UNU.RAN User Manual

Generating non-uniform random numbers
Version 0.1.2, 21 March 2001

Josef Leydold
Wolfgang Hörmann
Erich Janka
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 1.3 [Concepts of UNURAN], page 4 discribes the basics of UNURAN. It also has a short guideline for choosing an appropriate method. In Chapter 2 [Examples], page 9 gives examples 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 3 [Handling distribution objects], page 21 describes how to create and manipulate distribution objects. Chapter 6 [standard distributions], page 75 describes predefined distribution objects that are ready to use. Chapter 4 [Methods], page 35 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 A [Glossary], page 95.

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 5 [Using uniform random number generators], page 73.

UNURAN

1. First unzip and untar the package and change to the directory:
   ```
   tar zxvf unuran-0.1.2.tar.gz
   cd unuran-0.1.2
   ```
2. Edit the file '`src/unuran_config.h`'. Set the appropriate source of uniform random numbers: `#define UNUR_URNG_TYPE` (see Chapter 5 [URNG], page 73 for details).

   Important: If the prng library is not installed you must not use `UNUR_URNG_PRNG`.

   Warning: If `UNUR_URNG_POINTER` is used then the build-in default uniform random number generators should be used only for small sample sizes or for demonstration. They are not state of the art any more.
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.
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)/lib/libunuran.a
$(prefix)/info/unuran.info
```

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="HOMEDIRECTORY/lib"
export C_INCLURE_PATH="HOMEDIRECTORY/include"
```

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 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.

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 9.

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_sample_<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 3 [Handling distribution objects], page 21). 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 6 [Standard distributions], page 75.

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 A [Glossary], page 95 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
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.

Methods for **continuous multivariate distributions**
sample with `unur_sample_vec`

VMT: Requires the mean vector and the covariance matrix.

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

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 4 [Methods], page 35.

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 4 [Methods], page 35.

Sampling

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

Destroy

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

```
unur_free(generator);
```

Uniform random numbers

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

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

or

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

respectively. See Chapter 5 [Using uniform random number generators], page 73 for details.

1.4 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.

2.1 As short as possible

```
/* ------------------------------------------------------------ */
/* 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() */

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

2.2 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 1.3 [Concepts], page 4 for an overview and Chapter 4 [Methods], page 35 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, 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, 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() */

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

2.3 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 4 [Methods], page 35.

```
/* ------------------------------------------------------------- */
/* 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() */

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

2.4 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 5 [Using uniform random number generators], page 73 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_POINTER                   */
/* 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 (urng1 == 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");
#endif

} /* end of main() */

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

2.5 Sample pairs of antithetic random variates

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

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

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

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

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

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

/* 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_POINTER                    */
/* 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");
#endif

} /* end of main() */

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

2.6 More examples

See Section 4.2 [Methods for continuous univariate distributions], page 36.
See Section 4.3 [Methods for continuous empirical univariate distributions], page 57.
See Section 4.5 [Methods for continuous empirical multivariate distributions], page 62.
See Section 4.6 [Methods for discrete univariate distributions], page 64.

3 Handling distribution objects

Objects of type `UNUR_DISTR` are used for handling distributions. All data about a distribution are stored on this object. UNURAN provides functions that return such objects for standard distributions (see Chapter 6 [Standard distributions], page 75). 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 for each object type that returns an empty object of this 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 4 [Methods], page 35, 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?)

3.1 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** (`UNUR_DISTR*` *distribution*) –

Set and get name of distribution.

`int` **unur_distr_get_dim** (`UNUR_DISTR*` *distribution*) –

Get number of components of random vector (its dimension). For univariate distributions it returns dimension `1`.

`unsigned int` **unur_distr_get_type** (`UNUR_DISTR*` *distribution*) –

Get type of *distribution*. Possible types are

`UNUR_DISTR_CONT`
: univariate continuous distributions

`UNUR_DISTR_CEMP`
: empirical continuous univariate distributions (ie. samples)

`UNUR_DISTR_CVEC`
: continuous mulitvariate distributions

`UNUR_DISTR_CVEMP`
: empirical continuous multivariate distributions (ie. samples)

`UNUR_DISTR_DISCR`
: discrete univariate distributions

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

`int` **unur_distr_is_cont** (`UNUR_DISTR*` *distribution*) –

TRUE if *distribution* is a continuous univariate distribution.

`int` **unur_distr_is_cvec** (`UNUR_DISTR*` *distribution*) –

TRUE if *distribution* is a continuous multivariate distribution.

`int` **unur_distr_is_cemp** (`UNUR_DISTR*` *distribution*) –

TRUE if *distribution* is an empirical continuous univariate distribution, i.e. a sample.

`int` **unur_distr_is_cvemp** (`UNUR_DISTR*` *distribution*) –

TRUE if *distribution* is an empirical continuous multivariate distribution.

`int` **unur_distr_is_discr** (`UNUR_DISTR*` *distribution*) –

TRUE if *distribution* is a discrete univariate distribution.

3.2 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, 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 6 [Standard distributions], page 75) 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. A new distribution object has to be used instead.

UNUR_FUNCT_CONT* **unur_distr_cont_get_pdf** (UNUR_DISTR* *distribution*) –
UNUR_FUNCT_CONT* **unur_distr_cont_get_dpdf** (UNUR_DISTR* *distribution*) –
UNUR_FUNCT_CONT* **unur_distr_cont_get_cdf** (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, 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*, UNUR_DISTR* *distribution*) –
double **unur_distr_cont_eval_dpdf** (double *x*, UNUR_DISTR* *distribution*) –
double **unur_distr_cont_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`.

`int` **unur_distr_cont_set_pdfparams** (`UNUR_DISTR*` *distribution*, `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 `0` 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** (`UNUR_DISTR*` *distribution*, `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`. It is allowed to use this call to increase the domain.

`int` **unur_distr_cont_get_domain** (`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** (`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.

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_GET`.

`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`.

3.3 Continuous univariate order statistics

Function reference

`UNUR_DISTR*` **unur_distr_corder_new** (`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.

UNUR_DISTR* **unur_distr_corder_get_distribution** (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** (UNUR_DISTR* *distribution*, int* *n*, int* *k*) –
: Get sample size *n* and rank *k* of order statistics. In case of error `0` 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)

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)

3.4 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*, 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 `0` is returned.

int **unur_distr_cemp_get_data** (UNUR_DISTR* *distribution*, 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 *params*!

3.5 Continuous multivariate distributions

Function reference

UNUR_DISTR* **unur_distr_cvec_new** (int *dim*) –

Create a new (empty) object for multivariate continuous distribution. *dim* is the number of components of the random vector (i.e. its dimension). It 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*. The type of this function must be of type `double funct(double *x, 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, double *x, 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 `0` in case of an error and must return a non-zero value 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** (`UNUR_DISTR*` *distribution*) –

Get the pointer to the PDF of the *distribution*. 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.

`UNUR_VFUNCT_CVEC*` **unur_distr_cvec_get_dpdf** (`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, double *x, UNUR_DISTR *distr)`. If the corresponding function is not available for the *distribution*, the `NULL` pointer is returned.

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

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

Notice that *distribution* must not be the `NULL` pointer. If the corresponding function is not available for the *distribution*, `0` 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*, `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).

`double*` **unur_distr_cvec_get_mean** (`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`. (However note that the `NULL` pointer also indicates the zero vector to avoid unnecessary computations. But then `unur_errno` is not set.)

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

`int` **unur_distr_cvec_set_covar** (`UNUR_DISTR*` *distribution*, `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 positive. There is no check for the positive definitnes yet.

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

double* **unur_distr_cvec_get_covar** (UNUR_DISTR* *distribution*) –

Get covariance matrix of *distribution*. The function returns a pointer to an array of size `dim` x `dim`. The rows of the matrix have to be stored consecutively in this array. If the covariance matrix is not marked as known the `NULL` pointer is returned and `unur_errno` is set to `UNUR_ERR_DISTR_GET`. (However note that the `NULL` pointer also indicates the identity matrix to avoid unnecessary computations. But then `unur_errno` is not set.)

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

int **unur_distr_cvec_set_pdfparams** (UNUR_DISTR* *distribution*, int *par*, 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** (UNUR_DISTR* *distribution*, int *par*, 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 `0` 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*, double* *mode*) –

Set mode of *distribution*. *mode* must be a pointer to an array of the size returned by `unur_distr_get_dim`.

double* **unur_distr_cvec_get_mode** (UNUR_DISTR* *distribution*) –

Get mode of *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_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** (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`.

3.6 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*, 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 `0` is returned.

In case of an error no data are copied into the distribuion object and `0` is returned.

int **unur_distr_cvemp_get_data** (UNUR_DISTR* *distribution*, 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*.

3.7 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 a PV can be computed from PMF by means of a `unur_distr_discr_make_pv` call. If both a PV and a 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*, 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 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 10^-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 10^-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, `0` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_SET`.

int **unur_distr_discr_get_pv** (UNUR_DISTR* *distribution*, 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*. The type of each of these functions must be of type `double funct(int k, UNUR_DISTR *distr)`.

It is important to note that all these functions must return a result for all integers *k*. Eg., 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.

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 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*, `UNUR_DISTR*` *distribution*) –
`double` **unur_distr_discr_eval_pmf** (`int` *k*, `UNUR_DISTR*` *distribution*) –
`double` **unur_distr_discr_eval_cdf** (`int` *k*, `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`.

`int` **unur_distr_discr_set_pmfparams** (`UNUR_DISTR*` *distribution*, `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 `0` 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** (`UNUR_DISTR*` *distribution*, `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 [`INT_MIN`, `INT_MAX`] when a PMF is used for generation, and [0, size of PV - 1] when a PV is used.

int **unur_distr_discr_get_domain** (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.

4 Methods for generating non-uniform random variates

4.1 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*) –

Sample from generator object. The three routines depend on the type of the generator object (discrete or continuous univariate distribution, or multivariate distribution).

Important: These routines do **not** check if generator is an invalid `NULL` pointer.

`void` **unur_free** (`UNUR_GEN*` *generator*) –

Destroy (free) the given generator object.

`int` **unur_get_dimension** (`UNUR_GEN*` *generator*) –

Get the number of dimension of a (multivariate) distribution. For a univariate distribution `1` is return.

`const char*` **unur_get_genid** (`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.

`UNUR_DISTR*` **unur_get_distr** (`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?)

4.2 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
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

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, 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, 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() */

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

4.2.1 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 auxilliary 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 auxilliary 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** (`UNUR_DISTR*` *distribution*) –
: Get default parameters for generator.

`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** (`UNUR_GEN*` *generator*) –
: Get the current ratio (area inside squeeze) / (area inside envelope) for the generator. (In case of error `0` 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*, `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 4.6.3 [DGT], page 69). 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.

4.2.2 CSTD – Continuous STandarD distributions

Required: standard distribution from UNURAN library (see Chapter 6 [Standard distributions], page 75).

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 6 [Standard distributions], page 75). 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 6 [Standard distributions], page 75. 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** (`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 6 [Standard distributions], page 75).

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 6 [Standard distributions], page 75.

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 `0` 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 `0` 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 `0`.

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.

4.2.3 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. 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** (`UNUR_DISTR*` *distribution*) —

Get default parameters for generator.

`int` **unur_ninv_set_useregula** (`UNUR_PAR*` *parameters*) —

Switch to regula falsi. (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 in x. 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 `0`.

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 0 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 0 is returned and `unur_errno` is set to `UNUR_ERR_DISTR_NPARAMS`.

4.2.4 SROU – Simple Ratio-Of-Uniforms method

Required: T-concave PDF, mode, area

Speed: Set-up: fast, Sampling: slow

reference: [LJa01]

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 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. 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 to 2 and even a universal squeeze can (but need not be) used. A way to increase the performence of the algorithm when the CDF at the mode is not provided is the usage of the mirror principle. 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** (`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 `1` is returned, in case of a failure `0` is returned.

`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.

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 0 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 0 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 6 [Standard distributions], page 75). Otherwise unur_errno is set to UNUR_ERR_DISTR_DATA. unur_srou_reinit must be executed before sampling from the generator again.

4.2.5 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** (`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 `1` is returned, in case of a failure `0` 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 `0` 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 `0` 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 6 [Standard distributions], page 75). Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_srou_reinit` must be executed before sampling from the generator again.

4.2.6 TABL – a TABLe method with piecewise constant hats

Required: PDF, all local extrema

Optional: approximate area

Speed: Set-up: slow, Sampling: fast

reference: [AJa93] [AJa95]

TABL is an acceptance/rejection method that uses piecewise constant hat and squeezes. Immediate acceptance of points below the squeeze reduces the expected number of uniform random numbers to less than two and makes this method extremely fast.

The method only works for distributions with bounded domain. Thus for unbounded domains the left and right tails are cut off (the cutting points can be set by a `unur_tabl_set_boundary` call). This is no problem when the probability of falling into these tail regions is beyond computational relevance.

The method works for all probability density functions where the regions of monotonicity (called slopes) are given. This can be done explicitly by a `unur_tabl_set_slopes` call. If (and only if) no slopes are given, the domain and the mode of the PDF are used to compute the slopes. If neither slopes nor the mode and the domain are given initializing of the generator fails.

In the setup first the equal area rule is used to construct a hat function, i.e., the interval boundaries are chosen such that the area below each interval is equal to a given fraction of the total area below the given PDF. This fraction can be set by a `unur_tabl_set_areafraction` call. Additionally these intervals are split until the maximum number of intervals is reached or the ratio between the area below squeeze and the area below the hat is exceeded.

It is possible to switch off this second setup step. Then adaptive rejection sampling is used to split these intervals. There are three variants for adaptive rejection sampling. These differ in the way how an interval is split:

1. use the generated point to split the interval;
2. use the mean point of the interval; or
3. use the arcmean point.

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_tabl_set_verify` and `unur_tabl_chg_verify`, respectively. Notice however that sampling is (much) slower then.

Function reference

`UNUR_PAR*` **unur_tabl_new** (`UNUR_DISTR*` *distribution*) –

Get default parameters for generator.

`int` **unur_tabl_set_variant_setup** (`UNUR_PAR*` *parameters*, `unsigned` *variant*) –

Set variant for setup. Two modes are possible for *variant*:

`1` only use the equal area rule to construct hat.

`2` additionally split the intervals created by the equal area rule until the maximum number of intervals is reached or the ratio between the area below the squeeze and the area below the hat is exceeded.

Default is variant `2`.

`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 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. If *max_ratio* is close to one, many construction points are used.

Default is `0.9`.

`double` **unur_tabl_get_sqhratio** (`UNUR_GEN*` *generator*) –

Get the current ratio (area below squeeze) / (area below hat) for the generator. (In case of an error `0` is returned.)

`int` **unur_tabl_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*.

Default is `1000`.

`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*, `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 are given by an array *slopes* where each consecutive tuples (i.e. `(slopes[0], slopes[1])`, `(slopes[2], slopes[3])`, etc.) is one slopes. 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 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 4.6.3 [DGT], page 69). 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`.

`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`.

4.2.7 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 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`.

There are lots of parameters for these methods, see below.

It is possible to use this method for correlation induction by setting an auxilliary 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 auxilliary 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** (`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_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 it switches 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 `0`.

`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** (`UNUR_GEN*` *generator*) –

Get the current ratio (area below squeeze) / (area below hat) for the generator. (In case of an error 0 is returned.)

`int` **unur_tdr_set_max_intervals** (`UNUR_PAR*` *parameters*, `int` *max_ivs*) –

Set maximum number of intervals. No construction points are added after the setup when the number of intervals suceeds *max_ivs*.

Default is `100`.

`int` **unur_tdr_set_cpoints** (`UNUR_PAR*` *parameters*, `int` *n_stp*, `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 4.6.3 [DGT], page 69). 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`.

4.2.8 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** (`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 `1` is returned, in case of a failure `0` 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 `0` 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 `0` 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 6 [Standard distributions], page 75). Otherwise `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_srou_reinit` must be executed before sampling from the generator again.

4.3 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.

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() */

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

4.3.1 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.

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. Only exception is the case that you are especially interested in a fast sampling algorithm. Then 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.

Function reference

`UNUR_PAR*` **unur_empk_new** (`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 a Gaussian kernel.

int **unur_empk_set_kernelgen** (`UNUR_PAR*` *parameters*, `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 generator for the empirical distribution.

Default is a 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`.

4.4 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.

4.4.1 VMT – Vector Matrix Transformation

Required: mean vector, covariance matrix

Optional: marginal distribution (for non-Gaussian 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.

By default the standard normal distribution is used for the components of Y. Thus VMT produces multinormal random vectors when this distribution of Y is not set explicitly.

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 univariate distribution provided by a `unur_vmt_set_marginalgen` call should have mean 0 and standard deviation 1. Otherwise mu and Sigma are not the mean vector and covariance matrix, respectively, of the resulting distribution. Moreover notice that except for the multinormal distribution the given univariate distribution is *not* the marginal distribution of the resulting random vector.

Function reference

`UNUR_PAR*` **unur_vmt_new** (`UNUR_DISTR*` *distribution*) –

Get default parameters for generator.

`int` **unur_vmt_set_marginalgen** (`UNUR_PAR*` *parameters*, `UNUR_GEN*` *uvgen*) –

Set generator for (univariate) marginal distribution.

Default: Generator for (univariate) standard normal distribution.

4.5 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_cont.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() */

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

4.5.1 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** (`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`.

4.6 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

Example

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

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

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

/* Example how to sample from a discrete univariate distribution */
/* and sample from this 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() */

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

4.6.1 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 auxilliary 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** (`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 `1` is returned, in case of a failure `0` 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 6 [Standard distributions], page 75). Otherwise a (slow) numerical mode finder is called. If no mode can be found, then `0` 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 6 [Standard distributions], page 75). Otherwise `0` is returned and `unur_errno` is set to `UNUR_ERR_DISTR_DATA`. `unur_dari_reinit` must be executed before sampling from the generator again.

4.6.2 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** (`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`.

4.6.3 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. On the other hand the setup time for guide table is *O(N)* (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 (see `unur_distr_discr_make_pv` for details).

Function reference

`UNUR_PAR*` **unur_dgt_new** (`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.

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.

4.6.4 DSTD – Discrete STandarD distributions

Required: standard distribution from UNURAN library (see Chapter 6 [Standard distributions], page 75).

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 6 [Standard distributions], page 75). 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 6 [Standard distributions], page 75. 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** (`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 6 [Standard distributions], page 75).

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 6 [Standard distributions], page 75.

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.

4.7 Methods for uniform univariate distributions

4.7.1 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** `(void)` –

Get default parameters for generator. UNIF does not need a distribution object.

5 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 change this pointer via `unur_get_urng` or change the URNG for an existing generator object by means of `unur_get_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_POINTER

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).

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/`.

It is possible to use other interfaces to URNGs without much troubles. If you need such a new interface please email the authors of the UNURAN library.

Some generating methods provide the possibility of correlation induction. To use this feature a second auxilliary URNG is required. It can be set and changed by the `unur_set_urng_aux` and `unur_chg_urng_aux` call, respectively. Since the auxilliary generator is by default the same as the main generator, the auxilliary URNG must be set after any `unur_set_urng` ot `unur_chg_urng` call! Since in special cases mixing of two URNG might cause problems, we supply a default auxilliary generator that can be used by the `unur_use_urng_aux_default` call (after the main URNG has been set).

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.

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 auxilliary URNG to `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 auxilliary 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 auxilliary 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 auxilliary generator should be different to the main URNG, `unur_set_urng_aux` must be called after `unur_set_urng`. The auxilliary URNG is used as second stream of uniform random number for correlation induction. It is not possible to set an auxilliary URNG for a method that does not use one (i.e. the call returns 0).

int **unur_use_urng_aux_default** (UNUR_PAR* *parameters*) –

Use the default auxilliary URNG. (It must be set after `unur_get_urng`.) It is not possible to set an auxilliary URNG for a method that does not use one (i.e. the call returns 0).

UNUR_URNG* **unur_chg_urng_aux** (UNUR_GEN* *generator*, UNUR_URNG* *urng_aux*) –

Change the auxilliary URNG for the given generator. It returns the pointer to the old auxilliary URNG that has been used by the generator. It has to be called after each `unur_chg_urng` when the auxilliary URNG should be different from the main URNG. It is not possible to change the auxilliary 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 auxilliary URNG that is used by the generator. This is usefull if two generators should share the same URNG.

6 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.

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 B [Bibliography], page 97).

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 B [Bibliography], page 97).

Notice that these generators might be slower than universal methods.

If `DEF` is ommited, the first entry is the default generator.

6.1 UNURAN Library of continuous univariate distributions

6.1.1 `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

6.1.2 `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

6.1.3 `chi` – Chi distribution

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

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

domain: $0 \leq x < \infty$

parameters 1 (1): nu

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

reference: [JKBb94] ch.18, p.417

special generators:

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

6.1.4 `chisquare` – Chisquare distribution

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

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

domain: $0 \leq x < \infty$

parameters 1 (1): nu

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

reference: [JKBb94] ch.18, p.416

6.1.5 `exponential` – Exponential distribution

PDF: $\exp(-\frac{x-\theta}{\sigma})$

constant: $\frac{1}{\sigma}$

domain: $\theta \leq x < \infty$

parameters 0 (2): [sigma [, theta]]

No.	name		default	
`[0]`	σ	> 0	1	(*scale*)
`[1]`	θ		0	(*location*)

reference: [JKBb94] ch.19, p.494

special generators:

`INV` Inversion method

6.1.6 `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

6.1.7 `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

6.1.8 `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]

6.1.9 `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

6.1.10 `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

6.1.11 `lomax` – Lomax distribution (Pareto distribution of second kind)

PDF: $(x+C)^{-(a+1)}$

constant: $a\,C^a$

domain: $0 \leq x < \infty$

parameters 1 (2): a [, C]

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

reference: [JKBb94] ch.20, p.575

special generators:

`INV` Inversion method

6.1.12 `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)

6.1.13 `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

6.1.14 `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]

6.1.15 `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

6.1.16 `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

6.1.17 `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

6.1.18 `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

6.2 UNURAN Library of continuous multivariate distributions

6.3 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.

6.3.1 `binomial` – Binomial distribution

PMF: $\binom{n}{k}\, p^k\, (1-p)^{n-k}$

constant: 1

domain: $0 \leq k \leq n$

parameters 2 (2): n, p

No.	name		default	
[0]	n	≥ 1		(*no. of elements*)
[1]	p	$0 < p < 1$		(*shape*)

reference: [JKKa92] ch.3, p.105

special generators:

`DEF` Ratio of Uniforms/Inversion [STa89]

6.3.2 `geometric` – Geometric distribution

PMF: $p\,(1-p)^k$

constant: 1

domain: $0 \leq k < \infty$

parameters 1 (1): p

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

reference: [JKKa92] ch.5.2, p.201

special generators:

`INV` Inversion method

6.3.3 `hypergeometric` – Hypergeometric distribution

PMF: $\binom{M}{k}\binom{N-M}{n-k}/\binom{N}{n}$

constant: 1

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

parameters 3 (3): N, M, n

No.	name		default	
`[0]`	N	≥ 1		(*no. of elements*)
`[1]`	M	$1 \leq M \leq N$		(*shape*)
`[2]`	n	$1 \leq n \leq N$		(*shape*)

reference: [JKKa92] ch.6, p.237

special generators:

`DEF` Ratio of Uniforms/Inversion [STa89]

6.3.4 `logarithmic` – Logarithmic distribution

PMF: θ^k/k

constant: $-log(1.-\theta)$;

domain: $1 \leq k < \infty$

parameters 1 (1): theta

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

reference: [JKKa92] ch.7, p.285

special generators:

`DEF` Inversion/Transformation [KAa81]

6.3.5 `negativebinomial` – Negative Binomial distribution

PMF: $\binom{k+r-1}{r-1} p^r (1-p)^k$

constant: 1

domain: $0 \leq k < \infty$

parameters 2 (2): p, r

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

reference: [JKKa92] ch.5.1, p.200

6.3.6 `poisson` – Poisson distribution

PMF: $\theta^k / k!$

constant: $\exp(\theta)$

domain: $0 \leq k < \infty$

parameters 1 (1): theta

No.	name		default	
`[0]`	θ	> 0		(*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 Error handling

This chapter describes the way that UNURAN routines report errors.

7.1 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 `unuran_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 `unuran_errno` simultaneously). Thus when an error occurs the caller of the routine can examine the error code in `unuran_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

- 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.

- 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.

- Other run time errors.

`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_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 unsigned` **unur_errno** — Variable

Global variable for reporting diagnostics of error.

7.2 Output streams

In addition to reporting error via the `unuran_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.

8 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 til 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 they read from or write to the given adress. This does not hold for time-critical routines like all sampling routines. Then your 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.

9 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 marginal generation time and generation time for the underlying uniform random number (using the UNIF interface).

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*.

`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*.

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. There is no output when it is set to 0.

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*.

10 Miscelleanous

10.1 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 Glossary

CDF cumulative distribution function

PDF probability density function

dPDF derivative (gradient) of probability density function

PMF probability mass function

PV (finite) probability vector

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 B 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

[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. Hoermann (1995). *A rejection technique for sampling from T-concave distributions*, ACM Trans. Math. Software 21(2), pp. 182-193.

[HDa96] W. Hoermann and G. Derflinger (1996). *Rejection-inversion to generate variates from monotone discrete distributions*, ACM TOMACS 6(3), 169-184.

[HLa00] W. Hoermann 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 (2000). *A simple universal generator for continuous and discrete univariate T-concave distributions*, ACM Trans. Math. Software, to appear.

[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. Hoermann 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.

Appendix C Function Index