UNU.RAN User Manual

Generating non-uniform random numbers
Version 0.5.0, 6 August 2004

Josef Leydold
Wolfgang Hörmann
Erich Janka
Roman Karawatzki
Günter Tirler

Table of Contents

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 familar 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 consisent 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 choosen 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. Altough 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, discribes 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 43, describes how to create and manipulate distribution objects. Chapter 7 [standard distributions], page 121, describes predefined distribution objects that are ready to use. Chapter 5 [Methods], page 63 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 153.

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 117.

UNURAN

1. First unzip and untar the package and change to the directory:
   ```
   tar zxvf unuran-0.5.0.tar.gz
   cd unuran-0.5.0
   ```
2. Edit the file '`src/unuran_config.h`'. Set the appropriate source of uniform random numbers: `#define UNUR_URNG_TYPE` (see Chapter 6 [URNG], page 117 for details).

   Important: If neither `UNUR_URNG_FVOID` nor `UNUR_URNG_GENERIC` is used, check if the corresponding library is installed.
3. 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 *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 allmost all modern compting 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).

4. Compile and install the libray:

   ```
   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_INCLURE_PATH="<prefix>/include"
   ```

   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.

   If you do not want to make a shared library, than making such a library can be disabled using

   ```
   sh ./configure --disable-shared
   ```

5. Documentation in various formats (PS, PDF, info, dvi, HTML, plain text) can be found in the directory '`doc`'.

6. 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 UNURAN 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 defaut 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:

`cont`	Continuous univariate distributions.
`cvec`	Continuous multivariate distributions.
`discr`	Discrete univariate distributions.
`cemp`	Continuous empirical univariate distribution, ie. given by sample.
`cvemp`	Continuous empirical multivariate distribution, ie. given by sample.

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 43). 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 demonstated 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 fuctions `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 121.

Generation methods

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

Brackets indicate optional information while a tilde indicates that only an approximation must be provided. See Appendix B [Glossary], page 153, 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 = unur_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 = unur_<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 perforence 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 63.

Creating a generator object

Now it is possible to create a generator object:

```
UNUR_GEN *generator = unur_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 63.

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 `double`s). 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 117, 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 changed 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 choosen for sampling, see Section 1.4 [Concepts], page 5 for an overview and Chapter 5 [Methods], page 63 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_dpdf( 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 choosen for sampling, see Section 1.4 [Concepts], page 5 for an overview and Chapter 5 [Methods], page 63 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 63.

```
/* ---------------------------------------------------------------- */
/* 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 63.

```
/* ------------------------------------------------------------ */
/* 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 117, 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_PRNG                    */
/* in 'src/unuran_config.h'.                                  */

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

/* ---------------------------------------------------------- */

int main()
{
#if UNUR_URNG_TYPE == UNUR_URNG_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 = 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 Congruental Generator).         */
  urng2 = 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.*/
  prng_free(urng1);
  prng_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 117, 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_PRNG                    */
/* in 'src/unuran_config.h'.                                  */

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

/* ---------------------------------------------------------- */

int main()
{
#if UNUR_URNG_TYPE == UNUR_URNG_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 Congruental Generator).        */
  urng2 = 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.*/
  prng_free(urng1);
  prng_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_PRNG                       */
/* in 'src/unuran_config.h'.                                     */

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

/* ---------------------------------------------------------- */

int main()
{
#if UNUR_URNG_TYPE == UNUR_URNG_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 = 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 = 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 = 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.*/
prng_free(urng1);
prng_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_PRNG                    */
/* in 'src/unuran_config.h'.                                  */

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

/* ---------------------------------------------------------- */

int main()
{
#if UNUR_URNG_TYPE == UNUR_URNG_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.                                                 */
  prng_free(unur_get_urng(gen_normal));
  prng_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 64.
See Section 5.4 [Methods for continuous empirical univariate distributions], page 96.
See Section 5.6 [Methods for continuous empirical multivariate distributions], page 103.
See Section 5.7 [Methods for discrete univariate distributions], page 106.

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 133). 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 programm 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 29, 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 29, and Section 3.2 [Distribution String], page 31, 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:

`distr` definition of the distribution (see Section 3.2 [Distribution String], page 31).

`method` description of the transformation method (see Section 3.4 [Method String], page 35).

`urng` uniform random number generation (see Section 3.5 [Uniform RNG String], page 41).

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 31, and Section 3.4.1 [Keys for Method String], page 36, 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 `\"`.)

For the details of function strings see Section 3.3 [Function String], page 33.

3.2.1 Keys for Distribution String

List of standard distributions see Chapter 7 [Standard distributions], page 121

- `[distr =] beta(...)` ⇒ see Section 7.1.1 [beta], page 122
- `[distr =] binomial(...)` ⇒ see Section 7.3.1 [binomial], page 129
- `[distr =] cauchy(...)` ⇒ see Section 7.1.2 [cauchy], page 122
- `[distr =] chi(...)` ⇒ see Section 7.1.3 [chi], page 123
- `[distr =] chisquare(...)` ⇒ see Section 7.1.4 [chisquare], page 123
- `[distr =] exponential(...)` ⇒ see Section 7.1.5 [exponential], page 123
- `[distr =] extremeI(...)` ⇒ see Section 7.1.6 [extremeI], page 123
- `[distr =] extremeII(...)` ⇒ see Section 7.1.7 [extremeII], page 124
- `[distr =] gamma(...)` ⇒ see Section 7.1.8 [gamma], page 124
- `[distr =] geometric(...)` ⇒ see Section 7.3.2 [geometric], page 129
- `[distr =] hypergeometric(...)` ⇒ see Section 7.3.3 [hypergeometric], page 129
- `[distr =] laplace(...)` ⇒ see Section 7.1.9 [laplace], page 125
- `[distr =] logarithmic(...)` ⇒ see Section 7.3.4 [logarithmic], page 130
- `[distr =] logistic(...)` ⇒ see Section 7.1.10 [logistic], page 125
- `[distr =] lomax(...)` ⇒ see Section 7.1.11 [lomax], page 125
- `[distr =] negativebinomial(...)` ⇒ see Section 7.3.5 [negativebinomial], page 130
- `[distr =] normal(...)` ⇒ see Section 7.1.12 [normal], page 126

– `[distr =] pareto(...)` ⇒ see Section 7.1.13 [pareto], page 126
– `[distr =] poisson(...)` ⇒ see Section 7.3.6 [poisson], page 130
– `[distr =] powerexponential(...)` ⇒ see Section 7.1.14 [powerexponential], page 126
– `[distr =] rayleigh(...)` ⇒ see Section 7.1.15 [rayleigh], page 127
– `[distr =] student(...)` ⇒ see Section 7.1.16 [student], page 127
– `[distr =] triangular(...)` ⇒ see Section 7.1.17 [triangular], page 127
– `[distr =] uniform(...)` ⇒ see Section 7.1.18 [uniform], page 127
– `[distr =] weibull(...)` ⇒ see Section 7.1.19 [weibull], page 128

List of generic distributions see Chapter 4 [Handling Distribution Objects], page 43
– `[distr =] cemp` ⇒ see Section 4.4 [CEMP], page 51
– `[distr =] cont` ⇒ see Section 4.2 [CONT], page 44
– `[distr =] discr` ⇒ see Section 4.8 [DISCR], page 58

Notice: Order statistics for continuous distributions (see Section 4.3 [CORDER], page 49) 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 43

- `cemp` *(Distribution Type)* (see Section 4.4 [CEMP], page 51)

  `data = (<list>) [, <int>]`
  ⇒ see [unur_distr_cemp_set_data], page 51

- `cont` *(Distribution Type)* (see Section 4.2 [CONT], page 44)

  `cdf = "<string>"`
  ⇒ see [unur_distr_cont_set_cdfstr], page 46

  `domain = <double>, <double> | (<list>)`
  ⇒ see [unur_distr_cont_set_domain], page 46

  `hr = "<string>"`
  ⇒ see [unur_distr_cont_set_hrstr], page 48

  `mode = <double>`
  ⇒ see [unur_distr_cont_set_mode], page 48

  `pdf = "<string>"`
  ⇒ see [unur_distr_cont_set_pdfstr], page 46

  `pdfarea = <double>`
  ⇒ see [unur_distr_cont_set_pdfarea], page 48

  `pdfparams = (<list>) [, <int>]`
  ⇒ see [unur_distr_cont_set_pdfparams], page 46

`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 49)

- `discr` *(Distribution Type)* (see Section 4.8 [DISCR], page 58)

  `cdf = "<string>"`
  : ⇒ see [unur_distr_discr_set_cdfstr], page 59

  `domain = <int>, <int> | (<list>)`
  : ⇒ see [unur_distr_discr_set_domain], page 60

  `mode [= <int>]`
  : ⇒ see [unur_distr_discr_set_mode], page 60

  `pmf = "<string>"`
  : ⇒ see [unur_distr_discr_set_pmfstr], page 59

  `pmfparams = (<list>) [, <int>]`
  : ⇒ see [unur_distr_discr_set_pmfparams], page 60

  `pmfsum = <double>`
  : ⇒ see [unur_distr_discr_set_pmfsum], page 61

  `pv = (<list>) [, <int>]`
  : ⇒ see [unur_distr_discr_set_pv], page 58

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 17) 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 15). 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...) 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`	*digits*	`2343`
`.`	*decimal point*	`165.567`
`-`	*negative sign*	`-465.223`
`e`	*exponet*	`13.2e-4` (=0.00132)

Constants

`pi`	*pi = 3.1415...*	`3*pi+2`
`e`	*Euler's constant*	`3*e+2` (= 10.15...; do not cofuse with `3e2` = 300)
`inf`	*infinity*	(used for domains)

Special symbols

`(`	*opening parenthesis*	`2*(3+x)`
`)`	*closing parenthesis*	`2*(3+x)`
`,`	*(argument) list separator*	`mod(13,2)`

Relation operators (Indicator functions)

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

Arithmetic operators

`+`	*addition*	`2+x`
`-`	*subtraction*	`2-x`
`*`	*multiplication*	`2*x`
`/`	*division*	`x/2`
`^`	*power*	`x^2`

Functions

`mod`	`mod(m,n)` *remainder of devision m over n*	mod(x,2)
`exp`	*exponential function (same as* `e^x`*)*	`exp(-x^2)` (same as `e^(-x^2)`)
`log`	*natural logarithm*	`log(x)`
`sin`	*sine*	`sin(x)`
`cos`	*cosine*	`cos(x)`
`tan`	*tangent*	`tan(x)`
`sec`	*secant*	`sec(x*2)`
`sqrt`	*square root*	`sqrt(2*x)`
`abs`	*absolute value*	`abs(x)`
`sgn`	*sign function*	`sign(x)*3`

Variable

`x`	*variable*	`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 choosen standard distribution. E.g., if method AROU is chosen, use

```
method = arou
```

Of course the all following keys dependend on the method choosen 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 137, 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 67)

  `center = <double>`
  ⇒ see [unur_arou_set_center], page 69

  `cpoints = <int> [, (<list>)] | (<list>)`
  ⇒ see [unur_arou_set_cpoints], page 69

  `darsfactor = <double>`
  ⇒ see [unur_arou_set_darsfactor], page 68

  `guidefactor = <double>`
  ⇒ see [unur_arou_set_guidefactor], page 69

  `max_segments [= <int>]`
  ⇒ see [unur_arou_set_max_segments], page 69

  `max_sqhratio = <double>`
  ⇒ see [unur_arou_set_max_sqhratio], page 68

  `pedantic [= <int>]`
  ⇒ see [unur_arou_set_pedantic], page 69

  `usecenter [= <int>]`
  ⇒ see [unur_arou_set_usecenter], page 69

  `usedars [= <int>]`
  ⇒ see [unur_arou_set_usedars], page 68

  `verify [= <int>]`
  ⇒ see [unur_arou_set_verify], page 69

- `method = auto` ⇒ `unur_auto_new` (see Section 5.2 [AUTO], page 64)

  `logss [= <int>]`
  ⇒ see [unur_auto_set_logss], page 64

- `method = cstd` ⇒ `unur_cstd_new` (see Section 5.3.2 [CSTD], page 70)

  `variant = <unsigned>`
  ⇒ see [unur_cstd_set_variant], page 70

- `method = dari` ⇒ `unur_dari_new` (see Section 5.7.1 [DARI], page 108)

  `cpfactor = <double>`
  ⇒ see [unur_dari_set_cpfactor], page 109

`squeeze [= <int>]`
⇒ see [unur_dari_set_squeeze], page 109

`tablesize [= <int>]`
⇒ see [unur_dari_set_tablesize], page 109

`verify [= <int>]`
⇒ see [unur_dari_set_verify], page 109

- `method = dau` ⇒ `unur_dau_new` (see Section 5.7.2 [DAU], page 110)

  `urnfactor = <double>`
  ⇒ see [unur_dau_set_urnfactor], page 111

- `method = dgt` ⇒ `unur_dgt_new` (see Section 5.7.3 [DGT], page 111)

  `guidefactor = <double>`
  ⇒ see [unur_dgt_set_guidefactor], page 112

  `variant = <unsigned>`
  ⇒ see [unur_dgt_set_variant], page 112

- `method = dsrou` ⇒ `unur_dsrou_new` (see Section 5.7.4 [DSROU], page 112)

  `cdfatmode = <double>`
  ⇒ see [unur_dsrou_set_cdfatmode], page 113

  `verify [= <int>]`
  ⇒ see [unur_dsrou_set_verify], page 113

- `method = dstd` ⇒ `unur_dstd_new` (see Section 5.7.6 [DSTD], page 114)

  `variant = <unsigned>`
  ⇒ see [unur_dstd_set_variant], page 115

- `method = empk` ⇒ `unur_empk_new` (see Section 5.4.1 [EMPK], page 98)

  `beta = <double>`
  ⇒ see [unur_empk_set_beta], page 100

  `kernel = <unsigned>`
  ⇒ see [unur_empk_set_kernel], page 99

  `positive [= <int>]`
  ⇒ see [unur_empk_set_positive], page 100

  `smoothing = <double>`
  ⇒ see [unur_empk_set_smoothing], page 100

  `varcor [= <int>]`
  ⇒ see [unur_empk_set_varcor], page 100

- `method = hinv` ⇒ `unur_hinv_new` (see Section 5.3.3 [HINV], page 71)

  `boundary = <double>, <double> | (<list>)`
  ⇒ see [unur_hinv_set_boundary], page 73

  `cpoints = (<list>), <int>`
  ⇒ see [unur_hinv_set_cpoints], page 72

`guidefactor = <double>`
⇒ see [unur_hinv_set_guidefactor], page 73

`max_intervals [= <int>]`
⇒ see [unur_hinv_set_max_intervals], page 73

`order [= <int>]`
⇒ see [unur_hinv_set_order], page 72

`u_resolution = <double>`
⇒ see [unur_hinv_set_u_resolution], page 72

- `method = hrb` ⇒ `unur_hrb_new` (see Section 5.3.4 [HRB], page 74)

  `upperbound = <double>`
  ⇒ see [unur_hrb_set_upperbound], page 74

  `verify [= <int>]`
  ⇒ see [unur_hrb_set_verify], page 74

- `method = hrd` ⇒ `unur_hrd_new` (see Section 5.3.5 [HRD], page 74)

  `verify [= <int>]`
  ⇒ see [unur_hrd_set_verify], page 75

- `method = hri` ⇒ `unur_hri_new` (see Section 5.3.6 [HRI], page 75)

  `p0 = <double>`
  ⇒ see [unur_hri_set_p0], page 75

  `verify [= <int>]`
  ⇒ see [unur_hri_set_verify], page 76

- `method = ninv` ⇒ `unur_ninv_new` (see Section 5.3.7 [NINV], page 76)

  `max_iter [= <int>]`
  ⇒ see [unur_ninv_set_max_iter], page 77

  `start = <double>, <double> | (<list>)`
  ⇒ see [unur_ninv_set_start], page 77

  `table [= <int>]`
  ⇒ see [unur_ninv_set_table], page 77

  `usenewton`
  ⇒ see [unur_ninv_set_usenewton], page 77

  `useregula`
  ⇒ see [unur_ninv_set_useregula], page 76

  `x_resolution = <double>`
  ⇒ see [unur_ninv_set_x_resolution], page 77

- `method = nrou` ⇒ `unur_nrou_new` (see Section 5.3.8 [NROU], page 78)

  `center = <double>`
  ⇒ see [unur_nrou_set_center], page 79

  `u = <double>, <double> | (<list>)`
  ⇒ see [unur_nrou_set_u], page 79

`v = <double>`
⇒ see [unur_nrou_set_v], page 79

`verify [= <int>]`
⇒ see [unur_nrou_set_verify], page 79

- `method = srou` ⇒ `unur_srou_new` (see Section 5.3.9 [SROU], page 80)

  `cdfatmode = <double>`
  ⇒ see [unur_srou_set_cdfatmode], page 81

  `pdfatmode = <double>`
  ⇒ see [unur_srou_set_pdfatmode], page 81

  `r = <double>`
  ⇒ see [unur_srou_set_r], page 81

  `usemirror [= <int>]`
  ⇒ see [unur_srou_set_usemirror], page 82

  `usesqueeze [= <int>]`
  ⇒ see [unur_srou_set_usesqueeze], page 81

  `verify [= <int>]`
  ⇒ see [unur_srou_set_verify], page 82

- `method = ssr` ⇒ `unur_ssr_new` (see Section 5.3.10 [SSR], page 83)

  `cdfatmode = <double>`
  ⇒ see [unur_ssr_set_cdfatmode], page 84

  `pdfatmode = <double>`
  ⇒ see [unur_ssr_set_pdfatmode], page 84

  `usesqueeze [= <int>]`
  ⇒ see [unur_ssr_set_usesqueeze], page 84

  `verify [= <int>]`
  ⇒ see [unur_ssr_set_verify], page 84

- `method = tabl` ⇒ `unur_tabl_new` (see Section 5.3.11 [TABL], page 85)

  `areafraction = <double>`
  ⇒ see [unur_tabl_set_areafraction], page 88

  `boundary = <double>, <double> | (<list>)`
  ⇒ see [unur_tabl_set_boundary], page 88

  `darsfactor = <double>`
  ⇒ see [unur_tabl_set_darsfactor], page 87

  `guidefactor = <double>`
  ⇒ see [unur_tabl_set_guidefactor], page 88

  `max_intervals [= <int>]`
  ⇒ see [unur_tabl_set_max_intervals], page 88

  `max_sqhratio = <double>`
  ⇒ see [unur_tabl_set_max_sqhratio], page 87

  `nstp [= <int>]`
  ⇒ see [unur_tabl_set_nstp], page 88

`slopes = (<list>), <int>`
⇒ see [unur_tabl_set_slopes], page 88

`usedars [= <int>]`
⇒ see [unur_tabl_set_usedars], page 87

`variant_splitmode = <unsigned>`
⇒ see [unur_tabl_set_variant_splitmode], page 87

`verify [= <int>]`
⇒ see [unur_tabl_set_verify], page 89

- `method = tdr` ⇒ `unur_tdr_new` (see Section 5.3.12 [TDR], page 89)

`c = <double>`
⇒ see [unur_tdr_set_c], page 90

`center = <double>`
⇒ see [unur_tdr_set_center], page 92

`cpoints = <int> [, (<list>)] | (<list>)`
⇒ see [unur_tdr_set_cpoints], page 92

`darsfactor = <double>`
⇒ see [unur_tdr_set_darsfactor], page 91

`guidefactor = <double>`
⇒ see [unur_tdr_set_guidefactor], page 92

`max_intervals [= <int>]`
⇒ see [unur_tdr_set_max_intervals], page 92

`max_sqhratio = <double>`
⇒ see [unur_tdr_set_max_sqhratio], page 91

`pedantic [= <int>]`
⇒ see [unur_tdr_set_pedantic], page 93

`usecenter [= <int>]`
⇒ see [unur_tdr_set_usecenter], page 92

`usedars [= <int>]`
⇒ see [unur_tdr_set_usedars], page 90

`usemode [= <int>]`
⇒ see [unur_tdr_set_usemode], page 92

`variant_gw`
⇒ see [unur_tdr_set_variant_gw], page 90

`variant_ia`
⇒ see [unur_tdr_set_variant_ia], page 90

`variant_ps`
⇒ see [unur_tdr_set_variant_ps], page 90

`verify [= <int>]`
⇒ see [unur_tdr_set_verify], page 92

- `method = utdr` ⇒ `unur_utdr_new` (see Section 5.3.13 [UTDR], page 93)

`cpfactor = <double>`
⇒ see [unur_utdr_set_cpfactor], page 94

`deltafactor = <double>`
⇒ see [unur_utdr_set_deltafactor], page 94

`pdfatmode = <double>`
⇒ see [unur_utdr_set_pdfatmode], page 94

`verify [= <int>]`
⇒ see [unur_utdr_set_verify], page 94

- `method = vempk` ⇒ `unur_vempk_new` (see Section 5.6.1 [VEMPK], page 105)

  `smoothing = <double>`
  ⇒ see [unur_vempk_set_smoothing], page 105

  `varcor [= <int>]`
  ⇒ see [unur_vempk_set_varcor], page 106

- `method = vnrou` ⇒ `unur_vnrou_new` (see Section 5.5.2 [VNROU], page 101)

  `r = <double>`
  ⇒ see [unur_vnrou_set_r], page 103

  `v = <double>`
  ⇒ see [unur_vnrou_set_v], page 103

  `verify [= <int>]`
  ⇒ see [unur_vnrou_set_verify], page 103

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 such objects for standard distributions (see Chapter 7 [Standard distributions], page 121). It is then possible to change this distribution object by various set calls. Moreover it is possible to build a distribution object entirely from scratch. For this purpose there exists an `unur_distr_<type>_new` call that returns an empty object of this type for each object type (eg. univariate contiuous) which can be filled with the appropriate set calls.

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.

The library can handle truncated distributions, that is, distribution that are derived from (standard) distribution 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 this call 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 63, for details).

For the objects provided by the UNURAN library of standard distributions, calls for updating these parameters exist (one for each parameter to avoid computational overhead since not all parameters are required for all generator methods).

The calls listed below only handle distribution object. Since every generator object has its own copy of a distribution object, there are calls for a chosen method that change this copy of distribution object. NEVER extract the distribution object out of the generator object and run one of the below set calls on it. (How should the poor generator object know what has happend?)

4.1 Functions for all kinds of distribution objects

Function reference

`void` **unur_distr_free** (`UNUR_DISTR*` *distribution*) [–]

Destroy a 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.

`int` **unur_distr_get_dim** (`const` *`UNUR_DISTR* distribution`*) [–]
Get number of components of random vector (its dimension).

For univariate distributions it returns dimension `1`.

For matrix distributions it returns the number of components. 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 distributions

`UNUR_DISTR_CEMP`
: empirical continuous univariate distributions (i.e. samples)

`UNUR_DISTR_CVEC`
: continuous mulitvariate distributions

`UNUR_DISTR_CVEMP`
: empirical continuous multivariate distributions (i.e. samples)

`UNUR_DISTR_DISCR`
: discrete univariate distributions

`UNUR_DISTR_MATR`
: matrix distributions

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

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*. The type of each of these functions 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 121) 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 floats *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 PDF is given by the `unur_distr_cont_set_pdfstr` 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_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 33, for details. See also the remarks for the `unur_distr_cont_set_pdf` call.

It is not possible to call this funtion 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_dpdfstr** (`const` *UNUR_DISTR* distribution*) [–]
`char*` **unur_distr_cont_get_cdfstr** (`const` *UNUR_DISTR* distribution*) [–]

Get pointer to respective string for PDF, derivate of PDF, 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_cont_set_pdfparams** (`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 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. It these parameters are invalid, then no parameters are set and an error code is returned. Notice that optional parameters are (re-)set to their default values if not given for UNURAN standard distributions.

`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, `0` is returned and `params` is set to NULL.

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

`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 +/- infinity use `+/- 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 moden 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 explicitly `+/- 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. If the (truncated) domain is not set explicitly `+/- UNUR_INFINITY` is assumed and returned. No error is reported in this case.

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 33, for details. See also the remarks for the `unur_distr_cont_set_hr` call.

It is not possible to call this funtion 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 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_cont_set_mode** (`UNUR_DISTR* distribution, double mode`) [–]
Set mode of *distribution*.

`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 is used. If it failes `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_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

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 and `0` is returned. 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` **unur_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` **unur_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` **unur_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` **unur_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` **unur_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 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_corder_set_mode** (`UNUR_DISTR*` *distribution*, `double` *mode*) [–]
Set mode of distribution. See also `unur_distr_corder_set_mode`. (Macro)

`double` **unur_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` **unur_distr_corder_get_mode** (`UNUR_DISTR*` *distribution*) [–]
Get mode of distribution. See `unur_distr_cont_get_mode` for details. (Macro)

`int` **unur_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

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 double number from each line. 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, `0` 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 distribution. Moreover, the area of random vector generation is 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 5.5.1 [VMT], page 101) uses standard marginal distributions. However, the marginal distribution of the generated vectors might be transformed. Please read the discription of the choosen sampling method carfully.

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 must be at least 2; otherwise `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.

`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 proved the gradient of the function given by a `unur_distr_cvec_set_pdf` call.

`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, const UNUR_DISTR* distribution`) [–]

Evaluate the PDF of the *distribution* at *x*. *x* must be a pointers to a double arrays 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_dpdf** (`double* result, const double* x, const 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).

`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,. . . ,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 definite 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.

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: It might happen that a covariance matrix can be set but the inverse if the given matrix cannot be computed.

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_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 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 but not 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`) [–]

Get rank-correlation matrix 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 afterwarts 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*, `int` *par*, `const double*` *params*, `int` *n_params*) [–]

This function provides an interface for additional 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 *params* of size *n_params*. Notice that using this interface an An (n x 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 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** (`const` *`UNUR_DISTR*`* *`distribution`*`, int` *`par`*`, const` *`double**`* *`params`*) [–]

Get parameter of the PDF with number *par*. The pointer to the parameter array is stored in *params*, 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 *params*!

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 $(0, \ldots, 0)$.

`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 empirical multivariate distributions

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` double numbers from each line, where `dim` is the dimension of the *distribution* returned by `unur_distr_get_dim`. 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, `0` 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*.

4.7 MATRix distributions

Distributions for random matrices. Notice that UNURAN uses arrays of `double`s to handle matrices. There 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). Each must be at least 2; otherwise `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.

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. In case of an error `0` is returned.

4.8 Discrete univariate distributions

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 a *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 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 neglible 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 when a discrete distribution object is created from scratch, then the left boundary of the PV is set to `0` by default.

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, `0` 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,. . .,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`, `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 33, 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_cont_set_domain` call. It these parameters are invalid, then no parameters are set and an error code is returned. 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.

`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, `0` 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 +/- infinity use `INT_MAX` and `INT_MIN`, 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, than the right boudary is ignored (and internally set to *left* + size of PV - 1). Notice that `INT_MAX` and `INT_MIN` are interpreted as (minus) infinity.

Default is [`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 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_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 failes `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 `double`s 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?)

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 11.

Function reference

`UNUR_PAR*` **unur_auto_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get default parameters for generator.

`int` **unur_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                               */
/*                                                            */
/*          /  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.           */

  /* 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_sqhratio(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_sqhratio=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*` **unur_arou_new** (`const` *`UNUR_DISTR* distribution`*) [–]

Get default parameters for generator.

`int` **unur_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_sqhratio` 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 89)).

Default is `FALSE`.

`int` **unur_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` **unur_arou_set_max_sqhratio** (`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 segements. 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 5.7.3 [DGT], page 111). 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 squeeze(x) <= PDF(x) <= 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 121).

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 121). 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 (*variants*) is possible. These can be choosen by a `unur_cstd_set_variant` call. For possible variants see Chapter 7 [Standard distributions], page 121. 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.

Function reference

`UNUR_PAR*` **unur_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 121).

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` **unur_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 121.

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 chanced 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

HINV is a variant of numerical inversion, where the inverse CDF is approximated using Hermite interpolation. 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-resulution. 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 changed using the `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. for X = "approximate inverse CDF"(U) |U-CDF(X)|) can be controlled by means of the `unur_hinv_set_u_resolution`. 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. Setting such design points can be done 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` especially when a non-standard distribution is used.

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 resoultion 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 slighly 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 `+/- 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_guidefactor** (`UNUR_PAR*` *parameters*, `double` *factor*) [–]

Set factor for relative size of the guide table for indexed search (see also method DGT Section 5.7.3 [DGT], page 111). 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

Generates random variate with given hazard rate. It requires that the distribution object contains a hazard rate and it requires an upper bound for the hazard rate which must be set using `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.

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 decreasing hazard rates use method HRD, which is faster. For distributions with increasing hazard rate method HRI 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

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).

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. For distributions with increasing hazard rate method HRI is required.

Function reference

`UNUR_PAR*` **unur_hrd_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get default parameters for generator.

`int` **unur_hrd_set_verify** (`UNUR_PAR*` *`parameters`*, `int` *`verify`*) [–]
`int` **unur_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

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).

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 is required. For distributions which do not have increasing or decreasing hazard rates but are bounded from above use method HRB.

Function reference

`UNUR_PAR*` **unur_hri_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get default parameters for generator.

`int` **unur_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` **unur_hri_set_verify** (`UNUR_PAR*` *parameters*, `int` *verify*) [–]
`int` **unur_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.

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 standard 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 command `unur_ninv_[set|chg]_max_iter`.

To speed up this method (at the expense of the accuracy) it is possible to change the maximum error allowed in x with `unur_ninv_[set|chg]_x_resolution`.

NINV tries to use proper starting values for both the regala 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 command `unur_ninv_[set|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.

Default algorithm is regula falsi. It is slightly slower but numerically much more stable than Newton's algorithm.

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. It it is not possible to invert the CDF for a particular random number U when calling `unur_sample_cont`, `unur_error` is set to `UNUR_ERR_` and `UNUR_INFINITY` is returned. 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*) [–]
Set number of maximal iterations. Default is `40`.

`int` **unur_ninv_set_x_resolution** (`UNUR_PAR*` *parameters*, `double` *x_resolution*) [–]
Set maximal relative error. Default is `10^-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 follwing 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_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_max_iter** (`UNUR_GEN*` *generator*, `int` *max_iter*) [–]
Change the maximum number of iterations.

`int` **unur_ninv_chg_x_resolution** (`UNUR_GEN*` *generator*, `double` *x_resolution*) [–]
Change the maximal relative error in x.

`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_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* chanced 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. It 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`.

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]

NROU is an implementation of the ratio-of-uniforms method which uses (minimal) bounding rectangles, see Section A.4 [Ratio-of-Uniforms], page 148. The coordinates of this rectangles are given by

$$v^+ = \sup_x \sqrt{f(x)},$$

$$u^- = \inf_x (x-c)\sqrt{f(x)},$$

$$u^+ = \sup_x (x-c)\sqrt{f(x)}.$$

where c is the center of the distribution. These bounds can either be given directly, or these are computed automatically by means of an numerical routine. Of course this 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 that have to be provided are for the $PDF(x - center)$. This is important as otherwise the acceptance region can become a very long and skinny ellipsoid along a diagonal of the (huge) bounding rectangle.

How To Use

For using the NROU method UNURAN needs the PDF of the distribution. The bounding rectangle can be given by the `unur_vnrou_set_u` and `unur_vnrou_set_v` calls. If these are not called then the minimal bounding rectangle is computed automatically. Using `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. In particular for multimodal distributions this might fail. For T_c -concave distributions with $c = -1/2$ it should work.

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_center** (`UNUR_PAR*` *`parameters`*, `double` *`center`*) [–]
Set the center (μ) of the PDF. For distributions like the gamma distribution with large shape parameters the acceptance region becomes a long inclined skinny oval with a large bounding rectangle and thus an extremely large rejection constant. Using the *center* shifts the mode of the distribution near the origin and thus makes the bounding box of the acception region smaller.

Default: Mode if known, else `0`.

`int` **unur_nrou_set_verify** (`UNUR_PAR*` *parameters*, `int` *verify*) [–]

Turn verifying of algorithm while sampling on/off.

If the condition PDF(x) <= 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_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]

SROU is based on the ratio-of-uniforms method but 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)).

It requires the PDF, the (exact) location of the mode and the area below the given PDF. Moreover an (optional) 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)*. `r` should be set as small as possible but the given density must be T_c-concave for the corresponding *c*. The default setting for `r` is 1.

The parameter `r` can be any value larger than or equal 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.

Optionally the CDF at the mode can be given to increase the performance of the algorithm by means of the `unur_srou_set_cdfatmode` call. 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 not recommended in general (see below).

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 if no mode is given at all, a (slow) numerical mode finder will be used.

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.

It is even possible to give an upper bound for the area below 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). 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*` **unur_srou_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get default parameters for generator.

`int` **unur_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` **unur_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` **unur_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` **unur_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 squeeze(x) <= PDF(x) <= 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_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. It 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_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 correspondig `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 121). 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]

SSR is an acceptance/rejection method that uses universal inequalities for constructing (universal) hats and squeezes. 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 by means of the `unur_ssr_set_cdfatmode` call. Then the rejection constant is reduced by one half and even a universal squeeze can (but need not be) used. However using squeezes is not recommended unless the evaluation of the PDF is rather expensive. (The mirror principle is not implemented.)

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 if no mode is given at all, a (slow) numerical mode finder will be used.

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.

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. But 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). 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 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_DISTR* 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 distributions 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 squeeze(x) <= PDF(x) <= 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. It 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_srou_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 121). Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_srou_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: Cha5.1]

TABL (called Ahrens method in [HLD04]) is an acceptance/rejection method (see [ARVRej]) 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 (see [ARVRej]) 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" [Book,Cha5.1]. 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 "sqhratio" which must be smaller or equal to one. The expected number of iterations in the rejection algorithm is known to be smaller than 1/sqhratio and the expected number of evaluations of the density is bounded by 1/sqhratio - 1. So values of the sqhratio 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 sqhratio (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 is 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 `+/- 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.

The usage of the commands mentioned here are demonstrated in example_TABL1, example_TABL2 and example_TABL3 below.

Function reference

`UNUR_PAR*` **unur_tabl_new** (`const` *`UNUR_DISTR* distribution`*) [–]

Get default parameters for generator.

`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_sqhratio` 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_sqhratio** (`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 (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 suceeds *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_areafraction** (`UNUR_PAR*` *parameters*, `double` *fraction*) [–]
Set parameter for 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 distribution times *fraction* (which must be greater than zero).

Important: It the area below the PDF is not set, then 1 is assumed.

Default is `0.1`.

`int` **unur_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` **unur_tabl_set_slopes** (`UNUR_PAR*` *parameters*, `const` *double* slopes*, `int` *n_slopes*) [–]
Set slopes for the PDF. A slope <a,b> is an interval [a,b] or [b,a] where the PDF is monotone and PDF(a) >= PDF(b). The list of slopes is given by an array *slopes* where each consecutive tuple (i.e. `(slopes[0], slopes[1])`, `(slopes[2], slopes[3])`, etc.) defines one slope. Slopes must be sorted (i.e. both `slopes[0]` and `slopes[1]` must not be greater than any entry of the slope `(slopes[2], 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 5.7.3 [DGT], page 111). 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 `+/- UNUR_INFINITY` is not allowed.

Default is `-1.e20,1.e20`.

`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 squeeze(x) <= PDF(x) <= 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`.

5.3.12 TDR – Transformed Density Rejection

Required: T-concave PDF, dPDF

Optional: mode

Speed: Set-up: slow, Sampling: fast

reference: [GWa92] [HWa95]

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 <= -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 compositon 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_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_sqhratio` 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 amoung 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` **unur_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` **unur_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. Especially it becomes (very) small 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 adaptave 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 chanced and the call returns an error code.

`int` **unur_tdr_set_max_sqhratio** (`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_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_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_is_ARS_running** (`const UNUR_GEN* generator`) [–]
Check whether more points will be added by adaptive rejection sampling. (Internal call)

`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 suceeds *max_ivs*. It is increased automatically to twice the number of construction points if this is larger.

Default is `100`.

`int` **unur_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` **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_cpoints` 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 5.7.3 [DGT], page 111). 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 squeeze(x) <= PDF(x) <= 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 UTDR – Universal Transformed Density Rejection

Required: T-concave PDF, mode, approximate area

Speed: Set-up: moderate, Sampling: Moderate

reference: [HWa95]

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.

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*` **unur_utdr_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get default parameters for generator.

`int` **unur_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` **unur_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 squeeze(x) <= PDF(x) <= 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. It 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 121). 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 empirial 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 = unur_distr_cemp_new();
  unur_distr_cemp_set_data(distr, data, 15);

  /* Choose a method: EMPK.                                     */
  par = unur_empk_new(distr);

  /* Set smooting factor.                                       */
  unur_empk_set_smoothing(par, 0.8);

  /* Create the generator object.                               */
  gen = unur_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 empirial 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]

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 structur 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 standarddeviation of the noise, called bandwidth (or window width) of the kernel.

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. For other possible kernels see `unur_empk_set_kernel` and `unur_empk_set_kernelgen` below.

All other parameters 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 choosen. 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.

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) = Var(K)^(-2/5){ \int K(t)^2 dt}^(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 controlles 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]

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 Keltn (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 98).

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.

Important: Using this method is not recommended!

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 VMT – Vector Matrix Transformation

Required: mean vector, covariance matrix, standardized marginal distributions

Speed: Set-up: slow, Sampling: depends on dimension

VMT generates random vectors for distributions with given mean vector mu and covariance matrix Sigma. It produces random vectors of the form X = L Y + mu, where L is the Cholesky factor of Sigma, i.e. L L^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 mu and Sigma 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.

Function reference

`UNUR_PAR*` **unur_vmt_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get parameters for generator.

5.5.2 VNROU – Multivariate Naive Ratio-Of-Uniforms method

Required: PDF

Optional: mode, center, bounding rectangle for acceptance region

Speed: Set-up: fast, 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 148. It uses an additional parameter `r` that can be used to adjust the algorithm to the given distribution to improve performance and/or to make this method applicable. Moreover, this implementation uses the center c of the distribution (which is set to the mode or mean by default, see `unur_distr_cvec_set_center` for more details of its default values).

The minimal bounding has then the coordinates

$$v^+ = \sup_x (f(x))^{1/rk+1},$$

$$u_i^- = \inf_{x_i} (x_i - c_i) (f(x))^{r/rk+1},$$

$$u_i^+ = \sup_{x_i} (x_i - c_i) (f(x))^{r/rk+1},$$

where x_i is the i-th coordinate of point x; c_i is the i-th coordinate of the center c. These bounds can either be given directly, or these are computed automatically by means of an numerical routine by Hooke and Jeeves [HJa61] called direct search (see '`src/utils/hooke.c`' for further references and details). Of course this 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 that have to be provided are for the $PDF(x - center)$. 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 146). 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 increases when `r` is increased. On the other hand is is more unlikely that the bounding rectangle does not exist if `r` is small.

The bounding rectangle can be given by the `unur_vnrou_set_u` and `unur_vnrou_set_v` calls. If these are not called then the minimal bounding rectangle is computed automatically. Using `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_vnrou_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get default parameters for generator.

`int` **unur_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.

Notice: Computing the minimal bounding rectangle may fail under some circumstances. In particular for multimodal distributions this might fail.

Default: not set (i.e. computed automatically)

`int` **unur_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 numercally (and might fail).

Default: not set (i.e. computed automatically)

`int` **unur_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 $r>0$. Setting to a nonpositive value is ignored and in this case the default value value is used instead.

Default: `1`.

`int` **unur_vnrou_set_verify** (`UNUR_PAR*` *`parameters`*, `int` *`verify`*) [–]
Turn verifying of algorithm while sampling on/off.

If the condition PDF(x) <= 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_vnrou_chg_verify** (`UNUR_GEN*` *`generator`*, `int` *`verify`*) [–]
Change the verifying of algorithm while sampling on/off.

5.6 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 empirial 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 = unur_distr_cvemp_new( 2 );

  /* Set empirical sample.                                    */
  unur_distr_cvemp_set_data(distr, data, 4);

  /* Choose a method: VEMPK.                                  */
  par = unur_vempk_new(distr);

  /* Use variance correction.                                 */
  unur_vempk_set_varcor( par, 1 );

  /* Create the generator object.                             */
  gen = unur_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)

5.6.1 VEMPK – (Vector) EMPirical distribution with Kernel smoothing

Required: observed sample

Speed: Set-up: slow, Sampling: slow (depends on dimension)

reference: [HLa00]

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 standarddeviation 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.

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 controlles 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`.

5.7 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 = unur_distr_geometric(&param, 1);
  unur_distr_discr_set_mode(distr1, 0);

  /* Choose a method: DARI.                                   */
  par1 = unur_dari_new(distr1);
```

```
  gen1 = unur_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 = unur_distr_discr_new();
  unur_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() */

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

5.7.1 DARI – discrete automatic rejection inversion

Required: T-concave PMF, mode, approximate area

Speed: Set-up: moderate, Sampling: fast

reference: [HDa96]

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.

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*` **unur_dari_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get default parameters for generator.

`int` **unur_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` **unur_dari_set_squeeze** (`UNUR_PAR*` *`parameters`*, `int` *`squeeze`*) [–]
Turn utilization of the squeeze of the algorithm on/off. This squeeze does not resamble 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. 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 correspondig `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 121). Otherwise a (slow) numerical mode finder is called. If no mode can be found, then an error code is returnded 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 121). 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.

5.7.2 DAU – (Discrete) Alias-Urn method

Required: probability vector (PV)

Speed: Set-up: slow (linear with the vector-length), Sampling: very fast

reference: [WAa77]

DAU samples from distributions with arbitrary but finite probability vectors (PV) of length *N*. The algorithmus is based on an ingeneous 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).

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`.

5.7.3 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]

DGT samples from arbitrary but finite probability vectors. Random numbers are generated by the inversion method, i.e.,

1. Generate a random number U ~ U(0,1).
2. Find largest integer I such that F(I) = P(X<=I) <= 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'* <= *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 5.7.5 [DSS], page 114). 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 trucated (see `unur_distr_discr_make_pv` for details).

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 5.7.5 [DSS], page 114) 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.

5.7.4 DSROU – Discrete Simple Ratio-Of-Uniforms method

Required: T-concave PMF, mode, sum over PMF

Speed: Set-up: fast, Sampling: slow

reference: [LJa01]

DSROU is based on the ratio-of-uniforms method but uses universal inequalities for constructing a (universal) bounding rectangle. It works for all T-concave distributions with T(x) = -1/sqrt(x).

It 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 mode can be given to increase the performance of the algorithm by means of the `unur_dsrou_set_cdfatmode` call. Then the rejection constant is reduced to 2.

If the (exact) sum over the PMF is not known, then an upper bound can be used instead (which of course increases the rejection constant). But then `unur_dsrou_set_cdfatmode` must not be called.

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_pmfparams` and `unur_dsrou_chg_domain` call, respectively. But then `unur_dsrou_chg_pmfsum`, `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*` **unur_dsrou_new** (`const` *`UNUR_DISTR* distribution`*) [–]
Get default parameters for generator.

`int` **unur_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` **unur_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 squeeze(x) <= PMF(x) <= 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_pmfparams** (`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. It 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 correspondig `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_pmfsum** (`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_pmfsum** (`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 121). Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_dsrou_reinit` must be executed before sampling from the generator again.

5.7.5 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 ~ U(0,1).
2. Find largest integer I such that F(I) = P(X<=I) <= 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 5.7.3 [DGT], page 111) or DAU (see Section 5.7.2 [DAU], page 110) 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.

Function reference

`UNUR_PAR*` **unur_dss_new** (`const` *UNUR_DISTR* distribution*) [–]
Get default parameters for generator.

5.7.6 DSTD – Discrete STandarD distributions

Required: standard distribution from UNURAN library (see Chapter 7 [Standard distributions], page 121).

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 121). 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 (*variants*) is possible. These can be choosen by a `unur_dstd_set_variant` call. For possible variants see Chapter 7 [Standard distributions], page 121. However the following are common to all distributions:

`UNUR_STDGEN_DEFAULT`
: the default generator.

`UNUR_STDGEN_FAST`
: the fasted 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.

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 univariant distribution from the UNURAN library of standard distributions (see Chapter 7 [Standard distributions], page 121).

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 121.

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.

5.8 Methods for random matrices

5.8.1 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]

MCORR generates a random correlation matrix. Thus 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). There are many other possibilites (distributions) of sampling the random rows from a sphere. The chosen one is simple but does in not result in a uniform distriubution of the random correlation matrices.

It only works with distribution objects of random correlation matrices (see Section 7.4.1 [Random Correlation Matrix], page 131).

How To Use

Create a distibution object for random correlation matrices by a `unur_distr_correlation` call (see Section 7.4.1 [Random Correlation Matrix], page 131). 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.

5.9 Methods for uniform univariate distributions

5.9.1 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.

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

Each generator 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`; By this very flexible concept it is possible that each generator has its own (independent) URNG or several generators 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. A pointer to this default URNG can be obtained via `unur_get_default_urng`. Nevertheless, it is also possible to overwrite this default URNG by another one by means of the `unur_set_default_urng` call. However, this only takes effect for new parameter objects.

The pointer to a URNG is of type `UNUR_URNG*`. Its definition depends on the compilation switch `UNUR_URNG_TYPE` in '`unuran_config.h`'. Currently we have two possible switches (other values would result in a compilation error):

1. UNUR_URNG_TYPE == UNUR_URNG_FVOID

   This uses URNGs of type `double uniform(void)`. If independent versions of the same URNG should be used, a copy of the subroutine has to be implement 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_TYPE == UNUR_URNG_PRNG

   This uses the URNGs from the `prng` library. It provides a very flexible way to sample form arbitrary URNGs by means of an object oriented programing paradigma. Similarly to the UNURAN library independent generator objects can be build and used. Here `UNUR_URNG*` is simply a pointer to such a uniform generator object.

   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_TYPE == UNUR_URNG_RNGSTREAM

   Use Pierre L'Ecuyer's `RngStream` library for multiple independent streams of pseudo-random numbers. It is available from http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/c/.

4. UNUR_URNG_TYPE == UNUR_URNG_GSL

   Use the URNG from the GNU Scientific Library (GSL). It is available from http://www.gnu.org/software/gsl/.

5. UNUR_URNG_TYPE == UNUR_URNG_GENERIC

   This a generic interface with limited support. It uses a structure to store both a function call of type `double urng(void*)` and a void pointer to the parameter list. Both pointers must be set directly using the structure `struct unur_urng_generic` (there are currently no calls that support this URNG type). It is defined as

   ```
   struct unur_urng_generic {
      double (*getrand)(void *params);
      void *params;
   };
   ```

   All functions and parameters should be set at run time:

   1. Allocate variable of type `struct unur_urng_generic` (or of type `UNUR_URNG`, which is the same):

      ```
      UNUR_URNG *urng;
      urng = malloc(sizeof(UNUR_URNG));
      ```

2. Set function of type `double (*rand)(void *)` for sampling from URNG:
   ```
   urng->getrand = my_uniform_rng;
   ```
3. Set pointer to parameters of for this function (or `NULL` if no parameters are required):
   ```
   urng->params = my_parameters;
   ```
4. Use this URNG:
   ```
   unur_urng_set_default(urng);              (set default generator)
   unur_urng_set_default_aux(urng);    (set default aux. generator)
   ```
   Notice that this must be done before UNURAN generator object are created. Of course urng can also used just for a particular generator object. Use the following and similar calls:
   ```
   unur_set_urng(par,urng);
   unur_chg_urng(gen,urng);
   ```

It is possible to use other interfaces to URNGs without much troubles. All changes have to be done in file '`unuran_config.h`'. If you need such a new interface please feel free to contact the authors of the UNURAN library.

Some generating methods provide the possibility of correlation induction. To use this feature a second auxiliary URNG is required. It can be set and changed by the `unur_set_urng_aux` and `unur_chg_urng_aux` call, respectively. Since the auxiliary generator is by default the same as the main generator, 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 the `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.

Function reference

Default uniform RNGs

`UNUR_URNG*` **unur_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*` **unur_set_default_urng** (`UNUR_URNG*` *urng_new*) [–]
Change the default URNG for new parameter objects.

`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.

Uniform RNGs for generator objects

`int` **unur_set_urng** (`UNUR_PAR*` *parameters*, `UNUR_URNG*` *urng*) [–]
Use the URNG `urng` for the new generator. This overwrite 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 overwrite 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 use one (i.e. the call returns an error code).

`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.

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 explicitely. 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:

PDF probability density function, with variable x.

PMF probability mass function, with variable k.

constant normalization constant for given PDF and PMF, resp. They must be multiplied by *constant* to get the "real" PDF and PMF.

CDF gives information whether the CDF is implemented in UNURAN.

domain domain PDF and PMF, resp.

parameters *n_std (n_total):* list
list of parameters for distribution, where *n_std* is the number of parameters for the standard form of the distribution and *n_total* the total number for the (non-standard form of the) distribution. *list* 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 155).

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 155).

Notice that these generators might be slower than universal methods.

If `DEF` is ommited, 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		(*scale*)
`[1]`	q	> 0		(*scale*)
`[2]`	a		0	(*location, scale*)
`[3]`	b	$> a$	1	(*location, scale*)

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	(*location*)
`[1]`	λ	> 0	1	(*scale*)

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: $0 \le x < \infty$

parameters 1 (1): nu

No.	name		default	
`[0]`	ν	> 0		(*shape*)

reference: [JKBb94: Ch.18; p.417]

special generators:

`DEF` Ratio of Uniforms with shift (only for $\nu \ge 1$) [MJa87]

7.1.4 `chisquare` – Chisquare distribution

PDF: $x^{(\nu/2)-1} \exp(-x/2)$

constant: $1/(2^{\nu/2}\,\Gamma(\nu/2))$

domain: $0 \le x < \infty$

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: $\theta \le x < \infty$

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		(*shape*)
[1]	ζ		0	(*location*)
[2]	θ	> 0	1	(*scale*)

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		(*shape*)
[1]	β	> 0	1	(*scale*)
[2]	γ		0	(*location*)

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	(*location*)
`[1]`	ϕ	> 0	1	(*scale*)

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	(*location*)
`[1]`	β	> 0	1	(*scale*)

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: $0 \le x < \infty$

parameters 1 (2): a [, C]

No.	name		default	
`[0]`	a	> 0		(*shape*)
`[1]`	C	> 0	1	(*scale*)

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	(*location*)
`[1]`	σ	> 0	1	(*scale*)

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: $0 \le x < \infty$

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		(*shape*)

reference: [JKBc95: Ch.28; p.362]

7.1.17 `triangular` – Triangular distribution

PDF: $2\,x/H$, for $0 \le x \le H$
$2\,(1-x)/(1-H)$, for $H \le x \le 1$

constant: 1

domain: $0 \le x \le 1$

parameters 0 (1): [H]

No.	name		default	
`[0]`	H	$0 \le H \le 1$	1/2	(*shape*)

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	(*location*)
[1]	b	$> a$	1	(*location*)

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		(*shape*)
[1]	α	> 0	1	(*scale*)
[2]	ζ		0	(*location*)

reference: [JKBb94: Ch.21; p.628]

special generators:

INV	Inversion method

7.2 UNURAN Library of continuous multivariate distributions

7.2.1 multinormal – Multinormal distribution

reference: [KBJe00: Ch.45; p.105]

`UNUR_DISTR *unur_distr_multinormal(int dim, const double *mean, const double *covar)` creates a distribution object for the multinormal distribution with *dim* components. *mean* is an array of size *dim*. A `NULL` pointer for *mean* is interpreted as the zero vector (0,. . .,0). *covar* is an array of size *dim*x*dim* and holds the covariance matrix, 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 covariance matrix (i.e., not positive definite) then no distribution object is created and `NULL` is returned.

For standard form of the distribution use the null vector for *mean* and the identity matrix for *covar*.

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: $0 \le k \le n$

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: $0 \le k < \infty$

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: $\max(0, n-N+M) \le k \le \min(n, M)$

parameters 3 (3): N, M, n

No.	name		default	
[0]	N	≥ 1		(*no. of elements*)
[1]	M	$1 \le M \le N$		(*shape*)
[2]	n	$1 \le n \le N$		(*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: $1 \le k < \infty$

parameters 1 (1): theta

No.	name		default	
`[0]`	θ	$0 < \theta < 1$		(*shape*)

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: $0 \le k < \infty$

parameters 2 (2): p, r

No.	name		default	
`[0]`	p	$0 < p < 1$		(*shape*)
`[1]`	r	> 0		(*shape*)

reference: [JKKa92: Ch.5.1; p.200]

7.3.6 `poisson` – Poisson distribution

PMF: $\theta^k/k!$

constant: $\exp(\theta)$

domain: $0 \le k < \infty$

parameters 1 (1): theta

No.	name		default	
`[0]`	θ	> 0		(*shape*)

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 5.8.1 [Random Correlation Matrix], page 116) 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 `0` 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, chosing 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 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 occured.

`UNUR_ERR_NAN`
: NaN occured.

`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 abritrarily 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 destinguish 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 implementions. 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_CHI2`
: run chi^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*.

double **unur_test_chi2** (UNUR_GEN* *generator*, int *intervals*, int *samplesize*, int *classmin*, int *verbosity*, FILE* *out*) [–]

Run a Chi^2 test with the *generator*. The resulting p-value is returned.

It works with discrete und 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 to 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`.

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 resultung 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*:

q0	minimum
q1	25%
q2	median (50%)
q3	75%
q4	maximum

If *verbosity* is `TRUE` the result is written to the output stream *out*.

11 Miscelleanous

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`
: bolean 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.

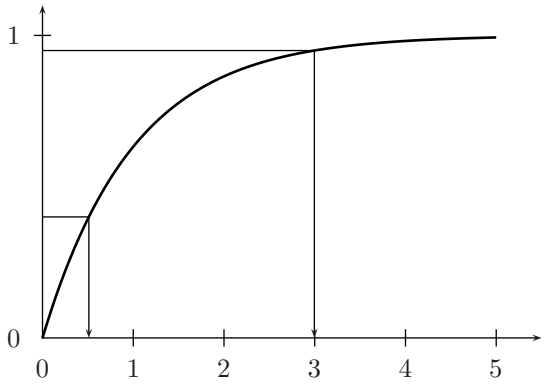
For a deeper treatment of the ideas presented here, for other basic methods and for automatic generators we refer the interested reader to our book [HLD04].

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

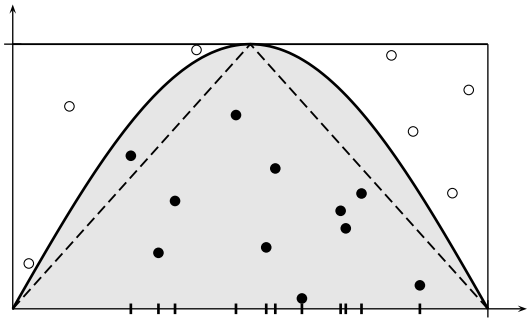
statistics. Moreover, the quality of the generated random variables depends only on the underlying uniform (pseudo-) random number generator. Another important advantage of the inversion method is that we can easily characterize its performance. To generate one random variate we always need exactly one 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 76) and HINV (see Section 5.3.3 [HINV], page 71) 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 146) 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 145).

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 algoritm. The expected number of evaluations of the density is bounded by $(1/sqhratio) - 1$.

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 seperatly 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 150).

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).

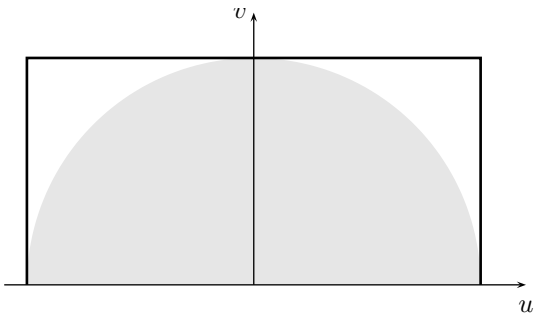
As a first example we consider the following simple algorithm for the Cauchy distribution.

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

$$(X, Y) \mapsto (U, V) = (2\,X\sqrt{Y}, 2\,\sqrt{Y})$$

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 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

$$v^+ \ = \ \sup_{b_l < x < b_r} \sqrt{f(x)},$$

$$u^- \ = \ \inf_{b_l < x < b_r} (x - \mu)\ \sqrt{f(x)},$$

$$u^+ \ = \ \sup_{b_l < x < b_r} (x - \mu)\ \sqrt{f(x)}.$$

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.)

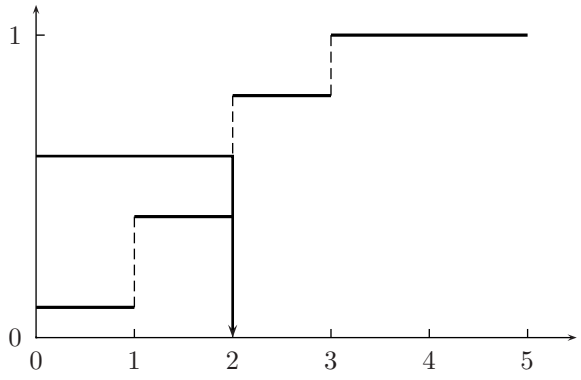
Nevertheless, we use more sophisticated method that construct better fitting envelopes, like method AROU (see Section 5.3.1 [AROU], page 67), or even avoid the computation of these design constants and thus have almost no setup, like method SROU (see Section 5.3.9 [SROU], page 80).

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 145). 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 $u_i = i/C$, $i = 0, ..., C-1$. 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 $k = floor(UC)$ 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

$U(a, b)$ continuous uniform distribution on the interval (a, b)

T-concave
T_c-concave

a function *f(x)* is called T-convace 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 != 0`	T(x) = sign(x) * 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.

[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.

[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

[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