of evaluations of all such functions required for a sample of non-uniform random variates of size *samplesize*. If *verbosity* is TRUE then a more detailed report is printed to the output stream *out*. In case of an error -1 is returned. This test is run on a copy of the given generator object.

Notice: The printed numbers of evaluation should be interpreted with care. For example, methods either use the PDF or the logPDF; if only the logPDF is given, but a method needs the PDF then both the logPDF and the PDF (a wrapper around the logPDF) are called and thus one call to the PDF is counted twice.

Notice: This test uses global variables to store function pointers and counters. Thus it is not thread save.

```
int unur_test_par_count_pdf (UNUR_PAR* parameters, int samplesize, int
                            verbosity, FILE* out)
```

Same as `unur_test_count_pdf` except that it is run on a parameter object. Thus it also prints the number of function evaluations for the setup. The temporary created generator object is destroyed before the results are returned.

```
double unur_test_chi2 (UNUR_GEN* generator, int intervals, int
    samplesize, int classmin, int verbosity, FILE* out)
```

Run a Chi² test with the *generator*. The resulting p-value is returned.

It works with discrete and continuous univariate distributions. For the latter the CDF of the distribution is required.

intervals is the number of intervals that is used for continuous univariate distributions. *samplesize* is the size of the sample that is used for testing. If it is set to 0 then a sample of size *intervals*² is used (bounded to some upper bound).

classmin is the minimum number of expected entries per class. If a class has too few entries then some classes are joined.

verbosity controls the output of the routine. If it is set to 1 then the result is written to the output stream *out*. If it is set to 2 additionally the list of expected and observed data is printed. If it is set to 3 then all generated numbers are printed. There is no output when it is set to 0.

Notice, for multivariate distributions also tests on the marginal distributions are performed. Then the minimal p-value of all these tests is returned.

```
int unur_test_moments (UNUR_GEN* generator, double* moments, int
    n_moments, int samplesize, int verbosity, FILE* out)
```

Computes the first *n_moments* central moments for a sample of size *samplesize*. The result is stored into the array *moments*. *n_moments* must be an integer between 1 and 4. For multivariate distributions the moments are stored consecutively for each dimension and the provided *moments*-array must have a length of at least $(n_moments+1) * dim$, where *dim* is the dimension of the multivariate distribution. The *m*'th moment for the *d*'th dimension ($0 \leq d < dim$) is thus stored in the array element *moments*[*d***n_moments*+*m*]

If *verbosity* is TRUE the result is written to the output stream *out*.

```
double unur_test_correlation (UNUR_GEN* generator1, UNUR_GEN*
    generator2, int samplesize, int verbosity, FILE* out)
```

Compute the correlation coefficient between streams from *generator1* and *generator2* for two samples of size *samplesize*. The resulting correlation is returned.

If *verbosity* is TRUE the result is written to the output stream *out*.

```
int unur_test_quartiles (UNUR_GEN* generator, double* q0, double* q1,
    double* q2, double* q3, double* q4, int samplesize, int verbosity, FILE* out)
```

Estimate quartiles of sample of size *samplesize*. The resulting quantiles are stored in the variables *q*:

<i>q0</i>	minimum
<i>q1</i>	25%
<i>q2</i>	median (50%)
<i>q3</i>	75%
<i>q4</i>	maximum

If *verbosity* is TRUE the result is written to the output stream *out*.

11 Miscellaneous

11.1 Mathematics

The following macros have been defined

`UNUR_INFINITY`

indicates infinity for floating point numbers (of type `double`). Internally `HUGE_VAL` is used.

`INT_MAX`

`INT_MIN` indicate infinity and minus infinity, resp., for integers (defined by ISO C standard).

`TRUE`

`FALSE` boolean expression for return values of `set` functions.

Appendix A A Short Introduction to Random Variate Generation

Random variate generation is the small field of research that deals with algorithms to generate random variates from various distributions. It is common to assume that a uniform random number generator is available. This is a program that produces a sequence of independent and identically distributed continuous $U(0, 1)$ random variates (i.e. uniform random variates on the interval $(0, 1)$). Of course real world computers can never generate ideal random numbers and they cannot produce numbers of arbitrary precision but state-of-the-art uniform random number generators come close to this aim. Thus random variate generation deals with the problem of transforming such a sequence of $U(0, 1)$ random numbers into non-uniform random variates.

Here we shortly explain the basic ideas of the *inversion*, *rejection*, and the *ratio of uniforms* method. How these ideas can be used to design a particular automatic random variate generation algorithms that can be applied to large classes of distributions is shortly explained in the description of the different methods included in this manual.

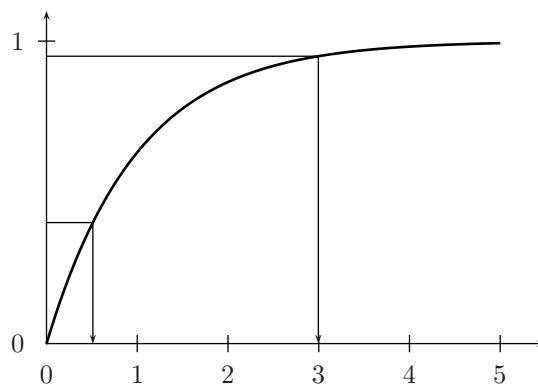
For a deeper treatment of the ideas presented here, for other basic methods and for automatic generators we refer the interested reader to our book [HLD04].

A.1 The Inversion Method

When the inverse F^{-1} of the cumulative distribution function is known, then random variate generation is easy. We just generate a uniformly $U(0, 1)$ distributed random number U and return

$$X = F^{-1}(U).$$

The following figure shows how the inversion method works for the exponential distribution.



This algorithm is so simple that inversion is certainly the method of choice if the inverse CDF is available in closed form. This is the case e.g. for the exponential and the Cauchy distribution.

The inversion method also has other special advantages that make it even more attractive for simulation purposes. It preserves the structural properties of the underlying uniform pseudo-random number generator. Consequently it can be used, e.g., for variance reduction techniques, it is easy to sample from truncated distributions, from marginal distributions, and from order statistics. Moreover, the quality of the generated random variables depends only on the underlying uniform (pseudo-) random number generator. Another important advantage of the inversion method is that we can easily characterize its performance. To generate one random variate we always need exactly one uniform variate and one evaluation of the inverse CDF. So its speed

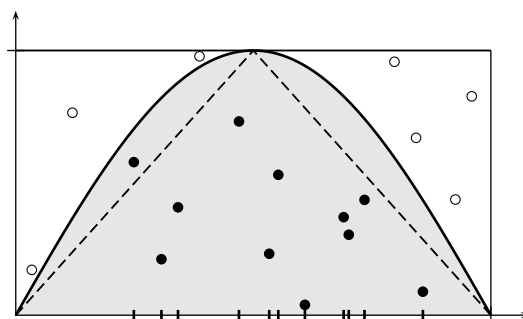
mainly depends on the costs for evaluating the inverse CDF. Hence inversion is often considered as the method of choice in the simulation literature.

Unfortunately computing the inverse CDF is, for many important standard distributions (e.g. for normal, student, gamma, and beta-distributions), comparatively difficult and slow. Often no such routines are available in standard programming libraries. Then numerical methods for inverting the CDF are necessary, e.g. Newton's method. Such procedures, however, have the disadvantage that they may be slow or not exact, i.e. they compute approximate values. The methods NINV (see [Section 5.3.7 \[NINV\]](#), page 103) and HINV (see [Section 5.3.3 \[HINV\]](#), page 97) of UNURAN are numerical inversion methods. Both require the CDF of the desired distribution. As the CDF is quite complicated and slow for many distributions this implies either that the generation is very slow (NINV) or a very slow setup step and large tables are necessary (HINV). Sometimes the CDF of a distribution is not available and alternative methods like the rejection method (see [Section A.2 \[Rejection\]](#), page 214) must be used.

A.2 The Rejection Method

The rejection method, often called *acceptance-rejection method*, has been suggested by John von Neumann in 1951. Since then it has proven to be the most flexible and most efficient method to generate variates from continuous distributions.

We explain the rejection principle first for the density $f(x) = \sin(x)/2$ on the interval $(0, \pi)$. To generate random variates from this distribution we also can sample random points that are uniformly distributed in the region between the graph of $f(x)$ and the x -axis, i.e., the shaded region in the below figure.



In general this is not a trivial task but in this example we can easily use the rejection trick: Sample a random point (X, Y) uniformly in the bounding rectangle $(0, \pi) \times (0, 0.5)$. This is easy since each coordinate can be sampled independently from the respective uniform distributions $U(0, \pi)$ and $U(0, 0.5)$. Whenever the point falls into the shaded region below the graph (indicated by dots in the figure), i.e., when $Y < \sin(X)/2$, we accept it and return X as a random variate from the distribution with density $f(x)$. Otherwise we have to reject the point (indicated by small circles in the figure), and try again.

It is quite clear that this idea works for every distribution with a bounded density on a bounded domain. Moreover, we can use this procedure with any multiple of the density, i.e., with any positive bounded function with bounded integral and it is not necessary to know the integral of this function. So we use the term density in the sequel for any positive function with bounded integral.

From the figure we can conclude that the performance of a rejection algorithm depends heavily on the area of the enveloping rectangle. Moreover, the method does not work if the target distribution has infinite tails (or is unbounded). Hence non-rectangular shaped regions

for the envelopes are important and we have to solve the problem of sampling points uniformly from such domains. Looking again at the example above we notice that the x -coordinate of the random point (X, Y) was sampled by inversion from the uniform distribution on the domain of the given density. This motivates us to replace the density of the uniform distribution by the (multiple of a) density $h(x)$ of some other appropriate distribution. We only have to take care that it is chosen such that it is always an upper bound, i.e., $h(x) \geq f(x)$ for all x in the domain of the distribution. To generate the pair (X, Y) we generate X from the distribution with density proportional to $h(x)$ and Y uniformly between 0 and $h(X)$. The first step (generate X) is usually done by inversion (see [Section A.1 \[Inversion\]](#), page 213).

Thus the general rejection algorithm for a hat $h(x)$ with inverse CDF H^{-1} consists of the following steps:

1. Generate a $U(0, 1)$ random number U .
2. Set X to $H^{-1}(U)$.
3. Generate a $U(0, 1)$ random number V .
4. Set Y to $Vh(X)$.
5. If $Y \leq f(X)$ accept X as the random variate.
6. Else try again.

If the evaluation of the density $f(x)$ is expensive (i.e., time consuming) it is possible to use a simple lower bound of the density as so called *squeeze function* $s(x)$ (the triangular shaped function in the above figure is an example for such a squeeze). We can then accept X when $Y \leq s(X)$ and can thus often save the evaluation of the density.

We have seen so far that the rejection principle leads to short and simple generation algorithms. The main practical problem to apply the rejection algorithm is the search for a good fitting hat function and for squeezes. We do not discuss these topics here as they are the heart of the different automatic algorithms implemented in UNURAN. Information about the construction of hat and squeeze can therefore be found in the descriptions of the methods.

The performance characteristics of rejection algorithms mainly depend on the fit of the hat and the squeeze. It is not difficult to prove that:

- The expected number of trials to generate one variate is the ratio between the area below the hat and the area below the density.
- The expected number of evaluations of the density necessary to generate one variate is equal to the ratio between the area below the hat and the area below the density, when no squeeze is used. Otherwise, when a squeeze is given it is equal to the ratio between the area between hat and squeeze and the area below the hat.
- The `sqhratio` (i.e., the ratio between the area below the squeeze and the area below the hat) used in some of the UNURAN methods is easy to compute. It is useful as its reciprocal is an upper bound for the expected number of trials of the rejection algorithm. The expected number of evaluations of the density is bounded by $(1/\text{sqhratio}) - 1$.

A.3 The Composition Method

The composition method is an important principle to facilitate and speed up random variate generation. The basic idea is simple. To generate random variates with a given density we first split the domain of the density into subintervals. Then we select one of these randomly with probabilities given by the area below the density in the respective subintervals. Finally we generate a random variate from the density of the selected part by inversion and return it as random variate of the full distribution.

Composition can be combined with rejection. Thus it is possible to decompose the domain of the distribution into subintervals and to construct hat and squeeze functions separately in

every subinterval. The area below the hat must be determined in every subinterval. Then the Composition rejection algorithm contains the following steps:

1. Generate the index J of the subinterval as the realisation of a discrete random variate with probabilities proportional to the area below the hat.
2. Generate a random variate X proportional to the hat in interval J .
3. Generate the $U(0, f(X))$ random number Y .
4. If $Y \leq f(X)$ accept X as random variate.
5. Else start again with generating the index J .

The first step can be done in constant time (i.e., independent of the number of chosen subintervals) by means of the indexed search method (see [Section A.6 \[IndexedSearch\]](#), page 218).

It is possible to reduce the number of uniform random numbers required in the above algorithm by recycling the random numbers used in Step 1 and additionally by applying the principle of *immediate acceptance*. For details see [HLD04: Sect. 3.1] .

A.4 The Ratio-of-Uniforms Method

The construction of an appropriate hat function for the given density is the crucial step for constructing rejection algorithms. Equivalently we can try to find an appropriate envelope for the region between the graph of the density and the x -axis, such that we can easily sample uniformly distributed random points. This task could become easier if we can find transformations that map the region between the density and the axis into a region of more suitable shape (for example into a bounded region).

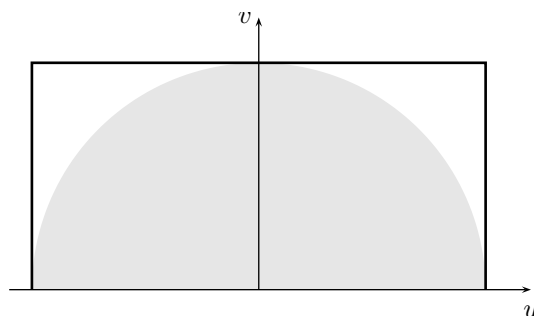
As a first example we consider the following simple algorithm for the Cauchy distribution.

1. Generate a $U(-1, 1)$ random number U and a $U(0, 1)$ random number V .
2. If $U^2 + V^2 \leq 1$ accept $X = U/V$ as a Cauchy random variate.
3. Else try again.

It is possible to prove that the above algorithm indeed generates Cauchy random variates. The fundamental principle behind this algorithm is the fact that the region below the density is mapped by the transformation

$$(X, Y) \mapsto (U, V) = (2X\sqrt{Y}, 2\sqrt{Y})$$

into a half-disc in such a way that the ratio between the area of the image to the area of the preimage is constant. This is due to the fact that the Jacobian of this transformation is constant.



The above example is a special case of a more general principle, called the *Ratio-of-uniforms (RoU) method*. It is based on the fact that for a random variable X with density $f(x)$ and some

constant μ we can generate X from the desired density by calculating $X = U/V + \mu$ for a pair (U, V) uniformly distributed in the set

$$A_f = \{(u, v): 0 < v \leq \sqrt{f(u/v + \mu)}\}.$$

For most distributions it is best to set the constant μ equal to the mode of the distribution. For sampling random points uniformly distributed in A_f rejection from a convenient enveloping region is used, usually the minimal bounding rectangle, i.e., the smallest possible rectangle that contains A_f (see the above figure). It is given by $(u^-, u^+) \times (0, v^+)$ where

$$\begin{aligned} v^+ &= \sup_{b_l < x < b_r} \sqrt{f(x)}, \\ u^- &= \inf_{b_l < x < b_r} (x - \mu) \sqrt{f(x)}, \\ u^+ &= \sup_{b_l < x < b_r} (x - \mu) \sqrt{f(x)}. \end{aligned}$$

Then the ratio-of-uniforms method consists of the following simple steps:

1. Generate a $U(u^-, u^+)$ random number U and a $U(0, v^+)$ random number V .
2. Set X to $U/V + \mu$.
3. If $V^2 \leq f(X)$ accept X as the random variate.
4. Else try again.

To apply the ratio-of-uniforms algorithm to a certain density we have to solve the simple optimization problems in the definitions above to obtain the design constants u^- , u^+ , and v^+ . This simple algorithm works for all distributions with bounded densities that have subquadratic tails (i.e., tails like $1/x^2$ or lower). For most standard distributions it has quite good rejection constants. (E.g. 1.3688 for the normal and 1.4715 for the exponential distribution.)

Nevertheless, we use more sophisticated method that construct better fitting envelopes, like method AROU (see [Section 5.3.1 \[AROU\]](#), page 92), or even avoid the computation of these design constants and thus have almost no setup, like method SROU (see [Section 5.3.9 \[SROU\]](#), page 108).

The Generalized Ratio-of-Uniforms Method

The Ratio-of-Uniforms method can be generalized: If a point (U, V) is uniformly distributed in the set

$$A_f = \{(u, v): 0 < v \leq (f(u/v^r + \mu))^{1/(r+1)}\}$$

then $X = U/V^r + \mu$ has the density $f(x)$. The minimal bounding rectangle of this region is given by $(u^-, u^+) \times (0, v^+)$ where

$$\begin{aligned} v^+ &= \sup_{b_l < x < b_r} (f(x))^{1/(r+1)}, \\ u^- &= \inf_{b_l < x < b_r} (x - \mu)(f(x))^{r/(r+1)}, \\ u^+ &= \sup_{b_l < x < b_r} (x - \mu)(f(x))^{r/(r+1)}. \end{aligned}$$

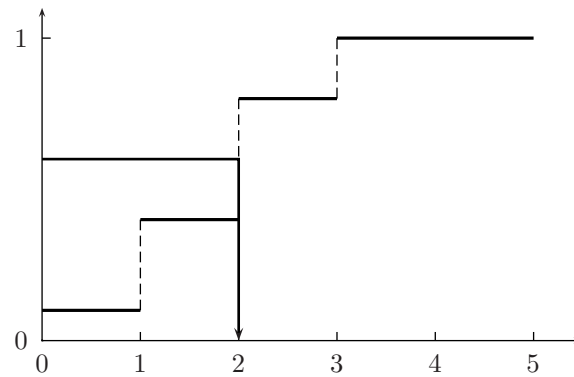
The above algorithm has then to be adjusted accordingly. Notice that the original Ratio-of-Uniforms method is the special case with $r = 1$.

A.5 Inversion for Discrete Distributions

We have already presented the idea of the inversion method to generate from continuous random variables (see [Section A.1 \[Inversion\]](#), page 213). For a discrete random variable X we can write it mathematically in the same way:

$$X = F^{-1}(U),$$

where F is the CDF of the desired distribution and U is a uniform $U(0, 1)$ random number. The difference compared to the continuous case is that F is now a step-function. The following figure illustrates the idea of discrete inversion for a simple distribution.



To realize this idea on a computer we have to use a search algorithm. For the simplest version called *Sequential Search* the CDF is computed on-the-fly as sum of the probabilities $p(k)$, since this is usually much cheaper than computing the CDF directly. It is obvious that the basic form of the search algorithm only works for discrete random variables with probability mass functions $p(k)$ for nonnegative k . The sequential search algorithm consists of the following basic steps:

1. Generate a $U(0,1)$ random number U .
2. Set X to 0 and P to $p(0)$.
3. Do while $U > P$
4. Set X to $X + 1$ and P to $P + p(X)$.
5. Return X .

With the exception of some very simple discrete distributions, sequential search algorithms become very slow as the while-loop has to be repeated very often. The expected number of iterations, i.e., the number of comparisons in the while condition, is equal to the expectation of the distribution plus 1. It can therefore become arbitrary large or even infinity if the tail of the distribution is very heavy. Another serious problem can be critical round-off errors due to summing up many probabilities $p(k)$. To speed up the search procedure it is best to use indexed search.

A.6 Indexed Search (Guide Table Method)

The idea to speed up the sequential search algorithm is easy to understand. Instead of starting always at 0 we store a table of size C with starting points for our search. For this table we compute $F^{-1}(U)$ for C equidistributed values of U , i.e., for $u_i = i/C$, $i = 0, \dots, C - 1$. Such a table is called *guide table* or *hash table*. Then it is easy to prove that for every U in $(0,1)$ the guide table entry for $k = \text{floor}(UC)$ is bounded by $F^{-1}(U)$. This shows that we can really start our sequential search procedure from the table entry for k and the index k of the correct table entry can be found rapidly by means of the truncation operation.

The two main differences between *indexed search* and *sequential search* are that we start searching at the number determined by the guide table, and that we have to compute and store the cumulative probabilities in the setup as we have to know the cumulative probability for the starting point of the search algorithm. The rounding problems that can occur in the sequential search algorithm can occur here as well. Compared to sequential search we have now the obvious drawback of a slow setup. The computation of the cumulative probabilities grows linear with the size of the domain of the distribution L . What we gain is really high speed as the marginal execution time of the sampling algorithm becomes very small. The expected number of comparisons is bounded by $1 + L/C$. This shows that there is a trade-off between speed and the size of the guide table. Cache-effects in modern computers will however slow down the speed-up for really large table sizes. Thus we recommend to use a guide table that is about two times larger than the probability vector to obtain optimal speed.

Appendix B Glossary

CDF	cumulative distribution function
HR	hazard rate (or failure rate)
PDF	probability density function
dPDF	derivative (gradient) of probability density function
PMF	probability mass function
PV	(finite) probability vector
URNG	uniform random number generator
$U(a, b)$	continuous uniform distribution on the interval (a, b)
T-concave	a function $f(x)$ is called T -convave if the transformed function $T(f(x))$ is concave. We only deal with transformations T_c , where
$c = 0$	$T(x) = \log(x)$
$c = -0.5$	$T(x) = -1/\sqrt{x}$
$c \neq 0$	$T(x) = \text{sign}(x) \cdot x^c$

Appendix C Bibliography

Standard Distributions

- [JKKa92] N.L. JOHNSON, S. KOTZ, AND A.W. KEMP (1992). *Univariate Discrete Distributions*, 2nd edition, John Wiley & Sons, Inc., New York.
- [JKBb94] N.L. JOHNSON, S. KOTZ, AND N. BALAKRISHNAN (1994). *Continuous Univariate Distributions*, Volume 1, 2nd edition, John Wiley & Sons, Inc., New York.
- [JKBc95] N.L. JOHNSON, S. KOTZ, AND N. BALAKRISHNAN (1995). *Continuous Univariate Distributions*, Volume 2, 2nd edition, John Wiley & Sons, Inc., New York.
- [JKBd97] N.L. JOHNSON, S. KOTZ, AND N. BALAKRISHNAN (1997). *Discrete Multivariate Distributions*, John Wiley & Sons, Inc., New York.
- [KBJe00] S. KOTZ, N. BALAKRISHNAN, AND N.L. JOHNSON (2000). *Continuous Multivariate Distributions*, Volume 1: Models and Applications, John Wiley & Sons, Inc., New York.

Universal Methods – Surveys

- [HLD04] W. HÖRMANN, J. LEYDOLD, AND G. DERFLINGER (2004). *Automatic Nonuniform Random Variate Generation*, Springer, Berlin.

Universal Methods

- [AJa93] J.H. AHRENS (1993). *Sampling from general distributions by suboptimal division of domains*, Grazer Math. Berichte 319, 30pp.
- [AJa95] J.H. AHRENS (1995). *An one-table method for sampling from continuous and discrete distributions*, Computing 54(2), pp. 127-146.
- [CAa74] H.C. CHEN AND Y. ASAU (1974). *On generating random variates from an empirical distribution*, AIIE Trans. 6, pp. 163-166.
- [DLa86] L. DEVROYE (1986). *Non-Uniform Random Variate Generation*, Springer Verlag, New York.
- [GWa92] W.R. GILKS AND P. WILD (1992). *Adaptive rejection sampling for Gibbs sampling*, Applied Statistics 41, pp. 337-348.
- [Hwa95] W. HÖRMANN (1995). *A rejection technique for sampling from T-concave distributions*, ACM Trans. Math. Software 21(2), pp. 182-193.
- [HDa96] W. HÖRMANN AND G. DERFLINGER (1996). *Rejection-inversion to generate variates from monotone discrete distributions*, ACM TOMACS 6(3), 169-184.
- [HLa00] W. HÖRMANN AND J. LEYDOLD (2000). *Automatic random variate generation for simulation input*. In: J.A. Joines, R. Barton, P. Fishwick, K. Kang (eds.), Proceedings of the 2000 Winter Simulation Conference, pp. 675-682.
- [HLa03] W. HÖRMANN AND J. LEYDOLD (2003). *Continuous Random Variate Generation by Fast Numerical Inversion*, ACM TOMACS 13(4), 347-362.
- [LJa00] J. LEYDOLD (2000). *Automatic Sampling with the Ratio-of-Uniforms Method*, ACM Trans. Math. Software 26(1), pp. 78-98.
- [LJa01] J. LEYDOLD (2001). *A simple universal generator for continuous and discrete univariate T-concave distributions*, ACM Trans. Math. Software 27(1), pp. 66-82.

- [LJa02] J. LEYDOLD (2003). *Short universal generators via generalized ratio-of-uniforms method*, Math. Comp. 72(243), pp. 1453-1471.
- [WGS91] J.C. WAKEFIELD, A.E. GELFAND, AND A.F.M. SMITH (1992). *Efficient generation of random variates via the ratio-of-uniforms method*, Statist. Comput. 1(2), pp. 129-133.
- [WAa77] A.J. WALKER (1977). *An efficient method for generating discrete random variables with general distributions*, ACM Trans. Math. Software 3, pp. 253-256.

Special Generators

- [ADa74] J.H. AHRENS, U. DIETER (1974). *Computer methods for sampling from gamma, beta, Poisson and binomial distributions*, Computing 12, 223-246.
- [ADa82] J.H. AHRENS, U. DIETER (1982). *Generating gamma variates by a modified rejection technique*, Communications of the ACM 25, 47-54.
- [ADb82] J.H. AHRENS, U. DIETER (1982). *Computer generation of Poisson deviates from modified normal distributions*, ACM Trans. Math. Software 8, 163-179.
- [BMa58] G.E.P. BOX AND M.E. MULLER (1958). *A note on the generation of random normal deviates*, Annals Math. Statist. 29, 610-611.
- [CHa77] R.C.H. CHENG (1977). *The Generation of Gamma Variables with Non-Integral Shape Parameter*, Appl. Statist. 26(1), 71-75.
- [HDa90] W. HÖRMANN AND G. DERFLINGER (1990). *The ACR Method for generating normal random variables*, OR Spektrum 12, 181-185.
- [KAa81] A.W. KEMP (1981). *Efficient generation of logarithmically distributed pseudo-random variables*, Appl. Statist. 30, 249-253.
- [KRa76] A.J. KINDERMAN AND J.G. RAMAGE (1976). *Computer Generation of Normal Random Variables*, J. Am. Stat. Assoc. 71(356), 893 - 898.
- [MJa87] J.F. MONAHAN (1987). *An algorithm for generating chi random variables*, ACM Trans. Math. Software 13, 168-172.
- [MGa62] G. MARSAGLIA (1962). *Improving the Polar Method for Generating a Pair of Random Variables*, Boeing Sci. Res. Lab., Seattle, Washington.
- [M0a84] G. MARSAGLIA AND I. OLKIN (1984). *Generating Correlation Matrices*, SIAM J. Sci. Stat. Comput 5, 470-475.
- [STa89] E. STADLOBER (1989). *Sampling from Poisson, binomial and hypergeometric distributions: ratio of uniforms as a simple and fast alternative*, Bericht 303, Math. Stat. Sektion, Forschungsgesellschaft Joanneum, Graz.
- [ZHa94] H. ZECHNER (1994). *Efficient sampling from continuous and discrete unimodal distributions*, Pd.D. Thesis, 156 pp., Technical University Graz, Austria.

Other references

- [CPa76] R. CRANLEY AND T.N.L. PATTERSON (1976). *Randomization of number theoretic methods for multiple integration*, SIAM J. Num. Anal., Vol. 13, pp. 904-914.
- [HJa61] R. HOOKE AND T.A. JEEVES (1961). *Direct Search Solution of Numerical and Statistical Problems*, Journal of the ACM, Vol. 8, April 1961, pp. 212-229.

Appendix D Function Index

F

FALSE 211

I

INT_MAX 211

INT_MIN 211

T

TRUE 211

U

unur_arou_chg_verify 93
 unsur_arou_get_hatarea 93
 unsur_arou_get_sqhratio 93
 unsur_arou_get_squeezearea 93
 unsur_arou_new 92
 unsur_arou_set_center 93
 unsur_arou_set_cpoints 93
 unsur_arou_set_darsfactor 92
 unsur_arou_set_guidefactor 93
 unsur_arou_set_max_segments 93
 unsur_arou_set_max_sqhratio 92
 unsur_arou_set_pedantic 94
 unsur_arou_set_usecenter 93
 unsur_arou_set_usedars 92
 unsur_arou_set_verify 93
 unsur_auto_new 87
 unsur_auto_set_logss 87
 unsur_ball_new 139
 unsur_ball_set_adaptive_ball 140
 unsur_ball_set_adaptive_factor 140
 unsur_ball_set_ball_radius 139
 unsur_ball_set_r 139
 unsur_ball_set_thinning 140
 unsur_ball_set_variant_pdf 139
 unsur_ball_set_variant_rou 139
 unsur_chg_debug 205
 unsur_chg_urng 179
 unsur_chg_urng_aux 179
 unsur_chgto_urng_aux_default 179
 unsur_cstd_chg_pdfparams 96
 unsur_cstd_chg_truncated 96
 unsur_cstd_new 95
 unsur_cstd_set_variant 95
 unsur_dari_chg_domain 163
 unsur_dari_chg_mode 163
 unsur_dari_chg_pmfparams 163
 unsur_dari_chg_pmfsum 164
 unsur_dari_chg_verify 163
 unsur_dari_new 162
 unsur_dari_reinit 162
 unsur_dari_set_cpfactor 163
 unsur_dari_set_squeeze 162
 unsur_dari_set_tablesize 163
 unsur_dari_set_verify 163
 unsur_dari_upd_mode 163
 unsur_dari_upd_pmfsum 164

unur_dau_new 165
 unsur_dau_set_urnfactor 165
 UNUR_DEBUG_ADAPT 205
 UNUR_DEBUG_ALL 205
 UNUR_DEBUG_INIT 205
 UNUR_DEBUG_OFF 205
 UNUR_DEBUG_SAMPLE 205
 UNUR_DEBUG_SETUP 205
 unsur_dgt_new 166
 unsur_dgt_set_guidefactor 166
 unsur_dgt_set_variant 167
 unsur_distr_<dname> 187
 unsur_distr_beta 189
 unsur_distr_binomial 197
 unsur_distr_cauchy 189
 unsur_distr_cemp_get_data 67
 unsur_distr_cemp_new 67
 unsur_distr_cemp_read_data 67
 unsur_distr_cemp_set_data 67
 unsur_distr_chi 189
 unsur_distr_chisquare 190
 unsur_distr_condi_get_condition 77
 unsur_distr_condi_get_distribution 77
 unsur_distr_condi_new 76
 unsur_distr_condi_set_condition 76
 unsur_distr_cont_eval_cdf 58
 unsur_distr_cont_eval_dlogpdf 59
 unsur_distr_cont_eval_dpdpf 58
 unsur_distr_cont_eval_hr 61
 unsur_distr_cont_eval_logpdf 59
 unsur_distr_cont_eval_pdf 58
 unsur_distr_cont_get_cdf 58
 unsur_distr_cont_get_cdfstr 59
 unsur_distr_cont_get_center 62
 unsur_distr_cont_get_dlogpdf 59
 unsur_distr_cont_get_dlogpdfstr 60
 unsur_distr_cont_get_domain 60
 unsur_distr_cont_get_dpdpf 58
 unsur_distr_cont_get_dpdpfstr 59
 unsur_distr_cont_get_hr 61
 unsur_distr_cont_get_hrstr 61
 unsur_distr_cont_get_logpdf 59
 unsur_distr_cont_get_logpdfstr 60
 unsur_distr_cont_get_mode 62
 unsur_distr_cont_get_pdf 58
 unsur_distr_cont_get_pdfarea 63
 unsur_distr_cont_get_pdfparams 60
 unsur_distr_cont_get_pdfparams_vec 60
 unsur_distr_cont_get_pdfstr 59
 unsur_distr_cont_get_truncated 61
 unsur_distr_cont_new 57
 unsur_distr_cont_set_cdf 57
 unsur_distr_cont_set_cdfstr 59
 unsur_distr_cont_set_center 62
 unsur_distr_cont_set_dlogpdf 59
 unsur_distr_cont_set_domain 60
 unsur_distr_cont_set_dpdpf 57
 unsur_distr_cont_set_hr 61
 unsur_distr_cont_set_hrstr 61
 unsur_distr_cont_set_logpdf 59
 unsur_distr_cont_set_logpdfstr 60

unur_distr_cont_set_mode	62	unur_distr_cvec_set_mode	74
unur_distr_cont_set_pdf	57	unur_distr_cvec_set_pdf	68
unur_distr_cont_set_pdfarea	62	unur_distr_cvec_set_pdfparams	73
unur_distr_cont_set_pdfparams	59	unur_distr_cvec_set_pdfparams_vec	73
unur_distr_cont_set_pdfparams_vec	60	unur_distr_cvec_set_pdfvol	75
unur_distr_cont_set_pdfstr	59	unur_distr_cvec_set_pdlogpdf	70
unur_distr_cont_upd_mode	62	unur_distr_cvec_set_pdpdf	69
unur_distr_cont_upd_pdfarea	62	unur_distr_cvec_set_rankcorr	72
unur_distr_copula	195	unur_distr_cvec_set_stdmarginal_array	72
unur_distr_corder_eval_cdf	64	unur_distr_cvec_set_stdmarginal_list	72
unur_distr_corder_eval_dpdf	64	unur_distr_cvec_set_stdmarginals	72
unur_distr_corder_eval_pdf	64	unur_distr_cvemp_get_data	78
unur_distr_corder_get_cdf	64	unur_distr_cvemp_new	78
unur_distr_corder_get_distribution	64	unur_distr_cvemp_read_data	78
unur_distr_corder_get_domain	65	unur_distr_cvemp_set_data	78
unur_distr_corder_get_dpdf	64	unur_distr_discr_eval_cdf	81
unur_distr_corder_get_mode	65	unur_distr_discr_eval_pmf	81
unur_distr_corder_get_pdf	64	unur_distr_discr_eval_pv	81
unur_distr_corder_get_pdfarea	66	unur_distr_discr_get_cdfstr	82
unur_distr_corder_get_pdfparams	65	unur_distr_discr_get_domain	82
unur_distr_corder_get_rank	64	unur_distr_discr_get_mode	83
unur_distr_corder_get_truncated	65	unur_distr_discr_get_pmfparams	82
unur_distr_corder_new	64	unur_distr_discr_get_pmfstr	82
unur_distr_corder_set_domain	65	unur_distr_discr_get_pmfsum	83
unur_distr_corder_set_mode	65	unur_distr_discr_get_pv	81
unur_distr_corder_set_pdfarea	65	unur_distr_discr_make_pv	80
unur_distr_corder_set_pdfparams	65	unur_distr_discr_new	80
unur_distr_corder_set_rank	64	unur_distr_discr_set_cdf	81
unur_distr_corder_set_upd_mode	65	unur_distr_discr_set_cdfstr	82
unur_distr_corder_upd_pdfarea	66	unur_distr_discr_set_domain	82
unur_distr_correlation	199	unur_distr_discr_set_mode	83
unur_distr_cvec_eval_dlogpdf	70	unur_distr_discr_set_pmf	81
unur_distr_cvec_eval_dpdf	70	unur_distr_discr_set_pmfparams	82
unur_distr_cvec_eval_logpdf	70	unur_distr_discr_set_pmfstr	81
unur_distr_cvec_eval_pdf	69	unur_distr_discr_set_pmfsum	83
unur_distr_cvec_eval_pdlogpdf	70	unur_distr_discr_set_pv	80
unur_distr_cvec_eval_pdpdf	70	unur_distr_discr_upd_mode	83
unur_distr_cvec_get_center	74	unur_distr_discr_upd_pmfsum	83
unur_distr_cvec_get_cholesky	71	unur_distr_exponential	190
unur_distr_cvec_get_covar	71	unur_distr_extremeI	190
unur_distr_cvec_get_covar_inv	71	unur_distr_extremeII	190
unur_distr_cvec_get_dlogpdf	70	unur_distr_free	55
unur_distr_cvec_get_dpdf	69	unur_distr_gamma	191
unur_distr_cvec_get_logpdf	70	unur_distr_geometric	197
unur_distr_cvec_get_marginal	73	unur_distr_get_dim	55
unur_distr_cvec_get_mean	70	unur_distr_get_name	55
unur_distr_cvec_get_mode	74	unur_distr_get_type	55
unur_distr_cvec_get_pdf	69	unur_distr_hypergeometric	197
unur_distr_cvec_get_pdfparams	73	unur_distr_is_cemp	56
unur_distr_cvec_get_pdfparams_vec	74	unur_distr_is_cont	55
unur_distr_cvec_get_pdfvol	75	unur_distr_is_cvec	55
unur_distr_cvec_get_rankcorr	72	unur_distr_is_cvemp	56
unur_distr_cvec_get_rk_cholesky	72	unur_distr_is_discr	56
unur_distr_cvec_get_stdmarginal	73	unur_distr_is_matr	56
unur_distr_cvec_new	68	unur_distr_laplace	191
unur_distr_cvec_set_center	74	unur_distr_logarithmic	198
unur_distr_cvec_set_covar	71	unur_distr_logistic	191
unur_distr_cvec_set_covar_inv	71	unur_distr_lomax	192
unur_distr_cvec_set_dlogpdf	70	unur_distr_matr_get_dim	79
unur_distr_cvec_set_dpdf	69	unur_distr_matr_new	79
unur_distr_cvec_set_logpdf	70	unur_distr_multicauchy	195
unur_distr_cvec_set_marginal_array	72	unur_distr_multiexponential	195
unur_distr_cvec_set_marginal_list	72	unur_distr_multinormal	195
unur_distr_cvec_set_marginals	72	unur_distr_multistudent	195
unur_distr_cvec_set_mean	70	unur_distr_negativebinomial	198

unur_distr_normal	192	UNUR_ERR_SHOULD_NOT_HAPPEN	203
unur_distr_pareto	193	UNUR_ERR_SILENT	203
unur_distr_poisson	198	UNUR_ERR_STR	203
unur_distr_powerexponential	193	UNUR_ERR_STR_INVALID	203
unur_distr_rayleigh	193	UNUR_ERR_STR_SYNTAX	203
unur_distr_set_name	55	UNUR_ERR_STR_UNKNOWN	203
unur_distr_student	193	UNUR_ERR_URNG	202
unur_distr_triangular	194	UNUR_ERR_URNG_MISS	203
unur_distr_uniform	194	unur_errno	203
unur_distr_weibull	194	unur_free	85
unur_dsrou_chg_cdfatmode	169	unur_gen_anti	181
unur_dsrou_chg_domain	169	unur_gen_nextsub	181
unur_dsrou_chg_mode	169	unur_gen_reset	181
unur_dsrou_chg_pmparams	169	unur_gen_resetsub	181
unur_dsrou_chg_pmfsun	169	unur_gen_seed	181
unur_dsrou_chg_verify	169	unur_gen_sync	181
unur_dsrou_new	168	unur_get_default_urng	178
unur_dsrou_reinit	168	unur_get_default_urng_aux	179
unur_dsrou_set_cdfatmode	168	unur_get_dimension	85
unur_dsrou_set_verify	169	unur_get_distr	85
unur_dsrou_upd_mode	169	unur_get_genid	85
unur_dsrou_upd_pmfsun	169	unur_get_stream	204
unur_dss_new	170	unur_get_strerror	204
unur_dstd_chg_pmparams	172	unur_get_urng	179
unur_dstd_new	171	unur_get_urng_aux	179
unur_dstd_set_variant	171	unur_gibbs_chg_state	150
unur_empk_chg_smoothing	136	unur_gibbs_get_state	150
unur_empk_chg_varcor	136	unur_gibbs_new	149
unur_empk_new	135	unur_gibbs_reset_state	150
unur_empk_set_beta	136	unur_gibbs_set_burnin	150
unur_empk_set_kernel	135	unur_gibbs_set_c	149
unur_empk_set_kernelgen	135	unur_gibbs_set_startingpoint	149
unur_empk_set_positive	136	unur_gibbs_set_thinning	149
unur_empk_set_smoothing	136	unur_gibbs_set_variant_coordinate	149
unur_empk_set_varcor	136	unur_gibbs_set_variant_random_direction	149
unur_empl_new	137	unur_hinv_chg_truncated	99
UNUR_ERR_COMPILE	203	unur_hinv_estimate_error	99
UNUR_ERR_COOKIE	203	unur_hinv_eval_approxinvcdf	99
UNUR_ERR_DISTR_DATA	202	unur_hinv_get_n_intervals	99
UNUR_ERR_DISTR_DOMAIN	202	unur_hinv_new	98
UNUR_ERR_DISTR_GEN	202	unur_hinv_set_boundary	98
UNUR_ERR_DISTR_GET	202	unur_hinv_set_cpoints	98
UNUR_ERR_DISTR_INVALID	202	unur_hinv_set_guidefactor	99
UNUR_ERR_DISTR_NPARAMS	202	unur_hinv_set_max_intervals	99
UNUR_ERR_DISTR_PROP	202	unur_hinv_set_order	98
UNUR_ERR_DISTR_REQUIRED	202	unur_hinv_set_u_resolution	98
UNUR_ERR_DISTR_SET	202	unur_hitro_chg_state	155
UNUR_ERR_DISTR_UNKNOWN	202	unur_hitro_get_state	155
UNUR_ERR_DOMAIN	203	unur_hitro_new	153
UNUR_ERR_FSTR_DERIV	203	unur_hitro_reset_state	155
UNUR_ERR_FSTR_SYNTAX	203	unur_hitro_set_adaptive_multiplier	154
UNUR_ERR_GEN	202	unur_hitro_set_burnin	155
UNUR_ERR_GEN_CONDITION	202	unur_hitro_set_r	154
UNUR_ERR_GEN_DATA	202	unur_hitro_set_startingpoint	155
UNUR_ERR_GEN_INVALID	202	unur_hitro_set_thinning	155
UNUR_ERR_GEN_SAMPLING	202	unur_hitro_set_u	154
UNUR_ERR_GENERIC	203	unur_hitro_set_use_adaptiveline	153
UNUR_ERR_INF	203	unur_hitro_set_use_adaptiverectangle	153
UNUR_ERR_MALLOC	203	unur_hitro_set_use_boundingrectangle	153
UNUR_ERR_NAN	203	unur_hitro_set_v	154
UNUR_ERR_NULL	203	unur_hitro_set_variant_coordinate	153
UNUR_ERR_PAR_INVALID	202	unur_hitro_set_variant_random_direction	153
UNUR_ERR_PAR_SET	202	unur_hrb_chg_verify	100
UNUR_ERR_PAR_VARIANT	202	unur_hrb_new	100
UNUR_ERR_ROUNDOff	203	unur_hrb_set_upperbound	100

unur_hrb_set_verify	100	unur_ssr_chg_domain	113
unur_hrd_chg_verify	101	unur_ssr_chg_mode	114
unur_hrd_new	101	unur_ssr_chg_pdfarea	114
unur_hrd_set_verify	101	unur_ssr_chg_pdfatmode	114
unur_hri_chg_verify	102	unur_ssr_chg_pdfparams	113
unur_hri_new	102	unur_ssr_chg_verify	113
unur_hri_set_p0	102	unur_ssr_new	112
unur_hri_set_verify	102	unur_ssr_reinit	113
UNUR_INFINITY	211	unur_ssr_set_cdfatmode	113
unur_init	85	unur_ssr_set_pdfatmode	113
unur_mcorr_new	174	unur_ssr_set_usesqueeze	113
unur_mcorr_set_eigenvalues	174	unur_ssr_set_verify	113
unur_ninv_chg_max_iter	104	unur_ssr_upd_mode	114
unur_ninv_chg_pdfparams	105	unur_ssr_upd_pdfarea	114
unur_ninv_chg_start	104	unur_str2distr	37
unur_ninv_chg_table	104	unur_str2gen	37
unur_ninv_chg_truncated	105	UNUR_SUCCESS (0x0u)	201
unur_ninv_chg_x_resolution	104	unur_tabl_chg_truncated	118
unur_ninv_eval_approxinvcdf	105	unur_tabl_chg_verify	119
unur_ninv_new	104	unur_tabl_get_hatarea	118
unur_ninv_set_max_iter	104	unur_tabl_get_n_intervals	118
unur_ninv_set_start	104	unur_tabl_get_sqhratio	118
unur_ninv_set_table	104	unur_tabl_get_squeezearea	118
unur_ninv_set_usenewton	104	unur_tabl_new	116
unur_ninv_set_useregula	104	unur_tabl_set_areafraction	117
unur_ninv_set_x_resolution	104	unur_tabl_set_boundary	118
unur_norta_new	141	unur_tabl_set_cpoints	116
unur_nrou_chg_verify	107	unur_tabl_set_darsfactor	117
unur_nrou_new	107	unur_tabl_set_guidefactor	118
unur_nrou_set_center	107	unur_tabl_set_max_intervals	118
unur_nrou_set_r	107	unur_tabl_set_max_sqhratio	117
unur_nrou_set_u	107	unur_tabl_set_nstp	116
unur_nrou_set_v	107	unur_tabl_set_pedantic	119
unur_nrou_set_verify	107	unur_tabl_set_slopes	118
unur_run_tests	207	unur_tabl_set_usedars	117
unur_sample_cont	85	unur_tabl_set_useear	116
unur_sample_discr	85	unur_tabl_set_variant_ia	116
unur_sample_matr	85	unur_tabl_set_variant_splitmode	117
unur_sample_vec	85	unur_tabl_set_verify	119
unur_set_debug	205	unur_tdr_chg_reinit_ncpoints	122
unur_set_default_debug	205	unur_tdr_chg_reinit_percentiles	122
unur_set_default_urng	178	unur_tdr_chg_truncated	122
unur_set_default_urng_aux	179	unur_tdr_chg_verify	124
unur_set_stream	204	unur_tdr_eval_invcdfhat	124
unur_set_urng	179	unur_tdr_get_hatarea	123
unur_set_urng_aux	179	unur_tdr_get_sqhratio	123
unur_set_use_distr_privatecopy	85	unur_tdr_get_squeezearea	123
unur_srou_chg_cdfatmode	110	unur_tdr_new	121
unur_srou_chg_domain	110	unur_tdr_reinit	121
unur_srou_chg_mode	110	unur_tdr_set_c	121
unur_srou_chg_pdfarea	111	unur_tdr_set_center	123
unur_srou_chg_pdfatmode	111	unur_tdr_set_cpoints	122
unur_srou_chg_pdfparams	110	unur_tdr_set_darsfactor	122
unur_srou_chg_verify	110	unur_tdr_set_guidefactor	124
unur_srou_new	109	unur_tdr_set_max_intervals	123
unur_srou_reinit	109	unur_tdr_set_max_sqhratio	123
unur_srou_set_cdfatmode	109	unur_tdr_set_pedantic	124
unur_srou_set_pdfatmode	109	unur_tdr_set_reinit_ncpoints	122
unur_srou_set_r	109	unur_tdr_set_reinit_percentiles	122
unur_srou_set_usemirror	110	unur_tdr_set_usecenter	123
unur_srou_set_usesqueeze	110	unur_tdr_set_usedars	121
unur_srou_set_verify	110	unur_tdr_set_usemode	124
unur_srou_upd_mode	110	unur_tdr_set_variant_gw	121
unur_srou_upd_pdfarea	111	unur_tdr_set_variant_ia	121
unur_ssr_chg_cdfatmode	114	unur_tdr_set_variant_ps	121

unur_tdr_set_verify	124	unur_urng_seed	180
unur_tdrwg_chg_reinit_ncpoints	126	unur_urng_set_anti	182
unur_tdrwg_chg_reinit_percentiles	126	unur_urng_set_delete	182
unur_tdrwg_chg_verify	126	unur_urng_set_nextsub	182
unur_tdrwg_eval_invcdfhat	127	unur_urng_set_reset	182
unur_tdrwg_get_loghatarea	127	unur_urng_set_resetsub	182
unur_tdrwg_new	125	unur_urng_set_sample_array	182
unur_tdrwg_reinit	125	unur_urng_set_seed	182
unur_tdrwg_set_cpoints	126	unur_urng_set_sync	182
unur_tdrwg_set_max_intervals	126	unur_urng_sync	180
unur_tdrwg_set_pedantic	127	unur_use_urng_aux_default	179
unur_tdrwg_set_reinit_ncpoints	126	unur_utdr_chg_domain	129
unur_tdrwg_set_reinit_percentiles	126	unur_utdr_chg_mode	129
unur_tdrwg_set_verify	126	unur_utdr_chg_pdfarea	129
unur_test_chi2	209	unur_utdr_chg_pdfatmode	129
unur_test_correlation	209	unur_utdr_chg_pdfparams	129
unur_test_count_pdf	208	unur_utdr_chg_verify	129
unur_test_count_urn	208	unur_utdr_new	128
unur_test_moments	209	unur_utdr_reinit	128
unur_test_par_count_pdf	208	unur_utdr_set_cpfactor	129
unur_test_printsample	207	unur_utdr_set_deltafactor	129
unur_test_quartiles	209	unur_utdr_set_pdfatmode	128
unur_test_timing	207	unur_utdr_set_verify	129
unur_test_timing_exponential	208	unur_utdr_upd_mode	129
unur_test_timing_total	208	unur_utdr_upd_pdfarea	130
unur_test_timing_uniform	208	unur_varou_chg_verify	142
unur_unif_new	176	unur_varou_new	142
unur_urng_anti	180	unur_varou_set_cones	142
unur_urng_free	181	unur_varou_set_verify	142
unur_urng_fvoid_new	183	unur_vempk_chg_smoothing	158
unur_urng_gsl_new	183	unur_vempk_chg_varcor	158
unur_urng_gslptr_new	183	unur_vempk_new	158
unur_urng_gslqrng_new	184	unur_vempk_set_smoothing	158
unur_urng_new	181	unur_vempk_set_varcor	158
unur_urng_nextsub	181	unur_vmt_new	143
unur_urng_prng_new	184	unur_vnrou_chg_verify	145
unur_urng_prngptr_new	185	unur_vnrou_get_volumehat	145
unur_urng_randomshift_new	186	unur_vnrou_new	145
unur_urng_randomshift_nextshift	186	unur_vnrou_set_r	145
unur_urng_reset	180	unur_vnrou_set_u	145
unur_urng_resetsub	181	unur_vnrou_set_v	145
unur_urng_rngstream_new	185	unur_vnrou_set_verify	145
unur_urng_rngstreamptr_new	185	unur_walk_new	146
unur_urng_sample	180	unur_walk_set_ball_radius	146
unur_urng_sample_array	180	unur_walk_set_thinning	146

