This version: February 4, 2024

TestU01

A Software Library in ANSI C for Empirical Testing of Random Number Generators

User's guide, compact version

Pierre L'Ecuyer and Richard Simard

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal

This document describes the software library *TestU01*, implemented in the ANSI C language, and offering a collection of utilities for the (empirical) statistical testing of uniform random number generators (RNG).

The library implements several types of generators in generic form, as well as many specific generators proposed in the literature or found in widely-used software. It provides general implementations of the classical statistical tests for random number generators, as well as several others proposed in the literature, and some original ones. These tests can be applied to the generators predefined in the library and to user-defined generators. Specific tests suites for either sequences of uniform random numbers in $[0, 1]$ or bit sequences are also available. Basic tools for plotting vectors of points produced by generators are provided as well.

Additional software permits one to perform systematic studies of the interaction between a specific test and the structure of the point sets produced by a given family of RNGs. That is, for a given kind of test and a given class of RNGs, to determine how large should be the sample size of the test, as a function of the generator's period length, before the generator starts to fail the test systematically.

Copyright

Contents

Chapter 1

INTRODUCTION

1.1 Design and testing of random number generators

Random numbers generators (RNGs) are small computer programs whose purpose is to produce sequences of numbers that *seem to behave* as if they were generated randomly from a specified probability distribution. These numbers are sometimes called *pseudorandom numbers*, to underline the fact that they are not truly random. Here, we just call them random numbers, with the usual (slight) abuse of language. These RNGs are crucial ingredients for a whole range of computer usages, such as statistical experiments, simulation of stochastic systems, numerical analysis, probabilistic algorithms, cryptology, secure communications, computer games, and gambling machines, to name a few.

The numbers must be generated quickly and easily by a computer program that is small, simple, and deterministic, except for its initial state which can be selected at random. In some cases, certain parameters of the generator are also selected at random, and can be viewed as part of the state. The quality criteria for an RNG may depend on the application. For simulation, one usually asks for speed, small memory requirement, and good statistical properties. For cryptology-related applications and for gambling machines in casinos, unpredictability is a crucial requirement for which speed can be sacrificed up to a certain point.

RNGs should be designed and selected based on a solid *theoretical* analysis of their mathematical structure. Here, we suppose that the goal is that the successive output values of the RNG, say $u_0, u_1, u_2, \ldots$, imitate independent random variables from the uniform distribution over the interval $[0, 1]$ (i.i.d. $U[0, 1]$), or over the two-element set $\{0, 1\}$ (independent random bits). In both cases (independent uniforms or random bits) we shall denote the hypothesis of perfect behavior by $\mathcal{H}_0$. These two situations are strongly related, because under the i.i.d. $U[0, 1]$ hypothesis, any pre-specified sequence of bits (e.g., the bit sequence formed by taking all successive bits of u_0, or every second bit, or the first five bits of each u_i, etc.) must be a sequence of independent random bits. So statistical tests for bit sequences can be used

as well (indirectly) for testing the null hypothesis in the first situation. In the remainder of this document, unless specified otherwise, $\mathcal{H}_0$ refers to this first situation.

In the $U[0,1]$ case, $\mathcal{H}_0$ is equivalent to saying that for each integer $t > 0$, the vector $(u_0, \ldots, u_{t-1})$ is uniformly distributed over the t-dimensional unit cube $[0,1]^t$. Clearly, this cannot be formally true, because these vectors always take their values only from the *finite* set Ψ_t of all t-dimensional vectors of t successive values that can be produced by the generator, from all its possible initial states (or seeds). The cardinality of this set cannot exceed the number of admissible seeds for the RNG. Assuming that the seed is chosen at random, vectors are actually generated over Ψ_t to approximate the uniform distribution over $[0,1]^t$. This suggests that Ψ_t should be very evenly distributed over the unit cube. Theoretical figures of merit for measuring this uniformity are discussed, e.g., in [90, 80, 93, 129, 158] and the references given there.

In the case of a sequence of random bits, the null hypothesis $\mathcal{H}_0$ cannot be formally true as soon as the length t of the sequence exceeds the number b of bits in the generator's state, for the number of distinct sequences of bits that can be produced cannot exceed 2^b. For $b < t$, the fraction of all sequences that can be visited is at most 2^{b-t}. The goal, then, is to make sure that those sequences that can be visited are "uniformly scattered" in the set of all 2^t possible sequences, and perhaps hard to distinguish.

Cryptologists use different quality criteria for RNGs. Their main concern is *unpredictability* of the forthcoming numbers. Their theoretical analysis of RNGs is usually asymptotic, in the framework of computational complexity theory [66, 69].

Once an RNG has been designed and implemented, based on some mathematical analysis of its structure, it is usually submitted to *empirical statistical tests* that try to detect statistical deficiencies by looking for empirical evidence against the hypothesis $\mathcal{H}_0$ introduced previously. A test is defined by a test statistic Y, which is a function of a finite number of u_n's (or a finite number of bits, in the case of bit generators), whose distribution under $\mathcal{H}_0$ is known (sometimes approximately). The number of different tests that can be defined is infinite and these different tests detect different problems with the RNGs. There is no universal test or battery of tests that can guarantee, when passed, that a given generator is fully reliable for all kinds of simulations. Passing many tests improves one's confidence in the RNG, although it never *proves* that the RNG is foolproof. In fact, no RNG can pass every conceivable statistical test. One could say that a *bad* RNG is one that fails *simple* tests, and a *good* RNG is one that fails only very complicated tests that are extremely hard to find or impractical to run.

Ideally, Y should mimic the random variable of practical interest in such a way that a bad structural interference between the RNG and the problem will show up in the test. But this is rarely practical. This cannot be done, for example, for testing RNGs for general-purpose software packages.

Experience with empirical testing tells us that RNGs with very long periods, good structure of their set Ψ_t, and based on recurrences that are not too simplistic, pass most reasonable

tests, whereas RNGs with short periods or bad structures are usually easy to crack by standard statistical tests. The simple structure that makes certain classes of generators very fast is also (often) the source of their major statistical deficiencies, which sometimes lead to totally wrong simulation results [13, 96, 87, 93, 83, 37, 160]. Practical tools for detecting these deficiencies are needed. Offering a rich variety of empirical tests for doing that is the purpose of the TestU01 library.

Some authors suggest that statistical tests should be used to identify and discard what they call *bad subsequences* from the output sequence of random number generators. We do not believe that this is a good idea. Such surgical procedures that cut out particular subsequences based on statistical test results would tend to remove some of the natural variability in the sequence, yielding a sequence that may lack some of the randomness properties of typical random sequences. Typically, when a generator fails a test decisively (e.g., with a significance level or p-value less than 10^{-15}, for example), it fails in pretty much the same way for all its subsequences of a given length. This is because failure typically depends on the structure of the point set Ψ_t. There are exceptions, but they are not frequent. Moreover, when a generator starts failing a test decisively, the p-value of the test usually converges to 0 or 1 exponentially fast as a function of the sample size when the sample size is increased further.

1.2 Organization of the library

The software tools of TestU01 are organized in four classes of modules: those implementing RNGs, those implementing statistical tests, those implementing pre-defined batteries of tests, and those implementing tools for applying tests to entire families of generators. The names of the modules in those four classes start with the letters `u`, `s`, `b`, and `f`, respectively, and we shall refer to them as the `u`, `s`, `b`, and `f` modules. The name of every public identifier (type, variable, function, ...) is prefixed by the name of the module to which it belongs. Chapters 2 to 5 of this guide describe these four classes of modules and give some examples. Some of these modules use definitions and functions from the ANSI C libraries MyLib and ProbDist [92, 94], also developed in our laboratory. Several platform-dependent switches are collected in module `gdef` of MyLib. They must be set to appropriate values, compatible with the environment in which TestU01 is running (see the installation notes, in file `README`).

1.2.1 The generator implementations

The module `unif01` provides the basic tools for defining and manipulating uniform RNGs. It contains the type `unif01_Gen`, which implements the definition of an arbitrary RNG object. Every RNG intrinsic to this package is of this type. Functions are also available to write the current state of a generator, to filter its output in different ways (e.g., combining successive values in the sequence to produce an output with more bits of precision, or taking

non-successive values, or selecting only specific bits from the output, etc.), to combine two or more generators and to test the speed of different generators.

One can create an arbitrary number of RNGs of a given kind or of different kinds in the same program, with the exception of a few specific RNG's that were programmed directly in C by their authors, and which use global variables. For the latter, only one copy of a particular generator can be in use at any given time, and this is indicated in the documentation of these specific RNG's. For example, one could use 3 LCGs with different parameters in the same program; each has its own private set of variables that does not interfere with the state or the parameters of the other two. Additional kinds of generators can be defined by the user if needed, by implementing functions that construct objects of type `unif01_Gen`.

The other `u` modules implement RNGs and offer functions of the form `u..._Create...` that return a generator object which can be used as a source of random numbers, and to which tests can be applied. A dummy generator that just reads numbers from a file, either in text or in binary format, is also available in module `ufile`. There are functions in module `unif01` that makes it very easy for the user to test his own generator or an external generator that is not pre-programmed in TestU01.

It is important to underline that most of the RNG implementations given here are *not* intended for direct use in simulation or other similar purposes. Other RNG packages, based on robust generators and with multiple streams and other convenient facilities, have been designed for that [84, 95]. The purpose of the RNG implementations provided here is essentially for empirical testing and experimentation with variants, combinations, etc.

1.2.2 The statistical tests

The statistical tests are implemented in the `s` modules, whose names start by `s`. They all test one of the two null hypotheses $\mathcal{H}_0$ defined previously, using different test statistics. To apply a test to a specific generator, the generator must first be created by the appropriate `Create` function in a `u` module, then it must be passed as a parameter to the function implementing the appropriate test. The test results are printed automatically to the standard output, with a level of detail that can be selected by the user (see module `swrite`).

It is also possible to recover information about what has happened in the tests, via data structures specific to each type of test. These data stuctures, if they are to be used outside of a test, must always be created by calling the appropriate `s..._Create...` function. They are described only in the *detailed* version of this user's guide. This could be used, for example, to examine or post-process the results of a test. There are also a few public functions that do not appear even in the *detailed* version of this guide. They are hidden since they will be useful only when developing new tests or modifying existing ones.

The testing procedures use several functions from the library ProbDist [94]. In particular, they use `statcoll` from that library to collect statistical observations, and `gofw` to apply the goodness-of-fit (GOF) tests.

The module `scatter` does not apply statistical tests per se, but permits one to draw scatter plots for vectors of points returned by a generator.

1.2.3 Batteries of tests

Many users find it convenient to have predefined suites (or batteries) of more or less standard statistical tests, with fixed parameters, that can be applied to a given RNG. Different types of tests should be included in such a battery, in order to detect different types of weaknesses in a given generator.

A number of predefined batteries of tests, some oriented towards sequences of uniform floating-point numbers in the interval $[0, 1)$, others towards sequences of bits, are available in TestU01. There are small batteries, that run quickly, and larger (more stringent) batteries that take longer to run. These batteries are implemented in the `b` modules.

1.2.4 Tools for testing families of generators

The `f` modules provide a set of tools, built on top of the modules that implement the generator families and the tests, designed to perform systematic studies of the interaction between certain types of tests and the structure of the point sets produced by given families of RNGs. Roughly, the idea is to see at which sample size n_0 the test starts to reject the RNG decisively, as a function of its period length ρ. In experiments already performed with certain classes of generators and specific tests [90, 87, 93], the results were often surprisingly regular, in the sense that a regression model of the form $\log n_0 = a \log \rho + \epsilon$, where a is a constant and ϵ a small noise, fits very well.

1.3 History and implementation notes

TestU01 started as a Pascal program implementing the tests suggested in the 1981 edition of volume 2 of "The Art of Computer Programming" [64]. This was around 1985. Three or four years later, a Modula-2 implementation was made, in the form of a library with a modular design. Other tests were added, as well as some generators implemented in generic form. Between 1990 and 2001, new generators and new tests were added regularly to the library and a detailed user's guide (in french) was kept up to date. The `f` modules, which contain tools for testing entire families of generators, were introduced in 1997, while the first author was on sabbatical at the University of Salzburg, Austria. In 2001 and 2002, we partially redesigned the library, translated it in the C language, and translated the user's guide in english.

These preliminary versions of the library were used for several articles (co)authored by P. L'Ecuyer, starting from his 1986 paper where he first proposed a combined LCG [71], and including [72, 74, 84, 78, 90, 96, 79, 91, 87, 93, 89, 83].

1.4 Other software for testing RNGs

Another well-known public-domain testing package for RNGs is DIEHARD [106]. It contains a large number of statistical tests. However, it has some drawbacks and limitations. Firstly, the sequence of tests to be applied to any generator, as well as the parameters of these tests (sample size, etc.) are fixed in the package. The sample sizes are moderate; all these tests run in a few seconds of CPU time on a desktop computer. For example, on a PC with an Athlon XP 2100+ processor at 1733 MHz and running Linux, the entire series of tests take approximately 12 seconds to run. Secondly, the package requires that the random numbers to be tested are 32-bit integers, placed in a huge file in binary format. This file is passed to the testing procedures. This setup is not always convenient. Many RNGs produce numbers with less than 32 bits of resolution (e.g., 31 bits is frequent) and DIEHARD does not care for that. TestU01 is more flexible on all these aspects.

The SPRNG library [120] is another public-domain software that implements the classical tests for RNGs given in [66], plus a few others. The National Institute of Standards and Technology (NIST), in the USA, has implemented a test suite (16 tests) for RNGs, mainly for the testing and certification of RNGs used in cryptographic applications (see [152] and `http://csrc.nist.gov/rng/`). The ENT test program (`http://www.fourmilab.ch/random/`) implements a few elementary tests. The Information Security Research Center, in Australia, offers a commercial testing package called Crypt-X, which contains a test suite designed for stream ciphers, block ciphers, and key generators used in cryptology (see `http://www.isrc.qut.edu.au/resource/cryptx/`). The GNU scientific library `gsl`, currently under development (see `https://www.gnu.org/software/gsl/manual/html_node/`), implements a large set of well-tested RNGs, but so far no statistical test per se.

Chapter 2

UNIFORM GENERATORS

This chapter contains a description of various uniform generators already programmed in this library and which were proposed by various authors over the past several years, as well as tools for managing and implementing additional types of generators. Related generators are regrouped in the same module. For example, the linear congruential generators (LCGs) are in module `ulcg`, the multiple recursive generators (MRGs) are in `umrg`, the inversive generators in `uinv`, the cubic generators in `ucubic`, etc. We emphasize that the generators provided here are not all recommendable; in fact, *most of them are not*.

The module `unif01` contains the basic utilities for defining, manipulating, filtering, combining, and timing generators. Each generator must be implemented as an object of type `unif01_Gen`. To implement one's own generator, one should create such an object and define all its fields. For each generator, the structure `unif01_Gen` must contain a function `GetU01` that returns values in the interval $[0, 1)$ and a function `GetBits` that returns a block of 32 bits. Most of the tests in the `s` modules call the generators to be tested only indirectly, through the use of the interface functions `unif01_StripD`, `unif01_StripL` and `unif01_StripB`. These functions drop the r most significant bits of each random number generated and returns a number built out of the remaining bits.

It is also possible to test one's own or an external generator (that is, a generator that is not predefined in TestU01) very easily with the help of the functions `unif01_CreateExternGen01` and `unif01_CreateExternGenBits` (see page 15 of this guide), as long as this generator is programmed in C.

Figure 2.1 gives simple examples of how to use predefined generators. The program creates a LCG with modulus $m = 2^{31} - 1$, multiplier $a = 16807$, and initial state $s = 12345$, generates and adds 100 uniforms produced by this generator, prints the sum, and deletes the generator. To illustrate the fact that there are different ways of getting the uniforms from the generator, we have generated the first 50 by calling the `GetU01` function and the next 50 via `unif01_StripD`. These two methods are equivalent. The program then instantiates the generator `lfsr113` available in module `ulec`, with the vector $(12345, \ldots, 12345)$ as initial seed, generates and prints five integers in the range $\{0, \ldots, 2^{10} - 1\}$ (i.e., 10-bit integers)

obtained by taking five successive output values from the generator, stripping out the four most significant bits from each value, and retaining the next 10 bits.

For each public identifier used in programs, it is important to include the corresponding header file before using it, so as to inform the compiler about the type and signature of functions and exported variables. For instance, in the following examples, the header files `unif01.h`, `ulcg.h` and `ulec.h` contain the declarations of `unif01_Gen`, `ulcg_CreateLCG` and `ulec_Createlfsr113`, respectively.

Other examples on how to use the facilities of module `unif01` are given at the end of its description.

```
#include <testu01/unif01.h>
#include <testu01/ulcg.h>
#include <testu01/ulec.h>
#include <stdio.h>

int main (void)
{
   int i;
   double x;
   unsigned long z;
   unif01_Gen *gen;

   gen = ulcg_CreateLCG (2147483647, 16807, 0, 12345);
   x = 0.0;
   for (i = 0; i < 50; i++)
      x += gen->GetU01(gen->param, gen->state);
   for (i = 0; i < 50; i++)
      x += unif01_StripD (gen, 0);
   printf ("Sum = %14.10f\n\n", x);
   ulcg_DeleteGen (gen);

   gen = ulec_Createlfsr113 (12345, 12345, 12345, 12345);
   for (i = 0; i < 5; i++) {
      z = unif01_StripB (gen, 4, 10);
      printf ("%10lu\n", z);
      }
   ulec_DeleteGen (gen);
   return 0;
}
```

Figure 2.1: Using pre-programmed generators

unif01

This module offers basic tools for defining, manipulating, and transforming uniform random number generators to which tests are to be applied or which could be used for other purposes. Each generator is implemented as a structure of type `unif01_Gen`. Several predefined generators are available in the `u` modules. Each such generator must be created by the appropriate `...Create...` function before being used, and should be deleted by the corresponding `...Delete...` function to free the memory used by the generator when it is no longer needed. One can create and use simultaneously any number of generators. These generators are usually passed to functions as pointers to objects of type `unif01_Gen`.

One may call an external generator for testing using the functions in this module. See Figure 2.2 for an example. One may also implement one's own generator, by creating a structure of type `unif01_Gen` and defining all its fields properly. See Figure 2.5 for an illustration.

Each implemented generator returns either a floating-point number in $[0, 1)$ (via its function `GetU01`) or a block of 32 bits (via its function `GetBits`). Ideally, these should follow the uniform distribution $(0, 1)$ and $\{0, \ldots, 2^{32} - 1\}$, respectively. Most of the tests in the `s` modules actually call the generator to be tested only indirectly through the use of one of the interface functions `unif01_StripD`, `unif01_StripL` and `unif01_StripB`. These functions drop the r most significant bits of each random number and return a number built out of the remaining bits.

Functions are also provided for adding one or many output *filters* to a given generator. These functions create another generator object which implements a mechanism that automatically transforms the output values of the original generator in a specified way. One can also combine the outputs of several generators in different ways. By using the output of several generators or several substreams of the same generator in a round-robin way, one can test the quality of these as examples of parallel generators. Finally, tools are provided for measuring the speed of generators and adding their output values (for testing purposes).

```
#include <testu01/gdef.h>
```

Basic types

```
typedef struct {
void *state;
void *param;
char *name;
double (*GetU01) (void *param, void *state);
unsigned long (*GetBits) (void *param, void *state);
void (*Write) (void *state);
} unif01_Gen;
```

Generic random number generator. The function `GetU01` returns a floating-point number in $[0, 1)$ while `GetBits` returns a block of 32 bits. If the generator delivers less than 32 bits, these

bits are left shifted so that the most significant bits are the relevant ones. The variable `state` keeps the current state of the generator and `param` is the set of specific parameters used in computing the next random number. The function `Write` will write the current state of the generator. The string `name` describes the current generator, its parameters, and its initial state. In the description of the generators in the u modules, one indicates how the `GetU01` function gets its value from the generator's recurrence; it is always understood that the `GetBits` function is equivalent to 2^{32} `GetU01`.

Environment variables

```
extern lebool unif01_WrLongStateFlag;
```

For generators whose state is a large array, determines whether the state will be written out in full (`TRUE`) or not (`FALSE`) in the printouts. The default value is `FALSE`.

Basic functions

```
double unif01_StripD (unif01_Gen *gen, int r);
```

Makes one call to the generator `gen`, drops the r most significant bits, left-shift the others by r positions, and returns the result, which is a floating-point number in $[0, 1)$. More specifically, returns $2^r u \bmod 1$, where u is the output of `gen`.

```
long unif01_StripL (unif01_Gen *gen, int r, long d);
```

Similar to `unif01_StripD`, but generates an integer "uniformly" over the set $\{0, \ldots, d-1\}$, by using the most significant bits of the output of `gen` after having dropped the first r bits. More specifically, returns $\lfloor d(2^r u \bmod 1) \rfloor$, where u is the output of `gen`.

```
unsigned long unif01_StripB (unif01_Gen *gen, int r, int s);
```

Calls the generator `gen`, drops the r most significant bits, and returns the s following bits as an integer in the set $\{0, \ldots, 2^s - 1\}$.

```
void unif01_WriteNameGen (unif01_Gen *gen);
```

Writes the character string `gen->name` that describes the generator.

```
void unif01_WriteState (unif01_Gen *gen);
```

Writes the current state of generator `gen`.

`void unif01_WrLongStateDef (void);`

Dummy function used when the state of the current generator is a large array and we do not want to write the full state. Writes the message "`Not shown here ... takes too much space`".

`unif01_Gen * unif01_CreateDummyGen (void);`

Creates a *dummy* generator, which does nothing and always returns zero. It can be used for instance to measure the overhead of function calls when comparing generator's speeds (see the timing tools below).

`void unif01_DeleteDummyGen (unif01_Gen *gen);`

Frees the dynamic memory used by the dummy generator above.

Output filters

The following describes some filters that can be added to transform the output of a given generator. In each case, a new generator object is created that will effectively apply the filter to the original generator. One may apply more than one filter at a time on a given generator (for example, one may apply the `Double`, the `Bias`, the `Trunc` and the `Lac` filters on top of one another). It suffices to create the appropriate filters as described below. The resulting filtered generator(s) will call the original generator behind the scenes. Thus the state of the original generator will evolve as usual, even though it is not called directly.

The different filters applied on an original generator are not independent but are related as the elements of a stack. When they are no longer in use, they must be deleted *in the reverse order of their creation*, the original generator being the last one of this group to be deleted. Figure 2.8 illustrates how these facilities can be used.

`unif01_Gen * unif01_CreateDoubleGen (unif01_Gen *gen, int s);`

Given a generator `gen`, this function creates and returns a generator with increased precision, such that every call to this new generator corresponds to two successive calls to the original generator. The method `GetU01` of this doubled generator returns $(U_1 + U_2/2^s) \bmod 1$, where U_1 and U_2 are the results of two successive calls to the method `GetU01` of `gen`. If the current generator has 31 bits of precision, for example, then one can obtain 53 bits of precision from `GetU01` by creating this new generator with `s` between 22 and 31.

`unif01_Gen * unif01_CreateDoubleGen2 (unif01_Gen *gen, double h);`

A more general version of `unif01_CreateDoubleGen` where the method `GetU01` of the double generator returns $(U_1 + hU_2) \bmod 1$. Restriction: $0 < h < 1$.

`unif01_Gen * unif01_CreateLacGen (unif01_Gen *gen, int k, long I[]);`

Given an original generator `gen`, this function creates and returns a generator involving lacunary indices, such that successive calls to this new generator will no longer provide successive values from the original generator, but rather selected values as specified by the table `I[0..k-1]`, in a circular fashion. More specifically, if $u_0, u_1, u_2, \ldots$ is the sequence produced by the original `gen`, if the table `I[0..k-1]` contains the non-negative integers $i_0, \ldots i_{k-1}$ (in increasing order), and if we put $L = i_{k-1} + 1$, then the output sequence of the new generator will be:

$$u_{i_0}, u_{i_1}, \ldots, u_{i_{k-1}}, u_{L+i_0}, u_{L+i_1}, \ldots, u_{L+i_{k-1}}, u_{2L+i_0}, u_{2L+i_1}, \ldots.$$

For example, if $k = 3$ and $I = \{0, 3, 5\}$, the output sequence will be the numbers

$$u_0, u_3, u_5, u_6, u_9, u_{11}, u_{12}, \ldots$$

of the original generator. To obtain every s-th number produced by the original generator for example (a *decimated sequence*), one should take $k = 1$ and $I = \{s - 1\}$.

`unif01_Gen * unif01_CreateLuxGen (unif01_Gen *gen, int k, int L);`

Given an original generator `gen`, this function creates and returns a new generator giving the output of the original generator with luxury level L: out of every group of L random numbers, the first k are kept and the next $L - k$ are skipped.

`unif01_Gen * unif01_CreateBiasGen (unif01_Gen *gen, double a, double p);`

Given an original generator `gen`, this function creates and returns a new generator giving a biased output of the original generator. The output is biased in such a way that the density becomes constant with total probability p over the interval $[0, a)$, and constant with total probability $1 - p$ over $[a, 1)$ (the two constant densities are different). For example, by choosing $p = 1$ and $a = 0.5$, all the random numbers generated by `GetU01` will fall on the interval [0, 0.5). This filter can be used, for example, to study the power of certain statistical tests. Restrictions: $0 < a < 1$ and $0 \leq p \leq 1$.

`unif01_Gen * unif01_CreateTruncGen (unif01_Gen *gen, int s);`

Given an original generator `gen`, this function creates and returns a new generator giving the output of the original generator truncated to its s most significant bits. Restriction: $s \leq 32$.

`unif01_Gen * unif01_CreateBitBlockGen (unif01_Gen *gen, int r, int s, int w);`

Consider a group of $v \leq 32$ successive 32-bit integers outputted by generator `gen`. For each of these, drop the r most significant bits and keep the s following bits numbered $b_{i1}, b_{i2}, \ldots, b_{is}$, starting with the most significant, for $1 \leq i \leq v$. Make with all these a $v \times s$ matrix of bits, say $\mathcal{B}$. The generator returned by this function is a filter that builds new 32-bit integers from $v \times w$ submatrices of $\mathcal{B}$. The number of columns of the submatrix w must be a power of 2 no

larger than 32 and it must be $\leq s$. If w does not divide s exactly, the last submatrix of $\mathcal{B}$ will have less than w columns and will be disregarded.

If the stream of bits thus obtained from `gen` is

$$b_{11}, b_{12}, \ldots, b_{1s}, b_{21}, b_{22}, \ldots, b_{2s}, \ldots, b_{v1}, b_{v2}, \ldots, b_{vs}, \ldots$$

then the new integers returned by the filter will be 32-bit integers taken from the rearranged stream of bits so that the first new number is (its most significant bit being given first)

$$b_{11}, b_{12}, \ldots, b_{1w}, b_{21}, b_{22}, \ldots, b_{2w}, \ldots, b_{v1}, b_{v2}, \ldots, b_{vw},$$

the second new number is made of the bits (its most significant bit first)

$$b_{1(w+1)}, b_{1(w+2)}, \ldots, b_{1(2w)}, b_{2(w+1)}, b_{2(w+2)}, \ldots, b_{2(2w)}, \ldots, b_{v(w+1)}, b_{v(w+2)}, \ldots, b_{v(2w)},$$

and so on.

The following examples illustrates how the filter works. If $r = 0$ and $w = s = 32$, then the filter has no effect, the new integers being the same as those outputted by `gen`. If $r = 0$ and $w = s = 1$, then the filter will return integers made only from the most significant bit of the original integers, all other bits being dropped. If $r = 0$, $w = 1$ and $s = 32$, then the filter will return integers made from the columns of $\mathcal{B}$, i.e., since the rows of $\mathcal{B}$ are made of the original integers, the filter will return the columns of $\mathcal{B}$ as the new integers. Restrictions: $r \geq 0$, $0 < s \leq 32$ and w in $\{1, 2, 4, 8, 16, 32\}$.

```
void unif01_DeleteDoubleGen (unif01_Gen *gen);
void unif01_DeleteLacGen    (unif01_Gen *gen);
void unif01_DeleteLuxGen    (unif01_Gen *gen);
void unif01_DeleteBiasGen   (unif01_Gen *gen);
void unif01_DeleteTruncGen  (unif01_Gen *gen);
void unif01_DeleteBitBlockGen (unif01_Gen *gen);
```

Frees the memory used by the generator created by the corresponding `Create` functions above.

Combining generators

These functions permit one to define the combination of two, three or more generators. The resulting generator calls the component generators behind the scenes, so it changes their state. *The component generators must not be destroyed as long as the combination generator is in use.* One can obtain the combinations of more than three generators by combining the generators obtained from combinations of two or three generators.

```
unif01_Gen * unif01_CreateCombAdd2 (unif01_Gen *gen1, unif01_Gen *gen2,
char *name);
```

This function creates and returns a generator whose output is the addition of the outputs modulo 1 of the method `GetU01` of the two generators `gen1` and `gen2`. The character string `name` may be printed in reports to identify this new combined generator.

```
unif01_Gen * unif01_CreateCombAdd3 (unif01_Gen *gen1, unif01_Gen *gen2,
unif01_Gen *gen3, char *name);
```

Same as `unif01_CreateCombAdd2`, except that the returned generator is the combination (the addition of the outputs modulo 1 of the method `GetU01`) of the three generators `gen1, gen2` and `gen3`.

```
unif01_Gen * unif01_CreateCombXor2 (unif01_Gen *gen1, unif01_Gen *gen2,
char *name);
```

This function creates and returns a generator whose output is the bitwise *exclusive-or (XOR)* of the outputs of the two generators `gen1` and `gen2`. The character string `name` may be printed in reports to identify this combined generator.

```
unif01_Gen * unif01_CreateCombXor3 (unif01_Gen *gen1, unif01_Gen *gen2,
unif01_Gen *gen3, char *name);
```

Same as `unif01_CreateCombXor2`, except that the returned generator is the combination of the three generators `gen1, gen2` and `gen3`.

```
void unif01_DeleteCombGen (unif01_Gen *gen);
```

Frees the memory used by one of the combination generators returned by the `Create` functions above, but does not delete any of its component generators.

Parallel generators

The following functions allow the joining of the output of several generators or of different substreams of the same generator into a single stream of random numbers. This can be used to test for apparent correlations between the output of several generators or several substreams used in parallel. For example, one may want to choose seeds that are far separated for the same generator, while making sure that such seed choice is statistically valid and does not introduce unwanted correlation between the substreams thus defined.

```
unif01_Gen * unif01_CreateParallelGen (int k, unif01_Gen *gen[], int L);
```

Creates and returns a generator whose output is obtained in a round-robin way L numbers at a time from each of the k generators `gen[i]` as follows: the first L numbers are generated from `gen[0]`, the next L numbers are generated from `gen[1]`, and so on until L numbers have been generated from `gen[k-1]`, after which, this whole process is repeated. *It is important that none of the generators* `gen[i]` *be destroyed as long as the parallel generator is in use.*

```
void unif01_DeleteParallelGen (unif01_Gen *gen);
```

Frees the memory allocated by the parallel generator returned by the `Create` function above, but *does not* delete any of its component generators, which is the responsibility of the program that created them.

External generators

Although TestU01 implements many generators both in generic and in specific forms, it is not possible to implement all those that are in existence because there are just too many and new ones are proposed regularly. The typical user would like to test his preferred generator with as little complications as possible. The functions below allows one to do just that. As long as the generator is programmed in C, one has but to pass the function implementing the generator to one of the functions below and call some of the tests available in TestU01. It is the responsibility of the user to ensure that his generator does not violate the conditions described in the functions below. For the call in `unif01_CreateExternGen01`, his generator must return floating-point numbers in $[0, 1)$. For the calls in `unif01_CreateExternGenBitsL` and `unif01_CreateExternGenBits`, his generator must return an integer in the interval $[0, 2^{32} - 1]$. If these conditions are violated, the results of the tests in TestU01 are unpredictable.

```
unif01_Gen *unif01_CreateExternGen01 (char *name, double (*gen01)(void));
```

Implements a pre-existing external generator `gen01` that is not part of TestU01. It must be a C function taking no argument and returning a `double` in the interval $[0, 1)$. Parameter `name` is the name of the generator. No more than one generator of this type can be in use at a time.

```
unif01_Gen *unif01_CreateExternGenBits (char *name,
unsigned int (*genB)(void));
```

Implements a pre-existing external generator `genB` that is not part of TestU01. It must be a C function taking no argument and returning an integer in the interval $[0, 2^{32} - 1]$. If the generator delivers less than 32 bits of resolution, then these bits must be left shifted so that the most significant bit is bit 31 (counting from 0). Parameter `name` is the name of the generator. No more than one generator of this type can be in use at a time.

```
unif01_Gen *unif01_CreateExternGenBitsL (char *name,
unsigned long (*genB)(void));
```

Similar to `unif01_CreateExternGenBits`, but with `unsigned long` instead of `unsigned int`. The generator `genB` must also return an integer in the interval $[0, 2^{32} - 1]$.

```
void unif01_DeleteExternGen01 (unif01_Gen * gen);
void unif01_DeleteExternGenBits (unif01_Gen * gen);
void unif01_DeleteExternGenBitsL (unif01_Gen * gen);
```

Frees the memory used by the generator created by the corresponding `Create` functions above.

As an example, Figure 2.2 shows how to apply `SmallCrush`, a small predefined battery of tests (described on page 143) to the generators `MRG32k3a` and `xorshift`, whose code is shown in Figures 2.3 and 2.4. One must compile and link the two external files with the main program and the TestU01 library. The generator `MRG32k3a` returns numbers in (0, 1)

and was proposed by L'Ecuyer in [80]. The generator `xorshift` returns 32-bit integers and was proposed by Marsaglia in [111, page 4].

```
#include <testu01/unif01.h>
#include <testu01/bbattery.h>

unsigned int xorshift (void);
double MRG32k3a (void);

int main (void)
{
   unif01_Gen *gen;

   gen = unif01_CreateExternGen01 ("MRG32k3a", MRG32k3a);
   bbattery_SmallCrush (gen);
   unif01_DeleteExternGen01 (gen);

   gen = unif01_CreateExternGenBits ("xorshift", xorshift);
   bbattery_SmallCrush (gen);
   unif01_DeleteExternGenBits (gen);

   return 0;
}
```

Figure 2.2: Example of a program to test two external generators

Timing devices

```
typedef struct {
unif01_Gen *gen;
long n;
double time;
double mean;
lebool fU01;
} unif01_TimerRec;
```

Structure to memorize the results of speed and sum tests on a given generator. Here, `gen` is the generator, `n` is the number of calls made to the generator, `time` is the total CPU time in seconds, and `mean` is the mean of the `n` output values of the generator. If `fU01` is `TRUE`, the function `GetU01` of `gen` is called, otherwise the function `GetBits` is called.

```
void unif01_TimerGen (unif01_Gen *gen, unif01_TimerRec *timer, long n,
lebool fU01);
```

This function computes the CPU time needed to generate `n` random numbers with the generator `gen`, and returns the result in `timer`. If `fU01` is `TRUE`, the random numbers will be generated by the method `GetU01` of `gen`, otherwise by the method `GetBits`.

```
#define norm 2.328306549295728e-10
#define m1   4294967087.0
#define m2   4294944443.0
#define a12     1403580.0
#define a13n     810728.0
#define a21      527612.0
#define a23n    1370589.0

static double s10 = 12345, s11 = 12345, s12 = 123,
              s20 = 12345, s21 = 12345, s22 = 123;

double MRG32k3a (void)
{
   long   k;
   double p1, p2;
   /* Component 1 */
   p1 = a12 * s11 - a13n * s10;
   k = p1 / m1;   p1 -= k * m1;   if (p1 < 0.0) p1 += m1;
   s10 = s11;   s11 = s12;   s12 = p1;

   /* Component 2 */
   p2 = a21 * s22 - a23n * s20;
   k  = p2 / m2;  p2 -= k * m2;   if (p2 < 0.0) p2 += m2;
   s20 = s21;   s21 = s22;   s22 = p2;

   /* Combination */
   if (p1 <= p2) return ((p1 - p2 + m1) * norm);
   else return ((p1 - p2) * norm);
}
```

Figure 2.3: External function for `MRG32k3a`.

```
static unsigned int y = 2463534242U;

unsigned int xorshift (void)
{
   y ^= (y << 13);
   y ^= (y >> 17);
   return y ^= (y << 5);
}
```

Figure 2.4: External function for `xorshift`.

```
void unif01_TimerSumGen (unif01_Gen *gen, unif01_TimerRec *timer, long n,
lebool fU01);
```

Same as `unif01_TimerGen`, but also adds the `n` random numbers and saves their mean in `timer->mean`.

```
void unif01_WriteTimerRec (unif01_TimerRec *timer);
```

Prints the results contained in `timer`, with some information about the generator and the current machine. One should make sure that the generator `gen` in `timer` has not been deleted when calling this function.

```
void unif01_TimerGenWr (unif01_Gen *gen, long n, lebool fU01);
```

Equivalent to calling `unif_TimerGen` followed by `unif01_WriteTimerRec`.

```
void unif01_TimerSumGenWr (unif01_Gen *gen, long n, lebool fU01);
```

Equivalent to calling `unif_TimerSumGen` followed by `unif01_WriteTimerRec`.

Examples

We now provide some examples of how to use the facilities of `unif01`. Figure 2.5 gives an example of how to implement one's own generator, using all the paraphernalia of TestU01. This is specially useful when one wants to implement a generator in generic form with one or more parameters. This is a simple LCG with hardcoded parameters $m = 2^{31} - 1$ and $a = 16807$. The function `My16807_U01` will advance the generator's state by one step and return a $U(0, 1)$ random number U each time it is called, whereas `My16807_Bits` will return the 32 most significant bits in the binary representation of U. The function `CreateMy16807` allocates the memory for the corresponding `unif01_Gen` structure and initializes all its fields.

Figure 2.6 shows how to use the timing facilities. The `main` program first sets the generator `gen` to an LCG with modulus $2^{31} - 1$, multiplier $a = 16807$, and initial state 12345, implemented in floating point. (This generator is well known, but certainly *not* to be recommended; its period length of $2^{31} - 2$ is much too small.) The program calls `unif01_TimerSumGenWr` which generates 10 million random numbers in $[0, 1)$, computes their mean, and prints the CPU time needed to do that. Next, the program deletes this `unif01_Gen` object and creates a new one, which is actually a user-defined implementation of the same LCG, taken from the home-made module `my16807` whose code is shown in Figure 2.5. In this implementation, the parameters have been placed as constants directly into the code. Ten million random numbers are generated with this alternative implementation, and the average and CPU time are printed. The same procedure is repeated for two additional predefined generators taken from modules `ulec`. Figure 2.7 shows the results of this program, run on a 2106 MHz computer running Linux, and compiled with `gcc -O2`.

```
#include "my16807.h"
#include <testu01/unif01.h>
#include <testu01/util.h>
#include <testu01/addstr.h>
#include <string.h>

typedef struct { double S; } My16807_state;

static double My16807_U01 (void *par, void *sta)
{
   My16807_state *state = sta;
   long k;
   state->S *= 16807.0;
   k = state->S / 2147483647.0;
   state->S -= k * 2147483647.0;
   return (state->S * 4.656612875245797E-10);
}

static unsigned long My16807_Bits (void *par, void *sta)
{
   return (unsigned long) (My16807_U01 (par, sta) * 4294967296.0);
}

static void WrMy16807 (void *sta)
{
   My16807_state *state = sta;
   printf (" S = %.0f\n", state->S);
}

unif01_Gen *CreateMy16807 (int s)
{
   unif01_Gen *gen;
   My16807_state *state;
   size_t leng;
   char name[60];

   gen = util_Malloc (sizeof (unif01_Gen));
   gen->state = state = util_Malloc (sizeof (My16807_state));
   state->S = s;
   gen->param = NULL;
   gen->Write = WrMy16807;
   gen->GetU01 = My16807_U01;
   gen->GetBits = My16807_Bits;

   strcpy (name, "My LCG implementation for a = 16807:");
   addstr_Int (name, "   s = ", s);
   leng = strlen (name);
   gen->name = util_Calloc (leng + 1, sizeof (char));
   strncpy (gen->name, name, leng);
   return gen;
}

void DeleteMy16807 (unif01_Gen * gen)
{
   gen->state = util_Free (gen->state);
   gen->name = util_Free (gen->name);
   util_Free (gen);
}
```

Figure 2.5: A user-defined generator, in file `my16807.c`.

```
#include <testu01/unif01.h>
#include <testu01/ulcg.h>
#include <testu01/ulec.h>
#include "my16807.h"
#include <stdio.h>

int main (void)
{
   unif01_Gen *gen;
   double x = 0.0;
   int i;

   gen = ulcg_CreateLCGFloat (2147483647, 16807, 0, 12345);
   unif01_TimerSumGenWr (gen, 10000000, TRUE);
   ulcg_DeleteGen (gen);

   gen = CreateMy16807 (12345);
   unif01_TimerSumGenWr (gen, 10000000, TRUE);
   DeleteMy16807 (gen);

   gen = ulec_CreateMRG32k3a (123., 123., 123., 123., 123., 123.);
   unif01_TimerSumGenWr (gen, 10000000, TRUE);
   ulec_DeleteGen (gen);

   gen = ulec_Createlfsr113 (12345, 12345, 12345, 12345);
   unif01_TimerSumGenWr (gen, 10000000, TRUE);
   for (i = 0; i < 100; i++)
      x += unif01_StripD (gen, 0);
   printf ("Sum = %14.10f\n", x);
   ulec_DeleteGen (gen);

   return 0;
}
```

Figure 2.6: Example of a program creating and timing generators.

```
-------------  Results of speed test  ---------------

 Host:
 Generator:   ulcg_CreateLCGFloat
 Method:      GetU01
 Mean =       0.499974546727091
 Number of calls:  10000000
 Total CPU time:   0.28 sec


-------------  Results of speed test  ---------------

 Host:
 Generator:   My LCG implementation for a = 16807
 Method:      GetU01
 Mean =       0.499974546727091
 Number of calls:  10000000
 Total CPU time:   0.28 sec


-------------  Results of speed test  ---------------

 Host:
 Generator:   ulec_CreateMRG32k3a
 Method:      GetU01
 Mean =       0.500045268775809
 Number of calls:  10000000
 Total CPU time:   0.50 sec


-------------  Results of speed test  ---------------

 Host:
 Generator:   ulec_Createlfsr113
 Method:      GetU01
 Mean =       0.500154672454091
 Number of calls:  10000000
 Total CPU time:   0.10 sec

Sum =  50.6276649707
```

Figure 2.7: Results of the program of Figure 2.6.

Figure 2.8 shows how to apply filters to generators and how to combine two or more generators by addition modulo 1 or bitwise exclusive-or. The program starts by creating a simple Tausworthe generator `gen1` and it generates 20 values from it. It then deletes `gen1`, creates a new copy of it with the same parameters and initial state, and applies a "lacunary indices" filter to create a second generator `gen2`. The output sequence of `gen2` will be (in terms of the original sequence numbering) $u_3, u_7, u_9, u_{13}, u_{17}, u_{19}, u_{23}, \ldots$. Next, the program creates a generator `gen3` for which each output value is constructed from two successive output values of `gen2`, generates some values from `gen3` and `gen2`, and deletes them.

After that, the program creates another Tausworthe generator `gen2` and a generator `gen3` which is a combination of `gen1` and `gen2` by bitwise exclusive-or. It generates a few values with `gen3` and deletes all the generators.

```
#include <testu01/unif01.h>
#include <testu01/utaus.h>
#include <stdio.h>

int main (void)
{
   unif01_Gen *gen1, *gen2, *gen3;
   long I[3] = { 3, 7, 9 };
   int i, n = 20;
   double x;

   gen1 = utaus_CreateTaus (31, 3, 12, 12345);
   for (i = 0; i < n; i++)
      printf ("%f\n", unif01_StripD (gen1, 0));
   utaus_DeleteGen (gen1);
   printf ("\n");

   gen1 = utaus_CreateTaus (31, 3, 12, 12345);
   gen2 = unif01_CreateLacGen (gen1, 3, I);
   for (i = 0; i < n; i++)
      printf ("%f\n", unif01_StripD (gen2, 0));

   gen3 = unif01_CreateDoubleGen (gen2, 24);
   for (i = 0; i < n; i++)
      x = unif01_StripD (gen3, 0);
   unif01_DeleteDoubleGen (gen3);
   unif01_DeleteLacGen (gen2);

   gen2 = utaus_CreateTaus (28, 7, 14, 12345);
   gen3 = unif01_CreateCombXor2 (gen1, gen2, "A Combined Tausworthe Gener.");
   for (i = 0; i < n; i++)
      x = unif01_StripD (gen3, 0);
   unif01_DeleteCombGen (gen3);
   utaus_DeleteGen (gen2);
   utaus_DeleteGen (gen1);
   return 0;
}
```

Figure 2.8: Applying filters and combining generators.

ulcg

This module implements linear congruential generators (LCGs), simple or combined, in generic form. The simple LCG is defined by the recurrence

$$x_i = (ax_{i-1} + c) \mod m, \tag{2.1}$$

and the output at step i is $u_i = x_i/m$. Two types of combinations are implemented: the one proposed by L'Ecuyer [72], and the one proposed by Wichmann and Hill [171]. See [97] for details. Some of the implementations use the GNU multiprecision package GMP. The macro `USE_GMP` is defined in module `gdef` in directory `mylib`.

The following table gives specific parameters taken from the literature or from widely available software. See also [40, 81] for other LCG parameters. Parameters for combined LCGs can be found in [72, 97, 84].

Table 2.1: Some specific (popular) LCGs

m	a	c	Reference
2^{24}	1140671485	12820163	in Microsoft VisualBasic
$2^{31}-1$	742938285	0	[41]
$2^{31}-1$	950706376	0	[41]
$2^{31}-1$	630360016	0	[70, 134]
$2^{31}-1$	397204094	0	in SAS [149]
$2^{31}-1$	16807	0	[101, 6, 70, 133]
$2^{31}-1$	45991	0	[88]
2^{31}	65539	0	RANDU [58, 70]
2^{31}	134775813	1	in Turbo Pascal
2^{31}	1103515245	12345	`rand()` in BSD ANSI C
2^{31}	452807053	0	[58, URN11]
2^{32}	1099087573	0	[39]
2^{32}	4028795517	0	[39]
2^{32}	663608941	0	[58, URN13]
2^{32}	69069	0	component of original SuperDuper
2^{32}	69069	1	on VAX/VMS [58, URN22]
2^{32}	2147001325	715136305	in BCLP language
2^{35}	5^{13}	0	Apple
2^{35}	5^{15}	7261067085	[64, p.102]
$10^{12}-11$	427419669081	0	`rand()` in Maple 9.5 or earlier
$2^{47}-115$	71971110957370	0	[85]

m	a	c	Reference
$2^{47} - 115$	-10018789	0	[85]
2^{48}	68909602460261	0	[39]
2^{48}	25214903917	11	Unix's `rand48()`
2^{48}	44485709377909	0	on CRAY system [20]
2^{59}	13^{13}	0	in NAG Fortran/C library
$2^{63} - 25$	2307085864	0	[85]
2^{64}	11^{13}	c	`prng` at Cornell Theory Center [135]

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

Simple LCGs

```
unif01_Gen * ulcg_CreateLCG (long m, long a, long c, long s);
```

Initializes a LCG of the form (2.1). The initial state is $x_0 = s$ and the output at step i is x_i/m. The actual implementation depends on the values of (m, a, c). Restrictions: a, c and s must be non-negative and less than m.

```
unif01_Gen * ulcg_CreateLCGFloat (long m, long a, long c, long s);
```

The same as `ulcg_CreateLCG`, except that the implementation is in floating-point arithmetic. Valid only if the IEEE floating-point standard is respected (all integers smaller than 2^{53} are represented exactly as `double`). Restrictions : $-m < a < m$, $0 \leq c < m$, $-m < s < m$, $|am| + c < 2^{53}$, and $c = 0$ when $a < 0$.

```
#ifdef USE_GMP
unif01_Gen * ulcg_CreateBigLCG (char *m, char *a, char *c, char *s);
```

The same as `ulcg_CreateLCG`, but using arbitrary large integers. The integers are given as strings of decimal digits. The implementation uses GMP. Restrictions: a, c and s non negative and less than m.

```
#endif
```

```
unif01_Gen * ulcg_CreateLCGWu2 (long m, char o1, unsigned int q, char o2,
unsigned int r, long s);
```

Implements a LCG of the kind proposed by Wu [177], and generalized by L'Ecuyer and Simard [91], for which the modulus and multiplier can be written as $m = 2^e - h$ and $a = \pm 2^q \pm 2^r$. The parameters $o1$ and $o2$ can be `'+'` or `'-'`; they give the sign in front of 2^q and 2^r, respectively. Uses an implementation proposed in [91, 177], which uses shifts instead of multiplications. The initial state is $x_0 = s$ and the output at step i is x_i/m. We use a fast implementation with shifts instead of multiplications, whenever possible. Restrictions: $0 < s < m$, $m < 2^{31}$, and

the parameters must also satisfy the conditions $h < 2^q$, $h(2^q - (h+1)/2^{e-q}) < m$ and $h < 2^r$, $h(2^r - (h+1)/2^{e-r}) < m$. .

`unif01_Gen * ulcg_CreateLCGPayne (long a, long c, long s);`

Same as `ulcg_CreateLCG`, with the additional restriction that $m = 2^{31} - 1$. Uses the fast implementation proposed by Payne et al. [134, 9]. See also Robin Whittle's WWW page at `http://www.firstpr.com.au/dsp/rand31/`.

`unif01_Gen * ulcg_CreateLCG2e31m1HD (long a, long s);`

Same as `ulcg_CreateLCG`, with the additional restrictions that $m = 2^{31} - 1$, $c = 0$ and $1 < a < 2^{30}$. Uses the specialized implementation proposed by Hörmann et Derflinger [54].

`unif01_Gen * ulcg_CreateLCG2e31 (long a, long c, long s);`

Same as `ulcg_CreateLCG`, but with $m = 2^{31}$. Uses a specialized implementation.

`unif01_Gen * ulcg_CreateLCG2e32 (unsigned long a, unsigned long c, unsigned long s);`

Same as `ulcg_CreateLCG`, but with $m = 2^{32}$. Uses a specialized implementation.

`unif01_Gen * ulcg_CreatePow2LCG (int e, long a, long c, long s);`

Implements a LCG as in `ulcg_CreateLCG`, but with $m = 2^e$. Restrictions: a, c and s non negative and smaller than m, and $e \leq 31$.

```
#ifdef USE_LONGLONG
unif01_Gen * ulcg_CreateLCG2e48L (ulonglong a, ulonglong c, ulonglong s);
```

A simple LCG of the form $x_{i+1} = (ax_i + c) \bmod 2^{48}$, where $x_0 = s$ is the seed. The generator `drand48` of the SUN C library is obtained with the parameters

$$a = 25214903917, \qquad c = 11.$$

Only the 32 most significant bits are kept. Restrictions: $a, c, s < 281474976710656 = 2^{48}$.

`unif01_Gen * ulcg_CreatePow2LCGL (int e, ulonglong a, ulonglong c, ulonglong s);`

Implements a LCG as in `ulcg_CreatePow2LCG`, but with $e \leq 64$. Only the 32 most significant bits are kept.

`#endif`

```
#ifdef USE_GMP
unif01_Gen * ulcg_CreateBigPow2LCG (long e, char *a, char *c, char *s);
```

Implements the same type of generator as `ulcg_CreatePow2LCG`, but using arbitrary large integers. The integers a, c and s are given as strings of decimal digits.

```
#endif
```

Combined LCGs

```
unif01_Gen * ulcg_CreateCombLEC2 (long m1, long m2, long a1, long a2,
long c1, long c2, long s1, long s2);
```

Combines two LCGs by the method of L'Ecuyer [72]. The first LCG has parameters `(m1, a1, c1, s1)` and the second has parameters `(m2, a2, c2, s2)`. The combination is via $x_i = (s_{i1} - s_{i2}) \bmod (m_1 - 1)$, where s_{i1} are s_{i2} are the states of the two components at step i. The output is $u_i = x_i/m_1$ if $x_i \neq 0$, and $u_i = (m_1 - 1)/m_1$ if $x_i = 0$. As for `ulcg_CreateLCG`, the implementation depends on the parameters. The same restrictions as for `ulcg_CreateLCG` apply to the two components and one must also have `m1` $>$ `m2`.

```
unif01_Gen * ulcg_CreateCombLEC2Float (long m1, long m2, long a1, long a2,
long c1, long c2, long s1, long s2);
```

Floating-point version of `ulcg_CreateCombLEC2`. Valid only if any positive integer smaller than 2^{53} is represented exactly as a `double` (this holds, e.g., if the IEEE floating-point standard is respected). Restrictions: $a_1 m_1 + c_1 - a_1 < 2^{53}$ and $a_2 m_2 + c_2 - a_2 < 2^{53}$.

```
unif01_Gen * ulcg_CreateCombLEC3 (long m1, long m2, long m3, long a1,
long a2, long a3, long c1, long c2,
long c3, long s1, long s2, long s3);
```

Same as `ulcg_CreateCombLEC2`, but combines 3 LCGs instead of 2. The combination is via $x_i = (s_{i1} - s_{i2} + s_{i3}) \bmod (m_1 - 1)$, where s_{i1}, s_{i2} et s_{i3} are the states of the components. One must have `m1` $>$ `m2` $>$ `m3`.

```
unif01_Gen * ulcg_CreateCombWH2 (long m1, long m2, long a1, long a2,
long c1, long c2, long s1, long s2);
```

Combines two LCGs as in `ulcg_CreateCombLEC2`, but using the Wichmann and Hill approach [171]: By adding modulo 1 the outputs of the two LCGs. The same restrictions apply.

```
unif01_Gen * ulcg_CreateCombWH2Float (long m1, long m2, long a1, long a2,
long c1, long c2, long s1, long s2);
```

Floating-point version of `ulcg_CreateCombWH2`. Valid only if the IEEE floating-point standard is respected (all integers smaller than 2^{53} are represented exactly as `double`). Restrictions: $a_1 m_1 + c_1 - a_1 < 2^{53}$ and $a_2 m_2 + c_2 - a_2 < 2^{53}$.

```
unif01_Gen * ulcg_CreateCombWH3 (long m1, long m2, long m3, long a1,
long a2, long a3, long c1, long c2,
long c3, long s1, long s2, long s3);
```

Same as `ulcg_CreateCombWH2`, but combines three LCGs. The recent version of Excel uses the original Wichmann-Hill combination of three small LCGs [171] for its new random number generator (see `usoft_CreateExcel2003` on page 81 of this guide).

Clean-up functions

```
#ifdef USE_GMP
void ulcg_DeleteBigLCG (unif01_Gen *gen);
```

Frees the dynamic memory used by the `BigLCG` generator and allocated by the corresponding `Create` function above.

```
void ulcg_DeleteBigPow2LCG (unif01_Gen *gen);
```

Frees the dynamic memory used by the `BigPow2LCG` generator and allocated by the corresponding `Create` function above.

```
#endif
```

```
void ulcg_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator of this module that does not have an explicit `Delete` function. This function should be called to clean up a generator object when it is no longer in use.

Other related generators

For other specific LCGs, see also

- `uwu_CreateLCGWu61a`
- `uwu_CreateLCGWu61b`

umrg

This module implements *multiple recursive generators* (MRGs), based on a linear recurrence of order k, modulo m:

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m. \tag{2.2}$$

and whose output is normally $u_n = x_n/m$. It implements combined MRGs as well. For more details about these generators, see for example [85, 75, 76, 80, 98, 129].

Lagged-Fibonacci generators are also implemented here. These generators are actually MRGs only when the selected operation is addition or subtraction. Multiplicative lagged-Fibonacci generators, for example, are *not* MRGs, but are implemented here nonetheless.

Some of the generators in this module use the GNU multiprecision package GMP. The macro `USE_GMP` is defined in module `gdef` in directory `mylib`.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

Simple MRGs

`unif01_Gen * umrg_CreateMRG (long m, int k, long A[], long S[]);`

Implements a MRG of the form (2.2), with $(a_1, \ldots, a_k)$ in `A[0..(k-1)]`, initial state $(x_{-1}, \ldots, x_{-k})$ in `S[0..(k-1)]`, and output $u_n = x_n/m$. Faster implementations are provided for the special cases $k = 2, 3, 5, 7$ when $A[0] > 0, A[k-1] > 0$, and all other $A[i] = 0$. Restrictions: $2 \leq k$, $|a_i|(m \bmod |a_i|) < m$, $-m < a_i < m$, and $-m < x_{-i} < m$, for $i = 1, \ldots, k$.

`unif01_Gen * umrg_CreateMRGFloat (long m, int k, long A[], long S[]);`

Similar to `umrg_CreateMRG` above, but uses a floating-point implementation, as described in [80]. Restrictions: $2 \leq k$, $-m < a_i < m$ and $-m < x_{-i} < m$ for $i = 1, \ldots, k$, and $m \max(Q^+, -Q^-) < 2^{53}$ where Q^+ is the sum of the positive coefficients a_i and Q^- is the sum of the negative coefficients a_i.

```
#ifdef USE_GMP
unif01_Gen * umrg_CreateBigMRG (char *m, int k, char *A[], char *S[]);
```

Similar to `umrg_CreateMRG` above, except that the modulus, coefficients, and initial state are given as decimal character strings in `m`, `A[0..(k-1)]` and `S[0..(k-1)]`. Restrictions: $-m < a_i < m$ and $-m < x_{-i} < m$ for $i = 1, \ldots, k$.

```
#endif
```

`unif01_Gen * umrg_CreateLagFibFloat (int k, int r, char Op, int Lux, unsigned long S[]);`

Implements a 2-lags Fibonacci generator [103, 66], using a floating-point implementation, with recurrence

$$u_n = (u_{n-k} \ \texttt{Op} \ u_{n-r}) \bmod 1,$$

where the binary operator `Op` can take the values `'+'` or `'-'`, which stand for addition and subtraction. The seed vector `S[0..(k-1)]` must contain the first `k` values $u_{-1}, \ldots, u_{-k}$. The parameter `Lux` gives the *luxury level* defined as follows: if `Lux` is larger than k, then for each block of `Lux` successive output values, the first k are used and the next `Lux` $- k$ are skipped. If `Lux` $\leq k$, no value is skipped. *Note:* for `Op = '-'`, one may choose either $k < r$ or $k > r$. For example, the case $k = 55$, $r = 24$ corresponds to $X_n = (X_{n-55} - X_{n-24}) \bmod 1$, while the case $k = 24$, $r = 55$ corresponds to $X_n = (X_{n-24} - X_{n-55}) \bmod 1$. *Restrictions:* `S[i]` $< 2^{32}$ and `Op` $\in$ {`'+', '-'`}.

```
unif01_Gen * umrg_CreateLagFib (int t, int k, int r, char Op, int Lux,
unsigned long S[]);
```

Similar to `umrg_CreateLagFibFloat`, except that the implementation uses t-bit integers

$$X_n = (X_{n-k} \texttt{ Op } X_{n-r}) \bmod 2^t.$$

The parameter `Op` may take one of the values {`'*', '+', '-', 'x'`}, which stands for multiplication, addition, subtraction, and exclusive-or respectively. Note that the resulting multiplicative lagged-Fibonacci generator is not an MRG. Assume that $k > r$. If M is a power of 2, say $M = 2^t$, then the maximal period length is $(2^k - 1)2^{t-1}$ for the additive and subtractive cases, and $(2^k - 1)2^{t-3}$ for the multiplicative case. This maximal period is reached if and only if the characteristic polynomial $f(x) = x^k - x^{k-r} - 1$ is a primitive polynomial modulo 2 (i.e., over the finite field $\mathbb{F}_2$) [64, 7, 11]. Pairs of lags (k, r) that give a maximal period can be found in [115, 66, 7]. *Note:* for `Op = '-'`, one may choose $k < r$ or $k > r$. For example, the case $k = 55$, $r = 24$ corresponds to $X_n = (X_{n-55} - X_{n-24}) \bmod 2^t$, while $k = 24$, $r = 55$ corresponds to $X_n = (X_{n-24} - X_{n-55}) \bmod 2^t$. Restrictions: $0 < t \leq 64$. In the case `Op = '*'`, all the $S[i]$ must be odd; if they are not, 1 will be added to the even values.

Combined MRGs

```
unif01_Gen * umrg_CreateC2MRG (long m1, long m2, int k, long A1[],
long A2[], long S1[], long S2[]);
```

Implements a generator that combines two MRGs of order k. The combination method is by subtracting the states modulo m_1 and the implementation is the same as in Figure 1 of [76]. Restrictions: assumes that $a_{11} = 0$, $a_{12} > 0$, $a_{13} < 0$, $a_{21} > 0$, $a_{22} = 0$ and $a_{23} < 0$, $k = 3$ and the coefficients must satisfy the conditions $a_{1j}(m_1 \bmod a_{1j}) < m_1$ and $a_{2j}(m_2 \bmod a_{2j}) < m_2$.

```
#ifdef USE_GMP
unif01_Gen * umrg_CreateBigC2MRG (char *m1, char *m2, int k, char *A1[],
char *A2[], char *S1[], char *S2[]);
```

Implements a combined generator obtained from 2 MRGs of order k, whose modulus are m_1 and m_2. The coefficients of the 2 components are given as decimal strings in `A1[0..(k-1)]`, `A2[0..(k-1)]`, and the initial values are in `S1[0..(k-1)]`, `S2[0..(k-1)]`, also given as decimal strings. Restrictions are as for `umrg_CreateMRG`.

```
#endif
```

Clean-up functions

```
void umrg_DeleteMRG    (unif01_Gen * gen);
void umrg_DeleteMRGFloat (unif01_Gen * gen);
void umrg_DeleteLagFib (unif01_Gen * gen);
void umrg_DeleteLagFibFloat (unif01_Gen * gen);
void umrg_DeleteC2MRG  (unif01_Gen * gen);

#ifdef USE_GMP
void umrg_DeleteBigMRG (unif01_Gen * gen);
void umrg_DeleteBigC2MRG (unif01_Gen * gen);
#endif
```

Frees the dynamic memory used by the generators of this module, and allocated by the corresponding `Create` function.

Some related generators

For some other specific lagged-Fibonacci generators, see also

- `uknuth_CreateRan_array1`
- `uknuth_CreateRan_array2`
- `uknuth_CreateRanf_array1`
- `uknuth_CreateRanf_array2`

ucarry

Generators based on linear recurrences with carry are implemented in this module. This includes the *add-with-carry* (AWC), *subtract-with-borrow* (SWB), *multiply-with-carry* (MWC), and *shift-with-carry* (SWC) generators. For the theoretical properties of these generators and other details, we refer the reader to [13, 14, 15, 67, 160].

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

```
unif01_Gen * ucarry_CreateAWC (unsigned int r, unsigned int s,
unsigned long c, unsigned long m,
unsigned long S[]);
```

Implements the add-with-carry (AWC) generator proposed by Marsaglia and Zaman [116], based on the recurrence

$$x_i = (x_{i-r} + x_{i-s} + c_{i-1}) \bmod m, \tag{2.3}$$

$$c_i = (x_{i-r} + x_{i-s} + c_{i-1}) \operatorname{div} m, \tag{2.4}$$

with output $u_i = x_i/m$. The vector `S[0..k-1]` contains the k initial values $(x_0, \ldots, x_{k-1})$, where $k = \max\{r, s\}$, and `c` contains c_0. Restrictions: $0 < s$, $0 < r$, $r \neq s$ and $c = 0$ or 1.

```
unif01_Gen * ucarry_CreateSWB (unsigned int r, unsigned int s,
unsigned long c, unsigned long m,
unsigned long S[]);
```

Implements the subtract-with-borrow (SWB) generator proposed by Marsaglia and Zaman [116], based on the recurrence

$$x_i = (x_{i-r} - x_{i-s} - c_{i-1}) \bmod m, \tag{2.5}$$

$$c_i = I[(x_{i-r} - x_{i-s} - c_{i-1}) < 0], \tag{2.6}$$

with output $u_i = x_i/m$, where I is the indicator function. The vector `S[0..(k-1)]` contains the k initial values $(x_0, \ldots, x_{k-1})$, where $k = \max\{r, s\}$, and `c` contains c_0. Restrictions : $0 < s$, $0 < r$, $r \neq s$ and $c = 0$ or 1.

```
unif01_Gen * ucarry_CreateRanlux (unsigned int L, long s);
```

Implements the specific modified SWB generator proposed by Lüscher [102]. This is an adapted version of the FORTRAN implementation of James [55]. The parameter `L` is the luxury level and `s` is the initial state. Restriction: $24 \leq L$. The precision of this generator is only 24 bits.

```
#ifdef USE_LONGLONG
unif01_Gen * ucarry_CreateMWC (unsigned int r, unsigned long c,
unsigned int w, unsigned long A[],
unsigned long S[]);
#endif
```

Implements the *multiply-with-carry* (MWC) generator, defined by [15]:

$$x_n = (a_1 x_{n-1} + \cdots + a_r x_{n-r} + c_{n-1}) \bmod 2^w; \tag{2.7}$$

$$c_n = (a_1 x_{n-1} + \cdots + a_r x_{n-r} + c_{n-1}) \operatorname{div} 2^w; \tag{2.8}$$

$$u_n = x_n / 2^w. \tag{2.9}$$

The array `A[0..(r-1)]` must contain the coefficients $a_1, \ldots, a_r$, the array `S[0..(r-1)]` gives the initial values $(x_0, \ldots, x_{-r+1})$, and `c` gives the value of c_0. This implementation uses 64-bit integers and therefore works only on platforms where these are available. Restrictions: $w \leq 32$, $a_i < 2^w$, $x_i < 2^w$, and $c + (2^w - 1)(|a_1| + \cdots + |a_r|) < 2^{64}$.

```
unif01_Gen * ucarry_CreateMWCFloat (unsigned int r, unsigned long c,
unsigned int w, unsigned long A[],
unsigned long S[]);
```

Same as `ucarry_CreateMWC`, but uses a floating-point implementation (in `double`). Restrictions: $w \leq 32$, $a_i < 2^w$, $x_i < 2^w$, and $c + (2^w - 1)(|a_1| + \cdots + |a_r|) < \min\{2^{53},\ 2^{32+w}\}$.

```
unif01_Gen * ucarry_CreateMWCfixCouture (unsigned int c,
unsigned int S[]);
```

Implements the following specific MWC, suggested by Couture and L'Ecuyer [15]:

$$\begin{aligned} x_n &= (14x_{n-8} + 18x_{n-7} + 144x_{n-6} + 1499x_{n-5} + 2083x_{n-4} \\ &\quad + 5273x_{n-3} + 10550x_{n-2} + 45539x_{n-1} + c_{n-1}) \bmod 2^{16}, \\ c_n &= (14x_{n-8} + \cdots + 45539x_{n-1} + c_{n-1}) \operatorname{div} 2^{16}, \\ u_n &= \frac{x_{2n}}{2^{32}} + \frac{x_{2n+1}}{2^{16}}. \end{aligned}$$

The initial state is in `S[0..7]`, and c is the initial carry. The lowest 16 bits and the highest 16 bits of each u_n come from two successive numbers x_j.

```
unif01_Gen * ucarry_CreateSWC (unsigned int r, unsigned int h,
unsigned int c, unsigned int w,
unsigned int A[], unsigned int S[]);
```

Implements the shift-with-carry (SWC) generator designed by R. Couture, based on the recurrence

$$x_n = (a_1 x_{n-1} \oplus \cdots \oplus a_r x_{n-r} \oplus c_{n-1}) \bmod 2^w \tag{2.10}$$

$$c_n = (a_1 x_{n-1} \oplus \cdots \oplus a_r x_{n-r} \oplus c_{n-1}) \operatorname{div} 2^w \tag{2.11}$$

$$u_n = x_n / 2^w, \tag{2.12}$$

The vector $(x_n, \ldots, x_{n-r+1}, c_n)$ is the state of the generator. The array `A[0..h-1]` contains the polynomials $a_1, \ldots, a_r$. Each even element stands for a polynomial number and the next element stands for the corresponding nonzero coefficient number of that polynomial. The vector `S[0..r-1]` gives the initial values of $(x_0, \ldots, x_{-r+1})$ and `c` is the initial carry. Restrictions: $0 < r$ and $w \leq 32$.

```
unif01_Gen * ucarry_CreateMWC1616 (unsigned int a, unsigned int b,
unsigned int x, unsigned int y);
```

Implements the combined generator of two 16-bit multiply-with-carry generators [106]

$$x_n = (ax_{n-1} + \texttt{carx}_{n-1}) \bmod 2^{16}, \tag{2.13}$$
$$\texttt{carx}_n = (ax_{n-1} + \texttt{carx}_{n-1}) \operatorname{div} 2^{16}, \tag{2.14}$$
$$y_n = (by_{n-1} + \texttt{cary}_{n-1}) \bmod 2^{16}, \tag{2.15}$$
$$\texttt{cary}_n = (by_{n-1} + \texttt{cary}_{n-1}) \operatorname{div} 2^{16}. \tag{2.16}$$

The rightmost 16 bits of the two above product make the new x (or y) and the leftmost 16 bits the new carry `carx` (or `cary`). The function returns $(x_n \ll 16) + (y_n \;\&\; 0\text{xffff})$; the output is a 32-bit integer, x_n making up its leftmost 16 bits and y_n its rightmost 16 bits.

Clean-up functions

These functions should be called to clean up generator objects of this module when they are no longer in use.

```
void ucarry_DeleteAWC (unif01_Gen *gen);
void ucarry_DeleteSWB (unif01_Gen *gen);
void ucarry_DeleteRanlux (unif01_Gen *gen);
void ucarry_DeleteMWC (unif01_Gen *gen);
void ucarry_DeleteMWCFloat (unif01_Gen *gen);
void ucarry_DeleteMWCfixCouture (unif01_Gen *gen);
void ucarry_DeleteSWC (unif01_Gen *gen);
```

Frees the dynamic memory used by the generators of this module, and allocated by the corresponding `Create` function.

```
void ucarry_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator of this module that does not have a specific `Delete` function.

utaus

Implements simple and combined Tausworthe generators using the definitions, the initialization methods and the algorithms given in [75, 77]. The current implementation is restricted to components whose characteristic polynomial is a *trinomial*. That is, for a simple generator and for each component of a combined generator, the basic recurrence has the form

$$x_n = x_{n-r} \oplus x_{n-k} = x_{n-k+q} \oplus x_{n-k}, \tag{2.17}$$

with characteristic polynomial $p(x) = x^p + x^q + 1$, where $q = k - r$, each x_n is 0 or 1, and $\oplus$ means exclusive-or (i.e., addition modulo 2). The output at step n is

$$u_n = \sum_{j=1}^{w} x_{ns+j-1} 2^{-j} \tag{2.18}$$

with $w = 32$. To obtain $w < 32$, it suffices to truncate the output. The parameters must satisfy the following conditions: $0 < 2q < k \leq 32$ (except in the case of the `LongTaus` generator for which k can take values as high as 64) and $0 < s \leq r$. In the functions defined below, the k most significant bits of the variable `Y` contain the initial values $x_0, \ldots, x_{k-1}$ (this is the seed). They must not be all zero.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

```
unif01_Gen * utaus_CreateTaus (unsigned int k, unsigned int q,
unsigned int s, unsigned int Y);
```

Implements a simple Tausworthe generator as described above. Restrictions: $0 < 2q < k \leq 32$ and $0 < s \leq k - q$.

```
unif01_Gen * utaus_CreateTausJ (unsigned int k, unsigned int q,
unsigned int s, unsigned int j,
unsigned int Y);
```

Implements a Tausworthe generator as in `utaus_CreateTaus`, except that it produces a j-decimated sequence. That is, at each call, it skips $j - 1$ values in the sequence defined by (2.17–2.18) and outputs the next one. The same restrictions as in `utaus_CreateTaus` apply.

```
#ifdef USE_LONGLONG
unif01_Gen * utaus_CreateLongTaus (unsigned int k, unsigned int q,
unsigned int s, ulonglong Y1);
```

Similar to `utaus_CreateTaus` but uses 64 bits integers for the state of the generator. However, it returns only the 32 most significant bits of each generated number, after having shifted them 32 bits to the right. Restrictions: $k \leq 64$, $0 < 2q < k$ and $0 < s \leq k - q$.

```
#endif
```

```
unif01_Gen * utaus_CreateCombTaus2 (
unsigned int k1, unsigned int k2, unsigned int q1, unsigned int q2,
unsigned int s1, unsigned int s2, unsigned int Y1, unsigned int Y2);
```

Combines two Tausworthe generators defined as in `utaus_CreateTaus`. The combination is via a bitwise exclusive-or, as in [77, 157, 159]. The same restrictions as in `utaus_CreateTaus` apply to each of the two components. Also assumes that $k_1 \geq k_2$.

```
unif01_Gen * utaus_CreateCombTaus3 (
unsigned int k1, unsigned int k2, unsigned int k3,
unsigned int q1, unsigned int q2, unsigned int q3,
unsigned int s1, unsigned int s2, unsigned int s3,
unsigned int Y1, unsigned int Y2, unsigned int Y3);
```

Similar to `utaus_CreateCombTaus2`, except that three Tausworthe generators are combined instead of two. Assumes that $k_1 \geq k_2 \geq k_3$.

```
unif01_Gen * utaus_CreateCombTaus3T (
unsigned int k1, unsigned int k2, unsigned int k3,
unsigned int q1, unsigned int q2, unsigned int q3,
unsigned int s1, unsigned int s2, unsigned int s3,
unsigned int Y1, unsigned int Y2, unsigned int Y3);
```

Similar to `utaus_CreateCombTaus3`, except that the generator has "triple" precision. Three successive output values u_i of the combined Tausworthe generator are used to build each output value U_i (uniform on [0, 1)) of this generator, as follows:

$$U_i = \left(u_{3i} + \frac{u_{3i+1}}{2^{17}} + \frac{u_{3i+2}}{2^{34}}\right) \bmod 1.$$

Clean-up functions

```
void utaus_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator returned by the `Create` functions of this module. This function should be called to clean up any generator object of this module when it is no longer in use.

Related generators

For specific Tausworthe generators, see also

- `utezu_CreateTezLec91`
- `utezu_CreateTez95`

- `ulec_Createlfsr88`
- `ulec_Createlfsr88T`
- `ulec_Createlfsr113`
- `ulec_Createlfsr258`

ugfsr

This module implements generalized feedback shift register (GFSR) generators, twisted GFSR (TGFSR) generators, and tempered TGFSR generators (TTGFSR).

The following table points to some specific generators based on trinomials, taken from the literature.

Table 2.2: Some specific GFSRs and TGFSRs

k	r	ℓ	Type	Reference
607	273	23	GFSR	[163, 125]
521	32	31	GFSR	[145]
521	32	31	GFSR	[45]
250	103	32	GFSR	[62]
25	7	32	TGFSR	T800 [125]
25	14	32	TTGFSR	TT400 [125]
13	11	31	TTGFSR	TT403 [125]
25	17	31	TTGFSR	TT775 [125]
25	7	32	TTGFSR	TT800 [125]
624	397	32	TTGFSR	MT19937 [126]

```
#include <testu01/unif01.h>
```

GFSR generators

```
unif01_Gen * ugfsr_CreateGFSR3 (unsigned int k, unsigned int r,
unsigned int l, unsigned long S[]);
```

Implements a generator GFSR based on the recurrence

$$x_i = x_{i-r} \oplus x_{i-k} \tag{2.19}$$

where each x_i is a 32-bit vector, $\oplus$ stands for bitwise addition modulo 2, and $r < k$. The output at step i is $u_i = \tilde{x}_i/2^l$, where $\tilde{x}_i$ is the integer formed by the first l bits of x_i. The array `S[0..(k-1)]` contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. Proper initialization techniques for this generator are discussed, e.g., in [44] and [158]. Restrictions: $0 < r < k$ and $l \leq 32$.

```
unif01_Gen * ugfsr_CreateToot73 (unsigned long S[]);
```

Implements the Tausworthe generator of parameters $(k, r, s, l) = (607, 273, 512, 23)$ proposed in [163], under the form of a GFSR. The generator is initialized as in [163] from the "arbitrary" bits given in `S[0..k-1]`. This generator is the same as G607 in [125].

```
unif01_Gen * ugfsr_CreateKirk81 (long s);
```

Implements the GFSR generator proposed by Kirkpatrick and Stoll [62], with their initialization procedure. The parameters are $(k, r, l) = (250, 103, 32)$ and s is the seed.

```
unif01_Gen * ugfsr_CreateRipley90 (long s);
```

Implements the GFSR generator given in the appendix of Ripley [145]. It is a GFSR with parameters $(k, r, l) = (521, 32, 31)$. The state of this GFSR is initialized as in [145] from a MLCG of modulus $m = 2^{31} - 1$ and multiplier $a = 16807$, whose initial state is s. The returned value is $y_i/(2^{31} - 1)$.

```
unif01_Gen * ugfsr_CreateFushimi (int k, int r, int s);
```

Implements a GFSR generator with $l = 31$, with the initialization procedure proposed by Fushimi [45], using `s` as a seed to construct the initial state.

```
unif01_Gen * ugfsr_CreateFushimi90 (int s);
```

Implements a specific GFSR generator proposed by Fushimi [45], with parameters $(k, r, l) = (1563, 1467, 31)$ and using `s` as a seed to construct the initial state.

```
unif01_Gen * ugfsr_CreateGFSR5 (unsigned int k, unsigned int r1,
unsigned int r2, unsigned int r3,
unsigned int l, unsigned long S[]);
```

Implements a GFSR generator whose characteristic polynomial is a pentanomial, i.e., based on the recurrence

$$x_i = x_{i-r_3} \oplus x_{i-r_2} \oplus x_{i-r_1} \oplus x_{i-k}$$

where the x_i's are vectors of 32 bits whose first l bits are used to create the output, as described in `ugfsr_CreateGFSR3`. The array `S[0..(k-1)]` contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. Restrictions: $1 \leq l \leq 32$ and $0 < r_3 < r_2 < r_1 < k$.

```
unif01_Gen * ugfsr_CreateZiff98 (unsigned long S[]);
```

Implements a specific pentanomial-based GFSR generator proposed by Ziff [178], with parameters $(k, r_1, r_2, r_3, l) = (9689, 6988, 1586, 471, 32)$. The array `S[0..9688]` must contain the initial state.

Twisted GFSR generators

```
unif01_Gen * ugfsr_CreateTGFSR (unsigned int k, unsigned int r,
unsigned int l, unsigned long Av,
unsigned long S[]);
```

Implements the original form of TGFSR generator proposed by Matsumoto and Kurita [124].

It is based on the recurrence

$$x_i = x_{i-k} \oplus (x_{i-r}A), \tag{2.20}$$

where k, r, l and the x_i's are as in (2.19), and A is a binary matrix of dimension $l \times l$ whose first superdiagonal has all its elements equal to 1, the last row is the vector `Av`, and all other elements are 0. The output at step i is $u_i = \tilde{x}_i/2^l$, where $\tilde{x}_i$ is the integer formed by the first l bits of x_i. Matsumoto and Kurita [125] later reported deficiencies of this generator. The array `S[0..(k-1)]` contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. Remark: the notation

$$x_{i+k} = x_{i-r'} \oplus x_i,$$

where $r' = k - r$, is used in [125]. In other words, their r correspond to our $k - r$.

```
unif01_Gen * ugfsr_CreateT800 (unsigned long S[]);
```

Implements the TGFSR generator T800 proposed by Matsumoto and Kurita [125], whose parameters are $(k, r, l) = (25, 18, 32)$ and `Av = 0x8EBD028`. The array `S[0..(k-1)]` contains the k initial bit vectors $x_0, \ldots, x_{k-1}$.

```
unif01_Gen * ugfsr_CreateTGFSR2 (unsigned int k, unsigned int r,
unsigned int l, unsigned int s,
unsigned int t, unsigned long Av,
unsigned long Bv, unsigned long Cv,
unsigned long S[]);
```

Implements the generator TGFSR-II proposed by Matsumoto and Kurita [125], based on the same recurrence as their original TGFSR, but where a *tempering* is added to improve the statistical quality of the output. It is defined by

$$\begin{aligned} x_i &= x_{i-k} \oplus (x_{i-r}A), & (2.21)\\ y_i &= x_i \oplus ((x_i \ll s) \;\&\; b), & (2.22)\\ z_i &= y_i \oplus ((y_i \ll t) \;\&\; c), & (2.23) \end{aligned}$$

where $\ll s$ means a left shift by s bits, & means the bitwise-and operation, and the bit vectors b and c are given by `Bv` and `Cv`. The output u_i is constructed as described in `ugfsr_CreateTGFSR`, but using z_i instead of x_i. The array `S[0..(k-1)]` contains the k initial bit vectors $x_0, \ldots, x_{k-1}$.

```
unif01_Gen * ugfsr_CreateTT400 (unsigned long S[]);
```

Implements the generator TT400 proposed by Matsumoto and Kurita [125], whose parameters are $(k, k-r, l) = (25, 11, 16)$, $s = 2$, $t = 7$, `Av = 0xA875` , `Bv = 0x6A68`, `Cv = 0x7500`. The array `S[0..(k-1)]` contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. The returned value is $z_i/(2^{16}-1)$.

```
unif01_Gen * ugfsr_CreateTT403 (unsigned long S[]);
```

Implements the generator TT403 proposed by Matsumoto and Kurita [125], whose parameters are $(k, k-r, l) = (13, 2, 31)$, $s = 8$, $t = 14$, `Av = 0x6B5ECCF6`, `Bv = 0x102D1200`, `Cv`

= 0x66E50000. The array S[0..(k-1)] contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. The returned value is $z_i/(2^{31}-1)$.

```
unif01_Gen * ugfsr_CreateTT775 (unsigned long S[]);
```

Implements the generator TT775 proposed by Matsumoto and Kurita [125], whose parameters are $(k, k-r, l) = (25, 8, 31)$, $s = 6$, $t = 14$, Av = 0x6C6CB38C, Bv = 0x1ABD5900, Cv = 0x776A0000. The array S[0..(k-1)] contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. The returned value is $z_i/(2^{31}-1)$.

```
unif01_Gen * ugfsr_CreateTT800 (unsigned long S[]);
```

Implements the generator TT800 proposed by Matsumoto and Kurita [125], whose parameters are $(k, r, l) = (25, 18, 32)$, $s = 7$, $t = 15$, Av = 0x8EBFD028, Bv = 0x2B5B2500, Cv = 0xDB8D0000. The array S[0..24] contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. The returned value is $z_i/2^{32}$.

```
unif01_Gen * ugfsr_CreateTT800M94 (unsigned long S[]);
```

The original implementation of TT800 provided by Matsumoto and Kurita [125], in 1994. The array S[0..24] contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. The returned value is $z_i/(2^{32}-1)$.

```
unif01_Gen * ugfsr_CreateTT800M96 (unsigned long S[]);
```

A second implementation of TT800, provided by Matsumoto and Kurita in 1996. The array S[0..24] contains the k initial bit vectors $x_0, \ldots, x_{k-1}$. The returned value is $z_i/(2^{32}-1)$.

```
unif01_Gen * ugfsr_CreateMT19937_98 (unsigned long seed);
```

The original implementation of the Mersenne twister generator of Matsumoto and Nishimura [126]. Its period length is $2^{19937}-1$. The returned value is $z_i/(2^{32}-1)$. This is the 1998 version.

```
unif01_Gen * ugfsr_CreateMT19937_02 (unsigned long seed,
unsigned long Key[], int len);
```

The 2002 version of the Mersenne twister generator of Matsumoto and Nishimura [126], which has a better initialization procedure than the original 1998 version. If len ≤ 0 or Key = NULL, then seed is used to initialize the state vector. If len > 0, the array Key of length len is used instead. If len is smaller than 624, then each array of 32-bit integers gives distinct initial state vectors. This is useful if one wants a larger seed space than a single 32-bit word.

Clean-up functions

```
void ugfsr_DeleteGFSR5 (unif01_Gen * gen);
```

Frees the dynamic memory allocated by ugfsr_CreateGFSR5.

```
void ugfsr_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator of this module that does not have an explicit `Delete` function. This function should be called when a generator is no longer in use.

uinv

This module implements different types of inversive generators.

```
#include <testu01/unif01.h>
```

```
unif01_Gen * uinv_CreateInvImpl (long m, long a1, long a2, long z0);
```

Implements a nonlinear inversive generator as defined in [28] and [73, p.93], with

$$\begin{aligned} z_n &= \begin{cases} (a_1 + a_2 \cdot z_{n-1}^{-1}) \bmod m & \text{if } z_{n-1} \neq 0 \\ a_1 & \text{if } z_{n-1} = 0 \end{cases} \\ u_n &= z_n/m. \end{aligned}$$

The generator computes z_{n-1}^{-1} via the modified Euclid algorithm (see [64] p. 325). If m is prime and if $p(x) = x^2 - a_1 x - a_2$ is a primitive polynomial modulo m, then the generator has maximal period m. Restrictions: $0 \leq z_0 < m$, $0 < a_1 < m$ and $0 < a_2 < m$. Furthermore, m must be a prime number, preferably large.

```
unif01_Gen * uinv_CreateInvImpl2a (int e, unsigned long a1,
unsigned long a2, unsigned long z0);
```

Implements a nonlinear inversive generator similar to `uinv_CreateInvImpl`, but with $m = 2^e$ (see [28] p. 172). The domain is limited to odd positive integers since the inverse modulo 2^e of a given x exists only if x is odd. For $e = 31$ or 32, the generator computes the inverse by exponentiation according to the formula: $x^{-1} = x^{m-1} = x^{(m \text{ div } 4)-1}$. For $e \leq 30$, the inverse is computed via the modified Euclid algorithm (faster than exponentiation, but our implementation of it is only valid in the domain of `long`, i.e. if $m \leq 2^{31} - 1$). If $e \geq 3$ and if $a_2 - 1$ and $a_1 - 2$ are multiples of 4, then the period is maximal and equal to $m/2$. *Restrictions*: $3 \leq e \leq 32$; z_0, a_1 and a_2 less than m; z_0 and a_2 must be odd and a_1 must be even.

```
unif01_Gen * uinv_CreateInvImpl2b (int e, unsigned long a1,
unsigned long a2, unsigned long z0);
```

Implements a nonlinear inversive generator with $m = 2^e$ as described in [31]. The recurrence is:

$$z_n = T(z_{n-1})$$

where

$$T(2^\ell z) = (a_1 + 2^\ell a_2 z^{-1}) \bmod 2^e$$

whenever z is odd. For $e = 31$ or 32, the inverse is computed by exponentiation according to the formula: $x^{-1} = x^{m-1} = x^{(m \text{ div } 4)-1}$. For $e \geq 3$, if $a_2 - 1$ is a multiple of 4 and if a_1 is odd, then the period is maximal and equal to m. *Restrictions*: $3 \leq e \leq 32$; z_0, a_1 and a_2 less than m and odd.

`unif01_Gen * uinv_CreateInvExpl (long m, long a, long c);`

Implements an *explicit* nonlinear inversive generator, as described in [29] and [75] (Section 10.2), with

$$z_n = \begin{cases} x_n^{-1} & \text{si } x_n \neq 0 \\ 0 & \text{si } x_n = 0 \end{cases}$$

where $x_n = (an + c) \bmod m$ for $n \geq 0$. The generator computes x_n^{-1} by the modified Euclid algorithm (see [64] p. 325). The initial state of the generator, x_0, is given by c. *Restrictions*: $0 < a < m$, $0 \leq c < m$ and m must be a prime number. In this case, the period has length m.

`unif01_Gen * uinv_CreateInvExpl2a (int e, long a, long c);`

Implements an *explicit* nonlinear inversive generator, similar to `uinv_CreateInvExpl`, but with $m = 2^e$, as described in [34]. *Restrictions*: $3 \leq e \leq 32$; a and c less than m, $a - 2$ multiple of 4 and c odd.

`unif01_Gen * uinv_CreateInvExpl2b (int e, long a, long c);`

Implements an *explicit modified* nonlinear inversive generator, with $m = 2^e$, as proposed in [30]. The recurrence has the form

$$x_n = n(an + c)^{-1} \bmod 2^e; \qquad u_n = x_n 2^{-e}.$$

Restrictions: $3 \leq e \leq 32$, $a < m$, $c < m$, $a - 2$ multiple of 4, and c odd. With these restrictions, the period is equal to m.

`unif01_Gen * uinv_CreateInvMRG (long m, int k, long A[], long S[]);`

Implements an inversive multiple recursive generator (MRG), based on the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m$$

as in `umrg_CreateMRG`, except that the output u_n is constructed using $x_n^{-1} \bmod m$ instead of x_n. *Restrictions*: The same restrictions as for `umrg_CreateMRG` apply here and m must be a prime number.

`unif01_Gen * uinv_CreateInvMRGFloat (long m, int k, long A[], long S[]);`

Provides a floating-point implementation of the same generator as in `uinv_CreateInvMRG`. The implementation is similar to that in `umrg_CreateMRGFloat`. *Restrictions*: The same restrictions apply here as for `umrg_CreateMRGFloat` and m must be a prime number.

Clean-up functions

`void uinv_DeleteInvMRG (unif01_Gen * gen);`

Frees the dynamic memory allocated by `uinv_CreateInvMRG`.

```
void uinv_DeleteInvMRGFloat (unif01_Gen * gen);
```

Frees the dynamic memory allocated by `uinv_CreateInvMRGFloat`.

```
void uinv_DeleteGen (unif01_Gen * gen);
```

Frees the dynamic memory allocated by the other `Create` functions of this module.

uquad

This module implements generators based on quadratic recurrences modulo m, of the form

$$x_{n+1} = (ax_n^2 + bx_n + c) \bmod m, \tag{2.24}$$

with output $u_n = x_n/m$ at step n. See, e.g., [27, 33, 35, 66] for analyses of such generators.

```
#include <testu01/unif01.h>
```

```
unif01_Gen * uquad_CreateQuadratic (long m, long a, long b, long c, long s);
```

Initializes a generator based on recurrence (2.24), with initial state $x_0 = s$. Depending on the values of the parameters, various implementations of different speeds are used. In general, this generator is slow. Restrictions: a, b, c and s non negative and less than m.

```
unif01_Gen * uquad_CreateQuadratic2 (int e, unsigned long a,
unsigned long b, unsigned long c, unsigned long s);
```

Similar to `uquad_CreateQuadratic`, but with $m = 2^e$. Restrictions: a, b, c and s non negative and less than 2^e; $e \leq 32$ for 32-bit machines, and $e \leq 64$ for 64-bit machines.

Clean-up functions

```
void uquad_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator returned by the `Create` functions of this module. This function should be called to clean up any generator object of this module when it is no longer in use.

ucubic

This module implements simple and combined cubic congruential generators, based on recurrences of the form

$$x_{n+1} = (ax_n^3 + bx_n^2 + cx_n + d) \bmod m, \tag{2.25}$$

with output $u_n = x_n/m$ at step n. See, e.g., [32, 90].

Generators based on a linear congruential recurrence, but with a cubic output transformation, are also available (see `ucubic_CreateCubicOut`).

```
#include <testu01/unif01.h>
```

```
unif01_Gen * ucubic_CreateCubic (long m, long a, long b, long c, long d,
long s);
```

Initializes a generator of the form (2.25), with initial state $x_0 = s$. Depending on the values of the parameters, various implementations of different speed are used. In general, this generator is rather slow. Restrictions: a, b, c, d, and s non negative and less than m.

```
unif01_Gen * ucubic_CreateCubicFloat (long m, long a, long b, long c,
long d, long s);
```

A floating-point implementation of the same generator as in `ucubic_CreateCubic`. The implementation depends on the parameter values and is slower when $m(m-1) > 2^{53}$. The same restrictions as for `ucubic_CreateCubic` apply. Also assumes that a `double` has at least 53 bits of precision.

```
unif01_Gen * ucubic_CreateCubic1 (long m, long a, long s);
```

Implements a cubic generator which is a special case of (2.25), with recurrence $x_{n+1} = (ax_n^3 + 1) \bmod m$. The initial state is $x_0 = s$ and the n-th generated value is $u_n = x_n/m$. Restrictions: a and s non negative and less than m.

```
unif01_Gen * ucubic_CreateCubic1Float (long m, long a, long s);
```

Floating-point implementation of the same generator as in `ucubic_CreateCubic1`. The implementation and restrictions are similar to those in `ucubic_CreateCubicFloat`.

```
unif01_Gen * ucubic_CreateCombCubic2 (long m1, long m2, long a1, long a2,
long s1, long s2);
```

Implements a generator that combines two cubic components of the same type as in the procedure `ucubic_CreateCubic1`. The output is $u_n = (x_{1,n}/m_1 + x_{2,m}/m_2) \bmod 1$, where $x_{1,n}$ and $x_{2,n}$ are the states of the two components at step n.

`unif01_Gen * ucubic_CreateCubicOut (long m, long a, long c, long s);`

Initializes a generator defined by the linear recurrence $x_{n+1} = (ax_n + c) \bmod m$, with initial state $x_0 = s$, and with output $u_n = (x_n^3 \bmod m)/m$. Restrictions: a, c and s non negative and less than m.

`unif01_Gen * ucubic_CreateCubicOutFloat (long m, long a, long c, long s);`

A floating-point implementation of `ucubic_CreateCubicOut`. The implementation and restrictions are similar to those in `ucubic_CreateCubicFloat`.

Clean-up functions

`void ucubic_DeleteGen (unif01_Gen *gen);`

Frees the dynamic memory used by any generator returned by the `Create` functions of this module. This function should be called to clean up any generator object of this module when it is no longer in use.

uxorshift

This module implements *xorshift* generators, a class of very fast generators proposed by Marsaglia in [111], and studied in depth by Panneton and L'Écuyer in [132]. The state of a xorshift generator is a vector of bits. At each step, the next state is obtained by applying a given number of xorshift operations to w-bit blocks in the current state, where $w = 32$ or 64. A *xorshift operation* is defined as follows: replace the w-bit block by a bitwise *xor* (exclusive or) of the original block with a shifted copy of itself by a positions either to the right or to the left, where $0 < a < w$.

Xorshifts are linear operations. The left shift of a w-bit vector $\mathbf{x}$ by one bit, $\mathbf{x} \ll 1$, can also be written as $\mathbf{L}\mathbf{x}$ where $\mathbf{L}$ is the $w \times w$ matrix with 1's on its main subdiagonal and 0's elsewhere. Similarly, the right shift $\mathbf{x} \gg 1$ can be written as $\mathbf{R}\mathbf{x}$ where $\mathbf{R}$ has 1's on its main superdiagonal and 0's elsewhere. Matrices of the forms $(\mathbf{I} + \mathbf{L}^a)$ and $(\mathbf{I} + \mathbf{R}^a)$, where $a \in \{1, \ldots, w-1\}$, are called *left* and *right xorshift matrices*, respectively. They represent *left* and *right a-bit xorshift operations*.

A *xorshift generator* is defined by a recurrence of the form

$$\mathbf{v}_i = \sum_{j=1}^{p} \tilde{\mathbf{A}}_j \mathbf{v}_{i-m_j} \bmod 2 \tag{2.26}$$

where p is a positive integer, the $\mathbf{v}_i$'s are w-bit vectors, the m_j's are integers, and $\tilde{\mathbf{A}}_j$ is either the identity or the product of ν_j xorshift matrices for some $\nu_j \geq 0$, for each j ($\tilde{\mathbf{A}}_j$ is the zero matrix if $\nu_j = 0$). The generator's state at step i is $\mathbf{x}_i = (\mathbf{v}_i^\mathsf{T}, \ldots, \mathbf{v}_{i-r+1}^\mathsf{T})^\mathsf{T}$ and the output is $u_i = \sum_{\ell=1}^{w} v_{i,\ell-1} 2^{-\ell}$ where $\mathbf{v}_i = (v_{i,0}, \ldots, v_{i,w-1})^\mathsf{T}$.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

```
unif01_Gen* uxorshift_CreateXorshift32 (int a, int b, int c, unsigned int x);
```

Implements the 32-bit *Xorshift* generators proposed by Marsaglia in [111, page 3]:

$$\begin{aligned} y &= y_{n-1} \oplus (y_{n-1}\, H_1\, a), \\ y &= y \oplus (y\, H_2\, b), \\ y_n &= y \oplus (y\, H_3\, c) \bmod 2^{32} \end{aligned}$$

where the operators H_1, H_2 and H_3 may be either the left bit-shift operator $\ll$ or the right bit-shift operator $\gg$, depending on whether the corresponding parameter (a, b or c) is positive (left shift) or negative (right shift). The initial seed is `x` and the generator returns $y_n/2^{32}$. Restrictions: $-32 < a, b, c < 32$.

```
#ifdef USE_LONGLONG
unif01_Gen* uxorshift_CreateXorshift64 (int a, int b, int c, ulonglong x);
#endif
```

Similar to `uxorshift_CreateXorshift32` but using 64-bit integers (see [111, page 3]). Only the 32 most significant bits of each generated number are returned, though the generator does all its calculations with 64 bits. Restrictions: $-64 < a, b, c < 64$.

```
unif01_Gen * uxorshift_CreateXorshiftC (int a, int b, int c, int r,
unsigned int X[]);
```

Generalizes the *Xorshift* generators proposed by Marsaglia in [111, page 4] to generators with maximal period $2^{32r}-1$. Given integers x_i, $i = 1, 2, \ldots, r$, representing the state of the generator, the next state is obtained through:

$$\begin{aligned} t &= x_1 \oplus (x_1 \, H_1 \, a) \\ x_i &= x_{i+1}, \qquad i = 1, 2, \ldots, r-1 \\ x_r &= x_r \oplus (x_r \, H_3 \, c) \oplus t \oplus (t \, H_2 \, b) \end{aligned}$$

where the operators H_1, H_2 and H_3 may be either the left bit-shift operator $\ll$ or the right bit-shift operator $\gg$, depending on whether the corresponding parameter (a, b or c) is positive (left shift) or negative (right shift). The initial state x_i is obtained from the seed `X` as $x_i =$ `X`$[i-1], i = 1, 2, \ldots, r$ and the generator returns $x_r/2^{32}$. Restrictions: $-32 < a, b, c < 32$.

```
unif01_Gen * uxorshift_CreateXorshiftD (int r, int a[], unsigned int X[]);
```

Generalizes the *Xorshift* generators proposed by Marsaglia in [111, page 5] to generators with maximal period $2^{32r}-1$. Given integers x_i, $i = 1, 2, \ldots, r$, representing the state of the generator, and shift parameters a_i, $i = 1, 2, \ldots, r$, the next state is obtained through:

$$\begin{aligned} t &= x_1 \oplus (x_1 \, H_1 \, a_1) \oplus x_2 \oplus (x_2 \, H_2 \, a_2) \oplus \cdots \oplus x_r \oplus (x_r \, H_r \, a_r) \\ x_i &= x_{i+1}, \qquad i = 1, 2, \ldots, r-1 \\ x_r &= t \end{aligned}$$

where the operators H_i may be either the left bit-shift operator $\ll$ or the right bit-shift operator $\gg$, depending on whether the corresponding parameter a_i is positive (left shift) or negative (right shift). The initial state x_i is obtained from the seed `X` as $x_i =$ `X`$[i-1], i = 1, 2, \ldots, r$ and the shift parameters are given by $a_i =$ `a`$[i-1], i = 1, 2, \ldots, r$. The generator returns $x_r/2^{32}$. Restrictions: $-32 <$ `a`$[i] < 32, \quad i = 0, 1, 2, \ldots, r-1$.

```
unif01_Gen* uxorshift_CreateXorshift7 (unsigned int S[8]);
```

Creates a full-period `Xorshift` generator of order 8 with 7 xorshifts, proposed in [132]. It has a period length of $2^{256} - 1$, its state $\mathbf{v}$ is made up of eight 32-bit integers, and it satisfies the recurrence

$$\begin{aligned} \mathbf{v}_n &= (\mathbf{I} + \mathbf{L}^9)(\mathbf{I} + \mathbf{L}^{13})\mathbf{v}_{n-1} + (\mathbf{I} + \mathbf{L}^7)\mathbf{v}_{n-4} + (\mathbf{I} + \mathbf{R}^3)\mathbf{v}_{n-5} + \\ &\quad (\mathbf{I} + \mathbf{R}^{10})\mathbf{v}_{n-7} + (\mathbf{I} + \mathbf{L}^{24})(\mathbf{I} + \mathbf{R}^7)\mathbf{v}_{n-8} \end{aligned}$$

where $\mathbf{L}^j$ stands for a j-bits left shift, $\mathbf{R}^j$ stands for a j-bits right shift, and $\mathbf{I}$ is the identity operator. All additions are done modulo 2. The `S` are the 8 seeds.

`unif01_Gen* uxorshift_CreateXorshift13 (unsigned int S[8]);`

Similar to the `uxorshift_CreateXorshift7` generator [132] but with 13 xorshifts and satisfying the recurrence

$$\begin{aligned}\mathbf{v}_n &= (\mathbf{I}+\mathbf{L}^{17})\mathbf{v}_{n-1} + (\mathbf{I}+\mathbf{L}^{10})\mathbf{v}_{n-2} + (\mathbf{I}+\mathbf{R}^{9})(\mathbf{I}+\mathbf{L}^{17})\mathbf{v}_{n-4} + (\mathbf{I}+\mathbf{R}^{3})\mathbf{v}_{n-4} + \\ &\quad (\mathbf{I}+\mathbf{R}^{12})\mathbf{v}_{n-5} + (\mathbf{I}+\mathbf{R}^{25})\mathbf{v}_{n-5} + (\mathbf{I}+\mathbf{R}^{3})(\mathbf{I}+\mathbf{R}^{2})\mathbf{v}_{n-6} + (\mathbf{I}+\mathbf{R}^{22})\mathbf{v}_{n-7} + \\ &\quad (\mathbf{I}+\mathbf{L}^{24})(\mathbf{I}+\mathbf{R}^{3})\mathbf{v}_{n-8}.\end{aligned}$$

Clean-up functions

`void uxorshift_DeleteXorshiftC (unif01_Gen * gen);`

Frees the dynamic memory allocated by `uxorshift_CreateXorshiftC`.

`void uxorshift_DeleteXorshiftD (unif01_Gen * gen);`

Frees the dynamic memory allocated by `uxorshift_CreateXorshiftD`.

`void uxorshift_DeleteGen (unif01_Gen * gen);`

Frees the dynamic memory used by any generator of this module. This function should be called when a generator is no longer in use.

ubrent

This module implements some random number generators proposed by Richard P. Brent (Web pages at `http://maths-people.anu.edu.au/~brent/` and `http://maths-people.anu.edu.au/~brent/random.html`).

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

The xorgens generators, version 2004

`unif01_Gen* ubrent_CreateXorgen32 (int r, int s, int a, int b, int c, int d, lebool hasWeyl, unsigned int seed);`

Some fast long-period random number generators [8] generalizing Marsaglia's *Xorshift* RNGs [111] (see page 48 of this guide). The output may be combined with a Weyl generator. The parameters r, s, a, b, c, d are chosen such that the $n \times n$ matrix T defining the recurrence has a minimal polynomial which is of degree n and primitive over $\mathbb{F}_2$. The state of the generator is made up of $n = 32r$ bits. The primary recurrence is $x_k = x_{k-r}A + x_{k-s}B$, where matrices A and B implement a combination of left and right shifts; in the notation of Marsaglia, $A = (I + L^a)(I + R^b)$ and $B = (I + L^c)(I + R^d)$ with I the identity matrix, L^a a left shift by a bits, and R^b a right shift by b bits. If `hasWeyl` is `TRUE`, then the Weyl combination is added to the output as in Brent original code. If it is `FALSE`, then no Weyl combination is added; this is useful for testing these *xorgens* by themselves. Restrictions: $r > 1$, $0 < s < r$ and $0 < a, b, c, d < 32$, and r must be a power of 2. The following table gives parameters recommended by Brent for the *best* 32-bit generators of this kind according to the criteria given in `http://maths-people.anu.edu.au/~brent/ftp/random/xortable.txt`.

Table 2.3: Good parameters for Brent's `xorgens` generators

n	r	s	a	b	c	d	Weight	delta
64	2	1	17	14	12	19	31	12
128	4	3	15	14	12	17	55	12
256	8	3	18	13	14	15	109	13
512	16	1	17	15	13	14	185	13
1024	32	15	19	11	13	16	225	11
2048	64	59	19	12	14	15	213	12
4096	128	95	17	12	13	15	251	12

`unif01_Gen* ubrent_CreateXor4096s (unsigned int seed);`

This is the 32-bit generator *xor4096s* with period $2^{32}(2^{4096} - 1)$ proposed by Brent [8]. It is a generalization of Marsaglia's *Xorshift* generators [111] (see page 48 of this guide). The initial seed is `seed`. This generator corresponds to the more general case above with parameters $n = 4096$, $r = 128$, $s = 95$, $a = 17$, $b = 12$, $c = 13$, $d = 15$, Weight $= 251$, delta $= 12$.

```
#ifdef USE_LONGLONG
```

```
unif01_Gen* ubrent_CreateXorgen64 (int r, int s, int a, int b, int c, int d,
lebool hasWeyl, ulonglong seed);
```

Similar to `ubrent_CreateXorgen32` above but with 64-bit generators. The state of the generator is made up of $n = 64r$ bits, but only the 32 most significant bits of each generated number are used here. Restrictions: $r > 1$, $0 < s < r$ and $0 < a, b, c, d < 64$, and r must be a power of 2. The following table gives parameters recommended by Brent for the *best* generators of their kind according to the criteria given in `http://maths-people.anu.edu.au/~brent/ftp/random/xortable.txt`.

Table 2.4: Good parameters for Brent's `xorgens` generators

n	r	s	a	b	c	d	Weight	delta
128	2	1	33	31	28	29	65	28
256	4	3	37	27	29	33	127	27
512	8	1	37	26	29	34	231	26
1024	16	7	34	29	25	31	439	25
2048	32	1	35	27	26	37	745	26
4096	64	53	33	26	27	29	961	26

```
unif01_Gen* ubrent_CreateXor4096l (ulonglong seed);
```

This is the 64-bit generator *xor4096l* with period at least $(2^{4096} - 1)$ proposed by Brent [8]. It is a generalization of Marsaglia's *Xorshift* generators [111] (see page 48 of this guide). The initial seed is `seed`. While Brent's original code returns 64-bit numbers, only the 32 most significant bits of each generated number are used here.

```
unif01_Gen* ubrent_CreateXor4096d (ulonglong seed);
```

This is the 53-bit floating-point generator *xor4096d* with period at least $(2^{4096} - 1)$ proposed by Brent [8]. It is based on *xor4096l* (implemented in `ubrent_CreateXor4096l` above) and uses its 53 most significant bits. The initial seed is `seed`.

```
#endif
```

The xorgens generators, version 2006

```
unif01_Gen* ubrent_CreateXor4096i (unsigned long seed);
```

This is the integer random number generator *xor4096i* with period at least $(2^{4096} - 1)$ proposed by Brent (see `http://maths-people.anu.edu.au/~brent/random.html`). This is the 2006 version of the generators *xor4096s* and *xor4096l*. It has a different initialization and a slightly different algorithm from the 2004 version.

```
unif01_Gen* ubrent_CreateXor4096r (unsigned long seed);
```

This is the floating-point generator *xor4096r* proposed by Brent (see `http://maths-people.anu.edu.au/~brent/random.html`). This is the 2006 version of the generators *xor4096f* and *xor4096d*. It is based on *xor4096i* implemented in `ubrent_CreateXor4096i` above. The initial seed is `seed`.

Clean-up functions

```
void ubrent_DeleteXorgen32 (unif01_Gen *);
void ubrent_DeleteXor4096s (unif01_Gen *);
void ubrent_DeleteXor4096i (unif01_Gen *);
void ubrent_DeleteXor4096r (unif01_Gen *);

#ifdef USE_LONGLONG
void ubrent_DeleteXorgen64 (unif01_Gen *);
void ubrent_DeleteXor4096l (unif01_Gen *);
void ubrent_DeleteXor4096d (unif01_Gen *);
#endif
```

Frees the dynamic memory allocated by the corresponding `Create` functions of this module.

ulec

This module collects several generators from the papers of L'Ecuyer and his co-authors.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

```
unif01_Gen * ulec_CreateCombLec88 (long S1, long S2);
```

Combined generator for 32-bit machines proposed by L'Ecuyer [72], in its original version. The integers S_1 and S_2 are the seed. They must satisfy: $0 < S_1 < 2147483563$ and $0 < S_2 < 2147483399$.

```
unif01_Gen * ulec_CreateCombLec88Float (long S1, long S2);
```

Same generator as `ulec_CreateCombLec88`, but implemented using floating-point arithmetic, as in `ulcg_CreateLCGFloat`.

```
unif01_Gen * ulec_CreateCLCG4 (long S1, long S2, long S3, long S4);
```

This generator is a combined LCG with four components, with period length near 2^{121}, proposed by L'Ecuyer and Andres [84].

```
unif01_Gen * ulec_CreateCLCG4Float (long S1, long S2, long S3, long S4);
```

Same generator as `ulec_CreateCLCG4`, but implemented using floating-point arithmetic.

```
unif01_Gen * ulec_CreateMRG93 (long S1, long S2, long S3, long S4, long S5);
```

MRG of order 5, with modulus $m = 2^{31} - 1$, multipliers $a_1 = 107374182$, $a_2 = a_3 = a_4 = 0$, $a_5 = 104480$, and period length $m^5 - 1$, proposed by L'Ecuyer, Blouin, and Couture [85], page 97. The integers `S1` to `S5` are the seed. They must be non-negative and not all zero.

```
unif01_Gen * ulec_CreateCombMRG96 (long S11, long S12, long S13,
long S21, long S22, long S23);
```

Combined MRG proposed by L'Ecuyer [76], implemented in integer arithmetic using the `long` type. This generator combines two MRGs of order 3 with distinct prime moduli less than 2^{31}. The six parameters of the function make the seed. They must all be non-negative, the first three not all zero, and the last three not all zero.

```
unif01_Gen * ulec_CreateCombMRG96Float (long S11, long S12, long S13,
long S21, long S22, long S23);
```

Same as `ulec_CreateCombMRG96`, except that the implementation is in floating-point arithmetic.

```
unif01_Gen * ulec_CreateCombMRG96D (long S11, long S12, long S13,
long S21, long S22, long S23);
```

Similar to `ulec_CreateCombMRG96`, except that the generator has "double" precision. Two successive output values u_i of the `ulec_CreateCombMRG96` generator are used to build each output value U_i (uniform on [0, 1)) of this generator, as follows:

$$U_i = \left(u_{2i} + \frac{u_{2i+1}}{2^{24}}\right) \bmod 1.$$

```
unif01_Gen * ulec_CreateCombMRG96FloatD (long S11, long S12, long S13,
long S21, long S22, long S23);
```

Similar to `ulec_CreateCombMRG96Float`, except that the generator has "double" precision. Two successive output values u_i of the `ulec_CreateCombMRG96Float` generator are used to build each output value U_i (uniform on [0, 1)) of this generator, as follows:

$$U_i = \left(u_{2i} + \frac{u_{2i+1}}{2^{24}}\right) \bmod 1.$$

```
unif01_Gen * ulec_CreateMRG32k3a (double x10, double x11, double x12,
double x20, double x21, double x22);
```

Implements the combined MRG `MRG32k3a` proposed by L'Ecuyer [80]. Its period length is near 2^{191}. This is a floating-point implementation. The six parameters represent the initial state and must be all *integers* represented as `double`'s. The first three must be integers in [0, 4294967086] and not all 0. The last three must be integers in [0, 4294944442] and not all 0.

```
unif01_Gen * ulec_CreateMRG32k3aL (long x10, long x11, long x12,
long x20, long x21, long x22);
```

Same as `MRG32k3a` above, but implemented assuming 64-bit `long` integers.

```
unif01_Gen * ulec_CreateMRG32k3b (double x10, double x11, double x12,
double x20, double x21, double x22);
```

Similar to `ulec_CreateMRG32k3a` but implements the Wichmann-Hill variant.

```
unif01_Gen * ulec_CreateMRG32k5a (double x10, double x11, double x12,
double x13, double x14, double x20,
double x21, double x22, double x23,
double x24);
```

Implements the combined MRG `MRG32k5a` proposed by L'Ecuyer [80]. Its period length is near 2^{319}. This is a floating-point implementation.

```
unif01_Gen * ulec_CreateMRG32k5b (double x10, double x11, double x12,
double x13, double x14, double x20,
double x21, double x22, double x23,
double x24);
```

Similar to `ulec_CreateMRG32k5b` but implements the Wichmann-Hill variant.

```
#ifdef USE_LONGLONG
unif01_Gen * ulec_CreateMRG63k3a (longlong s10, longlong s11, longlong s12,
longlong s20, longlong s21, longlong s22);
```

Implements the combined MRG `MRG63k3a` proposed by L'Ecuyer [80]. Uses 64-bit integers (see `gdef.h`) and works only if that type is fully supported by the compiler.

```
unif01_Gen * ulec_CreateMRG63k3b (longlong s10, longlong s11, longlong s12,
longlong s20, longlong s21, longlong s22);
```

Similar to `ulec_CreateMRG63k3a` but implements the Wichmann-Hill variant.

```
#endif
```

```
unif01_Gen * ulec_Createlfsr88 (unsigned int s1, unsigned int s2,
unsigned int s3);
```

Combined Tausworthe generator proposed by L'Ecuyer [77], with period length near 2^{88}. The initial seeds `s1, s2, s3` must be greater or equal than 2, 8, and 16 respectively.

```
unif01_Gen * ulec_Createlfsr113 (unsigned int s1, unsigned int s2,
unsigned int s3, unsigned int s4);
```

Combined Tausworthe generator proposed by L'Ecuyer [82], with period length near 2^{113}. Restrictions: the initial seeds `s1, s2, s3, s4` must be greater or equal than 2, 8, 16, and 128 respectively.

```
#ifdef USE_LONGLONG
unif01_Gen * ulec_Createlfsr258 (ulonglong s1, ulonglong s2, ulonglong s3,
ulonglong s4, ulonglong s5);
#endif
```

Combined Tausworthe generator proposed by L'Ecuyer [82], with period length near 2^{258}. This implementation uses 64-bits integers (see `gdef.h`), and works only with machines and compilers that support them. Restrictions: the initial seeds `s1, s2, s3, s4, s5` must be greater or equal than 2, 512, 4096, 131072 and 8388608 respectively.

```
unif01_Gen * ulec_CreateCombTausLCG11 (unsigned int k, unsigned  int q,
unsigned int s, unsigned S1,
long m, long a, long c, long S2);
```

Combines a Tausworthe generator of parameters (k, q, s) and initial state `S1` with an LCG of parameters (m, a, c) and initial state `S2`. The combination is made via addition modulo 1 of the outputs of the two generators.

```
unif01_Gen * ulec_CreateCombTausLCG21 (unsigned int k1, unsigned int q1,
unsigned int s1, unsigned int Y1,
unsigned int k2, unsigned int q2,
unsigned int s2, unsigned int Y2,
long m, long a, long c, long Y3);
```

Combines a combined Tausworthe generator with two components of parameters (k_1, q_1, s_1), (k_2, q_2, s_2) and initial states `Y1, Y2`, with a LCG of parameters (m, a, c) and initial state `Y3`. The combination is made by addition modulo 1 of the outputs of the two generators.

```
unif01_Gen * ulec_CreateMRG31k3p (long x10, long x11, long x12,
long x20, long x21, long x22);
```

Implements the combined MRG with two components of order 3 named `MRG31k3p` by L'Ecuyer and Touzin [98]. The two components have parameters (m, a_1, a_2, a_3) equal to ($2^{31} - 1$, 0, 2^{22}, $2^7 + 1$) and ($2^{31} - 21069$, 2^{15}, 0, $2^{15} + 1$). Its period length is close to 2^{185} and the six parameters represent the initial state. Restrictions: $0 \leq$ `x10`, `x11`, `x12` < 2147483647 and not all 0, and $0 \leq$ `x20`, `x21`, `x22` < 2147462579 and not all 0.

Clean-up functions

```
void ulec_DeleteCombTausLCG11 (unif01_Gen *gen);
```

Frees the dynamic memory allocated by `ulec_CreateCombTausLCG11`.

```
void ulec_DeleteCombTausLCG21 (unif01_Gen *gen);
```

Frees the dynamic memory allocated by `ulec_CreateCombTausLCG21`.

```
void ulec_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator of this module that does not have an explicit `Delete` function. This function should be called when a generator is no longer in use.

utezuka

This module collects some generators designed by S. Tezuka.

```
#include <testu01/unif01.h>
```

```
unif01_Gen * utezuka_CreateTezLec91 (unsigned int Y1, unsigned int Y2);
```

Implements a combined Tausworthe generator constructed by Tezuka and L'Ecuyer [159], and whose implementation is given in their paper. The initial values `Y1` and `Y2` must be positive and less than 2^{31} and 2^{29} respectively.

```
unif01_Gen * utezuka_CreateTez95 (unsigned int Y1, unsigned int Y2,
unsigned int Y3);
```

Implements the combined generator proposed in Figure A.1 of [158], page 194. The initial values `Y1, Y2, Y3` must be positive and less than 2^{28}, 2^{29} and 2^{31} respectively.

```
unif01_Gen * utezuka_CreateTezMRG95 (unsigned int Y1[5],
unsigned int Y2[7]);
```

Implements the combined generator proposed in Figure A.2 of [158], page 195. The initial values of the array elements of `Y1` and `Y2` must be positive and less than 2^{31} and 2^{29} respectively.

Clean-up functions

```
void utezuka_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator returned by the `Create` functions of this module. This function should be called to clean up any generator object of this module when it is no longer in use.

umarsa

This module implements several generators proposed in different places by George Marsaglia and his co-workers. See also the URL site `http://stat.fsu.edu/~geo/`. In the description of the generators, the symbols $\ll$ stands for the left shift operator, $\gg$ for the right shift operator, and $\oplus$ for the bitwise exclusive-or operator. In the implementations of the generators, multiplications and divisions by powers of 2 are implemented with left and right bit shifts.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

```
unif01_Gen * umarsa_CreateMarsa90a (int y1, int y2, int y3, int z0,
unsigned int Y0);
```

Implements the combination proposed by Marsaglia, Narasimhan and Zaman [113]. Its components are the subtract-with-borrow generator (SWB) (see `CreateSWB` in module `ucarry`) $X_n = (X_{n-22} - X_{n-43} - C) \bmod (2^{32} - 5)$ where C is the borrow, and the Weyl generator $Y_n = (Y_{n-1} - 362436069) \bmod 2^{32}$. The combination is done by subtraction modulo 2^{32}, i.e., $Z_n = (X_n - Y_n) \bmod 2^{32}$ and the value returned is $u_n = Z_n/2^{32}$. The first 43 values of the generator SWB are initialized by the combination of a 3-lag Fibonacci generator whose recurrence is $y_n = (y_{n-1}y_{n-2}y_{n-3}) \bmod 179$, with a LCG with recurrence $z_n = (53z_{n-1} + 1) \bmod 169$, as follows: the sixth bit of $y_i z_i \bmod 64$ is used to fill the seed numbers of the main generator, bit by bit. The parameters `y1`, `y2`, and `y3` are the seeds of the 3-lag Fibonacci sequence, while `z0` is the seed of the sequence $z_n = (53z_{n-1} + 1) \bmod 169$. Finally `Y0` is the seed of the Weyl generator. Restrictions: $0 <$ `y1, y2, y3` < 179 and $0 \leq$ `z0` < 169.

```
unif01_Gen * umarsa_CreateRANMAR (int y1, int y2, int y3, int z0);
```

Implements *RANMAR*, a combination proposed by Marsaglia, Zaman and Tsang in [119]. Its components are the lagged-Fibonacci generator $X_n = (X_{n-97} - X_{n-33}) \bmod 1$, implemented using 24-bit floating-point numbers, and the arithmetic sequence $S_n = (S_{n-1} - k) \bmod (2^{24} - 3)$. The first 97 values of the lagged-Fibonacci generator are initialized in exactly the same way as the main generator in `umarsa_CreateMarsa90a`. The parameters `y1`, `y2`, and `y3` are the seeds of the 3-lag Fibonacci sequence, while `z0` is the seed of the LCG. This generator has 24 bits of resolution. Restrictions: $0 <$ `y1, y2, y3` < 179 and $0 \leq$ `z0` < 169.

```
#ifdef USE_LONGLONG
unif01_Gen * umarsa_CreateMother0 (unsigned long x1, unsigned long x2,
unsigned long x3, unsigned long x4, unsigned long c);
#endif
```

Marsaglia [107] named this generator *"The Mother of all RNG's"*. It is a "multiply-with-carry" generator (MWC) whose recurrence is

$$\begin{aligned} Y &= 5115\, x_{n-1} + 1776\, x_{n-2} + 1492\, x_{n-3} + 2111111111\, x_{n-4} + C, \\ x_n &= Y \bmod 2^{32}, \\ C &= Y \,/\, 2^{32}, \end{aligned}$$

where C is the carry. The returned value is $x_n/2^{32}$. The four seeds `x1`, `x2`, `x3` and `x4` are the initial values of the x_i and `c` is the initial carry. Marsaglia uses `c` $= 0$ as initial value of the carry. Restrictions: $0 \leq$ `c` ≤ 2111119494 (= the sum of the coefficients of the x_n).

```
unif01_Gen * umarsa_CreateCombo (unsigned int x1, unsigned int x2,
unsigned int y1, unsigned int c);
```

Generator *Combo* proposed by Marsaglia [107]:

$$\begin{aligned} x_n &= (x_{n-1}\, x_{n-2}) \bmod 2^{32}, \\ y_n &= 30903\,(y_{n-1} \bmod 2^{16}) + y_{n-1} \operatorname{div} 2^{16}. \end{aligned}$$

The output is $u_n = z_n/2^{32}$ with the combination $z_n = (x_n + y_n) \bmod 2^{32}$. Marsaglia uses $c = 0$ as initial value of the carry. Restrictions: $y_1 < 2^{16}$ and $0 \leq c \leq 30903$.

```
unif01_Gen * umarsa_CreateECG1 (unsigned int x1, unsigned int x2,
unsigned int x3);
```

Marsaglia [107] named these "*extended congruential*" generators. This one is based on

$$x_n = (65065x_{n-1} + 67067x_{n-2} + 69069x_{n-3}) \bmod (2^{32} - 5)$$

and $u_n = x_n/(2^{32} - 5)$. Restrictions: $0 \leq x1, x2, x3 < 4294967291$.

```
unif01_Gen * umarsa_CreateECG2 (unsigned int x1, unsigned int x2,
unsigned int x3);
```

Generator based on the recurrence

$$x_n = 2^{10}(x_{n-1} + x_{n-2} + x_{n-3}) \bmod (2^{32} - 5)$$

and $u_n = x_n/(2^{32} - 5)$. Restrictions: $0 \leq x1, x2, x3 < 4294967291$.

```
unif01_Gen * umarsa_CreateECG3 (unsigned int x1, unsigned int x2,
unsigned int x3);
```

Generator based on the recurrence

$$x_n = (2000x_{n-1} + 1950x_{n-2} + 1900x_{n-3}) \bmod (2^{32} - 209)$$

and $u_n = x_n/(2^{32} - 209)$. Restrictions: $0 \leq x1, x2, x3 < 4294967087$.

```
unif01_Gen * umarsa_CreateECG4 (unsigned int x1, unsigned int x2,
unsigned int x3);
```

Generator based on the recurrence

$$x_n = 2^{20}(x_{n-1} + x_{n-2} + x_{n-3}) \bmod (2^{32} - 209)$$

and $u_n = x_n/(2^{32} - 209)$. Restrictions: $0 \leq x1, x2, x3 < 4294967087$.

`unif01_Gen * umarsa_CreateMWC97R (unsigned int x0, unsigned int y0);`

This generator proposed by Marsaglia in [108] concatenates two 16-bit multiply-with-carry generators based on the recurrences

$$\begin{aligned} x_n &= 36969\left(x_{n-1} \bmod 2^{16}\right) + x_{n-1} \operatorname{div} 2^{16}, \\ y_n &= 18000\left(y_{n-1} \bmod 2^{16}\right) + y_{n-1} \operatorname{div} 2^{16}, \\ Z_n &= \left(2^{16}x_n + y_n \bmod 2^{16}\right) \bmod 2^{32}. \end{aligned}$$

The 16 upper bits of x_n and y_n are the carries of the respective equation. The generator returns $Z_n/(2^{32}-1)$. It has been included as the default generator in the GNU package *R* under the name *Marsaglia-MultiCarry* [139].

`unif01_Gen * umarsa_CreateULTRA (unsigned int s1, unsigned int s2, unsigned int s3, unsigned int s4);`

Implements the *ULTRA* generator [107], a combination of a lagged Fibonacci generator (see `CreateLagFib` in module `umrg`) with a multiply-with-carry generator (see `CreateMWC` in module `ucarry`), proposed by Marsaglia with his test suite DIEHARD:

$$\begin{aligned} x_n &= \left(x_{n-97}\, x_{n-33}\right) \bmod 2^{32}, \\ y_n &= 30903\,\left(y_{n-1} \bmod 2^{16}\right) + y_{n-1} \operatorname{div} 2^{16}, \\ z_n &= \left(x_n + y_n\right) \bmod 2^{32}. \end{aligned}$$

The generator returns $z_n/2^{32}$. This agrees with the effective implementation in DIEHARD which does not agree with its documentation. The four seeds `s1`, `s2`, `s3` and `s4` are used in a complicated way to initialize the component generators.

`unif01_Gen * umarsa_CreateSupDup73 (unsigned int x0, unsigned int y0);`

Implements the original *SuperDuper* generator [112], a combination of a congruential generator with a shift-register generator:

$$\begin{aligned} x_n &= 69069\, x_{n-1} \bmod 2^{32}, \\ t &= y_{n-1} \oplus \left(y_{n-1} \operatorname{div} 2^{15}\right), \\ y_n &= t \oplus \left(2^{17}t \bmod 2^{32}\right), \\ z_n &= x_n \oplus y_n \end{aligned}$$

The generator returns $z_n/(2^{32}-1)$. The seeds `x0` and `y0` initializes the x_n and y_n. Restriction: `x0` must be odd.

`unif01_Gen * umarsa_CreateSupDup96Add (unsigned int x0, unsigned int y0, unsigned int c);`

Implements the *SuperDuper* generator, an additive combination of a congruential generator with a shift-register generator, proposed by Marsaglia with his test suite DIEHARD [107]:

$$x_n = \left(69069\, x_{n-1} + c\right) \bmod 2^{32},$$

$$
\begin{aligned}
t &= y_{n-1} \oplus (2^{13} y_{n-1}), \\
t &= t \oplus (t \text{ div } 2^{17}), \\
y_n &= (t \oplus (2^5 t)) \bmod 2^{32}, \\
z_n &= (x_n + y_n) \bmod 2^{32}
\end{aligned}
$$

The generator returns $z_n/2^{32}$. This is the uniform generator (called `randuni`) included in MATLAB that is used to generate normal random variables. Restriction: `c` odd.

```
unif01_Gen * umarsa_CreateSupDup96Xor (unsigned int x0, unsigned int y0,
unsigned int c);
```

Similar to `umarsa_CreateSupDup96Add` above, except that the combination of the two generators is with a bitwise exclusive-or:

$$z_n = x_n \oplus y_n$$

```
#ifdef USE_LONGLONG
unif01_Gen * umarsa_CreateSupDup64Add (ulonglong x0, ulonglong y0,
ulonglong a, ulonglong c,
int s1, int s2, int s3);
#endif
```

Implements the 64-bit generator *supdup64*, an additive combination of a congruential generator with a shift-register generator, proposed by Marsaglia in [110]:

$$
\begin{aligned}
x_n &= (a\, x_{n-1} + c) \bmod 2^{64}, \\
t &= y_{n-1} \oplus (2^{s_1} y_{n-1}), \\
t &= t \oplus (t \text{ div } 2^{s_2}), \\
y_n &= (t \oplus (2^{s_3} t)) \bmod 2^{64}, \\
z_n &= (x_n + y_n) \bmod 2^{64}
\end{aligned}
$$

The generator returns $z_n/2^{64}$ using only the 32 most significant bits of z_n and setting the others to 0. In his post, Marsaglia suggests the values $a = 6906969069$, $c = 1234567$, $s_1 = 13$, $s_2 = 17$ and $s_3 = 43$. Restrictions: $a = 3 \bmod 8$ or $a = 5 \bmod 8$.

```
#ifdef USE_LONGLONG
unif01_Gen * umarsa_CreateSupDup64Xor (ulonglong x0, ulonglong y0,
ulonglong a, ulonglong c,
int s1, int s2, int s3);
#endif
```

Similar to `umarsa_CreateSupDup64Add` above, except that the combination of the two generators is with a bitwise exclusive-or:

$$z_n = x_n \oplus y_n$$

```
unif01_Gen * umarsa_CreateKISS93 (unsigned int x0, unsigned int y0,
unsigned int z0);
```

Implements the generator *KISS* proposed by Marsaglia in [117], which is a combination of a LCG sequence with two 2-shifts register sequences:

$$\begin{aligned} x_n &= (69069\, x_{n-1} + 23606797) \bmod 2^{32}, \\ t &= y_{n-1} \oplus \left(2^{17} y_{n-1}\right), \\ y_n &= t \oplus \left(t \text{ div } 2^{15}\right) \bmod 2^{32}, \\ t &= \left(z_{n-1} \oplus \left(2^{18} z_{n-1}\right)\right) \bmod 2^{31}, \\ z_n &= t \oplus \left(t \text{ div } 2^{13}\right) \end{aligned}$$

The generator returns $\left((x_n + y_n + z_n) \bmod 2^{32}\right) / 2^{32}$. Restrictions: $0 \leq$ `z0` $< 2^{31}$.

```
unif01_Gen * umarsa_CreateKISS96 (unsigned int x0, unsigned int y0,
unsigned int z1, unsigned int z2);
```

Implements the generator *KISS* proposed by Marsaglia in his test suite DIEHARD [107]:

$$\begin{aligned} x_n &= (69069\, x_{n-1} + 1) \bmod 2^{32}, \\ t &= y_{n-1} \oplus \left(2^{13} y_{n-1}\right), \\ t &= t \oplus \left(t \text{ div } 2^{17}\right), \\ y_n &= \left(t \oplus \left(2^5 t\right)\right) \bmod 2^{32}, \\ z_n &= (2z_{n-1} + z_{n-2} + c_{n-1}) \bmod 2^{32}, \\ c_n &= (2z_{n-1} + z_{n-2} + c_{n-1}) \text{ div } 2^{32}, \end{aligned}$$

where the x_n are a LCG sequence, the y_n are a 3-shifts register sequence, and the z_n are a simple multiply-with-carry sequence with c_n as the carry (see `CreateMWC` in module `ucarry`). The variable x_0 is the seed of the LCG component, y_0 is the seed of the shift register component, and z_1, z_2 are the seeds of the multiply-with-carry sequence. The generator returns $\left((x_n + y_n + z_n) \bmod 2^{32}\right) / 2^{32}$.

```
unif01_Gen * umarsa_CreateKISS99 (unsigned int x0, unsigned int y0,
unsigned int z1, unsigned int z2);
```

Implements the generator *KISS* proposed by Marsaglia in [109]. It is a combination of a LCG, a 3-shifts register generator, and two multiply-with-carry generators:

$$\begin{aligned} x_n &= (69069\, x_{n-1} + 1234567) \bmod 2^{32}, \\ t &= y_{n-1} \oplus \left(2^{17} y_{n-1}\right), \\ t &= t \oplus \left(t \text{ div } 2^{13}\right), \\ y_n &= \left(t \oplus \left(2^5 t\right)\right) \bmod 2^{32}, \\ Z_n &= \text{as in } \texttt{umarsa\_CreateMWC97R} \text{ above} \end{aligned}$$

where x_0 is the seed of the LCG component, y_0 the seed of the 3-shifts register component, and z_1, z_2 the seeds of the multiply-with-carry generators. The generator returns $\left(((Z_n \oplus x_n) + y_n) \bmod 2^{32}\right) / 2^{32}$.

`unif01_Gen * umarsa_Create4LFIB99 (unsigned int T[256]);`

Implements the 4-lag lagged Fibonacci generator *LFIB4* proposed by Marsaglia in [109]. It uses addition in the form (see also `CreateLagFib` in module `umrg`)

$$T_n = (T_{n-55} + T_{n-119} + T_{n-179} + T_{n-256}) \bmod 2^{32}.$$

The generator returns $T_n/2^{32}$. Its period is close to 2^{287}.

`unif01_Gen * umarsa_Create3SHR99 (unsigned int y0);`

Implements the 3-shift random number generator *SHR3* proposed by Marsaglia in [109]:

$$\begin{aligned} t &= y_{n-1} \oplus \left(2^{17} y_{n-1}\right), \\ t &= t \oplus \left(t \text{ div } 2^{13}\right), \\ y_n &= \left(t \oplus \left(2^5 t\right)\right) \bmod 2^{32}. \end{aligned}$$

The generator returns $y_n/2^{32}$ and its period is $2^{32} - 1$.

`unif01_Gen * umarsa_CreateSWB99 (unsigned int T[256], int b);`

Implements the subtract-with-borrow generator *SWB* proposed by Marsaglia in [109]:

$$\begin{aligned} b_n &= I[T_{n-222} < T_{n-237} + b_{n-1}], \\ T_n &= (T_{n-222} - T_{n-237} - b_{n-1}) \bmod 2^{32}, \end{aligned}$$

where b_n is the borrow and I is the indicator function (see `CreateSWB` in module `ucarry`). The generator returns $T_n/2^{32}$ and its period is close to 2^{7578}.

Clean-up functions

`void umarsa_DeleteGen (unif01_Gen *gen);`

Frees the dynamic memory used by any generator returned by the `Create` functions of this module. This function should be called to clean up any generator object of this module when it is no longer in use.

uknuth

This module collects generators proposed by Donald E. Knuth. Knuth's code can be found at `http://www-cs-faculty.stanford.edu/~knuth/programs.html`. Since there are global variables in this module, no more than one generator of each type in this module can be in use at any given time.

```
#include <testu01/unif01.h>
```

```
unif01_Gen * uknuth_CreateRan_array1 (long s, long A[100]);
```

Implements the generator `ran_array` in its first version as appeared on Knuth's web site in 2000. It is based on the lagged Fibonacci sequence with subtraction [66], modified via Lüscher's method. It generates 1009 numbers from the recurrence

$$X_j = (X_{j-100} - X_{j-37}) \bmod 2^{30}$$

out of which only the first 100 are used and the next 909 are discarded, and this process is repeated. The generator returns $U_j = X_j/2^{30}$. Gives 30 bits of precision.

If the seed $s \geq 0$, then Knuth's initialization procedure is performed: it shifts and transforms the bits of s in order to get the 100 numbers that make up the initial state; in that case, array `A` is unused. If $s < 0$, then the initial state is taken from the array `A[0..99]`. This could be convenient for restarting the generator from a previously saved state. Restrictions: $s \leq 1073741821$.

```
unif01_Gen * uknuth_CreateRan_array2 (long s, long A[100]);
```

This implements the new version of `ran_array` with a new initialization procedure as appeared on Knuth's web site in 2002.

```
unif01_Gen * uknuth_CreateRanf_array1 (long s, double B[100]);
```

Similar generator to `ran_array1` above, but where the recurrence is

$$U_j = (U_{j-100} + U_{j-37}) \bmod 1$$

and is implemented directly in floating-point arithmetic. This implements the first version of Knuth's `ranf_array` as it appeared on his web site in 2000. Array `B` contains numbers in [0, 1).

```
unif01_Gen * uknuth_CreateRanf_array2 (long s, double B[100]);
```

This implements the new version of `ranf_array` as it appeared on Knuth's web site in 2002.

Clean-up functions

```
void uknuth_DeleteRan_array1  (unif01_Gen *gen);
void uknuth_DeleteRan_array2  (unif01_Gen *gen);
void uknuth_DeleteRanf_array1 (unif01_Gen *gen);
void uknuth_DeleteRanf_array2 (unif01_Gen *gen);
```

Frees the dynamic memory used by the generators of this module, and allocated by the corresponding `Create` function.

utouzin

This module is an interface to random number generators proposed by P. L'Ecuyer and R. Touzin [164]. They are multiple recursive generators (MRG) or combinations of MRG's with coefficients of the form $\pm 2^p \pm 2^q$ (see the description of MRG's in module `umrg`).

```
#include <testu01/unif01.h>
```

```
unif01_Gen * utouzin_CreateMRG00a (long s1, long s2, long s3, long s4,
long s5);
```

Creates a MRG of order 5 of the form

$$x_n = \left((2 - 2^{24})x_{n-1} - 2^{18}x_{n-3} - 2^4 x_{n-4} + (2^{11} - 1)x_{n-5}\right) \bmod m$$

where $m = 2^{31} - 1$ and $u_n = x_n/m$. The parameters s are the seeds.

```
unif01_Gen * utouzin_CreateMRG00b (long s1, long s2, long s3, long s4,
long s5, long s6);
```

Creates a MRG of order 6 of the form

$$x_n = \left((-2^{21} - 1)x_{n-1} - 2^{12}x_{n-2} + 2^{16}x_{n-3} + 2^7 x_{n-5} + (1 - 2^{27})x_{n-6}\right) \bmod m$$

where $m = 2^{31} - 1$ and $u_n = x_n/m$. The parameters s are the seeds.

```
unif01_Gen * utouzin_CreateMRG00c (long s1, long s2, long s3, long s4,
long s5, long s6, long s7);
```

Creates a MRG of order 7 of the form

$$x_n = \left(-2^{12}x_{n-1} - 2^{20}x_{n-2} + 2^{14}x_{n-3} + 2^{25}x_{n-5} - 2^6 x_{n-6} + (2^4 + 1)x_{n-7}\right) \bmod m$$

where $m = 2^{31} - 19$ and $u_n = x_n/m$. The parameters s are the seeds.

```
unif01_Gen * utouzin_CreateMRG00d (long s1, long s2, long s3, long s4,
long s5, long s6, long s7, long s8);
```

Creates a MRG of order 8 of the form

$$x_n = \left(-2^4 x_{n-1} + 2^{15}x_{n-3} - 2^{12}x_{n-4} + 2^{22}x_{n-5} + 2^9 x_{n-6} + 2^{27}x_{n-7} + (2^{18} - 2)x_{n-8}\right) \bmod m$$

where $m = 2^{31} - 1$ and $u_n = x_n/m$. The parameters s are the seeds.

```
unif01_Gen * utouzin_CreateMRG00e (long s10, long s11, long s12,
long s20, long s21, long s22);
```

Creates a combined MRG with two components of order 3 of the form

$$\begin{aligned} x_n &= \left(2^{22}x_{n-2} + (2^7 + 1)x_{n-3}\right) \bmod m_1 \\ y_n &= \left(2^{15}y_{n-1} + (2^{15} + 1)y_{n-3}\right) \bmod m_2 \end{aligned}$$

where $m_1 = 2^{31} - 1$, $m_2 = 2^{31} - 21069$, and $u_n = ((x_n - y_n) \bmod m_1)/(m_1 + 1)$ with the exception of the value 0 which is replaced by $u_n = m_1/(m_1 + 1)$. Thus the generator cannot return the values 0 or 1. The parameters s are the seeds.

```
unif01_Gen * utouzin_CreateMRG00f (long s10, long s11, long s12,
long s20, long s21, long s22);
```

Creates a combined MRG with two components of order 3 of the form

$$\begin{aligned} x_n &= \left(2^{14}x_{n-2} + (-2^{26} + 1)x_{n-3}\right) \bmod m_1 \\ y_n &= \left(2^{17}y_{n-1} + 2^{11}y_{n-3}\right) \bmod m_2 \end{aligned}$$

where $m_1 = 2^{31} - 1$, $m_2 = 2^{31} - 19$, and $u_n = ((x_n - y_n) \bmod m_1)/(m_1 + 1)$ with the exception of the value 0 which is replaced by $u_n = m_1/(m_1 + 1)$. Thus the generator cannot return the values 0 or 1. The parameters s are the seeds.

```
unif01_Gen * utouzin_CreateMRG00g (long s10, long s11, long s12,
long s20, long s21, long s22,
long s30, long s31, long s32);
```

Creates a combined MRG with three components of order 3 of the form

$$\begin{aligned} x_n &= \left(2^{30}x_{n-1} + (2^{19} - 1)x_{n-3}\right) \bmod m_1 \\ y_n &= \left(2^{23}y_{n-2} + 2^{19}y_{n-3}\right) \bmod m_2 \\ z_n &= \left(2^{11}z_{n-1} + 2^{9}z_{n-2} + 2z_{n-3}\right) \bmod m_3 \end{aligned}$$

where $m_1 = 2^{31} - 1$, $m_2 = 2^{31} - 19$, $m_3 = 2^{31} - 61$ and $u_n = ((x_n - y_n + z_n) \bmod m_1)/(m_1 + 1)$ with the exception of the value 0 which is replaced by $u_n = m_1/(m_1 + 1)$. Thus the generator cannot return the values 0 or 1. The parameters s are the seeds.

```
unif01_Gen * utouzin_CreateMRG00h (long s10, long s11, long s12, long s13,
long s20, long s21, long s22, long s23);
```

Creates a combined MRG with two components of order 4 of the form

$$\begin{aligned} x_n &= \left(-x_{n-1} - 2^{13}x_{n-2} + (2^{23} + 1)x_{n-4}\right) \bmod m_1 \\ y_n &= \left(2^{10}y_{n-1} - 2^{20}y_{n-3} + 2^{7}y_{n-4}\right) \bmod m_2 \end{aligned}$$

where $m_1 = 2^{31} - 1$, $m_2 = 2^{31} - 19$, and $u_n = ((x_n - y_n) \bmod m_1)/(m_1 + 1)$ with the exception of the value 0 which is replaced by $u_n = m_1/(m_1 + 1)$. Thus the generator cannot return the values 0 or 1.The parameters s are the seeds.

Clean-up functions

```
void utouzin_DeleteGen (unif01_Gen * gen);
```

Frees the dynamic memory used by any generator of this module. This function should be called when a generator is no longer in use.

ugranger

This module collects combined generators implemented by Jacinthe Granger-Piché for her master thesis. Some of the generators in this module use the GNU multiprecision package GMP. The macro `USE_GMP` is defined in module `gdef` in directory `mylib`.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

```
unif01_Gen * ugranger_CreateCombLCGInvExpl (
long m1, long a1, long c1, long s1, long m2, long a2, long c2);
```

Combines an LCG of parameters (m_1, a_1, c_1) and initial state s_1 with a non-linear explicit inversive generator with parameters (m_2, a_2, c_2). The implementation of the LCG uses either `ulcg_CreateLCGFloat` or `ulcg_CreateLCG`, depending on the parameters (m_1, a_1, c_1), and the implementation of the inversive generator uses `uinv_CreateInvExpl`. The combination is done by adding mod 1 the outputs of the two generators. Restrictions: the same as those for `ulcg_CreateLCG` and `uinv_CreateInvExpl`.

```
#ifdef USE_GMP
unif01_Gen * ugranger_CreateCombBigLCGInvExpl (
char *m1, char *a1, char *c1, char *s1, long m2, long a2, long c2);
```

Same as `ugranger_CreateCombLCGInvExpl`, but the LCG is implemented using arbitrary large integers with `ulcg_CreateBigLCG`. Restrictions: the same as those for `ulcg_CreateBigLCG` and `uinv_CreateInvExpl`.

```
#endif
```

```
unif01_Gen * ugranger_CreateCombLCGCub (
long m1, long a1, long c1, long s1, long m2, long a2, long s2);
```

Combines an LCG of parameters (m_1, a_1, c_1) and initial state s_1 with a cubic generator of parameters (m_2, a_2) and initial state s_2. The LCG implementation is either `ulcg_CreateLCGFloat` or `ulcg_CreateLCG`, depending on the parameters (m_1, a_1, c_1), and the implementation of the cubic generator is `ucubic_CreateCubic1Float`. The combination is done by adding mod 1 the outputs of the two generators. Restrictions: the same as those for `ulcg_CreateLCG` and `ucubic_CreateCubic1Float`.

```
#ifdef USE_GMP
unif01_Gen * ugranger_CreateCombBigLCGCub (
char *m1, char *a1, char *c1, char *s1, long m2, long a2, long c2);
```

Same as `ugranger_CreateCombCubLCG`, but the LCG is implemented using arbitrary large integers with `ulcg_CreateBigLCG`. Restrictions: the same as those for `ulcg_CreateBigLCG` and `ucubic_CreateCubic1Float`.

```
unif01_Gen * ugranger_CreateCombTausBigLCG (
unsigned int k1, unsigned int q1, unsigned int s1, unsigned int SS1,
unsigned int k2, unsigned int q2, unsigned int s2, unsigned int SS2,
char *m, char *a, char *c, char *SS3);
```

Combines a Tausworthe generator with two components of parameters (k_1, q_1, s_1), (k_2, q_2, s_2) and initial states *SS1*, *SS2* with an LCG of parameters (m, a, c) and initial state *SS3*. The combination is done by adding mod 1 the outputs of the LCG and of the combined Tausworthe. The implementation of the LCG is the one in `ulcg_CreateBigLCG`, and the implementation of the combined Tausworthe is the one in `utaus_CreateCombTaus2`. Restrictions: the same as those for `utaus_CreateCombTaus2` and `ulcg_CreateBigLCG`.

```
#endif
```

```
unif01_Gen * ugranger_CreateCombTausLCG21xor (
unsigned int k1, unsigned int q1, unsigned int s1, unsigned int SS1,
unsigned int k2, unsigned int q2, unsigned int s2, unsigned int SS2,
long m, long a, long c, long SS3);
```

Combines a Tausworthe generator with two components of parameters (k_1, q_1, s_1), (k_2, q_2, s_2) and initial states *SS1*, *SS2* with an LCG of parameters (m, a, c) and initial state *SS3*. The combination is done using a bitwise exclusive-or of the outputs of the two generators. The implementation of the LCG is either the one in `ulcg_CreateLCGFloat` or in `ulcg_CreateLCG`, depending on the parameters (m, a, c), and the implementation of the combined Tausworthe is the one in `utaus_CreateCombTaus2`. Restrictions: the same as those for `utaus_CreateCombTaus2` and `ulcg_CreateLCG`.

```
unif01_Gen * ugranger_CreateCombTausCub21xor (
unsigned int k1, unsigned int q1, unsigned int s1, unsigned int SS1,
unsigned int k2, unsigned int q2, unsigned int s2, unsigned int SS2,
long m, long a, long SS3);
```

Combines a Tausworthe generator with two components of parameters (k_1, q_1, s_1), (k_2, q_2, s_2) and initial states *SS1*, *SS2* with a cubic generator of parameters (m, a) and initial state *SS3*. The combination is done using a bitwise exclusive-or of the outputs of the two generators. The implementation of the combined Tausworthe is the one in `utaus_CreateCombTaus2`, and the implementation of the cubic generator is the one in `ucubic_CreateCubic1Float`. Restrictions: the same as those for `utaus_CreateCombTaus2` and `ucubic_CreateCubic1Float`.

```
unif01_Gen * ugranger_CreateCombTausInvExpl21xor (
unsigned int k1, unsigned int q1, unsigned int s1, unsigned int SS1,
unsigned int k2, unsigned int q2, unsigned int s2, unsigned int SS2,
long m, long a, long c);
```

Combines a Tausworthe generator with two components of parameters (k_1, q_1, s_1), (k_2, q_2, s_2) and initial states *SS1*, *SS2* with an explicit inversive generator of parameters (m, a, c). The combination is done using a bitwise exclusive-or of the outputs of the two generators. The implementation of the combined Tausworthe is the one in `utaus_CreateCombTaus2`, and the implementation of the inversive generator is the one in `uinv_CreateInvExpl`. Restrictions: the same as those for `utaus_CreateCombTaus2` and `uinv_CreateInvExpl`.

Clean-up functions

```
void ugranger_DeleteCombLCGInvExpl (unif01_Gen *gen);
void ugranger_DeleteCombLCGCub (unif01_Gen *gen);
void ugranger_DeleteCombTausLCG21xor (unif01_Gen *gen);
void ugranger_DeleteCombTausCub21xor (unif01_Gen *gen);
void ugranger_DeleteCombTausInvExpl21xor (unif01_Gen *gen);

#ifdef USE_GMP
void ugranger_DeleteCombBigLCGInvExpl (unif01_Gen *gen);
void ugranger_DeleteCombBigLCGCub (unif01_Gen *gen);
void ugranger_DeleteCombTausBigLCG (unif01_Gen *gen);
#endif
```

Frees the dynamic memory allocated by the corresponding `Create` function of this module.

uwu

This module collects some generators from Pei-Chi Wu.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>


#ifdef USE_LONGLONG
unif01_Gen * uwu_CreateLCGWu61a (longlong s);
```

Implements a LCG proposed by Wu [177], with $m = 2^{61} - 1$, $a = 2^{30} - 2^{19}$, $c = 0$. Uses a fast implementation with shifts rather than multiplications. It uses 64-bits integers.

```
unif01_Gen * uwu_CreateLCGWu61b (longlong s);
```

Similar to `uwu_CreateLCGWu61a`, but with $a = 2^{42} - 2^{31}$.

```
#endif
```

Clean-up functions

```
void uwu_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator of this module that does not have an explicit `Delete` function. This function should be called to clean up a generator object when it is no longer in use.

See also

- `ulcg_CreateLCGWu2`

udeng

This module collects some generators from Lih-Yuan Deng and his collaborators.

```
#include <testu01/unif01.h>
```

```
unif01_Gen * udeng_CreateDL00a (unsigned long m, unsigned long b, int k,
unsigned long S[]);
```

Creates a multiple recursive generator proposed by Deng and Lin [22] in the form:

$$x_i \;=\; ((m-1)x_{i-1} + bx_{i-k}) \bmod m \;=\; (-x_{i-1} + bx_{i-k}) \bmod m.$$

The generator returns $u_i = x_i/m$. The initial state $(x_{-1}, \ldots, x_{-k})$ is in `S[0..(k-1)]`. Restriction: $k \leq 128$.

```
unif01_Gen * udeng_CreateDX02a (unsigned long m, unsigned long b, int k,
unsigned long S[]);
```

Creates a multiple recursive generator proposed by Deng and Xu [23] in the form:

$$x_i \;=\; b(x_{i-1} + x_{i-k}) \bmod m.$$

The generator returns $u_i = x_i/m$. The initial state $(x_{-1}, \ldots, x_{-k})$ is in `S[0..(k-1)]`. Restriction: $k \leq 128$.

Clean-up functions

```
void udeng_DeleteGen (unif01_Gen * gen);
```

Frees the dynamic memory used by any generator of this module that does not have an explicit `Delete` function. This function should be called when a generator is no longer in use.

uweyl

This module implements simple and combined generators based on Weyl sequences, proposed by Holian et al. [53].

```
#include <testu01/unif01.h>
```

```
unif01_Gen * uweyl_CreateWeyl (double alpha, long n0);
```

Implements a generator defined by the Weyl sequence:

$$u_n = n\alpha \bmod 1 = (u_{n-1} + \alpha) \bmod 1, \tag{2.27}$$

where $\alpha =$ `alpha` is a real number in the interval $(0, 1)$. The initial value of n is `n0`. In theory, if α is irrationnal, this sequence is asymptotically equidistributed over (0,1) [170]. However, this is not true for the present implementation, because α is represented only with finite precision. The implementation is only a rough approximation, valid when n is not too large. Some possible values for α are:

$$\begin{aligned} \sqrt{2} \bmod 1 &= 0.414213562373095 \\ \sqrt{3} \bmod 1 &= 0.732050807568877 \\ \pi \bmod 1 &= 0.141592653589793 \\ e \bmod 1 &= 0.718281828459045 \\ \gamma &= 0.577215664901533 \end{aligned}$$

```
unif01_Gen * uweyl_CreateNWeyl (double alpha, long n0);
```

Implements a nested Weyl generator, as suggested in [53], defined by

$$u_n = (n\,(n\alpha \bmod 1)) \bmod 1, \tag{2.28}$$

where `alpha` $= \alpha \in (0, 1)$. The initial value of n is `n0`.

```
unif01_Gen * uweyl_CreateSNWeyl (long m, double alpha, long n0);
```

Implements a nested Weyl generator with "shuffling", proposed in [53], and defined by

$$\nu_n = m\,(n\,(n\alpha \bmod 1) \bmod 1) + 1/2, \qquad u_n = (\nu_n\,(\nu_n \alpha \bmod 1)) \bmod 1,$$

where m is a large positive integer and `alpha` $= \alpha \in (0, 1)$. The initial value of n is `n0`.

Clean-up functions

```
void uweyl_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator returned by the `Create` functions of this module. This function should be called to clean up any generator object of this module when it is no longer in use.

unumrec

Implements the generators proposed in *Numerical Recipes: Portable Random Number Generators* [138, 137].

```
#include <testu01/unif01.h>
```

```
unif01_Gen * unumrec_CreateRan0 (long s);
```

Creates and initializes the generator `Ran0` with the seed s. Restriction: $0 < s < 2^{31}$.

```
unif01_Gen * unumrec_CreateRan1 (long s);
```

Creates and initializes the generator `Ran1` with the seed s. Restriction: $0 < s < 2^{31}$.

```
unif01_Gen * unumrec_CreateRan2 (long s);
```

Creates and initializes the generator `Ran2` with the seed s. Restriction: $0 < s < 2^{31}$.

Clean-up functions

```
void unumrec_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator returned by the `Create` functions of this module. This function should be called to clean up any generator object of this module when it is no longer in use.

uautomata

This module implements generators based on cellular automata. A cellular automaton consists of a d-dimensional grid of cells, whose coordinates are the integer points in the d-dimensional euclidean lattice $\mathcal{L} = \mathbb{Z}^d$. Each cell can hold a value taken from a discrete set (for now, only binary values 0 and 1 are implemented). The value x at each cell i evolves deterministically with (discrete) time according to a set of rules involving the values of its nearest neighbours. For a one-dimensional cellular automaton with neighbourhood of radius r, the value of a cell at a given time depends on its value at the previous time step as well as the values of the r closest cells on the left and the r closest cells on the right, all at the previous time step. The evolution of cell i can thus be written as

$$x_i^{(t+1)} = F\left[x_{i-r}^{(t)},\ \ldots, x_i^{(t)},\ \ldots,\ x_{i+r}^{(t)}\right],$$

where t represents discrete time. These rules are applied synchronously to each cell at every time step. Here, only uniform cellular automata are considered, for which the rules are identical for all cell. See [175, 176, 162] for the theory of cellular automata.

In the current implementation, periodic boundary conditions are imposed on the grid of cells, so that cells on oppposite boundaries are considered adjacent. For example, for a one-dimensional grid with N cells, the condition $S_N = S_0$ applies. Similarly, a two-dimensional grid is considered as a torus.

```
#include <testu01/unif01.h>

unif01_Gen * uautomata_CreateCA1 (int N, int S[ ], int r, int F[ ],
int k, int ts, int cs, int rot);
```

Initializes a generator based on a 1-dimensional boolean uniform cellular automaton made up of N cells, with a rule F of radius r, and an initial state S. A rule of radius r is such that only the r nearest neighbors on each side of a cell are involved in determining the value of the cell at the next time step. Thus each cell has $2r + 1$ neighbors, including itself. The initial value of cell i is given in $S[i]$ and can take values 0 and 1 only.

The rule is specified by the 2^{2r+1} elements of array F indexed in standard numerical order, with an entry for every possible neighborhood configuration of states. A given entry is such that at the next time step, cell i takes the value obtained from the rule when the $2r + 1$ neighbour cell values are given by the binary representation of j. The following table shows an example of a local rule when $r = 1$.

j	7	6	5	4	3	2	1	0
x_{i-1}, x_i, x_{i+1}	111	110	101	100	011	010	001	000
new x_i	0	0	0	1	1	1	1	0

Each of the $2^{2r+1} = 8$ possible sets of values for a cell and its 2 nearest neighbours appear on the middle line, while the lower line gives the value to be taken by the central cell on the next time step. For this rule, array F must have the following form: $F = \{0, 1, 1, 1, 1, 0, 0, 0\}$. In

Wolfram's numbering scheme for one-dimensional automata [175] and in the litterature, this rule is called rule 30 because the lower line of the table is the binary representation of 30.

In order to generate random numbers from this automaton, only k cells are used, starting count at the center of the grid. Assuming that the parameters `ts` and `cs` are 0, then 32 time steps will be used to generate k random integers, the 32 bits of a cell over time making up one random number. For example, if $N = 10$ and $k = 3$, then only cells 4, 5, 6 will be used to generate random numbers, though all the cells contribute to the evolution of the cellular automaton.

The parameters `ts` and `cs` implements time spacings and cell spacings respectively. Thus only the bits generated at every `ts` + 1 time step are considered as part of the random sequence, the bits generated at the `ts` successive time steps in-between are disregarded. For example, if `ts` = 1, one keeps only the bits at 1 time step out of 2 to build the random numbers. The default value is `ts` = 0. Similarly, only cells spaced `cs` + 1 apart are used to generate random numbers; the output of the `cs` cells in-between is not considered part of the random sequence, though they still contribute to the evolution of the cellular automaton. For example, if $N = 20$, $k = 3$ and `cs` = 2, then only the bits generated by cells 7, 10, 13 are used to make up the random numbers returned by the cellular automaton. The default value is `cs` = 0.

The parameter `rot` indicates a circular shift of the cells at each time step. If `rot` > 0, the value of cell i at the end of each time step will become the value of cell $(i +$ `rot`$)$ mod N before going to the next time step. If `rot` < 0, cell i will become the value of cell $(i -$ `rot`$)$ mod N instead. There is no shift when `rot` $= 0$.

Restrictions: $k * ($`cs` $+ 1) \leq N +$ `cs`.

```
unif01_Gen * uautomata_CreateCA90mp (int m, int S[]);
```

Implements Matsumoto's cellular automaton $CA90(m)'$ (see [123]). It is a uniform boolean one-dimensional automaton with m cells based on rule 90 (as defined by Wolfram in [174]), i.e., the value of a cell at time $t + 1$ depends only on the state of its two closest neighbors at time t and is given by $x_i(t + 1) = x_{i-1}(t) + x_{i+1}(t)$ mod 2. There are two extra cells that implements the boundary conditions at both ends. The null boundary condition, $x_0(t) \equiv 0$, is applied permanently at the left end, while the mirror boundary condition, $x_{m+1}(t) = x_m(t)$, is applied permanently at the right end. The output is the value of cell m. Thus each time step generates one bit of output and 32 time steps generate one 32-bit integer. The initial state of the cells must be given in $S[j]$ for $j = 1, 2, \ldots, m$. Restriction: $S[j] \in \{0, 1\}$.

Clean-up functions

```
void uautomata_DeleteCA90mp (unif01_Gen *gen);
```

Frees the dynamic memory allocated by `uautomata_CreateCA90mp`.

```
void uautomata_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator of this module that does not have an explicit `Delete` function. This function should be called when a generator is no longer in use.

ucrypto

This module implements different versions of some random number generators proposed or used in the world of cryptology.

```
#include <testu01/unif01.h>
```

```
typedef enum {
ucrypto_OFB,                    /* Output Feedback mode */
ucrypto_CTR,                    /* Counter mode */
ucrypto_KTR                     /* Key counter mode */
} ucrypto_Mode;
```

Block modes of operation [26] for this module. Given an algorithm (for example, encryption or hashing) used as a generator of random numbers, then the output feedback mode (`OFB`) uses the result of the last application of the algorithm as input block for the current application. The counter mode (`CTR`) applies the algorithm on a counter used as input and incremented by 1 at each application. The key counter mode (`KTR`) applies the algorithm on the seed with a different key at each application of the algorithm; the key is incremented by 1 before each application.

```
unif01_Gen * ucrypto_CreateAES (unsigned char *Key, int klen,
unsigned char *Seed, ucrypto_Mode mode,
int r, int s);
```

Uses the *Advanced Encryption standard* (AES) as a source of random numbers [19, 130, 52, 3], based on the optimized C code for the Rijndael cipher written by V. Rijmen, A. Bosselærs and P. Barreto [144]. `klen` is the number of bits in the cipher `Key`, which must be given as an array of $16, 24$ or 32 bytes for a key of 128, 192 or 256 bits, respectively. `Seed` is the initial state, which must be an array of 16 bytes making in all 128 bits. At each encryption step j, the AES encryption algorithm is applied on the input block to obtain a new block of 128 bits (16 bytes). Of these, the first r bytes are dropped and the next s bytes are used to build 32-bit random numbers. Each call to the generator returns a 32-bit random number. For example, if $r = 2$ and $s = 8$, then the 16 $(8r)$ most significant bits of the block are dropped and the next 64 $(8s)$ bits are used to make two 32-bit random numbers which will be returned by the next two calls to the generator. Restrictions: `klen` $\in \{128, 192, 256\}$, $0 \leq r \leq 15$, $1 \leq s \leq 16$, and $r + s \leq 16$.

Let $C = E(K, T)$ denote the AES encryption operation with key K on plain text T resulting in encrypted text C.

- For the `OFB` mode, each new block of 128 bits C_j is obtained by $C_j = E(K, C_{j-1})$, where $C_0 =$ `Seed`.
- The `CTR` mode uses a 128-bit counter i whose initial value is equal to `Seed`, and which is incremented by 1 at each encryption step j. Each new block of 128 bits C_j is obtained by $C_j = E(K, i)$.
- The `KTR` mode uses a counter i as the key which is incremented by 1 at each encryption step j as $i \leftarrow i + 1$. Each new block of 128 bits C_j is obtained by $C_j = E(i,$ `Seed`$)$, where the initial value of i is `Key` viewed as an integer.

```
unif01_Gen * ucrypto_CreateSHA1 (unsigned char *Seed, int len,
ucrypto_Mode mode, int r, int s);
```

Uses the *Secure Hash Algorithm* SHA-1 as a source of random numbers [131, 3]. `Seed` is an array of size `len` used to initialize the generator. At each hashing step j, the SHA-1 algorithm is applied on the input block to obtain a hashed string of 160 bits (20 bytes). Of these, the first r bytes are dropped and the next s bytes are used to build 32-bit random numbers. Each call to the generator returns a 32-bit random number. For example, if $r = 2$ and $s = 8$, then the 16 ($8r$) most significant bits of the 160-bit string are dropped and the next 64 ($8s$) bits are used to make two 32-bit random numbers which will be returned by the next two calls to the generator. Restrictions: `len` ≤ 55, $0 \leq r \leq 19$, $1 \leq s \leq 20$, and $r + s \leq 20$.

Let $C = H(T)$ denote the SHA-1 operation applied on the original text T hashed to the 160-bit string C. (When T is too short, it is padded automatically by the SHA-1 algorithm to have the required block length of 512 bits.)

- For the `OFB` mode, each new block of 160 C_j is obtained by $C_j = H(C_{j-1})$, where $C_0 = H($`Seed`$)$.
- The `CTR` mode uses a 440-bit counter i whose initial value is equal to `Seed`, and which is incremented by 1 at each hashing step j. Each new block of 160 bits C_j is obtained by $C_j = H(i)$.

```
unif01_Gen * ucrypto_CreateISAAC (int flag, unsigned int A[256]);
```

This is the generator `ISAAC` (Indirection, Shift, Accumulate, Add, and Count), proposed and implemented by Bob Jenkins Jr. in [57]. The version used here is the one recommended for cryptography, with `RANDSIZL = 8`. If `flag = 0`, the array `A` is not used and the initial state is obtained from a complicated initialization procedure used in Jenkins' implementation. [1] If `flag = 1`, the array `A` is used and transformed by Jenkins' initialization procedure to obtain the initial state. If `flag = 2`, the array `A` is used as the starting state. Restriction: `flag` $\in \{0, 1, 2\}$.

Clean-up functions

```
void ucrypto_DeleteAES (unif01_Gen * gen);
void ucrypto_DeleteSHA1 (unif01_Gen * gen);
void ucrypto_DeleteISAAC (unif01_Gen * gen);
```

Frees the dynamic memory used by the generators of this module, and allocated by the corresponding `Create` function.

[1] From Richard: In his test program in file `rand.c`, Jenkins outputs the ISAAC random numbers as `randrsl[0], randrsl[1], randrsl[2]`, ... In TestU01, they are outputted in the order `randrsl[255], randrsl[254], randrsl[253]`, ..., because it is simpler.

usoft

This module implements (or, in some cases, provides an interface to) some random number generators used in popular software products. The macros of the form `USE_...` are defined in module `gdef` in directory `mylib`.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```

```
unif01_Gen * usoft_CreateSPlus (long S1, long S2);
```

Generator used in the statistical software environment *S-PLUS* [146, 122]. It is based on Marsaglia's Super-Duper generator of 1973 (see the description of `SupDup73` on page 61 of this guide). The generator never returns 0. Restrictions: $0 <$ `S1` $< 2^{31} - 1$ and $0 <$ `S2` $< 2^{31} - 1$.

```
#ifdef HAVE_RANDOM
unif01_Gen * usoft_CreateUnixRandom (unsigned int s);
#endif
```

Provides an interface to the set of five additive feedback random number generators implemented in the function `random()` in the Unix or Linux `C` library `stdlib` (see the documentation of `random`). It uses a default table of long integers to return successive pseudo-random numbers. The size of the state array determines the period of the random number generator; increasing the state array size increases the period. The parameter `s` determines the order of the recurrence. This generator is not part of the standard ANSI C library. Since it uses global variables, no more than one generator of this type can be in use at any given time. Restrictions: $s \in \{8, 32, 64, 128, 256\}$.

```
#ifdef USE_LONGLONG
unif01_Gen * usoft_CreateJava48 (ulonglong s, int jflag);
#endif
```

Implements the same generator as the method `nextDouble`, in class `java.util.Random` of the Java standard library (`http://java.sun.com/j2se/1.4.2/docs/api/java/util/Random.html`). It is based on a linear recurrence with period length 2^{48}, but each output value is constructed by taking two successive values from the linear recurrence, as follows:

$$\begin{aligned} x_{i+1} &= (25214903917\, x_i + 11) \bmod 2^{48} \\ u_i &= \frac{2^{27}\lfloor x_{2i}/2^{22} \rfloor + \lfloor x_{2i+1}/2^{21} \rfloor}{2^{53}}. \end{aligned}$$

Note that the generator `rand48` in the Unix standard library uses exactly the same recurrence, but produces its output simply via $u_i = x_i/2^{48}$. If `jflag > 0`, `s` is transformed via "`s = s^0x5DEECE66D`" at initialization, as is done in the Java class `Random`; one will then obtain the same numbers as in Java `Random` with the given seed. If `jflag = 0`, `s` is used directly as initial seed. Restriction: $s < 281474976710656$.

`unif01_Gen * usoft_CreateExcel2003 (int x0, int y0, int z0);`

This is the generator implemented by the RAND function in Microsoft Office Excel 2003 (see `https://support.microsoft.com/en-us/kb/828795/`). It uses the Wichmann-Hill generator [171, 172]

$$\begin{aligned} x_i &= 170\, x_{i-1} \bmod 30323 \\ y_i &= 172\, y_{i-1} \bmod 30307 \\ z_i &= 171\, z_{i-1} \bmod 30269 \\ u_i &= \left(\frac{x_i}{30323} + \frac{y_i}{30307} + \frac{z_i}{30269}\right) \bmod 1. \end{aligned}$$

The Wichmann-Hill generators are described in this guide on page 26. The Excel generator is equivalent to the call `ulcg_CreateCombWH3 (30323, 30307, 30269, 170, 172, 171, 0, 0, 0, x0, y0, z0)`. The initial seeds are `x0`, `y0` and `z0`. Restrictions: $0 <$ `x0` < 30323, $0 <$ `y0` < 30307 and $0 <$ `z0` < 30269.

`unif01_Gen * usoft_CreateVisualBasic (unsigned long s);`

The random number generator included in Microsoft VisualBasic. It is an LCG defined as:

$$x_i = (1140671485\, x_{i-1} + 12820163) \bmod 2^{24}; \qquad u_i = x_i / 2^{24}$$

(see `http://support.microsoft.com/support/kb/articles/Q231/8/47.ASP`). The parameter `s` gives the seed x_0. Note that the multiplier 1140671485 in the equation above is equivalent to 16598013, since $1140671485 \bmod 2^{24} = 16598013$.

```
#if defined(USE_GMP) && defined(USE_LONGLONG)
unif01_Gen * usoft_CreateMaple_9 (longlong s);
#endif
```

Implements the generator included in MAPLE 9.5 and earlier versions. It is a linear congruential generator (see the definition on page 23) with $m = 999999999989$, $a = 427419669081$ and $c = 0$. The seed is s. Restriction: $0 < s < 999999999989$. *Note:* MAPLE 10 uses the Mersenne twister MT19937 as its basic generator (see page 40 of this guide).

```
#ifdef USE_LONGLONG
unif01_Gen * usoft_CreateMATLAB (int i, unsigned int j, int bf,
double Z[]);
#endif
```

Implements the basic generator (function `rand`) included in MATLAB [128] to generate uniform random numbers. It is a combination of the subtract-with-borrow generator (2.29) proposed in [116], where z is an array of 32 floating-point numbers in $[0, 1)$ and b is a borrow flag, with the Xorshift generator (2.30) described in [111]:

$$z_i = z_{i+20} - z_{i+5} - b \tag{2.29}$$

$$j \;\hat{}= (j \ll 13); \quad j \;\hat{}= (j \gg 17); \quad j \;\hat{}= (j \ll 5); \tag{2.30}$$

The combination is done by taking the bitwise *exclusive-or* of the bits of the mantissa of z_i with a 52-bit shifted version of j, and this gives the mantissa of the returned number in $[0, 1)$. If `i` < 0, then `j`, `bf` and `Z` are unused, and the generator is initialized using the same procedure as the one described in Cleve Moler's MATLAB *M*-file `randtx.m` (see `http://www.mathworks.com/moler/ncm/randtx.m`) when z is empty. If `i` ≥ 0, then `j`, `bf` and `Z` are used as initial values for the generator state. If the flag `bf` $= 0$, then the initial borrow is set to $b = 0$, while if `bf` $\neq 0$, then it is set to $b = 2^{-53}$. Restrictions: `i` < 32, `bf` $\in \{0, 1\}$, and $0 <$ `Z[i]` < 1.

Another uniform generator included in MATLAB is used to generate normal random variables. It is Marsaglia's additive SuperDuper of 1996, with $c = 1234567$, described on page 61 of this guide (see `umarsa_CreateSupDup96Add`). MATLAB includes also the Mersenne twister generator of Matsumoto and Nishimura [126] (see `ugfsr_CreateMT19937` on page 40 of this guide).

```
#ifdef HAVE_MATHEMATICA
unif01_Gen * usoft_CreateMathematicaReal (int argc, char * argv[],
long s);
#endif
```

This provides an interface to the random number generator for real numbers in $[0, 1)$ implemented by function "`Random[ ]`" of Mathematica 5 and earlier releases (see the web site of Wolfram Research Inc. at `http://www.wolfram.com`). It is a subtract-with-borrow generator (described on page 31 of this guide) of the type proposed by Marsaglia and Zaman in [116], apparently of the form $x_i = (x_{i-8} - x_{i-48} - c) \bmod 2^{31}$, and each returned number in [0,1) uses two successive numbers of the recurrence to get a double of 53 bits. The parameters `argc` and `argv` are the usual arguments of the "`main`" function and the parameter `s` is the initial seed. The random numbers are generated in batches of $2^{18} = 262144$ numbers, for greater speed. Since this generator uses file variables, no more than one generator of this type can be in use at any given time. See the documentation in module `gdef` of MyLib concerning the macro `HAVE_MATHEMATICA`. If the executable program is called, say `tulip`, then the program is launched on a Unix/Linux platform by the command `tulip -linkname 'math -mathlink' -linklaunch`.

```
#ifdef HAVE_MATHEMATICA
unif01_Gen * usoft_CreateMathematicaInteger (int argc, char * argv[],
long s);
#endif
```

Provides an interface to the random number generator for integers in $[0, 2^{30} - 1]$ implemented by function "`Random[Integer, 2^30 - 1]`" of Mathematica 5 and earlier releases. It is based on a cellular automata with rule 30 proposed by Wolfram [175]. See also the documentation of `usoft_CreateMathematicaReal` above.

```
#ifdef USE_LONGLONG
void usoft_DeleteMATLAB (unif01_Gen *gen);
#endif
```

Frees the dynamic memory used by the MATLAB generator and allocated by the corresponding `Create` function above.

```
#ifdef HAVE_MATHEMATICA
void usoft_DeleteMathematicaReal (unif01_Gen *);
void usoft_DeleteMathematicaInteger (unif01_Gen *);
```

Frees the dynamic memory used by the MATHEMATICA generators and allocated by the corresponding `Create` function above.

```
#endif
```

```
#ifdef HAVE_RANDOM
void usoft_DeleteUnixRandom (unif01_Gen *);
```

Frees the dynamic memory used by the `UnixRandom` generator and allocated by the corresponding `Create` function above.

```
#endif
```

```
#if defined(USE_GMP) && defined(USE_LONGLONG)
void usoft_DeleteMaple_9 (unif01_Gen *gen);
```

Frees the dynamic memory used by the MAPLE generator and allocated by the corresponding `Create` function above.

```
#endif
```

```
void usoft_DeleteGen (unif01_Gen *gen);
```

Frees the dynamic memory used by any generator of this module that does not have an explicit `Delete` function. This function should be called to clean up a generator object when it is no longer in use.

uvaria

Implements various special generators proposed in the litterature.

```
#include <testu01/unif01.h>
```

```
unif01_Gen * uvaria_CreateACORN (int k, double S[]);
```

Initializes a generator ACORN (Additive COngruential Random Number) [173] of order k and whose initial state is given by the vector `S[0..(k-1)]`.

```
unif01_Gen * uvaria_CreateCSD (long v, long s);
```

Implements the generator proposed by Sherif and Dear in [151]. The initial state of the generator is given by `v`. The generator uses a MLCG generator whose initial state is given by `s`. Restrictions: $0 \leq$ `v` ≤ 9999 and $0 <$ `s` $< 2^{31} - 1$.

```
unif01_Gen * uvaria_CreateRanrotB (unsigned int seed);
```

This is a lagged-Fibonacci-type random number generator, but with a rotation of bits, called RANROT, and proposed by Fog [42]. The variant programmed here is RANROT of type B. The algorithm is:

$$X_n = ((X_{n-j} \text{ rotl } r_1) + (X_{n-k} \text{ rotl } r_2)) \mod 2^b$$

where rotl denotes a left rotation of the bits, each X_n is an `unsigned int`, and b is the number of bits in an `unsigned int`. The output value is $u_n = X_n/2^b$. The last k values of X are stored in a circular buffer (here of size 17, with $r_1 = 5$ and $r_2 = 3$). Information about RANROT generators can be found at `http://www.agner.org/random/`.

Since Fog's code is copied verbatim here, there are global variables in the implementation. Thus no more than one generator of that type can be in use at any given time.

```
unif01_Gen * uvaria_CreateRey97 (double a1, double a2, double b2, long n0);
```

Generator proposed by W. Rey [143]. It uses the recurrence:

$$z_i = a_1 \sin(b_1(i + n_0)) \mod 1; \tag{2.31}$$

$$u_i = (a_2 + z_i) \sin(b_2 z) \mod 1, \tag{2.32}$$

where $b_1 = (\sqrt{5} - 1)\pi/2$. According to the author, a_1, a_2 and b_2 should be chosen sufficiently large.

```
unif01_Gen * uvaria_CreateTindo (long b, long Delta, int s, int k);
```

Initializes the parameters of the generator proposed by Tindo in [161], with $a_0 = b -$ `Delta` and $a_1 =$ `Delta` $+ 1$. Assumes that $0 <$ `Delta` $< b - 1$ and $b < 2^{15} = 32768$. Restrictions: $1 \leq k \leq 32$, $1 \leq s \leq 32$.

Clean-up functions

`void uvaria_DeleteACORN (unif01_Gen *gen);`

Frees the dynamic memory used by the `ACORN` generator and allocated by the corresponding `Create` function above.

`void uvaria_DeleteRanrotB (unif01_Gen *gen);`

Frees the dynamic memory used by the `RanrotB` generator and allocated by the corresponding `Create` function above.

`void uvaria_DeleteGen (unif01_Gen *gen);`

Frees the dynamic memory used by any generator of this module that does not have an explicit `Delete` function. This function should be called to clean up a generator object when it is no longer in use.

ufile

This module allows the implementation of generators in the form of numbers read directly from an arbitrary file. No more than one generator of each type in this module can be in use at any given time.

```
#include <testu01/unif01.h>
```

```
unif01_Gen * ufile_CreateReadText (char *fname, long nbuf);
```

Reads numbers (assumed to be in text format) from input file `fname`. The numbers must be floating-point numbers in [0, 1), separated by whitespace characters. Numbers in the file can be grouped in any way: there may be blank lines, some lines may contain many numbers, others only one. The file must contain only valid real numbers, nothing else. The numbers are read in batches of `nbuf` at a time and kept in an array (if `nbuf` is very large, a smaller but still large array will be used instead).

```
void ufile_InitReadText (void);
```

Reinitializes the generator obtained from `ufile_CreateReadText` to the beginning of the file.

```
unif01_Gen * ufile_CreateReadBin (char *fname, long nbuf);
```

Reads numbers from input file `fname`. This file is assumed to be in binary format. The numbers are read in batches of `4 nbuf unsigned char`'s at a time, transformed into `nbuf` unsigned 32-bit integers and kept in an array (if `nbuf` is very large, a smaller but still large array will be used instead). This function is used in order to test (random) bit sequences kept in a file.

```
void ufile_InitReadBin (void);
```

Reinitializes the generator obtained from `ufile_CreateReadBin` to the beginning of the file.

Clean-up functions

```
void ufile_DeleteReadText (unif01_Gen *);
```

Closes the file and frees the dynamic memory allocated by `ufile_CreateReadText`.

```
void ufile_DeleteReadBin (unif01_Gen *);
```

Closes the file and frees the dynamic memory allocated by `ufile_CreateReadBin`.

`void ufile_Gen2Bin (unif01_Gen *gen, char *fname, double n, int r, int s);`

Creates the file `fname` containing n random bits using the output of generator `gen`. From each random number returned by `gen`, the r most significant bits will be dropped and the s following bits will be written to the file until n bits have been written. Restriction: $s \in \{8, 16, 24, 32\}$.

Chapter 3

STATISTICAL TESTS

This chapter describes the different statistical tests available in TestU01 and how they can be applied. These tests are organized in different modules, sometimes according to their similarity and sometimes according to the author of the book/article from which they were taken. Each test looks, in its own way, for empirical evidence against the null hypothesis $\mathcal{H}_0$ defined in the introduction. It computes a test statistic Y whose distribution under $\mathcal{H}_0$ is known (or for which a good approximation is available).

Single-level tests.

A *first-order* (or *single-level*) test observes the value of Y, say y, and rejects $\mathcal{H}_0$ if the *p-value* (or *significance level*)

$$p = P[Y \geq y \mid \mathcal{H}_0]$$

is much too close to either 0 or 1. If the distribution of Y is [approximately] continuous, p is [approximately] a $U(0,1)$ random variable under $\mathcal{H}_0$. Sometimes, this p can be viewed as a *measure of uniformity*, in the sense that it will be close to 1 if the generator produces its values with excessive uniformity, and close to 0 in the opposite situation (see, e.g., the module `smultin`).

In the case where Y has a *discrete distribution* under $\mathcal{H}_0$, we distinguish the *right p-value* $p_R = P[Y \geq y \mid \mathcal{H}_0]$ and the *left p-value* $p_L = P[Y \leq y \mid \mathcal{H}_0]$. We then define the p-value as

$$p \;=\; \begin{cases} p_R, & \text{if } p_R < p_L \\ 1 - p_L, & \text{if } p_R \geq p_L \text{ and } p_L < 0.5 \\ 0.5 & \text{otherwise.} \end{cases}$$

Why such a definition? Consider for example a Poisson random variable Y with mean 1 under $\mathcal{H}_0$. If Y takes the value 0, the right p-value is $p_R = P[Y \geq 0 \mid \mathcal{H}_0] = 1$. In the uniform case, this would obviously lead to rejecting $\mathcal{H}_0$ on the basis that the p-value is too close to 1. However, $P[Y = 0 \mid \mathcal{H}_0] = 1/e \approx 0.368$, so it does not really make sense to reject

$\mathcal{H}_0$ in this case. In fact, the left p-value here is $p_L = 0.368$, and the p-value computed with the above definition is $p = 1 - p_L \approx 0.632$. Note that if p_L is very small, with this definition, p becomes close to 1. If the left p-value was defined as $p_L = 1 - p_R = P[Y < y \mid \mathcal{H}_0]$, this would also lead to problems; in the example, one would have $p_L = 0$.

Two-level tests.

In a *second-order* (or *two-level*) test, one generates N "independent" copies of Y, say $Y_1, \ldots, Y_N$, by replicating the first-order test N times. Let F be the theoretical distribution function of Y under $\mathcal{H}_0$. In the case where F is *continuous*, the transformed observations $U_1 = F(Y_1), \ldots, U_N = F(Y_N)$ should behave as i.i.d. uniform random variables. One way of performing the two-level test is to compare the empirical distribution of these U_j's to the uniform distribution, via a *goodness-of-fit* (GOF) test such as those of Kolmogorov-Smirnov, Anderson-Darling, Crámer-von Mises, etc. These GOF test statistics are defined in module `gofs` and their p-values are computed by the functions of module `gofw` (these two modules are in library ProbDist). For example, if d_N^+ is the sample value taken by the Kolmogorov-Smirnov statistic D_N^+ (defined in module `gofs`), the corresponding p-value at the second level is $\delta^+ = P[D_N^+ > d_N^+ | \mathcal{H}_0]$. Under $\mathcal{H}_0$, δ^+ has the $U(0,1)$ distribution.

In TestU01, several of these GOF tests can actually be applied simultaneously, and all their p-values are reported in the results. Those that are too close to 0 or 1 are marked by special indicators in the printouts. The GOF tests that are applied are those that belong to the set `gofw_ActiveTests`. This kind of flexibility is sometimes convenient for comparing the power of these GOF tests to detect the weaknesses of specific classes of generators.

This type of two-level testing procedure has been widely applied for testing RNGs [40, 66, 74, 99, 103]. The arguments supporting it are that (i) it sometimes permits one to apply the test with a larger total sample size to increase its power (for example, if the memory size of the computer limits the sample size of a single-level test), and (ii) it tests the RNG sequence at the local level, not only at the global level (i.e., there could be very bad behavior over short subsequences, which cancels out when averaging over larger subsequences). As an example of this, consider the extreme case of a generator whose output values are $i/2^{31}$, for $i = 1, 2, \ldots, 2^{31} - 1$, in this order. A simple test of uniformity over the entire sequence would give a perfect fit, whereas the same test applied repeatedly over (disjoint) shorter sub-sequences would easily detect the anomaly.

Another way of performing the test at the second level is to simply add the N observations of the first level and reject $\mathcal{H}_0$ if the sum is too large or too small. For the great majority of the tests in this library, the distribution of Y is either chi-square, normal, or Poisson. In these three cases, the sum $\tilde{Y} = Y_1 + \cdots + Y_N$ has the same type of distribution. That is, if Y is chi-square with k degrees of freedom [resp., normal with mean μ and variance σ^2, Poisson with mean λ], $\tilde{Y}$ is chi-square with Nk degrees of freedom [resp., normal with mean $N\mu$ and variance $N^2\sigma^2$, Poisson with mean $N\lambda$]. TestU01 usually reports the results of the test based on $\tilde{Y}$ in these situations, in addition to the second-order GOF tests specified by `gofs_ActiveTests` (for the Poisson case, where the second-order GOF tests are not valid

unless λ is large enough for the Poisson distribution to be well approximated by a normal, only the results of the tests base on $\tilde{Y}$ are reported).

Our empirical investigations indicate that for a fixed total sample size Nn, when testing RNGs, a test with $N = 1$ is often more efficient than the corresponding test with $N > 1$. . This means that for typical RNGs, the type of structure found in one (reasonably long) subsequence is usually found in (practically) all subsequences of the same length. In other words, when a RNG started from a given seed fails spectacularly a certain test, it usually fails that test for *most* admissible seeds, though there are some exceptions. In the case where $N > 1$, the test based on $\tilde{Y}$ is usually more powerful than the second-order GOF tests comparing the empirical distribution of $F(Y_1), \ldots, F(Y_N)$ to the uniform, according to our experience.

Rejecting $\mathcal{H}_0$.

In statistical studies where a limited amount of data is available, people sometimes fix the significance level α in advance to arbitrary values such as 0.05 or 0.01, and reject H_0 if and only if the p-value is below α. However, statisticians often recommend to just report the p-value, because this provides more information than reporting a "reject" or "do not reject" verdict based on a fixed α.

When a p-value is extremely close to 0 or to 1 (for example, if it is less than 10^{-10}), one can obviously conclude that the generator *fails* the test. If the p-value is suspicious but failure is not clear enough, ($p = 0.0005$, for example), then the test can be replicated independently until either failure becomes obvious or suspicion disappears (i.e., one finds that the suspect p-value was obtained only by chance). This approach is possible because there is no limit (other than CPU time) on the amount of data that can be produced by a RNG to increase the sample size and the power of the test.

Common parameters and tools.

Three parameters, called N, n, and r, are common to all the functions that apply a test in the `s` modules. The parameter N gives the number of independent replications of the base test, i.e. the number of distinct subsequences on which it is applied, and n is the sample size for each replication. The parameter r gives the number of bits that are discarded from each generated random number. That is, each real-valued random number is multiplied by 2^r modulo 1, to drop its r most significant bits. These three parameters are not re-explained in each test description. It is implicit that the first r bits of each uniform are always discarded, that the test explained in the function description is always replicated N times, and that a two-level test is applied whenever $N > 1$.

For the tests based on bit strings, another parameter that usually appears is s. It represents the number of bits of each uniform that are effectively used by the test. That is, when s appears, the test drops the r most significant bits and takes the s bits that follow. In this case, it is important to make sure that $r + s$ does not exceed the number of bits of precision provided by the RNG. For example, if the RNG's output is always a multiple of $1/2^{31}$, $r + s$ should not exceed 31.

Reports.

By default, each test prints a report, on standard output, giving the name of the test, the name of the tested generator, the test parameters, the values of the statistics, the significance level of those statistics and the CPU time used by the test. This report may also contain information specific to a given test.

It is possible to print more or less detailed statistical reports by setting one or more of the `lebool` flags defined in module `swrite`. One may wish to see, for example, the value of the test statistic Y for each of the N replications, the values of the counters, the groupings of the classes, their expected and observed numbers for the chi-square test, etc. For some of the tests, printing the counters would generate huge reports and is not practically useful. For other tests (for example those based on a chi-square test), seeing the counters and the classes may be enlightening as to why a given generator fails a test. It is even possible to have no output at all from any of the `s` modules of TestU01 by setting all the `lebool` flags in module `swrite` to `FALSE`.

The test functions automatically print the state of the generator at the beginning of an experiment and at the end of each test. If more than one test are called in a program, the initial state of the generator at the beginning of a test will be the final state of the generator at the end of the preceding test. This permits one to keep track of which segment of the stream of random numbers has been used by each test.

A more flexible way of examining detailed information about what has happened in the tests, to have a closer look at specific details or perhaps for post-processing the results of the tests, is via the `..._Res` structures. These data structures are specific to each type of test and are described explicitly in the detailed version of this guide (see also module `sres`). Each function implementing a test has a parameter `..._Res *` pointing to a structure that keeps the results.

Perhaps in the majority of situations, the automatic printout made by the testing function suffices and there is no need to examine the `..._Res` structure(s) after the test(s). In this case, it suffices to pass a `NULL` pointer for the `..._Res *` parameter. The structure will then be created internally and destroyed automatically after the results are printed.

Scatter plots.

There is a module `scatter` that permits one to plot points produced by a generator in the t-dimensional hypercube $[0, 1)^t$. A rectangular box is defined in this hypercube, and the points lying in this box are projected on a selected two-dimensional subspace and placed on a 2-dimensional scatter plot. The plot is put in a file ready to be processed by LaTeX or *Gnuplot*.

An example: The birthday spacings tests applied to an LCG.

Figure 3.1 shows how to apply a test to a generator. The call to `ulcg_CreateLCG` creates and initializes the generator `gen` to the LCG with modulus $m = 2147483647$, multiplier $a = 397204094$, additive constant $c = 0$, and initial state $x_0 = 12345$. This LCG is used in the SAS statistical software [149]. Then the birthday spacings test is applied twice to this generator, with $N = 1$, $r = 0$, in $t = 2$ dimensions. The sample sizes are $n = 10^3$ and $n = 10^4$, and the number d of divisions along each coordinate is chosen so that the expected number of collisions $\lambda = n^3/(4d^t)$ is 2.5 in the first case and 0.25 in the second case (the values of d are 10^4 and 10^6, respectively). Under $\mathcal{H}_0$, the number of collisions is approximately a Poisson random variable with mean λ.

```
#include <testu01/unif01.h>
#include <testu01/ulcg.h>
#include <testu01/smarsa.h>
#include <stddef.h>

int main (void)
{
   unif01_Gen *gen;
   gen = ulcg_CreateLCG (2147483647, 397204094, 0, 12345);
   smarsa_BirthdaySpacings (gen, NULL, 1, 1000, 0, 10000, 2, 1);
   smarsa_BirthdaySpacings (gen, NULL, 1, 10000, 0, 1000000, 2, 1);
   ulcg_DeleteGen (gen);
   return 0;
}
```

Figure 3.1: Applying two birthday spacings tests to a LCG.

The results are in Figure 3.2. These results are printed to the standard output, which may be redirected to a file if desired. At sample size $n = 10^3$, there are 6 collisions and the p-value is 0.04, which is not extreme enough to reject $\mathcal{H}_0$. At sample size $n = 10^4$, there are 44 collisions and the p-value is close to 10^{-81} (i.e., if Y is Poisson with mean 0.25, $P[Y \geq 44] < 10^{-81}$). The generator fails miserably in this case, with a sample size as small as ten thousands. This test took approximately 0.02 second to run.

```
*********************************************************
HOST =

ulcg_CreateLCG:   m = 2147483647,   a = 397204094,   c = 0,   s = 12345


smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 1000,  r =  0,    d = 10000,    t = 2,    p = 1


      Number of cells = d^t =          100000000
      Lambda = Poisson mean =       2.5000


-------------------------------------------------
Total expected number = N*Lambda      :       2.50
Total observed number                 :       6
p-value of test                       :    0.04


-----------------------------------------------
CPU time used                    :  00:00:00.00

Generator state:
 s = 1858647048



*********************************************************
HOST =

ulcg_CreateLCG:   m = 2147483647,   a = 397204094,   c = 0,   s = 12345


smarsa_BirthdaySpacings test:
-----------------------------------------------
   N =  1,  n = 10000,  r =  0,    d = 1000000,    t = 2,    p = 1


      Number of cells = d^t =      1000000000000
      Lambda = Poisson mean =       0.2500


-------------------------------------------------
Total expected number = N*Lambda      :       0.25
Total observed number                 :      44
p-value of test                       : 9.5e-82    *****


-----------------------------------------------
CPU time used                    :  00:00:00.00

Generator state:
 s = 731506484
```

Figure 3.2: Results of the two birthday spacings tests.

swrite

This module contains some functions used in writing statistical test results, inside the implementation of other modules.

Each testing function in the `s` modules normally writes a report (on standard output, by default) that contains the description of the generator being tested, the name of the experiment, the name of the test and its parameters, the values and significance levels of statistics, and the CPU time used by each test. This report may contain additional information specific to a given test. More detailed results in the printouts can be obtained by setting the `lebool` variables below to `TRUE` before calling the test. If all `lebool` flags below are set to `FALSE`, then no output will be printed.

```
#include <testu01/gdef.h>
#include <testu01/chrono.h>
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

Environment variables

```
extern lebool swrite_Basic;           /* Prints basic results           */
extern lebool swrite_Parameters;      /* Prints details on parameters   */
extern lebool swrite_Collectors;      /* Prints statistical collectors  */
extern lebool swrite_Classes;         /* Prints classes for ChiSquare   */
extern lebool swrite_Counters;        /* Prints counters                */
```

These environment variables (which are *boolean switches*) are used to control the level of detail in the output printed by the tests. By default, all are set to `FALSE`, except for `swrite_Basic` which is set to `TRUE`. When `swrite_Basic` is `TRUE`, the test results are printed with a standard level of detail. If it is `FALSE`, then nothing from the `u` or `s` modules is printed.

The other switches permit one to obtain more detailed information than usual, in a selective way. The details are printed when the corresponding switch is set to `TRUE`. This could be useful, for example, to examine more closely the kind of defect exhibited by a random number generator that fails a test.

The switch `swrite_Parameters` controls the printing of internal parameters that are specific to each test. The switch `swrite_Collectors` controls the printing of the statistical collectors holding the N values of the main statistics Y of the test. The switch `swrite_Classes` controls the printing of details concerning the regroupings into classes (or categories), with the expected numbers of observations in each class, in the situations where such regrouping is performed in order to apply a chi-square test (see function `gofs_MergeClasses` in module `gofs` of library ProbDist). The switch `swrite_Counters` controls the printing of the different counters that hold the numbers of observations.

```
extern lebool swrite_Host;
```

If this variable is `TRUE`, the name of the machine on which the tests are run is printed before each test; otherwise it is not printed.

sres

This module defines common structures used to keep the results of tests in the `s` modules. They are described in the detailed version of this guide.

The first argument of each testing function is the random number generator to be tested. It must be created by calling the appropriate function in one of the module `u`, and deleted when no longer needed. The second argument of each testing function is a structure `s..._Res` that can keep the test results (intermediate and final). This is useful if one wishes to do something else with the results or the information generated during a test. If one does not want to post-process or use the results after a test, it suffices to set the `..._Res` argument to the `NULL` pointer. Then, the structure is created and deleted automatically inside the testing function.

smultin

Testing the uniformity and independence of a RNG amounts to testing that t-dimensional vectors $(u_i, \ldots, u_{i+t-1})$ of successive output values of the RNG behave like random points uniformly distributed over the unit hypercube $[0,1]^t$, for all t. A natural approach for testing this is to generate such vectors and measure (in some way) the uniformity of their distribution in the unit hypercube.

A class of tests based on the multinomial distribution.

One simple way of measuring this uniformity is as follows. For some integer d, partition the interval $[0,1)$ into d equal segments. This determines a partition of $[0,1)^t$ into $k = d^t$ small hypercubes (or cubic *cells*) of equal sizes. Then, generate n random points in the unit hypercube, using nt output values from the generator, and let X_j be the number of points falling into cell j, for $0 \le j \le k-1$. Under $\mathcal{H}_0$, the vector $(X_0, \ldots, X_{k-1})$ has the *multinomial distribution* with parameters $(n, 1/k, \ldots, 1/k)$. The next step is to measure how well the observed vector $(X_0, \ldots, X_{k-1})$ "agrees" with this multinomial distribution. For example, if $n \gg k$, the X_j's should not be too far from their expected values $E[X_j] = \lambda = n/k$. The most popular test statistic in this context is Pearson's chi-square [66, 70, 142]:

$$X^2 = \sum_{j=0}^{k-1} \frac{(X_j - \lambda)^2}{\lambda} = -n + \frac{1}{\lambda} \sum_{j=0}^{k-1} X_j^2, \tag{3.1}$$

Its distribution under $\mathcal{H}_0$ is approximately chi-square with $k-1$ degrees of freedom, if λ is large enough. Other test statistics can be used as well, and some of them turn out to be better than the chi-square for detecting deficiencies in typical RNGs.

The present module implements several such tests for the multinomial distribution, and for a variant of it where the points are formed by overlapping vectors. These tests are described and studied by L'Ecuyer, Simard, and Wegenkittl [96]. The test statistic has the general form

$$Y = \sum_{j=0}^{k-1} f_{n,k}(X_j) \tag{3.2}$$

where $f_{n,k}$ is a real-valued function which may depend on n and k. A subclass is the *power divergence* statistic

$$D_\delta = \sum_{j=0}^{k-1} \frac{2}{\delta(1+\delta)} X_j \left[\left(\frac{X_j}{\lambda} \right)^\delta - 1 \right], \tag{3.3}$$

studied in [142], where $\delta > -1$ is a real-valued parameter and $\delta = 0$ means the limit as $\delta \to 0$. One has $D_1 = X^2$ as a special case.

Other choices of $f_{n,k}$ are given in Table 3.1. In each case, Y is a measure of clustering: It decreases when the points are more evenly distributed between the cells. The loglikelihood statistic G^2 is also a special case of D_δ for $\delta \to 0$ [142], and it is related to H via the relation

$H = \log_2(k) - G^2/(2n \ln 2)$. The statistic N_b counts the number of cells that contain exactly b points (for $b \geq 0$), W_b is the number of cells that contain at least b points (for $b \geq 1$), and C is the number of collisions (i.e., the number of times a point falls in a cell that already has one or more points in it). These statistics are related by $N_0 = k - W_1 = k - n + C$, $W_b = N_b + \cdots + N_n$, and $C = W_2 + \cdots + W_n$.

Table 3.1: Some choices of $f_{n,k}$ and the corresponding statistics Y.

Y	$f_{n,k}(x)$	name
D_δ	$2x[(x/\lambda)^\delta - 1]/(\delta(1+\delta))$	power divergence
X^2	$(x-\lambda)^2/\lambda$	Pearson
G^2	$2x \ln(x/\lambda)$	loglikelihood
$-H$	$(x/n)\log_2(x/n)$	negative entropy
N_b	$I[x = b]$	number of cells with exactly b points
W_b	$I[x \geq b]$	number of cells with at least b points
N_0	$I[x = 0]$	number of empty cells
C	$(x-1)\, I[x > 1]$	number of collisions

How to generate the cell numbers.

The standard way of generating the cell numbers is as described earlier, by generating one fresh uniform for each coordinate of each point. That is, nt random numbers $u_0, \ldots, u_{nt-1}$ are generated and the n points are $(u_{ti}, \ldots, u_{ti+t-1})$, for $i = 0, \ldots, n-1$. This is the *non-overlapping serial* approach used in the classical *serial test* [66]. It is the method used by default by the function `smultin_Multinomial`. Here, the points are independent and $(X_0, \ldots, X_{k-1})$ has the *multinomial distribution* as mentioned earlier.

Another way of producing the cell numbers is as follows: Generate nt random numbers $u_0, \ldots, u_{nt-1}$. For $i = 0, \ldots, n-1$, let $v_{ti} = (u_{ti}, \ldots, u_{ti+t-1})$, and find which of the $t!$ *permutations* of t objects would reorder the coordinates of v_{ti} in increasing order. Let $k = t!$ number the $t!$ possible permutations from 0 to $k-1$, and let X_j be the number of vectors v_{ti} that are reordered by permutation j, for $j = 0, \ldots, k-1$. Here, since the permutations are independent, $(X_0, \ldots, X_{k-1})$ has the multinomial distribution with parameters $(n, 1/k, \ldots, 1/k)$.

Yet another method is to examine which coordinate in each v_{ti} has the *largest value*, and let X_j be the number of vectors v_{ti} whose largest coordinate is the jth. In this case, $k = t$ and $(X_0, \ldots, X_{k-1})$ is again multinomially distributed with parameters $(n, 1/k, \ldots, 1/k)$.

To change the method for generating the cells numbers in `smultin_Multinomial`, it suffices to set the environment variable `smultin_GenerCell` to the appropriate function (`smultin_GenerCellSerial`, `smultin_GenerCellPermut`, etc.) or to a user-defined function having the same types of parameters and that generates cell numbers uniformly and independently.

It is also possible and often advantageous to generate cell numbers that are *dependent*. This is what happens in the *overlapping serial* approach, where only n uniforms $u_0, \ldots, u_{n-1}$ are generated; they are placed in a circle and each one starts a new vector. The n points are thus defined as $v_0 = (u_0, \ldots, u_{t-1})$, $v_1 = (u_1, \ldots, u_t)$, $\ldots$, $v_{n-t+1} = (u_{n-t+1}, \ldots, u_n)$, $v_{n-t+2} = (u_{n-t+2}, \ldots, u_n, u_0)$, $\ldots$, $v_{n-1} = (u_{n-1}, u_n, u_0, \ldots, u_{t-3})$, $v_n = (u_n, u_0, \ldots, u_{t-2})$. These points are dependent, because their coordinates overlap, so $(X_0, \ldots, X_{k-1})$ is no longer multinomially distributed in this case. The function `smultin_MultinomialOver` uses this approach.

Another set of methods generate the cell numbers from a long string of "random" bits, which are produced s bits at a time by the RNG. Let $k = 2^L$ be the number of cells, for some integer L. Each cell number is generated by taking L successive bits from the string, either with or without overlap. The *non-overlapping* version requires nL bits ($\lceil nL/s \rceil$ calls to the generator) whereas the *overlapping* one requires n bits ($\lceil n/s \rceil$ calls). The overlapping version operates similarly as the overlapping serial approach described earlier, except that the uniforms u_j are replaced by bits. The functions `smultin_MultinomialBits` and `smultin_MultinomialBitsOver` implement these tests. (See also the tests `sentrop_EntropyDisc` and `sentrop_EntropyDiscOver`.)

Distribution of Y for the non-overlapping case.

We now examine the distribution of the different statistics Y in Table 3.1, under $\mathcal{H}_0$, assuming that $(X_0, \ldots, X_{k-1})$ has the multinomial distribution. Exact expressions for $E[Y]$ and $\mathrm{Var}[Y]$ for this situation are given in Eqs. (2.1) and (2.2) of [96]. In fact, one has $E[Y] = k\mu$ where $\mu = E[f_{n,k}(X_j)]$. These expressions for the mean and variance are cheap to compute when $\lambda = n/k$ is small but become very expensive to compute when n and λ are large. For the case where $\lambda \gg 1$, approximations with $o(1/n)$ error are provided in [142], page 65.

The following propositions, taken from [96], provide approximations for the distribution of Y under various conditions. Define $\sigma_N^2 = \mathrm{Var}[Y]$, $\sigma_C^2 = \mathrm{Var}[Y]/(2(k-1))$,

$$Y^{(N)} = \frac{Y - k\mu}{\sigma_N},$$

and

$$Y^{(C)} = \frac{Y - k\mu + (k-1)\sigma_C}{\sigma_C}.$$

Observe that $Y^{(N)}$ has mean 0 and variance 1 (the same as a standard normal) and that $Y^{(C)}$ has mean $k-1$ and variance $2(k-1)$ (the same as the chi-square random variable with $k-1$ degrees of freedom). Let $\Rightarrow$ denote convergence in distribution, $N(0,1)$ the standard normal distribution, and $\chi^2(k-1)$ the chi-square distribution with $k-1$ degrees of freedom.

Proposition 1 *For $\delta > -1$, under $\mathcal{H}_0$.*

(i) [Dense case] *If k is fixed and $n \to \infty$, $D_\delta^{(C)} \Rightarrow \chi^2(k-1)$.*

(ii) [Sparse case] *If* $k \to \infty$, $n \to \infty$, *and* $n/k \to \lambda_0$ *where* $0 < \lambda_0 < \infty$, *then* $D_\delta^{(N)} \Rightarrow N(0,1)$.

The counting variables N_b, W_b, and C do not obey the preceding proposition. For example if k is fixed and $n \to \infty$, eventually N_b becomes 0 and W_b becomes equal to k. If both k and n are large, then each X_j is approximately Poisson with mean λ, so $P[X_j = b] \approx e^{-\lambda}\lambda^b/b!$ for $b \geq 0$. If k is large and $P[X_j = b]$ is small, N_b is thus approximately Poisson with mean

$$E[N_b] = \frac{n^b e^{-\lambda}}{k^{b-1} b!}. \tag{3.4}$$

for $b \geq 0$. The following proposition is also taken from [96].

Proposition 2 *Under* $\mathcal{H}_0$, *suppose* $k \to \infty$ *and* $n \to \infty$, *and let* λ_∞, γ_0, *and* λ_0 *denote positive constants.*

(i) [Very sparse case] *If* $b \geq 2$ *and* $n^b/(k^{b-1}b!) \to \lambda_\infty$, *then* $W_b \Rightarrow N_b \Rightarrow \text{Poisson}(\lambda_\infty)$. *For* $b = 2$, *one also has* $C \Rightarrow N_2$.

(ii) *For* $b = 0$, *if* $n/k - \ln(k) \to \gamma_0$, *then* $N_0 \Rightarrow \text{Poisson}(e^{-\gamma_0})$.

(iii) [Sparse case] *If* $k \to \infty$ *and* $n/k \to \lambda_0 > 0$, *for* $Y = N_b$, W_b, *or* C, *one has* $Y^{(N)} \Rightarrow N(0,1)$.

The exact distributions of C and N_0 under $\mathcal{H}_0$ (one has $P(C = c) = P(N_0 = k - n + c)$), for the multinomial setup, can be found in Knuth's book (see [66], page 71), where an algorithm is also given to compute all the non-negligible exact probabilities in time $O(n \log n)$. Computing the exact distribution is very slow when n is large. The probability of having exactly c collisions is given by

$$P[C = c] = \frac{k(k-1)\dots(k-n+c+1)}{k^n} \left\{ \begin{matrix} n \\ n-c \end{matrix} \right\} \tag{3.5}$$

where the $\left\{ \begin{matrix} n \\ k \end{matrix} \right\}$ are the Stirling numbers of the second kind [65]. The expected number of collisions is

$$\mu_c \stackrel{\text{def}}{=} E[C] = k\left[\tfrac{n}{k} - 1 + \left(1 - \tfrac{1}{k}\right)^n\right] \approx \frac{n^2}{2k}.$$

The current implementation of the test based on C uses the following: If $n \leq 10^5$, the exact distribution is used, else if $n/k \leq 1$, C is approximated by the Poisson distribution with mean μ_c, else it is approximated by a normal distribution with the exact mean and standard deviation. For two-level tests, the Poisson approximation for the total number of collisions is used.

Distribution of Y for the overlapping serial approach.

Let $X_{t,j}^{(o)}$ be the number of overlapping vectors v_i, $i = 0, \ldots, n-1$, falling into cell j. The distributions of the statistics Y are more difficult to analyze in this case, because of the more complicated dependence relationship between the $X_{t,j}^{(o)}$.

For $\delta > -1$, let

$$D_{\delta,(t)} = \sum_{j=0}^{k-1} \frac{2}{\delta(1+\delta)} X_{t,j}^{(o)} \left[(X_{t,j}^{(o)}/\lambda)^{\delta} - 1 \right], \tag{3.6}$$

the power divergence statistic for the t-dimensional overlapping vectors, and define

$$\tilde{D}_{\delta,(t)} = D_{\delta,(t)} - D_{\delta,(t-1)}. \tag{3.7}$$

The following is taken from [96] (The case $\delta = 1$ was proved long ago by Good [46]):

Proposition 3 [Dense case] *Under $\mathcal{H}_0$, if k is fixed and $n \to \infty$, $\tilde{D}_{\delta,(t)} \Rightarrow \chi^2(d^t - d^{t-1})$.*

For the sparse case, where $k, n \to \infty$ and $n/k \to \lambda_0$ where $0 < \lambda_0 < \infty$, simulation experiments support the conjecture that

$$\frac{\tilde{D}_{1,(t)}^2 - (k - k')}{\sqrt{2(k-k')}} \Rightarrow N(0,1) \tag{3.8}$$

but this has not been proved.

Marsaglia and Zaman [118] speculate that N_0 is approximately normal with mean $ke^{-\lambda}$ and variance $ke^{-\lambda}(1 - 3e^{-\lambda})$. This approximation is reasonably accurate for $2 \leq \lambda \leq 5$ (roughly), but no longer makes sense when λ is too large or close to zero. Marsaglia [103] calls the test based on N_0 with $t = 2$ the *overlapping pairs sparse occupancy* (OPSO) test (see `smarsa_CollisionOver` and `smarsa_Opso`).

Proposition 2 (i) and (ii) probably holds in the overlapping case as well, but we do not have a formal proof. Simulation experiments indicate that the Poisson approximation for C is very accurate for (say) $\lambda < 1/32$, and already quite good for $\lambda \leq 1$, when n is large.

Calling the tests.

Four functions are available here to launch a test: `smultin_Multinomial` for the case where the cell numbers are independent of each other (so we have the multinomial distribution) and `smultin_MultinomialOver` for the case where the cell numbers are generated by overlapping t-tuples. In each of these two procedures, the type of approximation that is used for the distributions of the power divergence statistics is determined by the boolean parameter `Sparse`: The normal approximation is used when it is true, and the chi-square approximation otherwise. The two analog functions for bit tests are `smultin_MultinomialBits` and `smultin_MultinomialBitsOver`.

```
#include <testu01/gdef.h>
#include <testu01/fmass.h>
#include <testu01/statcoll.h>
#include <testu01/gofw.h>
#include <testu01/unif01.h>


#define smultin_MAX_DELTA  8
```

Maximal number of distinct values of δ for which the statistic D_δ can be computed in a single test (with $\delta = -1$ representing the family of collision-type tests).

```
#define smultin_MAXB  10
```

Maximal value of b for which the statistic W_b can be computed (see Table 3.1).

Environment variables

The parameters in `smultin_Envir` are environment variables that should be fixed once for all and will not normally be changed.

```
typedef struct {

double Maxk;
```

Maximal value of $k = d^t$, the number of cells. The default value is either 2^{63} if 64-bit integers are available, or else 2^{53}, the largest number of cells that can be enumerated by `double's` without loss of precision.

```
} smultin_Envir;


extern smultin_Envir smultin_env;
```

This is the environment variable used to keep the values of the fields described above in `smultin_Envir`.

Functions to generate the cell numbers

```
#ifdef USE_LONGLONG
typedef ulonglong smultin_CellType;
#else
typedef double smultin_CellType;
#endif
```

Type used to enumerate cell numbers, either 64-bit integers if they are available, or else `double's` with a 53-bit mantissa.

```
typedef smultin_CellType (*smultin_GenerCellType) (unif01_Gen *, int, int,
long);
```

Type of function used to generate a cell number. It returns the cell number.

```
smultin_CellType smultin_GenerCellSerial (unif01_Gen *gen, int r, int t,
long d);
```

Generates and returns a cell number in $\{0, \ldots, k-1\}$ for the multinomial test, with $k = d^t$. The function generates t integers $y_0, \ldots, y_{t-1}$ in $\{0, \ldots, d-1\}$, using the most significant bits of t successive uniforms (after throwing away their leading r bits). The cell number $c = y_0 d^{t-1} + \cdots + y_{t-2} d + y_{t-1}$ is returned.

```
smultin_CellType smultin_GenerCellSerial2 (unif01_Gen *gen, int r, int t,
long d);
```

Similar to `smultin_GenerCellSerial`, except that the cell number is $c = y_{t-1} d^{t-1} + \cdots + y_1 d + y_0$. This is equivalent to using `smultin_GenerCellSerial`, because it only changes the numbering of the cells and the power divergence test statistics do not depend on the numbering.

```
smultin_CellType smultin_GenerCellPermut (unif01_Gen *gen, int r, int t,
long junk);
```

Similar to `smultin_GenerCellSerial`, except that there are $t!$ cells, corresponding to the $t!$ permutations of t real numbers, The function generates t uniforms and returns the number of the permutation that corresponds to their ordering.

```
smultin_CellType smultin_GenerCellMax (unif01_Gen *gen, int r, int t,
long junk);
```

Similar to `smultin_GenerCellSerial`, except that $k = t$ and the cell number is the number of the largest coordinate in v_{ti}.

Test parameters having default values

The parameters in the following global structure `smultin_Param` often remain the same in a given experiment. They can practically be viewed as environment variables. To change their default values, one should create a `smultin_Param` structure and pass it as a pointer to the tests. When a `NULL` pointer is passed as argument to a test, the default values are used instead.

```
typedef struct {

int NbDelta;
double ValDelta [smultin_MAX_DELTA];
```

The number of values of δ, and the list of values of δ, for which the multinomial tests are

applied (simultaneously) when calling a test. If `NbDelta` $= m$, then the tests are applied for the m values $\delta_0, \ldots, \delta_{m-1}$ which must be given in array `ValDelta[0..m-1]`. The value $\delta = -1$ corresponds to the collision test and the other values of δ correspond to power divergence tests. The default values are $m = 2$, $\delta_0 = -1$, $\delta_1 = 1$.

`smultin_GenerCellType GenerCell;`

This function is used to generate the cell numbers in the tests. It is one of the function `smultin_GenerCell` described above. The default value is `smultin_GenerCellSerial`.

`int bmax;`

In the non-overlapping case for $\delta = -1$, in addition to applying the collision test, the statistic W_b and its p-value are computed for $b = 0, \ldots,$`bmax`. For $b = 0$, the test is actually based on N_0, the number of empty cells. If the value is negative, none of these statistics is computed. The default value is `bmax = -1`.

`} smultin_Param;`

```
smultin_Param * smultin_CreateParam (int NbDelta, double ValDelta[],
smultin_GenerCellType GenerCell,
int bmax);
```

Function creating and returning a structure that will hold the parameters described above. The parameters have the same meaning as in the structure `smultin_Param`.

```
void smultin_DeleteParam (smultin_Param *par);
```

Procedure freeing the memory allocated by `smultin_CreateParam`.

The tests

```
void smultin_Multinomial (unif01_Gen *gen, smultin_Param *par,
smultin_Res *res, long N, long n, int r, long d, int t, lebool Sparse);
```

This function applies the power divergence test, based on statistic D_δ, for each value of δ specified in the array `par->ValDelta[0..NbDelta-1]` (where $\delta = -1$ corresponds to the collision test, based on C). The theoretical distribution of D_δ (with a two-moment correction) is approximated by a normal if `Sparse = TRUE`, and by a chi-square with $k-1$ degrees of freedom if `Sparse = FALSE`. This function also applies tests based on W_b for $b = 0, 2, \ldots,$`par->bmax`. All these tests are applied simultaneously, using the same cell countings. The cell numbers are generated by the function in variable `par->GenerCell`. By default, it is `smultin_GenerCellSerial`. Both `par` or `res` can be set to the `NULL` pointer, in which case these structures are created and deleted internally. It is recommended to have $n/k > 8$ if `Sparse = FALSE`, except for $\delta = 1$. If `Sparse = TRUE`, n and k should both be very large. Restriction: $k = d^t \leq$ `smultin_Maxk`.

```
void smultin_MultinomialOver (unif01_Gen *gen, smultin_Param *par,
smultin_Res *res, long N, long n, int r, long d, int t, lebool Sparse);
```

Similar to `smultin_Multinomial`, but where the n cell numbers are generated using the *overlapping serial* approach, as described earlier in the paragraph "How to generate the cell numbers" and used in `smarsa_SerialOver`.

```
void smultin_MultinomialBits (unif01_Gen *gen, smultin_Param *par,
smultin_Res *res, long N, long n, int r, int s, int L, lebool Sparse);
```

Similar to `smultin_Multinomial`, except that the cells are generated from a string of bits obtained by taking `s` bits from each output value. There are $k = 2^L$ cells and each cell number is determined by taking L successive bits from the string. In the case where $L = ts$ for some integer t, this test is equivalent to `smultin_Multinomial` with $d = 2^s$ and $t = L/s$. The present function is to cover the other cases (e.g., if $L < s$). Restrictions: $L \bmod s = 0$ when $L > s$, and $s \bmod L = 0$ when $s > L$.

```
void smultin_MultinomialBitsOver (unif01_Gen *gen, smultin_Param *par,
smultin_Res *res, long N, long n, int r, int s, int L, lebool Sparse);
```

Similar to `smultin_MultinomialBits`, except that the n cell numbers are generated using the *overlapping* approach at the bit level. The n bits are placed in a circle and each block of L successive bits determines a cell number. L and s do not have to divide each other.

sentrop

This module implements tests based on discrete and continuous empirical entropies, defined and studied by Dudewicz and van der Meulen [24, 25] and L'Ecuyer, Compagner, and Cordeau [86]. The tests of [24, 25] are actually superseded by the tests with $\delta = 0$ in module `smultin`, where a much better approximation is used for the distribution of the test statistic. The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/statcoll.h>
#include <testu01/gofw.h>
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void sentrop_EntropyDisc (unif01_Gen *gen, sentrop_Res *res,
long N, long n, int r, int s, int L);
```

Applies the entropy-based test proposed in [86]. It builds n blocks of L bits by taking s-bit strings (the s most significant bits after dropping the first r) from each of NL/s successive output values from the generator, and concatenating these strings. There are $k = 2^L$ possible L-bit blocks, which can be numbered from 0 to $k-1$. Let X_i be the observed frequency of occurrence of the ith possibility, for $i = 0, \ldots, k-1$. The test is based on the *empirical entropy*

$$T = -\sum_{i=0}^{k-1} X_i \log_2 X_i,$$

whose distribution is approximated by the normal if $n/2^L \leq 8$ (the sparse case) and by a chi-square distribution if $n/2^L > 8$ (the dense case). This test is equivalent to `smultin_MultinomialBits` with the power divergence test statistic, using $\delta = 0$ only. Restrictions: Either L divides s or s divides L.

```
void sentrop_EntropyDiscOver (unif01_Gen *gen, sentrop_Res *res,
long N, long n, int r, int s, int L);
```

Applies an entropy-based test described in [86], similar to `sentrop_EntropyDisc`, but with overlap of the blocks. It constructs a sequence of n bits, by taking s bits from each of n/s output values, puts these n bits on a circle, and examines all n blocks of L successive bits on this circle. The test computes the empirical entropy, defined by

$$T = -\sum_{i=0}^{k-1} X_i \log_2 X_i,$$

where the X_i are the observed frequencies of the L-bit strings. This test is equivalent to `smultin_MultinomialBitsOver` with the power divergence test statistic, using $\delta = 0$ only.

For $N > 1$, the function also tests the empirical correlation between pairs of successive values of T, as well as the average of these values. This average is compared with the exact expectation in the cases where it is known. Restrictions: $r \leq 31$, $s \leq 31$, $n \leq 31$, $L \leq n/2$, $n \bmod s = 0$, and $N \gg n$.

```
void sentrop_EntropyDiscOver2 (unif01_Gen *gen, sentrop_Res *res,
long N, long n, int r, int s, int L);
```

A version of `sentrop_EntropyDiscOver` that accepts $n > 31$. For $n > 30$, it tests only the correlation between successive values of T. Restrictions: $L \leq 15$, $r \leq 31$, $s \leq 31$, $L + s \leq 31$, $n \bmod s = 0$, $\lceil L/s \rceil s \leq 31$, and $N \gg n$.

```
void sentrop_EntropyDM (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, long m);
```

Applies the entropy test described by Dudewicz and van der Meulen [24, 25]. It uses n successive output values from the generator and computes the empirical entropy $H_{m,n}$ defined in [24, 25], whose theoretical distribution is approximated by a normal distribution.

```
void sentrop_EntropyDMCirc (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, long m);
```

Similar to `sentrop_EntropyDM`, except that a circular definition of the Y_i's (defined in [24, 25]) is used. This function defines $Y_i = Y_{n+i} - 1$ for $i < 1$, and $Y_i = Y_{i-n} + 1$ for $i > n$.

snpair

This module implements tests based on the distances between the closest points in a sample of n uniformly distributed points in the unit torus in t dimensions. These tests are studied by L'Ecuyer, Cordeau, and Simard [87].

Distances between points are measured using the L_p norm $\|\cdot\|_p^o$ in the unit torus $[0,1)^t$, as defined in [87]. The unit torus is obtained by identifying (pairwise) the opposite sides of the unit hypercube, so that points that are face to face on opposite sides are "close" to each other. Each point is generated using t calls to the generator, for a total of nt calls to get the n points $X_1, \ldots, X_n$.

Let $D_{n,i,j} = \|X_j - X_i\|_p$ be the distance between X_i and X_j. Put $\lambda(n) = n(n-1)V_t(1)/2$, where $V_t(r)$ is the volume of the ball $\{x \in \mathbb{R}^t \mid \|x\|_p \le r\}$. For each $\tau \ge 0$, let $Y_n(\tau)$ be the number of distinct pairs of points (X_i, X_j), with $i < j$, such that $D_{n,i,j} \le (\tau/\lambda(n))^{1/t}$. The following is proved in [87]:

Proposition 4 *Under $\mathcal{H}_0$, for any fixed $\tau_1 > 0$ and $n \to \infty$, the truncated process $\{Y_n(\tau), 0 \le \tau \le \tau_1\}$ converges weakly to a Poisson process with unit rate. Moreover, for $\tau \le \lambda(n)/2^t$, one has $E[Y_n(\tau)] = \tau$ and $\mathrm{Var}[Y_n(\tau)] = \tau - 2\tau^2/(n(n-1))$.*

Let $T_{n,i} = \inf\{\tau \ge 0 \mid Y_n(\tau) \ge i\}$, $i = 1, 2, 3, \ldots$, be the jump times of Y_n, with $T_{n,0} = 0$, and let $W^*_{n,i} = 1 - \exp[-(T_{n,i} - T_{n,i-1})]$ be the transformed spacings between these jump times. Proposition 4 implies that for any fixed integer $m > 0$, for large enough n, the random variables $W^*_{n,1}, \ldots, W^*_{n,m}$ are approximately i.i.d. $U(0,1)$. The function `snpair_ClosePairs` applies the *m-nearest-pairs* (m-NP) test, which simply compares the empirical distribution of these random variables with the uniform distribution, using the Anderson-Darling test statistic (see module `gofs`). Why Anderson-Darling? Because typically, when a generator has too much structure, the jump times of Y_n tend to cluster, so there tends to be several $W^*_{n,i}$'s near zero, and the Anderson-Darling test is particularly sensitive to detect that type of behavior.

For a two-level test, when $N > 1$, the standard approach is to apply a goodness-of-fit test to the N values of the Anderson-Darling statistic. A second approach is to pool the N batches of m observations $W^*_{n,i}$ in a single sample of size Nm and apply the Anderson-Darling test to it. These two possibilities are referred to by the acronyms m-NP and m-NP1 respectively.

A third one is to superpose the N copies of the process Y_n, to obtain a process Y defined as the sum of the n copies of Y_n. Fix a constant $\tau_1 > 0$ and let J be the number of jumps of Y in the interval $[0, \tau_1]$. Set $T_0 = 0$ and let $T_1, \ldots, T_J$ be the sorted times of these jumps. Under $\mathcal{H}_0$, J is approximately Poisson with mean $N\tau_1$, and conditionally on J, the jump times T_j are distributed as J independent uniforms over $[0, \tau_1]$ sorted in increasing order. We can test this uniformity with an AD test on the J observations and this is called the m-NP2 test. It is also worthwhile to compare the realization of J with the Poisson distribution and this is the NJumps test.

In yet another variant of the uniformity test conditional on J, we apply a "spacings" transformation to the uniforms before applying the AD test (this is called the m-NP2S test). This latter test is very powerful but it also turns out to be very sensitive to the number of bits of "precision" in the output of the generator. For example, in dimension $t = 2$, for $N = 20$, $n = 10^6$ and $m = 20$, all generators returning less than 32 bits of precision will fail this test.

We can finally apply a "spacings" transformation to the N closest distances $W^*_{n,1}$ and to the mN nearest pairs $W^*_{n,i}$ before applying the AD test, and these are called the NPS and the m-NP1S tests, respectively.

The function `snpair_ClosePairs` implements the tests that we just described. The function `snpair_ClosePairsBitMatch` implements a similar type of test, but based on a different notion of distance: The distance between two points is measured by counting how many of the most significant bits of their coordinates are the same. The function `snpair_BickelBreiman` implements a multivariate goodness-of-fit test proposed by Bickel and Breiman [5], also based on distances between nearest points.

```
#include <testu01/gdef.h>
#include <testu01/chrono.h>
#include <testu01/statcoll.h>
#include <testu01/unif01.h>
```

Constants

```
#define snpair_MAXM 512
```

Maximal value of m for `snpair_ClosePairs`.

```
#define snpair_MAXREC 12
```

Maximal number of dimensional recursions.

The tests

The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
extern lebool snpair_mNP2S_Flag;
```

If this flag is set `TRUE`, the `mNP2S` statistic in `snpair_ClosePairs` will be printed as well as its p-value, otherwise not. The default value is `TRUE`. This statistic requires many bits of resolution

and for large enough values of N, n and m in 2 dimensions, all generators with less than 32 bits of resolution will fail the `mNP2S` test. For example, for $N = 20$, $n = 10^6$, $m = 20$, $t = 2$, all generators returning less than 32 bits of resolution will fail the test.

```
void snpair_ClosePairs (unif01_Gen *gen, snpair_Res *res,
long N, long n, int r, int t, int p, int m);
```

Applies the close-pairs tests NP, m-NP, m-NP1, and m-NP2, and m-NP2S by generating n points in the t-dimensional unit torus and computing the m nearest distinct pairs of points, where the distances between the points are measured using the L_p norm if $p \geq 1$, and the L_∞ norm if $p = 0$. Recommendation: $n \geq 4m^2\sqrt{N}$ for $t \leq 8$. Restrictions: $m \leq$ `snpair_MAXM`.

```
void snpair_ClosePairsBitMatch (unif01_Gen *gen, snpair_Res *res,
long N, long n, int r, int t);
```

Generates n points in the unit hypercube in t dimensions as in `snpair_ClosePairs` and computes the closest pair, but using a different definition of distance. The distance between two points X_i and X_j is $2^{-b_{i,j}}$, where $b_{i,j}$ is the maximal value of b such that the first b bits in the binary expansion of each coordinate are the same for both X_i and X_j. This means that if the unit hypercube is partitioned into 2^{tb} cubic boxes by dividing each axis into 2^b equal parts, the two points are in the same box for $b \leq b_{i,j}$, but they are in different boxes for $b > b_{i,j}$. Let $D = \min_{1 \leq i < j \leq n} 2^{-b_{i,j}}$ be the minimal distance between any two points. For any two points, $P[b_{i,j} \geq b] = 2^{-tb}$. One has $D \leq 2^{-b}$ if and only if $-\log_2 D = \max_{1 \leq i < j \leq n} b_{i,j} \geq b$, if and only if at least b bits agree for at least one pair, and the probability that this happens is approximately

$$P[D \leq 2^{-b}] \approx 1 - \left(1 - 2^{-tb}\right)^{n(n-1)/2} \stackrel{\text{def}}{=} q_b.$$

If exactly $b = -\log_2 D$ bits agree, the left and right p-values are $p_L = q_b$ and $p_R = 1 - p_{b-1}$, respectively. If $N > 1$, the two-level test computes the minimum of the N copies of D and uses it as a test statistic. The p-value is obtained from

$$P[\min\{D_1, D_2, \ldots, D_N\} \leq 2^{-b}] \approx 1 - \left(1 - 2^{-tb}\right)^{Nn(n-1)/2}.$$

```
void snpair_BickelBreiman (unif01_Gen *gen, snpair_Res *res, long N,
long n, int r, int t, int p, lebool Torus);
```

Applies a test based on the statistic proposed by Bickel and Breiman [5] to test the fit of a set of points to a multidimensional density. The test is described in [87]. As for `snpair_ClosePairs`, n points are generated in the unit hypercube in t dimensions. If `Torus = FALSE`, the distances are computed in the hypercube as usual, whereas if `Torus = TRUE`, the hypercube is treated as a torus for computing distances, so that two points near opposite faces of the cube can be close to each other [87].

For each point i, let D_i be the distance to its nearest neighbour, and let $W_i = \exp(-nV_i)$ where V_i is the volume of a hypersphere of radius D_i. The computed statistic is $T = \sum_{i=1}^{n} (W_{(i)} - i/n)^2$ where the $W_{(i)}$ are the W_i sorted in increasing order. If $N > 1$, the empirical distribution of the N values of T is compared to the theoretical law which is approximated using interpolation tables obtained by simulation as explained in [87]. Restrictions: $\{p, t\} = \{0, 2\}$, $\{0, 15\}$ or $\{2, 2\}$.

sknuth

This module implements the classical statistical tests for RNGs described in Knuth's book [64]. (They were actually taken from the 1981 edition, and implemented some years before the 1998 edition [66] appeared.) Some of these tests are special cases of the *multinomial* tests described and implemented in module `smultin`. In these cases, the functions here simply call the appropriate functions in `smultin`. The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void sknuth_Serial (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, long d, int t);
```

Applies the *serial test* of uniformity in t dimensions, using a chi-square, as described by Knuth [66], p.62. It divides the interval $[0, 1)$ in d equal segments, thus dividing the unit hypercube $[0, 1)^t$ in d^t small hypercubes. It generates n points (t-dimensional vectors) in $[0, 1)^t$, using n non-overlapping vectors of t successive output values from the generator, counts the number of points falling into each small hypercube, and compares those counts with the expected values via a chi-square test. This test is a special case of `smultin_Multinomial` with `Sparse = FALSE` (see [96]). It assumes that we are in the *dense* case, where $n > k = d^t$. Restriction: $n/d^t \geq$ `gofs_MinExpected`.

```
void sknuth_SerialSparse (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, long d, int t);
```

Similar to `sknuth_Serial`, except that a normal approximation is used for the distribution of the chi-square statistic. This is valid asymptotically when n/d^t is bounded and $n \to \infty$. The implementation uses a hashing table in order to allow for larger values of d^t. This test is a special case of `smultin_Multinomial` with `Sparse = TRUE` (see [96]). It corresponds to the sparse case of the serial test, where we assume that n is large and $n \leq k = d^t$. Restrictions: $d^t < 2^{53}$ and $d^t/n < 2^{31}$.

```
void sknuth_Permutation (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int t);
```

Applies the *permutation test* (Knuth [66], page 65). It generates n non-overlapping vectors of t values, each vector using t successive values obtained from the generator, and determines to which permutation each vector corresponds (the permutation that would place the

values in increasing order). The test counts the number of times each permutation has appeared and compares these counts with the expected values ($n/t!$) via a chi-square test. This is a special case of `smultin_Multinomial` with `Sparse = FALSE` and `smultin_GenerCell = smultin_GenerCellPermut`. It corresponds to the dense case, where n, the number of points, should be larger than $t!$, the number of cells. Restrictions: $n/t! \geq$ `gofs_MinExpected` and $2 \leq t \leq 18$.

```
void sknuth_Gap (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, double Alpha, double Beta);
```

Applies the *gap test* described by Knuth [59, 60, 66]. Let $\alpha =$ `Alpha`, $\beta =$ `Beta`, and $p = \beta - \alpha$. The test generates n values in $[0, 1)$ and, for $s = 0, 1, 2, \ldots$, counts the number of times that a sequence of exactly s successive values fall outside the interval $[\alpha, \beta]$ (this is the number of *gaps* of length s between visits to $[\alpha, \beta]$). It then applies a chi-square test to compare the expected and observed number of observations for the different values of s. Restrictions: $0 \leq \alpha < \beta \leq 1$.

```
void sknuth_SimpPoker (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int d, int k);
```

Applies the simplified poker test described by Knuth [59, 60, 66]. It generates n groups of k integers from 0 to $d - 1$, by making nk calls to the generator, and for each group it computes the number s of distinct integers in the group. It then applies a chi-square test to compare the expected and observed number of observations for the different values of s. Restrictions: $d < 128$ and $k < 128$.

```
void sknuth_CouponCollector (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int d);
```

Applies the *coupon collector test* proposed in [49] and described in [66]. The test generates a sequence of random integers in $\{0, \ldots, d - 1\}$, and counts how many must be generated before each of the d possible values appears at least once. This is repeated n times. The test counts how many times exactly s integers were needed, for each s, and compares these counts with the expected values via a chi-square test. Restriction: $1 < d < 62$. If d is too large for a given n, there will be only 1 class for the chi-square and the test will not be done.

```
void sknuth_Run (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, lebool Up);
```

Applies the test of increasing or decreasing subsequences (*runs up* or *runs down*) [61, 100, 66]. It measures the lengths of subsequences of successive values in $[0, 1)$ that are generated in increasing (or decreasing) order. If `Up = TRUE`, it considers *runs up*, otherwise it considers *runs down*. These subsequences are the *runs*. The test thus generates n random numbers, counts how many runs of each length there are after merging all run lengths larger or equal to 6, and computes the statistic V defined in [66], page 67, Eq. (10) (the new version of the test incorporating small $O(1/n)$ corrections is used, as described in the 3rd edition of [66]). For large n, this V should follow approximately the chi-square distribution with 6 degrees of freedom.

```
void sknuth_RunIndep (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, lebool Up);
```

A simplified version of `sknuth_Run`, where the increasing or decreasing subsequences are independent, as suggested in Exercice 3.3.2–14 of [66], page 77. This test skips one value between any two successive runs. In this case, a run has length t with probability $1/t! - 1/(t+1)!$. The function merges all values larger or equal to 6, and applies a chi-square test.

```
void sknuth_MaxOft (unif01_Gen *gen, sknuth_Res1 *res,
long N, long n, int r, int d, int t);
```

Applies the *maximum-of-t* test (Knuth [66], page 70). This test generates n groups of t values in $[0, 1)$, computes the maximum X for each group, and then compares the empirical distribution function of these n values of X with the theoretical distribution function of the maximum, $F(x) = x^t$, via a chi-square test and an Anderson-Darling (AD) test. To apply the chi-square test, the values of X are partitioned into d categories in a way that the expected number in each category, under $\mathcal{H}_0$, is exactly n/d. For $N > 1$, the empirical distribution of the p-values of the AD test is compared with the AD distribution. Restriction: $n/d \geq$ `gofs_MinExpected`.

```
void sknuth_Collision (unif01_Gen *gen, sknuth_Res2 *res,
long N, long n, int r, long d, int t);
```

Applies the *collision test* (Knuth [66], pp. 70–71, and [96]). Similar to `sknuth_Serial`, except that the test computes the number of *collisions* (the number of times a point hits a cell already occupied) instead of computing the chi-square statistic. This is a special case of `smultin_Multinomial` with `Sparse = TRUE`. This test is meaningfull only in the sparse case, with n smaller than k. See the documentation in `smultin`. Restrictions: $d^t < 2^{53}$ (assuming that a `double's` mantissa uses 53 bits of precision) and $d^t/n < 2^{31}$.

```
void sknuth_CollisionPermut (unif01_Gen *gen, sknuth_Res2 *res,
long N, long n, int r, int t);
```

Similar to `sknuth_Collisions`, except that instead of generating vectors as in `sknuth_Serial`, it generates permutations as in `sknuth_Permutation`. It then computes the number of collisions between these permutations. This is a special case of `smultin_Multinomial` with `Sparse = TRUE` and `smultin_GenerCell = smultin_GenerCellPermut`. It corresponds to the sparse case where n, the number of points, should be much smaller than $t!$, the number of cells. Restrictions: $2 \leq t \leq 18$ and $t!/n < 2^{31}$.

smarsa

Implements the statistical tests suggested by George Marsaglia and his collaborators in [103] and other places. Some of these tests are special cases of the overlapping versions of the tests implemented in module `smultin` and in these cases, the functions here simply call `smultin_MultinomialOver`. The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void smarsa_SerialOver (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, long d, int t);
```

Implements the *overlapping t-tuple* test described in [1, 103]. It is similar to `sknuth_Serial`, except that the n vectors are generated with overlap, as follows. A sequence of uniforms $u_0, \ldots, u_{n-1}$ is generated, and the n points are defined as $(u_0, \ldots, u_{t-1})$, $(u_1, \ldots, u_t)$, ..., $(u_{n-1}, u_n, u_0, \ldots, u_{t-3})$, $(u_n, u_0, \ldots, u_{t-2})$. This test is a special case of `smultin_MultinomialOver`, with `Sparse = FALSE` (see also [96]). Restriction: $n/d^t \geq$ `gofs_MinExpected`.

```
void smarsa_CollisionOver (unif01_Gen *gen, smarsa_Res *res,
long N, long n, int r, long d, int t);
```

Similar to the collision test, except that the vectors are generated with overlap, exactly as in `smarsa_SerialOver`. This test corresponds to the test *overlapping pairs sparse occupancy* (OPSO) test described in [103] and studied by Marsaglia and Zaman [105]. Let $\lambda = (n-t+1)/d^t$, called the *density*. If n (the number of points) and d^t (the number of cells) are very large and have the same order of magnitude, then, under $\mathcal{H}_0$, the number of collisions C is a random variable which is approximately normally distributed with mean $\mu \approx d^t(\lambda - 1 + e^{-\lambda})$ (this follows from Theorem 2 of [136]), and variance $\sigma^2 \approx d^t e^{-\lambda}(1 - 3e^{-\lambda})$, according to the speculations of [105] (see `smultin`). However, Rukhin [148] gave a better approximation for the variance as $\sigma^2 \approx d^t e^{-\lambda}(1 - (1+\lambda)e^{-\lambda})$ and this is the formula that is used. When $n \ll d^t$, the number of collisions should be approximately Poisson with mean μ, whereas if λ is large enough (e.g., $\lambda > 6$), then the number of empty cells $(d^t - n + C)$ should be approximately Poisson with mean $d^t e^{-\lambda}$. This test is a special case of `smultin_MultinomialOver`.

```
void smarsa_Opso (unif01_Gen *gen, smarsa_Res *res,
long N, int r, int p);
```

Three special cases of `smarsa_CollisionOver`. Implements the OPSO test with the same three sets of parameters as in the examples of [103]. The parameters (n, d, t) are $(2^{21}, 2^{10}, 2)$, $(2^{22}, 2^{11}, 2)$, and $(2^{23}, 2^{11}, 2)$, for $p = 1$, 2, and 3, respectively.

Restriction: $p \in \{1, 2, 3\}$.

```
void smarsa_CAT (unif01_Gen *gen, sres_Poisson *res,
long N, long n, int r, long d, int t, long S[]);
```

Applies the *CAT test*, one of the *monkey test* proposed by Marsaglia in [118] and analyzed by Percus and Whitlock in [136]. This test is a variation of the collision test with overlapping (`smarsa_CollisionOver`), except that only *one* cell is observed. For this reason, this test is typically less powerful than `smarsa_CollisionOver` unless the target cell happens to be visited very frequently due to a particular weakness of the generator. This target cell is specified by the vector `S[0..t-1]`. For each point, the generator provides t integers $y_0, \ldots, y_{t-1}$ in $\{0, \ldots, d-1\}$ and the target cell is hit whenever $(y_0, \ldots, y_{t-1})$ = `(S[0],...,S[t-1])`. The target cell number should make an aperiodic pattern, i.e., it should not be possible to write it as ABA where A is a prefix of the pattern.

The test generates n numbers (giving $n-t+1$ points with overlapping coordinates) and computes Y, the number of points that hit the target cell. Under $\mathcal{H}_0$, Y is approximately Poisson with mean $\lambda = (n - t + 1)/d^t$, and the sum of all values of Y for the N replications is approximately Poisson with mean $N\lambda$. The test computes this sum and the corresponding p-value, using the Poisson distribution. Note: The pair (N, n) may be replaced by $(1, nN)$, as this is equivalent. Normally, λ should be larger than 1, so this corresponds to the dense case, where $n > k$.

```
void smarsa_CATBits (unif01_Gen *gen, sres_Poisson *res, long N, long n,
int r, int s, int L, unsigned long Key);
```

Similar to `smarsa_CAT`, except that the cell is generated from a string of bits. This test is a variation of the multinomial test on bits with overlapping (`smultin_MultinomialBitsOver`), except that only *one* cell is observed (for this reason, this test is typically less powerful than `smultin_MultinomialBitsOver`). This target cell is specified by `Key`. Each point is made of L bits and the target cell is hit whenever the L bits are numerically equal to `Key`. The test compares each group of L bits to the key in a sequence of n bits. When the key is not found, one moves 1 bit forward in the sequence. But when the key is found, one jumps L bits forward. The bits of `Key` should make an aperiodic pattern, i.e., it should not be possible to write `Key` (in binary form) as ABA where A is a binary prefix of `Key`.

The test generates n bits and computes Y, the number of points that hit the target cell. Under $\mathcal{H}_0$, Y is approximately Poisson with mean $\lambda = (n - L + 1)/2^L$, and the sum of all values of Y for the N replications is approximately Poisson with mean $N\lambda$. The test computes this sum and the corresponding p-value, using the Poisson distribution. Normally, λ should be larger than 1, so this corresponds to the dense case, where $n > 2^L$. Restrictions: $L \leq 32$, $r + s \leq 32$, and if $L > s$ then $L \bmod s = 0$.

```
void smarsa_BirthdaySpacings (unif01_Gen *gen, sres_Poisson *res,
long N, long n, int r, long d, int t, int p);
```

Implements the *birthday spacings* test proposed in [103] and studied further by Knuth [66] and L'Ecuyer and Simard [93]. This is a variation of the collision test, in which n points are thrown into $k = d^t$ cells in t dimensions as in `smultin_Multinomial`. The cells are numbered from 0 to $k - 1$. To generate a point, t integers $y_0, \ldots, y_{t-1}$ in $\{0, \ldots, d - 1\}$ are generated from

t successive uniforms. The parameter p decides in which order these t integers are used to determine the cell number: The cell number is $c = y_0 d^{t-1} + \cdots + y_{t-2} d + y_{t-1}$ if $p = 1$ and $c = y_{t-1} d^{t-1} + \cdots + y_1 d + y_0$ if $p = 2$. This corresponds to using `smultin_GenerCellSerial` for $p = 1$ and `smultin_GenerCellSerial2` for $p = 2$.

The points obtained can be viewed as n birth dates in a year of k days [1, 103]. Let $I_1 \leq I_2 \leq \cdots \leq I_n$ be the n cell numbers obtained, sorted in increasing order. The test computes the differences $I_{j+1} - I_j$, for $1 \leq j < n$, and count the number Y of collisions between these differences. Under $\mathcal{H}_0$, Y is approximately Poisson with mean $\lambda = n^3/(4k)$, and the sum of all values of Y for the N replications (the total number of collisions) is approximately Poisson with mean $N\lambda$. The test computes this total number of collisions and computes the corresponding p-value using the Poisson distribution. Recommendation: k should be very large and λ relatively small. Restrictions: $k \leq$ `smarsa_Maxk`, $8N\lambda \leq k^{1/4}$ or $4n \leq k^{5/12}/N^{1/3}$, and $p \in \{1, 2\}$. [1]

```
void smarsa_MatrixRank (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int s, int L, int k);
```

Applies the test based on the *rank of a random binary matrix*, as suggested in [103, 115]. It fills a $L \times k$ matrix with random bits as follows. A sequence of uniforms are generated and s bits are taken from each. The matrix is filled one row at a time, using $\lceil k/s \rceil$ uniforms per row. The test then computes the rank of the matrix (the number of linearly independent rows). It thus generates n matrices and counts how many there are of each rank. Finally it compares this empirical distribution with the theoretical distribution of the rank of a random matrix, via a chi-square test, after merging classes if neeeded (as usual). The probability that the rank R of a random matrix is x is given by

$$\begin{aligned} P[R = 0] &= 2^{-Lk} \\ P[R = x] &= 2^{x(L+k-x)-Lk} \prod_{i=0}^{x-1} \frac{(1 - 2^{i-L})(1 - 2^{i-k})}{1 - 2^{i-x}}, \qquad 1 \leq x \leq \min(L, k). \end{aligned}$$

Restrictions: $n/2 >$ `gofs_MinExpected`. Recommendation: $L = k$. The difference $|L - k|$ should be small, otherwise almost all the probability will be lumped in a single class of the chi-square.

```
void smarsa_Savir2 (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, long m, int t);
```

Applies a modified version of the Savir test, as proposed by Marsaglia [104]. The test generates a random integer I_1 uniformly in $\{1, \ldots, m\}$, then a random integer I_2 uniformly in $\{1, \ldots, I_1\}$, then a random integer I_3 uniformly in $\{1, \ldots, I_2\}$, and so on until I_t. It thus generates n values of I_t and compares their empirical distribution with the theoretical one, via a chi-square test. The algorithm given in [104] is used to compute the theoretical distribution of I_t under the null hypothesis. Restrictions: $n/2 >$ `gofs_MinExpected`. Recommendation: $m \approx 2^t$.

[1] From Richard: Ce test est beaucoup plus sensible pour un très grand nombre de cellules avec n grand. Pour Nn constant, choisir n aussi grand que possible et N petit. Mais on est limité par la valeur maximale d'un entier pour numéroter les cases, et aussi par le fait que la densité doit être suffisamment petite pour que l'approximation de Poisson soit valide.

```
void smarsa_GCD (unif01_Gen *gen, smarsa_Res2 *res,
long N, long n, int r, int s);
```

Applies the tests based on the greatest common divisor (GCD) between two random integers in $[1, 2^s]$ as proposed by Marsaglia [114]. The first test considers the value of the GCD itself for which the probability that the GCD takes the value j is $P_j = 6/(\pi^2 j^2)$ (see [66]). A chi-square test is applied on the values obtained. The second test considers the number of iterations needed to find the GCD. The theoretical distribution is unknown and the binomial approximation proposed by Marsaglia has to be corrected by an empirical factor. This second test is not used for the moment and is left as a future project. Restrictions: $n \geq 30$ and $\log_2 n \leq 3s/2$.

svaria

This module implements various tests of uniformity based on relatively simple statistics, as well as a few specific tests proposed in the literature. The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void svaria_SampleMean (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r);
```

This test generates n uniforms $u_1, \ldots, u_n$ and computes their average

$$\overline{u}_n = \frac{1}{n} \sum_{j=1}^{n} u_j.$$

The distribution of the N values of $\overline{u}_n$ is compared with the exact theoretical distribution

$$P[n\overline{u}_n \le z] = \frac{1}{n!} \sum_{j=0}^{\lfloor z \rfloor} (-1)^j \binom{n}{j} (z-j)^n, \qquad 0 \le z \le n$$

given by Stephens [153] for $n < 60$, and to the normal distribution with mean $1/2$ and variance $1/(12n)$ for $n \ge 60$.

```
void svaria_SampleCorr (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, int k);
```

This test generates n uniforms $u_1, \ldots, u_n$ and computes the empirical autocorrelation [38] of lag k,

$$\hat{\rho}_k = \frac{1}{(n-k)} \sum_{j=1}^{n-k} \left(u_j u_{j+k} - \tfrac{1}{4}\right).$$

The empirical distribution of the N values of $\sqrt{12(n-k)}\hat{\rho}_k$ is compared with the standard normal distribution, which is its asymptotic theoretical distribution when $n \to \infty$. The approximation is valid only when n is very large. Restriction: $k \ll n$.

```
void svaria_SampleProd (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, int t);
```

This test generates tn uniforms $u_1, \ldots, u_{tn}$ and computes the empirical distribution of the products of n nonoverlapping successive groups of t values, $\{u_{(j-1)t+1}, u_{(j-1)t+2}, \ldots, u_{jt} : j =$

$1, \ldots, n\}$. For $N = 1$, this test is equivalent to calling `svaria_SumLogs (gen, res, n, t, r)`. This distribution is compared with the theoretical distribution of the product of t independent $U(0,1)$ random variables, given by

$$P[U_1 U_2 \ldots U_t \leq x] = F(x) = x \sum_{j=0}^{t-1} \frac{(-\ln x)^j}{j!}$$

for $0 \leq x \leq 1$, via an Anderson-Darling (AD) test. For $N > 1$, the empirical distribution of the p-values of the AD test is compared with the AD distribution.

```
void svaria_SumLogs (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r);
```

This test generates n uniforms, $u_1, \ldots, u_n$, and computes

$$P = -2 \sum_{j=1}^{n} \ln(u_j).$$

Under $\mathcal{H}_0$, $P/2$ is a sum of n i.i.d. exponentials of mean 1, so P has the chi-square distribution with $2n$ degrees of freedom (see [154]).

```
void svaria_WeightDistrib (unif01_Gen *gen, sres_Chi2 *res, long N, long n,
int r, long k, double alpha, double beta);
```

Applies the test proposed by Matsumoto and Kurita [125], page 264. This test generates k uniforms, $u_1, \ldots, u_k$, and computes

$$W = \sum_{j=1}^{k} I[\alpha \leq u_j < \beta],$$

the number of u_j's falling in the interval $[\alpha, \beta)$. Under $\mathcal{H}_0$, W is a binomial random variable with parameters k and $p = \beta - \alpha$. This is repeated n times, thus obtaining $W_1, \ldots, W_n$, whose empirical distribution is compared with the binomial distribution via a chi-square test. For the chi-square test, classes (possible values of W) are regrouped as needed to ensure that the expected numbers in each class is larger or equal to `gofs_MinExpected`.

```
void svaria_CollisionArgMax (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, long k, long m);
```

Applies a generalization of the test proposed by Sullivan [155]. This test generates k uniforms, $u_1, \ldots, u_k$, and computes

$$I = \min \Big\{ i \mid 1 \leq i \leq k \text{ and } u_i = \max(u_1, \ldots, u_k) \Big\},$$

the index of the largest value. It repeats this n times and counts the number C of collisions among the n values of I, which is equal to n minus the number of distinct values of I. If $m > 1$, this is repeated m times and the empirical distribution of the m values of C is compared with the theoretical distribution of C, given in [66] (see also `sknuth_Collisions`) by

applying a chi-square test. For $m = 1$, this test is equivalent to the collision test applied by `smultin_Multinomial` with `smultin_GenerCell = smultin_GenerCellMax`. Recommendations: $n \leq k$, and either $m = 1$ or m large enough for the chi-square test to make sense.

```
void svaria_SumCollector (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, double g);
```

Applies a test proposed by Ugrin-Sparac [165]. It generates a sequence of uniforms $u_0, u_1, \ldots$ adds them up until their sum exceeds g. It then defines $J = \min\{\ell \geq 0 : u_0 + \cdots + u_\ell > g\}$. This is repeated n times, to obtain n copies of J, say $J_1, \ldots, J_n$, whose empirical distribution is compared to the theoretical distribution given in [165], by a chi-square test. Restriction: $1 \leq g \leq 10$.

```
void svaria_AppearanceSpacings (unif01_Gen *gen, sres_Basic *res,
long N, long Q, long K, int r, int s, int L);
```

Applies the "universal test" proposed by Maurer [127]. The goal of this test is to measure the entropy of a sequence of random bits (it is somewhat related to the test applied by `sentrop_EntropyDisc`). The test takes the s most significant bits (after dropping the first r) from each uniform, and concatenates these s-bit blocks to construct $Q + K$ blocks of L bits. The first Q blocks are used for the initialization, and the K following blocks serve for the test proper. For each of these K blocks, the function finds the number of blocks generated since the most recent occurrence of the same block in the sequence. If it is generating the j-th block and its most recent occurrence was in block $(j - i)$-th, it sets $A_j = i$; if it is its first occurrence, it sets $A_j = j$. It then computes the average

$$Y = \frac{1}{K} \sum_{j=Q+1}^{Q+K} \log_2 A_j,$$

whose distribution under $\mathcal{H}_0$ is approximately normal with mean and variance given in [127]. A better formula for the variance was given in [12] and that is the one used. The approximation is good only when Q and K are very large. Recommendations: $Q \geq 10 \times 2^L$ and $K \gg 2^L$. Restrictions: $s \bmod L = 0$ if $s > L$.

swalk

This module implements statistical tests based on discrete random walks over the set $\mathbb{Z}$ of all integers. Similar tests are discussed, e.g., by Vattulainen [166, 167] and Takashima [156].

The main function, `swalk_RandomWalk1`, applies simultaneously several tests based on a random walk of length ℓ over the integers, for several (even) values of ℓ. The random walk is generated from a bit string produced by the generator by taking s bits per output value. Each bit determines one step of the walk: move by 1 to the left if the bit is 0 and by 1 to the right if the bit is 1. The function `swalk_RandomWalk1a` implements a variant where each move of the walk is determined by taking a linear combination modulo 2 (i.e., exclusive-or) of certain bits in the string.

The functions `swalk_VarGeoP` and `swalk_VarGeoN` implement "random walk" tests defined in terms of real numbers. These tests were proposed in [150] and turn out to be special cases of the gap test implemented in module `sknuth_gap`.

The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/bitset.h>
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void swalk_RandomWalk1 (unif01_Gen *gen, swalk_Res *res, long N, long n,
int r, int s, long L0, long L1);
```

Applies various tests based on a *random walk* over the set of integers $\mathbb{Z}$. The walk starts at 0 and at each step, it moves by one unit to the left with probability 1/2, and by one unit to the right with probability 1/2. The test considers ℓ-step random walks, for all *even* integers ℓ in the interval $[L_0, L_1]$. It first generates a random walk of length $\ell = L_0$, then adds two steps to obtain a random walk of length $\ell = L_0 + 2$, then adds two more steps for a walk of length $\ell = L_0 + 4$, and so on until $\ell = L_1$.

To generate the moves, the test uses one bit b_i at each step i. It takes s bits from each uniform (after dropping the first r). So for the entire walk of length L_1, it needs $\lceil L_1/s \rceil$ uniforms. Let $X_i = 1$ if $b_i = 1$, and $X_i = -1$ if $b_i = 0$. Define also $S_0 = 0$ and

$$S_k = \sum_{i=1}^{k} X_i, \quad k \geq 1.$$

The process $\{S_k,\, k \geq 0\}$ is a random walk over the integers. Under $\mathcal{H}_0$, we have

$$p_{k,y} = P[S_k = y] = \begin{cases} 2^{-k} \binom{k}{(k+y)/2} & \text{if } k+y \text{ is even;} \\ 0 & \text{if } k+y \text{ is odd.} \end{cases}$$

For ℓ even, we define the statistics:

$$\begin{aligned} H &= \ell/2 + S_\ell/2 \;=\; \sum_{i=1}^{\ell} I[X_i = 1], \\ M &= \max\left\{S_k,\;\; 0 \leq k \leq \ell\right\}, \\ J &= 2\sum_{k=1}^{\ell/2} I[S_{2k-1} > 0], \\ P_y &= \min\left\{k : S_k = y\right\}, \quad y > 0, \\ R &= \sum_{k=1}^{\ell} I[S_k = 0], \\ C &= \sum_{k=3}^{\ell} I[S_{k-2} S_k < 0], \end{aligned}$$

where I denotes the indicator function. Here, H is the number of steps to the right, M is the maximum value reached by the walk, J is the fraction of time spent on the right of the origin, P_y is the first passage time at y, R is the number of returns to 0, and C is the number of sign changes.

The test thus generates n random walks and computes the n values of each of these statistics. It compares the empirical distribution of these n values with the corresponding theoretical law, via a chi-square test (regrouping classes if needed).

The theoretical probabilities for these statistics under $\mathcal{H}_0$ are as follows:

$$\begin{aligned} P[H = k] &= P[S_\ell = 2k - \ell] \;=\; p_{\ell,2k-\ell} \;=\; 2^{-\ell}\binom{\ell}{k}, \quad 0 \leq k \leq \ell, \\ P[M = y] &= p_{\ell,y} + p_{\ell,y+1}, \quad 0 \leq y \leq \ell, \\ P[J = k] &= p_{k,0}\; p_{\ell-k,0}, \quad 0 \leq k \leq \ell,\; k \text{ even}, \\ P[P_y = k] &= (y/k) p_{k,y}, \\ P[R = y] &= p_{\ell-y,y}, \quad 0 \leq y \leq \ell/2, \\ P[C = y] &= 2p_{\ell-1,2y+1}, \quad 0 \leq y \leq (\ell-1)/2. \end{aligned}$$

The size of the memory used by the test is approximately $6L_1(L_1 - L_0 + 1)/(10^5)$ megabytes. Restrictions: $r + s \leq 32$, L_0 and L_1 even, and $L_0 \leq L_1$.

```
void swalk_RandomWalk1a (unif01_Gen *gen, swalk_Res *res, long N, long n,
int r, int s, int t, long L, bitset_BitSet C);
```

Applies the same collection of tests as `swalk_RandomWalk1`, except that ℓ takes a single value, `L`, and that the X_i's are defined in a different (more general) way, as follows. The parameter `C`

contains a vector of fixed binary coefficients (bits) $c_0, \ldots, c_{t-1}$, not all zero. The test generates a long sequence of bits $b_0, b_1, \ldots$ and puts $X_i = 1$ if $y_i = c_0 b_i + \ldots + c_{t-1} b_{i+t-1}$ is odd, $X_i = -1$ otherwise. Restrictions: $r + s \leq 32$, $t \leq 31$, and L even.

```
void swalk_VarGeoP (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, double Mu);
```

Applies a test based on the length of a certain "random walk", proposed by [150]. This test turns out to be a special case of the *gap test* implemented in `sknuth_gap`. It generates uniforms until one of these uniforms is larger or equal to `Mu` and counts how many uniforms were needed (the number of steps in the random walk). This is repeated n times, the number of walks of each length is counted, and a chi-square test is applied to compare these counts to the their expectations under $\mathcal{H}_0$. Restriction: `Mu` $\in (0, 1)$.

```
void swalk_VarGeoN (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, double Mu);
```

Same as `swalk_VarGeoP`, but with "larger or equal to `Mu`" replaced by "less than `1 - Mu`".

scomp

This module contains three tests based on the evolution of the *linear complexity* of a sequence of bits as it grows, and a test based on the compressibility of the bit sequence, as measured by the Lempel-Ziv complexity. For the compressibility test, we use the Lempel-Ziv compression algorithm of 1978 (see [180]). A similar test is implemented in [36, 147], but according to the authors, it uses a version of the Lempel-Ziv algorithm of 1977 instead [179]. The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void scomp_LinearComp (unif01_Gen *gen, scomp_Res *res,
long N, long n, int r, int s);
```

This procedure applies simultaneously the jump complexity test and the jump size test, two tests based on the linear complexity of a sequence of bits and described in [10, 36]. A sequence of n bits is constructed by taking s bits from each random number. For each ℓ, $1 \leq \ell \leq n$, the linear complexity of the subsequence formed by the first ℓ bits is computed by the Berlekamp-Massey algorithm [4, 121].

The *jump complexity test* counts the number of jumps occurring in the linear complexity of the sequence. A *jump* occurs whenever adding the next bit to the sequence increases its linear complexity. Under $\mathcal{H}_0$, for n sufficiently large, the number of jumps, say J, is approximately normally distributed with mean and variance given by [169]:

$$E(J) = \frac{n}{4} + \frac{4 + R_n}{12} - \frac{1}{3(2^n)} \tag{3.9}$$

$$\mathrm{Var}(J) = \frac{n}{8} - \frac{2 - R_n}{9 - R_n} + \frac{n}{6(2^n)} + \frac{6 + R_n}{18(2^n)} - \frac{1}{9(2^{2n})}, \tag{3.10}$$

where R_n is the parity of n ($= 0$ for even n, 1 for odd n). The test compares the standardized value of the observed number of jumps to the standard normal distribution.

The *jump size test* looks at the *size* of the jumps (there is a jump of size h if the complexity increases by h when we consider the next bit of the sequence), and counts how many jumps of each size have occurred. It then compares these countings with the expected values via a chi-square test. Carter has shown [10] that under $\mathcal{H}_0$, the jump sizes are i.i.d. random variables which obey a geometric law with parameter 1/2.

When N is large enough, the procedure also applies a third test, taken from [147]. It is a chi-square test based on the N replications of the statistic

$$T_n = (-1)^n (L_n - \xi_n) + 2/9,$$

where L_n is the linear complexity of the sequence, and $\xi_n = n/2+(4+R_n)/18$ is an approximation of $E[L_n]$ under $\mathcal{H}_0$. The statistic T_n takes only integer values, with probabilities given by (see [147]):

$$P[T=k] \;=\; \begin{cases} 0.5 & \text{for } k=0, \\ 2^{-2k} & \text{for } k=1,2,3,\ldots \\ 2^{-2|k|-1} & \text{for } k=-1,-2,-3,\ldots \end{cases}$$

Recommendation: take $N = 1$ and n large (however, computing the linear complexity takes $O(n^2 \log n)$ time).

```
void scomp_LempelZiv (unif01_Gen *gen, sres_Basic *res,
long N, int k, int r, int s);
```

Given a string of $n = 2^k$ bits, this test [147, 36] counts the number W of distinct patterns in the string in order to determine its compressibility by the Lempel-Ziv compression algorithm of 1978 (see [180]). According to [63], under $\mathcal{H}_0$,

$$Z = \frac{W - n/\log_2 n}{\sqrt{0.266\, n/(\log_2 n)^3}},$$

has approximately the standard normal distribution for large n. However, our tests show that even for n as large as 2^{24}, the approximation is not very good. Our implementation of the test assumes that W has approximately the normal distribution, but uses estimates of its mean and standard deviation that have been obtained through simulation with two differents reliable generators. Restriction: $k \leq 28$ and N not too large.

sspectral

This module contains tests based on spectral methods. The tests currently available compute the *discrete Fourier transform* for a string of n bits and look for deviations in the spectrum that are inconsistent with $\mathcal{H}_0$. The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/statcoll.h>
#include <testu01/gofw.h>
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void sspectral_Fourier1 (unif01_Gen *gen, sspectral_Res *res,
long N, int k, int r, int s);
```

This test is taken from [147]. Given a string of $n = 2^k$ bits, let $A_j = -1$ if the jth bit is 0 and $A_j = 1$ if the jth bit is 1. Define the discrete Fourier coefficients

$$f_\ell = \sum_{j=0}^{n-1} A_j e^{2\pi i j\ell/n}, \qquad \ell = 0, 1, \ldots, n-1, \tag{3.11}$$

where $i = \sqrt{-1}$, and let $|f_\ell|$ be the modulus of the complex number f_ℓ. Note that since the A_j are real, the f_ℓ for $\ell > n/2$ can be obtained simply from the f_ℓ with $\ell \leq n/2$. Let O_h denote the observed number of $|f_\ell|$'s, for $\ell \leq n/2$, that are smaller than h. According to [147], under $\mathcal{H}_0$, for large enough n and $h = \sqrt{2.995732274n}$, O_h has approximately the normal distribution with mean $\mu = 0.95n/2$ and variance $\sigma^2 = 0.05\mu$. The test computes the N values of the standardized statistic $(O_h - \mu)/\sigma$ and compares their distribution to the standard normal. Restrictions: $8 \leq k \leq 20$ and N very small.

```
void sspectral_Fourier2 (unif01_Gen *gen, sspectral_Res *res,
long N, int k, int r, int s);
```

This test, proposed and studied by Erdmann [36], computes $S_\ell = |f_\ell|^2/n$, for $\ell = 0, 1, \ldots, n-1$, where the Fourier coefficients f_ℓ are defined in (3.11). It is shown in [36] that under $\mathcal{H}_0$, each S_ℓ has mean 1 and variance $1 - 2/n$ for $\ell \neq 0$. The test computes the sum

$$X = \sum_{\ell=1}^{n/4} S_\ell,$$

which should be approximately normal with mean $n/4$ and variance equal to $(n-2)/4$. It compares the distribution of the N values of X with the normal distribution. Restrictions: $4 \leq k \leq 26$ and N very small. Recommendations: $N = 1$.

```
void sspectral_Fourier3 (unif01_Gen *gen, sspectral_Res *res,
long N, int k, int r, int s);
```

For each ℓ, let X_ℓ denote the sum of the N copies of S_ℓ, where S_ℓ is computed and defined as in `sspectral_Fourier2`. The central limit theorem ensures that for N large enough, X_ℓ should be approximately normal with mean N and variance NV_ℓ. This test compares the empirical distribution of the $n/4$ normal variables X_ℓ, $\ell = 1, 2, \ldots, n/4$, to the standard normal distribution. Restriction: $4 \leq k \leq 26$. Recommendation: $N \geq 2^k$.

sstring

This module implements different tests that are applied to strings of random bits. Each test takes a block of s bits from each uniform and concatenate them to construct bit strings. The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/tables.h>
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void sstring_PeriodsInStrings (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int s);
```

This function applies a test based on the distribution of the correlations in bit strings of length s. The *correlation* of a bit string $b = b_0 b_1 \cdots b_{s-1}$ is defined as the bit vector $c = c_0 c_1 \cdots c_{s-1}$ such that $c_p = 1$ if and only if p is a period of b, i.e. if and only if $b_i = b_{i+p}$ for $i = 0, \ldots, s-p-1$ (see Guibas and Odlyzko [50]). One always has $c_0 = 1$.

The function first enumerates all possible correlations c for bit strings of length s, and computes the expected number of strings of correlation c, $E_s(c) = nL_s(c)/2^s$, where $L_s(c)$ is the number of strings, among all 2^s strings of length s, having correlation c. These numbers are computed as explained in [50]. Then, n strings of length s are generated, by generating n uniforms and keeping the s most significant bits of each (after discarding the first r), the corresponding correlations are computed, and the number of occurrences of each correlation is computed. These countings are compared with the corresponding expected values via a chi-square test, after regrouping classes if needed to make sure that the expected number of values in each class is at least `gofs_MinExpected`. Restrictions: $2 \leq s \leq 31$, $r + s \leq 31$, $n/2 >$ `gofs_MinExpected`.

```
void sstring_LongestHeadRun (unif01_Gen *gen, sstring_Res2 *res,
long N, long n, int r, int s, long L);
```

This test generates n blocks of L bits by taking s bits from each of $n\lfloor L/s \rfloor$ successive uniforms. In each block, it finds the length ℓ of the longest run of successive 1's, and counts how many times each value of ℓ has occurred. It then compares these countings to the corresponding expected values via a chi-square test, after regrouping classes if needed. The expected values (i.e., the theoretical distribution) are computed as described in [43], [47] and [147, p. 21]. It also finds the length ℓ of the longest run of 1's over all blocks of all N replications and compares it with the theoretical distribution. Restrictions: $L \geq 1000$, $r + s \leq 32$.

```
void sstring_HammingWeight (unif01_Gen *gen, sres_Chi2 *res,
long N, long n, int r, int s, long L);
```

This test is a one-dimensional version of `sstring_HammingIndep`. It generates n blocks of L bits and examines the proportion of 1's within each (non-overlapping) L-bit block. Under $\mathcal{H}_0$, the number of 1's in each block are i.i.d. binomial random variables with parameters L and $1/2$ (with mean $L/2$ and variance $L/4$). Let X_j be the number of blocks amongst n, having j 1's (i.e., with Hamming weight j). The observed numbers of blocks having j 1's are compared with the expected numbers via a chi-square test. Restrictions: $r+s \leq 32$ and $n \geq 2*$ `gofs_MinExpected`.

```
void sstring_HammingWeight2 (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, int s, long L);
```

This test is taken from [147]. Given a string of n bits, the test examines the proportion of 1's within (non-overlapping) L-bit blocks. It partitions the bit string into $K = \lfloor n/L \rfloor$ blocks. Let X_j be the number of 1's in block j (i.e., its Hamming weight). Under $\mathcal{H}_0$, the X_j's are i.i.d. binomial random variables with parameters L and $1/2$ (with mean $L/2$ and variance $L/4$). The test computes the chi-square statistic

$$X^2 = \sum_{j=1}^{K} \frac{(X_j - L/2)^2}{L/4} = \frac{4}{L} \sum_{j=1}^{K} (X_j - L/2)^2,$$

which should have approximately the chi-square distribution with K degrees of freedom if L is large enough. For $L = n$, this test degenerates to the *monobit* test used in [147], which simply counts the proportion of 1's in a string of n bits. Restrictions: $r + s \leq 32$, $L \leq n$ and L large.

```
void sstring_HammingCorr (unif01_Gen *gen, sstring_Res *res,
long N, long n, int r, int s, int L);
```

Applies a correlation test on the Hamming weights of successive blocks of L bits (see [91]). It is strongly related to the test `sstring_HammingIndep` below. The test uses the s most significant bits from each generated random number (after dropping the first r bits) to build n blocks of L bits. Let X_j be the Hamming weight (the numbers of bits equal to 1) of the jth block, for $j = 1, \ldots, n$. The test computes the empirical correlation between the successive X_j's,

$$\hat{\rho} = \frac{4}{(n-1)L} \sum_{j=1}^{n-1} (X_j - L/2)\,(X_{j+1} - L/2)\,.$$

Under $\mathcal{H}_0$, as $n \to \infty$, $\hat{\rho}\sqrt{n-1}$ has asymptotically the standard normal distribution. This is what is used in the test. The test is valid only for large n.

```
void sstring_HammingIndep (unif01_Gen *gen, sstring_Res *res,
long N, long n, int r, int s, int L, int d);
```

Applies two tests of independence between the Hamming weights of successive blocks of L bits. These tests were introduced by L'Ecuyer and Simard [91]. They use the s most significant bits from each generated random number (after dropping the first r bits) to build $2n$ blocks

of L bits. Let X_j be the Hamming weight (the numbers of bits equal to 1) of the jth block, for $j = 1, \ldots, 2n$. Each vector (X_i, X_{i+1}) can take $(L+1) \times (L+1)$ possible values. The test counts the number of occurrences of each possibility among the non-overlapping pairs $\{(X_{2j-1}, X_{2j}),\ 1 \le j \le n\}$, and compares these observations with the expected numbers under $\mathcal{H}_0$, via a chi-square test, after lumping together in a single class all classes for which the expected number is less than `gofs_MinExpected`. Restriction: $n \ge 2$*`gofs_MinExpected`.

The function also applies the following (second) test on these countings, which are placed in a $(L+1) \times (L+1)$ matrix in the natural way. First, the $2d-1$ rows and $2d-1$ columns at the center of the matrix are discarded if L is even, and the $2d-2$ central rows and columns are discarded if L is odd. There remain four submatrices, at the four corners. Now, let Y_1 be the sum of the counters in the lower left and upper right submatrices, Y_2 the sum of the counters in the lower right and upper left submatrices, and $Y_3 = n - Y_1 - Y_2$. The observed values of Y_1, Y_2 and Y_3 are compared with their expected values with a chi-square test. The chi-square statistic has 1 degree of freedom if $d = 1$ and m is odd (because $Y_3 = 0$ in that case), and 2 degrees of freedom otherwise. Restrictions: $d \le 8$, $d \le (L+1)/2$. If d is too large for a given n, the expected numbers in the categories will be too small for the chi-square to be valid.

```
void sstring_Run (unif01_Gen *gen, sstring_Res3 *res,
long N, long n, int r, int s);
```

This is a version of the run test applicable to a bit string. It is also related to the gap test. In a bit string of length n, the runs of successive 1's can be seen as *gaps* between the blocks of successive 0's. These gap (or run) lengths are independent geometric random variables, plus 1; i.e., each run has length i with probability 2^{-i}, for $i = 1, 2, \ldots$. Of course, this is also true if we swap 0 and 1 and look at the runs of 0's.

Suppose we construct a bit string until we have n runs of 1's and n runs of 0's, i.e., a total of $2n$ runs. Select some positive integer k. For $j = 0$ and 1, let $X_{j,i}$ be the number of runs of j's of length i for $i = 1, \ldots, k-1$, and let $X_{j,k}$ be the number of runs of j's of length $\ge k$. Under $\mathcal{H}_0$, for each j, $(X_{j,1}, \ldots, X_{j,k-1}, X_{j,k})$ is a multinomial random vector with parameters $(n, p_1, p_2, \ldots, p_{k-1}, p_k)$, where $p_i = 2^{-i}$ for $1 \le i < k$ and $p_k = p_{k-1} = 2^{-k+1}$. Then, if np_k is large enough, the chi-square statistic

$$X_j^2 = \sum_{i=1}^{k} \frac{(X_{j,i} - np_i)^2}{np_i(1-p_i)}$$

has approximately the chi-square distribution with $k-1$ degrees of freedom. Moreover, X_0^2 and X_1^2 are independent, so $X^2 = X_0^2 + X_1^2$ should be approximately chi-square with $2(k-1)$ degrees of freedom. The test is based on the statistic X^2 and uses $k = 1 + \lfloor \log_2(n/$`gofs_MinExpected`$) \rfloor$.

Another test, applied simultaneously, looks at the total number Y of bits to get the $2n$ runs. Under $\mathcal{H}_0$, Y is the sum of $2n$ independent geometric random variables with parameter $1/2$, plus $2n$. For large n, it is approximately normal with mean $4n$ and variance $8n$, so $Z = (Y - 4n)/\sqrt{8n}$ is approximately standard normal. This second test, based on Z, is practically equivalent to the run test proposed in [147], which counts the total number of runs for a fixed number of bits. Restrictions: $r + s \le 32$.

```
void sstring_AutoCor (unif01_Gen *gen, sres_Basic *res,
long N, long n, int r, int s, int d);
```

This test measures the autocorrelation between bits with lag d [36]. It generates a n-bit string and computes

$$A_d = \sum_{i=1}^{n-d} b_i \oplus b_{i+d},$$

where b_i is the ith bit in the string and $\oplus$ denotes the exclusive-or operation. Under $\mathcal{H}_0$, A_d has the binomial distribution with parameters $(n-d, 1/2)$, so that

$$\frac{2A_d - (n-d)}{\sqrt{n-d}}$$

should be approximately standard normal for large $n-d$. Restrictions: $r+s \leq 32$, $1 \leq d \leq \lfloor n/2 \rfloor$, $n-d$ large enough for the normal approximation to be valid.

sspacings

This module implements tests based on sum functions of the spacings between the sorted observations of a sample of independent uniforms. These tests are studied by L'Ecuyer [78].

Let $u_1, \ldots, u_n$ be a sample of n uniforms over $(0, 1)$, and $u_{(1)}, \ldots, u_{(n)}$ their values sorted by increasing order. Define $u_{(0)} = 0$ and $u_{(n+i)} = 1 + u_{(i-1)}$ for $i > 1$. For $m < n$, the *overlapping spacings of order* m are

$$G_{m,i} = U_{(i+m)} - U_{(i)}, \qquad 0 \le i \le n.$$

The general classes of goodness-of-fit statistics considered here are

$$H_{m,n} = \sum_{i=0}^{n-m+1} h(nG_{m,i}) \tag{3.12}$$

and

$$H^{(c)}_{m,n} = \sum_{i=0}^{n} h(nG_{m,i}). \tag{3.13}$$

where h is some smooth function. Such statistics are discussed, e.g., in [140, 68, 141, 51, 78]. The form (3.13) is a *circular* version that puts the observations in a circle before computing the spacings.

Under mild assumptions on h and after being properly standardized, these statistics are asymptotically normal as $n \to \infty$, either for fixed m or when $m \to \infty$ no faster than $O(n^p)$ for $0 < p < 1$ (see, e.g., [21, 51, 56, 68]). In this module, they are standardized using the *exact* mean and variance whenever this is possible.

The functions `sspacings_AllSpacings` and `sspacings_AllSpacings2` apply several tests of the form (3.12) and (3.13) simultaneously on the same sample. Other functions apply only specific tests.

The parameter `res` is usually set to the `NULL` pointer. However, if one wants to examine or post-process the results after a test, then one must explicitly create a `res` structure. See the detailed version of this guide for the definition of the structures and the relevant instructions.

```
#include <testu01/unif01.h>
#include <testu01/sres.h>
```

The tests

```
void sspacings_SumLogsSpacings (unif01_Gen *gen, sspacings_Res *res,
long N, long n, int r, int m);
```

Applies a test based on the sum of the logarithms of the spacings of order m in a sample of n uniforms. The test statistic is

$$L^{(c)}_{m,n} = \sum_{i=0}^{n} \ln(nG_{m,i}).$$

This is the circular version. Under $\mathcal{H}_0$, this statistic is approximately normally distributed for large n, and exact formulas for $E[L_{m,n}^{(c)}]$ and $\mathrm{Var}[L_{m,n}^{(c)}]$ are given in [16, 78].

```
void sspacings_SumSquaresSpacings (unif01_Gen *gen, sspacings_Res *res,
long N, long n, int r, int m);
```

Similar to `sspacings_SumLogsSpacings`, except that the test statistic is the sum of the squares of the spacings of order m,

$$S_{m,n}^{(c)} = \sum_{i=0}^{n} (nG_{m,i})^2.$$

Under $\mathcal{H}_0$, it is approximately normally distributed for large n. Exact formulas for $E[S_{m,n}^{(c)}]$ and $\mathrm{Var}[S_{m,n}^{(c)}]$ (as well as for its non-circular version) are given in [17, 78].

```
void sspacings_ScanSpacings (unif01_Gen *gen, sspacings_Res *res,
long N, long n, int r, double d);
```

The test statistic here is the largest number of values $U_{(i)}$ falling in a window of witdh d when this window scans the interval $[0, 1]$:

$$S(L) = \max_{0 \le i \le n} \max \left\{ j - i + 1 \mid j \ge 1 \text{ and } U_{(j)} - U_{(i)} \le d \right\}.$$

The (discrete) distribution of $S(L)$ has been investigated in [2, 18, 168], and other references given in [154] (see also `fbar_Scan`).

```
void sspacings_AllSpacings (unif01_Gen *gen, sspacings_Res *res,
long N, long n, int r, int m0, int m1, int d,
int LgEps);
```

This function applies simultaneously different tests based on the statistics defined in (3.12) and (3.13), for $h(x) = \ln x$ and $h(x) = x^2$, for m varying from m_0 to m_1 by steps of length d. For example, if $(m_0, m_1, d) = (1, 10, 3)$, the tests will be performed for $m = 1, 4, 7, 10$, while if $(m_0, m_1, d) = (1, 10, 2)$, it will be performed for $m = 1, 3, 5, 7, 9$. If $m_0 = 0$, the program considers all m in $\{1, d, 2d, \ldots\}$ such that $m \le m_1$. For example, if $(m_0, m_1, d) = (0, 11, 3)$, the tests will be performed for $m = 1, 3, 6, 9$. When $h(x) = \ln x$, any spacing equal to zero is reset to ϵ, where $\log_2 \epsilon = -$`LgEps`. If the generator being tested returns b bits of precision, it is recommended to choose `LgEps` $= b + 1$.

```
void sspacings_AllSpacings2 (unif01_Gen *gen, sspacings_Res *res,
long N, long n, int r, int m0, int m1, int d,
int LgEps);
```

Similar to `sspacings_AllSpacings`, except that only the circular versions using the exact mean and variance are applied.

scatter

This module is useful for producing 2-dimensionnal scatter plots of N points obtained from a uniform random number generator. The N points are generated in the t-dimensional unit hypercube $[0,1]^t$, either by taking vectors of t successive output values from the generator, or by taking t *non-successive* values at pre-specified lags. The vectors can overlap or not. Thus, in the case of successive values for overlapping vectors, for instance, $N + t - 1$ uniforms are needed.

A rectangular box R is defined in $[0,1]^t$ by defining bounds $L_i < H_i$ for each coordinate i, for $1 \leq i \leq t$. All the points falling outside that box are discarded. Two coordinate numbers are selected in $\{1, \ldots, t\}$, say r_x and r_y, then all the points are projected on the two-dimensional subspace determined by these two coordinates, and these projected points are shown on the plot.

The plots can appear directly on the computer screen (using *Gnuplot*) or can be stored in a file, in a format chosen by the user (currently, the format is either for LaTeX or for *Gnuplot*). Three different functions are available for producing the scatter plots: `scatter_PlotUnif` reads the data in a file, `scatter_PlotUnif1` receives all the data in its parameters, and `scatter_PlotUnifInterac` gets the data interactively from the user.

N	Number of points
t	Dimension of vectors
Over	`TRUE` if we want overlapping vectors, `FALSE` otherwise
r_x r_y	Components to be plotted
r_1 L_{r_1} H_{r_1}	Axis number and bounds for x_{r_1}
$\vdots$ $\vdots$	
t L_t H_t	Dimension and bounds for x_t
Width Height	Physical dimensions of plot (in cm)
Output	Output format: `latex, gnu_ps, gnu_term`
Precision	Number of decimal digits for the points
Lacunary	`TRUE` if we want lacunary indices, `FALSE` otherwise
i_1	First lacunary index
$\vdots$	$\vdots$
i_t	tth lacunary index

Figure 3.3: General format of the data file for `scatter`.

Figure 3.3 gives the general format of the data file needed by `scatter_PlotUnif`. This file must have the extension "`.dat`". The right side (in the figure and in the file) contains optional (but useful) comments that are disregarded by the program. The values of the

variables on the left must appear in the file, in the same format. There should be no blank line. The name of the file is passed as an argument to the function, without the "`.dat`" extention.

The first line contains an integer N, the total number of points to generate. The second line gives the dimension t of the hypercube. In the third line, *Over* is a boolean indicating whether the vector coordinates overlap (`TRUE`) or not (`FALSE`). The next line gives the two coordinate numbers r_x and r_y selected for the plot. Each of the following lines contains a coordinate number r_i (an integer from 1 to t), and the lower and upper boundaries L_{r_i} and H_{r_i} (real) of the box R along the coordinate x_{r_i}. One must have $0 \leq L_i < H_i \leq 1$. For the coordinates that do not appear here, the boundaries are set to 0.0 and 1.0 by default. The last coordinate ($r_i = t$) must always appear. On the next line, *Width* and *Height* (real) specify the physical dimensions (in *cm*) of the rectangle for the plot (on paper). The variable *Output* specifies the format of the output file. The values currently allowed are `latex, gnu_ps, gnu_term`, and they correspond to creating a file for LaTeX, creating a file for *Gnuplot*, and showing the plot on the screen using *Gnuplot*, respectively. The variable *Precision* specifies the number of decimal digits to be printed for the points coordinates. If the boolean variable *Lacunary* is `FALSE`, the vectors are constructed from successive output values of the generator, and all the lines that follow are discarded. If *Lacunary* is `TRUE`, the t lines that follow must give the values of the t lacunary indices $i_1 < \cdots < i_t$ (integers). The points used in the plot are $\{(u_{i_1+n}, \ldots, u_{i_t+n}),\ n = 0, \ldots, N-1\}$ in the overlapping case.

As an illustration, suppose the data file is called `dice.dat`. If the output format is `latex`, the output file `dice.tex` will be created by the program. The command `latex dice.tex` can then produce a file `dice.dvi` that contains the plot. If the output format is `gnu_ps` or `gnu_term`, the two files `dice.gnu` and `dice.gnu.points` are created. The file `dice.gnu` contains a set of gnuplot commands to plot the points, which are kept in `dice.gnu.points`. The command `gnuplot dice.gnu` can then produce the scatter plot either on the terminal (if the output format was `gnu_term`) or in a PostScript file (if the output format was `gnu_ps`).

An example. Figure 3.4 gives an example of a small program that creates a scatter plot of numbers produced by the random number generator `RAND()` of Microsoft Excel 1997. A long sequence of numbers was previously generated by Excel and saved in the ASCII file `excel.pts`. In the program, the instruction `gen = ufile_CreateReadText ("excel.pts")` says that the generator `gen` will now simply reads its numbers from that file. Then, the instruction `scatter_PlotUnif (gen, "excel")` calls a function that plots the points after reading the data related to the plot in file `excel.dat`. Figure 3.5 show this data file.

The program will generate $N = 1.5$ million points (u_i, u_{i+1}), with overlapping, and show those whose first coordinate is between 0 and 0.0005. The output will be in the LaTeX file `excel.tex`. Figure 3.6 shows the scatter plot created by LaTeX.

The values produced by this generator obey the linear recurrence $u_i = (9821.0\, u_{i-1} + 0.211327) \bmod 1$, where the numbers u_i are represented with 15 decimal digits of precision. This linear relationship shows up very well in the graph: All the points are on equidistant parallel lines with slope 9821. This is obviously a bad generator.

```
#include <testu01/unif01.h>
#include <testu01/ufile.h>
#include <testu01/scatter.h>

int main (void)
{
   unif01_Gen *gen;
   gen = ufile_CreateReadText ("excel.pts", 100000);
   scatter_PlotUnif (gen, "excel");
   ufile_DeleteReadText (gen);
   return 0;
}
```

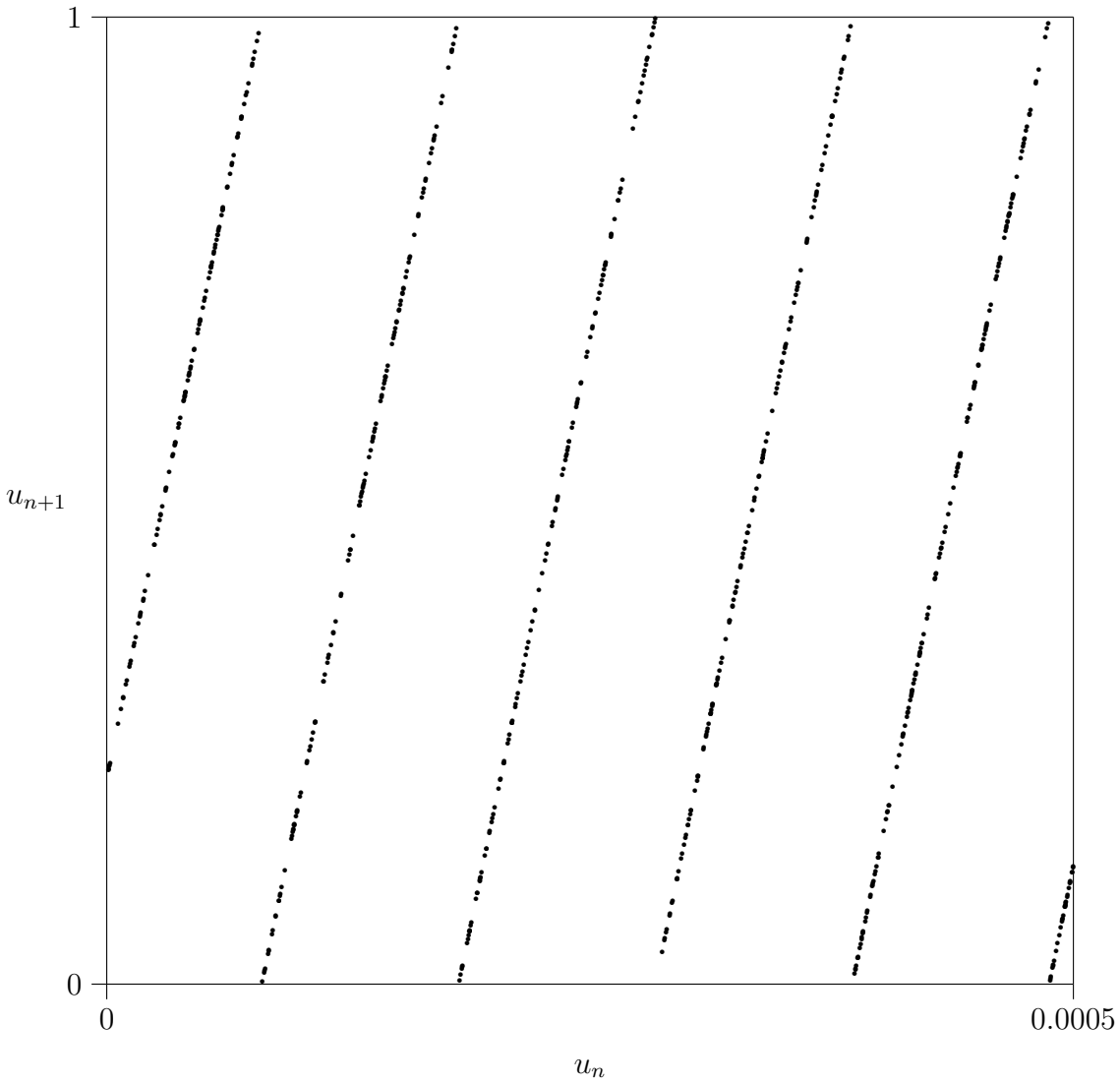
Figure 3.4: Example of a program to create a scatter plot.

```
1500000                      Number of points
2                            Dimension = t
TRUE                         Overlapping
1  2                         Components shown
1  0.0  0.0005               Component and bounds on x1
2  0.0  1.0                  Component and bounds on xt
13.0  13.0                   Size of plot in centimeters
latex                        Output format: latex, gnu_term or gnu_ps
8                            Precision
FALSE                        Lacunary
1                            Lacunary indices
3
```

Figure 3.5: The data file `excel.dat`.

Figure 3.7 shows another example using the random number generator in Microsoft VisualBasic. The example shows how to plot a scatter diagram without reading any data file. Here, the parameters are passed directly to the function `scatter_PlotUnif1`. We use the hypercube in 3 dimensions, and the plotting procedure is called for coordinates x_1 and x_3. Since the Lacunary flag is TRUE, the procedure will not use every random number generated, but will use only those selected by the lacunary indices $\{1, 2, 6\}$ (see the documentation in `unif01_CreateLacGen`). The results will be written in file `bone.tex`.

```
#include <testu01/gdef.h>
#include <testu01/unif01.h>
```



Generator: ufile_CreateReadText: excel.pts
Hypercube in 2 dimensions.
Over = TRUE
$0 < u_n < 0.0005$
Number of vectors generated: 1500000.
Number of points plotted: 711.
Total CPU time : 5.98 seconds.

Figure 3.6: Scatter plot for the Microsoft Excel 97 generator

```
#include <testu01/unif01.h>
#include <testu01/usoft.h>
#include <testu01/scatter.h>

int Proj[] = { 1, 3 };
long LacI[] = { 1, 2, 6};
double Lower[] = { 0.0, 0.0, 0.0 };
double Upper[] = { 0.0001, 0.5, 1.0 };

int main (void)
{
   unif01_Gen *gen;

   gen = usoft_CreateVisualBasic (12345);
   scatter_PlotUnif1 (gen, 10000000, 3, FALSE, Proj, Lower, Upper,
        scatter_latex, 8, TRUE, LacI, "bone");
   usoft_DeleteGen (gen);
   return 0;
}
```

Figure 3.7: Example, with lacunary indices, for creating a scatter plot.

Types

```
typedef enum {
scatter_latex,                  /* Latex format */
scatter_gnu_ps,                 /* gnuplot format for Postscript file */
scatter_gnu_term                /* Interactive gnuplot format */
} scatter_OutputType;
```

Possible formats for the output files containing the plots.

Constant

```
#define scatter_MAXDIM 64
```

Maximal number of dimensions.

The plotting functions

```
void scatter_PlotUnif (unif01_Gen *gen, char *F);
```

Creates a scatter plot, using the generator `gen` and the parameters $N, t, \ldots$ given in file `F.dat`, in the format specified in Figure 3.3. (The data file must have the extension `.dat`, but the argument `F` must be the file name without the extension.) The results are written in file `F.tex` or `F.gnu`, depending on the value of the field `scatter_Output` in the data file. For example, the instruction `scatter_PlotUnif (gen, "dice")` will read the data in file `dice.dat` and plot the figure using the parameters in this file.

```
void scatter_PlotUnif1 (unif01_Gen *gen, long N, int t, lebool Over,
int Proj[2], double Lower[], double Upper[], scatter_OutputType Output,
int Prec, lebool Lac, long LacI[], char *Name);
```

Similar to `scatter_PlotUnif`, except that the data are passed as arguments to the function instead of being read in a file. Here, `N` is the number of points to generate, `t` is the dimension, `Proj[0..1]` are the two values of the coordinates to be projected, `Lower[0..(t-1)]` gives the lower bounds of the values to be considered, `Upper[0..(t-1)]` gives the upper bounds of the values to be considered, `Over` is `TRUE` iff the coordinates of the points overlap, `Lac` is `TRUE` iff we consider lacunary values of the generator, `LacI[0..(t-1)]` gives the `t` lacunary indices, `Name` is the name (without extension) of the output file, and `Prec` is the number of decimal digits required for each written value. The constraints on these values are as explained earlier.

```
void scatter_PlotUnifInterac (unif01_Gen *gen);
```

Similar to `scatter_PlotUnif`, except that the data are given interactively on the terminal.

Chapter 4

BATTERIES OF TESTS

This chapter describes predefined batteries of tests available in TestU01. Some batteries are fast and small, and may be used as a first step in detecting gross defects in generators or errors in their implementation. Other batteries are more stringent and take longer to run. Special batteries are also available to test a stream of random bits taken from a file.

An example: The battery SmallCrush applied to a generator.

Figure 4.1 shows how to apply a battery of tests to a generator. The function call to `ulcg_CreateLCG` creates and initializes the generator `gen` to the linear congruential generator (LCG) with modulus $m = 2147483647$, multiplier $a = 16807$, additive constant $c = 0$, and initial state $x_0 = 12345$. Then the small battery `SmallCrush`, defined in module `bbattery`, is applied on this generator. Figure 4.2 shows a summary report of the results (assuming that 64-bits integers are available; otherwise, the results could be slightly different). Out of the 15 tests applied, the generator failed three with a p-value practically equal to 0, so it is clear that it failed this battery. It took 20.3 seconds to run this battery on a machine with a 2400 MHz Athlon processor running under Linux.

Another example: The battery Rabbit applied to a binary file.

Figure 4.3 shows how to apply the battery `Rabbit` to a binary file (presumably, a file of random bits). The tests will use at most 1048576 ($= 2^{20}$) bits from the binary file named `vax.bin`. (Incidentally, these bits were obtained by taking the 32 most significant bits from each uniform number generated by the well-known LCG with parameters $m = 2^{32}$, $a = 69069$ and $c = 1$. This was the random number generator used under VAX/VMS.) Since the variable `swrite_Basic` is set to `FALSE`, no detailed output is written, only the summary report shown in Figure 4.4 is printed after running the tests. Seven tests were failed with a p-value practically equal to 0 or 1. It is clear that the null hypothesis $\mathcal{H}_0$ must be rejected for this bit stream. It took 1.9 seconds to run the entire battery on a machine with a 2400 MHz Athlon processor running under Linux.

```
#include <testu01/ulcg.h>
#include <testu01/unif01.h>
#include <testu01/bbattery.h>

int main (void)
{
   unif01_Gen *gen;
   gen = ulcg_CreateLCG (2147483647, 16807, 0, 12345);
   bbattery_SmallCrush (gen);
   ulcg_DeleteGen (gen);
   return 0;
}
```

Figure 4.1: Applying the battery SmallCrush on a LCG generator.

```
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 Generator:        ulcg_CreateLCG
 Number of statistics:  15
 Total CPU time:   00:00:19.35
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                  eps
  2  Collision                         eps
  6  MaxOft                            eps
 ----------------------------------------------
 All other tests were passed
```

Figure 4.2: Results of applying SmallCrush.

```
#include <testu01/gdef.h>
#include <testu01/swrite.h>
#include <testu01/bbattery.h>

int main (void)
{
   swrite_Basic = FALSE;
   bbattery_RabbitFile ("vax.bin", 1048576);
   return 0;
}
```

Figure 4.3: Applying the battery Rabbit on a file of random bits.

```
========= Summary results of Rabbit =========

 Version:          TestU01 1.2.3
 File:             vax.bin
 Number of bits:   1048576
 Number of statistics:  38
 Total CPU time:   00:00:01.75
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  4  AppearanceSpacings              1.1e-4
  7  Fourier1                         eps
  8  Fourier3                       3.2e-213
 13  HammingCorr, L = 64             1 - eps1
 16  HammingIndep, L = 32             eps
 17  HammingIndep, L = 64             eps
 24  RandomWalk1 M                    eps
 24  RandomWalk1 J                    eps
 ----------------------------------------------
 All other tests were passed
```

Figure 4.4: Results of applying Rabbit.

bbattery

This module contains predefined batteries of statistical tests for sources of random bits or sequences of uniform random numbers in the interval $[0, 1)$. To test a RNG for general use, one could first apply the small and fast battery `SmallCrush`. If it passes, one could then apply the more stringent battery `Crush`, and finally the yet more time-consuming battery `BigCrush`. The batteries `Alphabit` and `Rabbit` can be applied on a binary file considered as a source of random bits. They can also be applied on a programmed generator. `Alphabit` has been defined primarily to test *hardware* random bits generators. The battery PseudoDIEHARD applies most of the tests in the well-known *DIEHARD* suite of Marsaglia [106]. The battery `FIPS_140_2` implements the small suite of tests of the *FIPS-140-2* standard from NIST.

The batteries described in this module will write the results of each test (on standard output) with a standard level of details (assuming that the boolean switches of module `swrite` have their default values), followed by a summary report of the suspect p-values obtained from the specific tests included in the batteries. It is also possible to get only the summary report in the output, with no detailed output from the tests, by setting the boolean switch `swrite_Basic` to `FALSE`.

Some of the tests compute more than one statistic using the same stream of random numbers and these statistics are thus not independent. *That is why the number of statistics in the summary reports is larger than the number of tests in the description of the batteries.*

```
#include <testu01/unif01.h>
```

```
extern int bbattery_NTests;
```

The maximum number of p-values in the array `bbattery_pVal`. For small sample size, some of the tests in the battery may not be done. Furthermore, some of the tests computes more than one statistic and its p-value, so `bbattery_NTests` will usually be larger than the number of tests in the battery.

```
extern double bbattery_pVal[];
```

This array keeps the p-values resulting from the battery of tests that is currently applied (or the last one that has been called). It is used by any battery in this module. The p-value of the j-th test in the battery is kept in `bbattery_pVal[`$j-1$`]`, for $1 \leq j \leq$ `bbattery_NTests`.

```
extern char *bbattery_TestNames[];
```

This array keeps the names of each test from the battery that is currently applied (or the last one that has been called). It is used by any battery in this module. The name of the j-th test in the battery is kept in `bbattery_TestNames[`$j-1$`]`, for $1 \leq j \leq$ `bbattery_NTests`.

```
void bbattery_SmallCrush (unif01_Gen *gen);

void bbattery_SmallCrushFile (char *filename);
```

Both functions applies `SmallCrush`, a small and fast battery of tests, to a RNG. The function `bbattery_SmallCrushFile` applies `SmallCrush` to a RNG given as a text file of floating-point numbers in $[0, 1)$; the file requires slightly less than 51320000 random numbers. The file will be rewound to the beginning before each test. Thus `bbattery_SmallCrush` applies the tests on one unbroken stream of successive numbers, while `bbattery_SmallCrushFile` applies each test on the same sequence of numbers. Some of these tests assume that `gen` returns at least 30 bits of resolution; if this is not the case, then the generator is most likely to fail these particular tests.

The following tests are applied:

1. `smarsa_BirthdaySpacings` with $N = 1$, $n = 5 * 10^6$, $r = 0$, $d = 2^{30}$, $t = 2$, $p = 1$.
2. `sknuth_Collision` with $N = 1$, $n = 5 * 10^6$, $r = 0$, $d = 2^{16}$, $t = 2$.
3. `sknuth_Gap` with $N = 1$, $n = 2 * 10^5$, $r = 22$, `Alpha` $= 0$, `Beta` $= 1/256$.
4. `sknuth_SimpPoker` with $N = 1$, $n = 4 * 10^5$, $r = 24$, $d = 64$, $k = 64$.
5. `sknuth_CouponCollector` with $N = 1$, $n = 5 * 10^5$, $r = 26$, $d = 16$.
6. `sknuth_MaxOft` with $N = 1$, $n = 2 * 10^6$, $r = 0$, $d = 10^5$, $t = 6$.
7. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^5$, $r = 27$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/8$.
8. `smarsa_MatrixRank` with $N = 1$, $n = 20000$, $r = 20$, $s = 10$, $L = k = 60$.
9. `sstring_HammingIndep` with $N = 1$, $n = 5 * 10^5$, $r = 20$, $s = 10$, $L = 300$, $d = 0$.
10. `swalk_RandomWalk1` with $N = 1$, $n = 10^6$, $r = 0$, $s = 30$, $L_0 = 150$, $L_1 = 150$.

```
void bbattery_RepeatSmallCrush (unif01_Gen *gen, int rep[]);
```

This function applies specific tests of `SmallCrush` on generator `gen`. Test numbered i in the enumeration above will be applied `rep[`i`]` times successively on `gen`. Those tests with `rep[`i`]` $= 0$ will not be applied. This is useful when a test in `SmallCrush` had a suspect p-value, and one wants to reapply the test a few more times to find out whether the generator failed the test or whether the suspect p-value was a statistical fluke. Restriction: Array `rep` must have one more element than the number of tests in `SmallCrush`.

`void bbattery_Crush (unif01_Gen *gen);`

Applies the battery `Crush`, a suite of stringent statistical tests, to the generator `gen`. The battery includes the classical tests described in Knuth [66] as well as many other tests. Some of these tests assume that `gen` returns at least 30 bits of resolution; if that is not the case, then the generator will certainly fail these particular tests. One test requires 31 bits of resolution: the `BirthdaySpacings` test with $t = 2$. On a PC with an AMD Athlon 64 Processor 4000+ of clock speed 2400 MHz running with Red Hat Linux, Crush will require around 1 hour of CPU time. `Crush` uses approximately 2^{35} random numbers. The following tests are applied:

1. `smarsa_SerialOver` with $N = 1$, $n = 5 * 10^8$, $r = 0$, $d = 2^{12}$, $t = 2$.
2. `smarsa_SerialOver` with $N = 1$, $n = 3 * 10^8$, $r = 0$, $d = 2^6$, $t = 4$.
3. `smarsa_CollisionOver` with $N = 10$, $n = 10^7$, $r = 0$, $d = 2^{20}$, $t = 2$.
4. `smarsa_CollisionOver` with $N = 10$, $n = 10^7$, $r = 10$, $d = 2^{20}$, $t = 2$.
5. `smarsa_CollisionOver` with $N = 10$, $n = 10^7$, $r = 0$, $d = 2^{10}$, $t = 4$.
6. `smarsa_CollisionOver` with $N = 10$, $n = 10^7$, $r = 20$, $d = 2^{10}$, $t = 4$.
7. `smarsa_CollisionOver` with $N = 10$, $n = 10^7$, $r = 0$, $d = 32$, $t = 8$.
8. `smarsa_CollisionOver` with $N = 10$, $n = 10^7$, $r = 25$, $d = 32$, $t = 8$.
9. `smarsa_CollisionOver` with $N = 10$, $n = 10^7$, $r = 0$, $d = 4$, $t = 20$.
10. `smarsa_CollisionOver` with $N = 10$, $n = 10^7$, $r = 28$, $d = 4$, $t = 20$.
11. `smarsa_BirthdaySpacings` with $N = 5$, $n = 2 * 10^7$, $r = 0$, $d = 2^{31}$, $t = 2$, $p = 1$.
12. `smarsa_BirthdaySpacings` with $N = 5$, $n = 2 * 10^7$, $r = 0$, $d = 2^{21}$, $t = 3$, $p = 1$.
13. `smarsa_BirthdaySpacings` with $N = 5$, $n = 2 * 10^7$, $r = 0$, $d = 2^{16}$, $t = 4$, $p = 1$.
14. `smarsa_BirthdaySpacings` with $N = 3$, $n = 2 * 10^7$, $r = 0$, $d = 2^9$, $t = 7$, $p = 1$.
15. `smarsa_BirthdaySpacings` with $N = 3$, $n = 2 * 10^7$, $r = 7$, $d = 2^9$, $t = 7$, $p = 1$.
16. `smarsa_BirthdaySpacings` with $N = 3$, $n = 2 * 10^7$, $r = 14$, $d = 2^8$, $t = 8$, $p = 1$.
17. `smarsa_BirthdaySpacings` with $N = 3$, $n = 2 * 10^7$, $r = 22$, $d = 2^8$, $t = 8$, $p = 1$.
18. `snpair_ClosePairs` with $N = 10$, $n = 2 * 10^6$, $r = 0$, $t = 2$, $p = 0$, $m = 30$.
19. `snpair_ClosePairs` with $N = 10$, $n = 2 * 10^6$, $r = 0$, $t = 3$, $p = 0$, $m = 30$.
20. `snpair_ClosePairs` with $N = 5$, $n = 2 * 10^6$, $r = 0$, $t = 7$, $p = 0$, $m = 30$.

21. `snpair_ClosePairsBitMatch` with $N = 4$, $n = 4 * 10^6$, $r = 0$, $t = 2$.
22. `snpair_ClosePairsBitMatch` with $N = 2$, $n = 4 * 10^6$, $r = 0$, $t = 4$.
23. `sknuth_SimpPoker` with $N = 1$, $n = 4 * 10^7$, $r = 0$, $d = 16$, $k = 16$.
24. `sknuth_SimpPoker` with $N = 1$, $n = 4 * 10^7$, $r = 26$, $d = 16$, $k = 16$.
25. `sknuth_SimpPoker` with $N = 1$, $n = 10^7$, $r = 0$, $d = 64$, $k = 64$.
26. `sknuth_SimpPoker` with $N = 1$, $n = 10^7$, $r = 24$, $d = 64$, $k = 64$.
27. `sknuth_CouponCollector` with $N = 1$, $n = 4 * 10^7$, $r = 0$, $d = 4$.
28. `sknuth_CouponCollector` with $N = 1$, $n = 4 * 10^7$, $r = 28$, $d = 4$.
29. `sknuth_CouponCollector` with $N = 1$, $n = 10^7$, $r = 0$, $d = 16$.
30. `sknuth_CouponCollector` with $N = 1$, $n = 10^7$, $r = 26$, $d = 16$.
31. `sknuth_Gap` with $N = 1$, $n = 10^8$, $r = 0$, `Alpha` $= 0$, `Beta` $= 1/8$.
32. `sknuth_Gap` with $N = 1$, $n = 10^8$, $r = 27$, `Alpha` $= 0$, `Beta` $= 1/8$.
33. `sknuth_Gap` with $N = 1$, $n = 5 * 10^6$, $r = 0$, `Alpha` $= 0$, `Beta` $= 1/256$.
34. `sknuth_Gap` with $N = 1$, $n = 5 * 10^6$, $r = 22$, `Alpha` $= 0$, `Beta` $= 1/256$.
35. `sknuth_Run` with $N = 1$, $n = 5 * 10^8$, $r = 0$, `Up = TRUE`.
36. `sknuth_Run` with $N = 1$, $n = 5 * 10^8$, $r = 15$, `Up = FALSE`.
37. `sknuth_Permutation` with $N = 1$, $n = 5 * 10^7$, $r = 0$, $t = 10$.
38. `sknuth_Permutation` with $N = 1$, $n = 5 * 10^7$, $r = 15$, $t = 10$.
39. `sknuth_CollisionPermut` with $N = 5$, $n = 10^7$, $r = 0$, $t = 13$.
40. `sknuth_CollisionPermut` with $N = 5$, $n = 10^7$, $r = 15$, $t = 13$.
41. `sknuth_MaxOft` with $N = 10$, $n = 10^7$, $r = 0$, $d = 10^5$, $t = 5$.
42. `sknuth_MaxOft` with $N = 5$, $n = 10^7$, $r = 0$, $d = 10^5$, $t = 10$.
43. `sknuth_MaxOft` with $N = 1$, $n = 10^7$, $r = 0$, $d = 10^5$, $t = 20$.
44. `sknuth_MaxOft` with $N = 1$, $n = 10^7$, $r = 0$, $d = 10^5$, $t = 30$.
45. `svaria_SampleProd` with $N = 1$, $n = 10^7$, $r = 0$, $t = 10$.
46. `svaria_SampleProd` with $N = 1$, $n = 10^7$, $r = 0$, $t = 30$.
47. `svaria_SampleMean` with $N = 10^7$, $n = 20$, $r = 0$.

48. `svaria_SampleCorr` with $N = 1$, $n = 5 * 10^8$, $r = 0$, $k = 1$.

49. `svaria_AppearanceSpacings` with $N = 1$, $Q = 10^7$, $K = 4 * 10^8$, $r = 0$, $s = 30$, $L = 15$.

50. `svaria_AppearanceSpacings` with $N = 1$, $Q = 10^7$, $K = 10^8$, $r = 20$, $s = 10$, $L = 15$.

51. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^6$, $r = 0$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/8$.

52. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^6$, $r = 8$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/8$.

53. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^6$, $r = 16$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/8$.

54. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^6$, $r = 24$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/8$.

55. `svaria_SumCollector` with $N = 1$, $n = 2 * 10^7$, $r = 0$, $g = 10$.

56. `smarsa_MatrixRank` with $N = 1$, $n = 10^6$, $r = 0$, $s = 30$, $L = k = 60$.

57. `smarsa_MatrixRank` with $N = 1$, $n = 10^6$, $r = 20$, $s = 10$, $L = k = 60$.

58. `smarsa_MatrixRank` with $N = 1$, $n = 50000$, $r = 0$, $s = 30$, $L = k = 300$.

59. `smarsa_MatrixRank` with $N = 1$, $n = 50000$, $r = 20$, $s = 10$, $L = k = 300$.

60. `smarsa_MatrixRank` with $N = 1$, $n = 2000$, $r = 0$, $s = 30$, $L = k = 1200$.

61. `smarsa_MatrixRank` with $N = 1$, $n = 2000$, $r = 20$, $s = 10$, $L = k = 1200$.

62. `smarsa_Savir2` with $N = 1$, $n = 2 * 10^7$, $r = 0$, $m = 2^{20}$, $t = 30$.

63. `smarsa_GCD` with $N = 1$, $n = 10^8$, $r = 0$, $s = 30$.

64. `smarsa_GCD` with $N = 1$, $n = 4 * 10^7$, $r = 10$, $s = 20$.

65. `swalk_RandomWalk1` with $N = 1$, $n = 5 * 10^7$, $r = 0$, $s = 30$, $L_0 = L_1 = 90$.

66. `swalk_RandomWalk1` with $N = 1$, $n = 10^7$, $r = 20$, $s = 10$, $L_0 = L_1 = 90$.

67. `swalk_RandomWalk1` with $N = 1$, $n = 5 * 10^6$, $r = 0$, $s = 30$, $L_0 = L_1 = 1000$.

68. `swalk_RandomWalk1` with $N = 1$, $n = 10^6$, $r = 20$, $s = 10$, $L_0 = L_1 = 1000$.

69. `swalk_RandomWalk1` with $N = 1$, $n = 5 * 10^5$, $r = 0$, $s = 30$, $L_0 = L_1 = 10000$.

70. `swalk_RandomWalk1` with $N = 1$, $n = 10^5$, $r = 20$, $s = 10$, $L_0 = L_1 = 10000$.

71. `scomp_LinearComp` with $N = 1$, $n = 120000$, $r = 0$, $s = 1$.

72. `scomp_LinearComp` with $N = 1$, $n = 120000$, $r = 29$, $s = 1$.
73. `scomp_LempelZiv` with $N = 10$, $k = 25$, $r = 0$, $s = 30$.
74. `sspectral_Fourier3` with $N = 50000$, $k = 14$, $r = 0$, $s = 30$.
75. `sspectral_Fourier3` with $N = 50000$, $k = 14$, $r = 20$, $s = 10$.
76. `sstring_LongestHeadRun` with $N = 1$, $n = 1000$, $r = 0$, $s = 30$, $L = 10^7$.
77. `sstring_LongestHeadRun` with $N = 1$, $n = 300$, $r = 20$, $s = 10$, $L = 10^7$.
78. `sstring_PeriodsInStrings` with $N = 1$, $n = 3 * 10^8$, $r = 0$, $s = 30$.
79. `sstring_PeriodsInStrings` with $N = 1$, $n = 3 * 10^8$, $r = 15$, $s = 15$.
80. `sstring_HammingWeight2` with $N = 100$, $n = 10^8$, $r = 0$, $s = 30$, $L = 10^6$.
81. `sstring_HammingWeight2` with $N = 30$, $n = 10^8$, $r = 20$, $s = 10$, $L = 10^6$.
82. `sstring_HammingCorr` with $N = 1$, $n = 5 * 10^8$, $r = 0$, $s = 30$, $L = 30$.
83. `sstring_HammingCorr` with $N = 1$, $n = 5 * 10^7$, $r = 0$, $s = 30$, $L = 300$.
84. `sstring_HammingCorr` with $N = 1$, $n = 10^7$, $r = 0$, $s = 30$, $L = 1200$.
85. `sstring_HammingIndep` with $N = 1$, $n = 3 * 10^8$, $r = 0$, $s = 30$, $L = 30$, $d = 0$.
86. `sstring_HammingIndep` with $N = 1$, $n = 10^8$, $r = 20$, $s = 10$, $L = 30$, $d = 0$.
87. `sstring_HammingIndep` with $N = 1$, $n = 3 * 10^7$, $r = 0$, $s = 30$, $L = 300$, $d = 0$.
88. `sstring_HammingIndep` with $N = 1$, $n = 10^7$, $r = 20$, $s = 10$, $L = 300$, $d = 0$.
89. `sstring_HammingIndep` with $N = 1$, $n = 10^7$, $r = 0$, $s = 30$, $L = 1200$, $d = 0$.
90. `sstring_HammingIndep` with $N = 1$, $n = 10^6$, $r = 20$, $s = 10$, $L = 1200$, $d = 0$.
91. `sstring_Run` with $N = 1$, $n = 10^9$, $r = 0$, $s = 30$.
92. `sstring_Run` with $N = 1$, $n = 10^9$, $r = 20$, $s = 10$.
93. `sstring_AutoCor` with $N = 10$, $n = 10^9$, $r = 0$, $s = 30$, $d = 1$.
94. `sstring_AutoCor` with $N = 5$, $n = 10^9$, $r = 20$, $s = 10$, $d = 1$.
95. `sstring_AutoCor` with $N = 10$, $n = 10^9$, $r = 0$, $s = 30$, $d = 30$.
96. `sstring_AutoCor` with $N = 5$, $n = 10^9$, $r = 20$, $s = 10$, $d = 10$.

`void bbattery_RepeatCrush (unif01_Gen *gen, int rep[]);`

Similar to `bbattery_RepeatSmallCrush` above but applied on `Crush`.

`void bbattery_BigCrush (unif01_Gen *gen);`

Applies the battery `BigCrush`, a suite of very stringent statistical tests, to the generator `gen`. Some of these tests assume that `gen` returns at least 30 bits of resolution; if that is not the case, then the generator will certainly fail these particular tests. One test requires 31 bits of resolution: the `BirthdaySpacings` test with $t = 2$. On a PC with an AMD Athlon 64 Processor 4000+ of clock speed 2400 MHz running with Linux, BigCrush will take around 8 hours of CPU time. `BigCrush` uses close to 2^{38} random numbers. The following tests are applied:

1. `smarsa_SerialOver` with $N = 1$, $n = 10^9$, $r = 0$, $d = 2^8$, $t = 3$.
2. `smarsa_SerialOver` with $N = 1$, $n = 10^9$, $r = 22$, $d = 2^8$, $t = 3$.
3. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 0$, $d = 2^{21}$, $t = 2$.
4. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 9$, $d = 2^{21}$, $t = 2$.
5. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 0$, $d = 2^{14}$, $t = 3$.
6. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 16$, $d = 2^{14}$, $t = 3$.
7. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 0$, $d = 64$, $t = 7$.
8. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 24$, $d = 64$, $t = 7$.
9. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 0$, $d = 8$, $t = 14$.
10. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 27$, $d = 8$, $t = 14$.
11. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 0$, $d = 4$, $t = 21$.
12. `smarsa_CollisionOver` with $N = 30$, $n = 2 * 10^7$, $r = 28$, $d = 4$, $t = 21$.
13. `smarsa_BirthdaySpacings` with $N = 100$, $n = 10^7$, $r = 0$, $d = 2^{31}$, $t = 2$, $p = 1$.
14. `smarsa_BirthdaySpacings` with $N = 20$, $n = 2 * 10^7$, $r = 0$, $d = 2^{21}$, $t = 3$, $p = 1$.
15. `smarsa_BirthdaySpacings` with $N = 20$, $n = 3 * 10^7$, $r = 14$, $d = 2^{16}$, $t = 4$, $p = 1$.
16. `smarsa_BirthdaySpacings` with $N = 20$, $n = 2 * 10^7$, $r = 0$, $d = 2^9$, $t = 7$, $p = 1$.
17. `smarsa_BirthdaySpacings` with $N = 20$, $n = 2 * 10^7$, $r = 7$, $d = 2^9$, $t = 7$, $p = 1$.
18. `smarsa_BirthdaySpacings` with $N = 20$, $n = 3 * 10^7$, $r = 14$, $d = 2^8$, $t = 8$, $p = 1$.

19. `smarsa_BirthdaySpacings` with $N = 20$, $n = 3 * 10^7$, $r = 22$, $d = 2^8$, $t = 8$, $p = 1$.
20. `smarsa_BirthdaySpacings` with $N = 20$, $n = 3 * 10^7$, $r = 0$, $d = 2^4$, $t = 16$, $p = 1$.
21. `smarsa_BirthdaySpacings` with $N = 20$, $n = 3 * 10^7$, $r = 26$, $d = 2^4$, $t = 16$, $p = 1$.
22. `snpair_ClosePairs` with $N = 30$, $n = 6 * 10^6$, $r = 0$, $t = 3$, $p = 0$, $m = 30$.
23. `snpair_ClosePairs` with $N = 20$, $n = 4 * 10^6$, $r = 0$, $t = 5$, $p = 0$, $m = 30$.
24. `snpair_ClosePairs` with $N = 10$, $n = 3 * 10^6$, $r = 0$, $t = 9$, $p = 0$, $m = 30$.
25. `snpair_ClosePairs` with $N = 5$, $n = 2 * 10^6$, $r = 0$, $t = 16$, $p = 0$, $m = 30$.
26. `sknuth_SimpPoker` with $N = 1$, $n = 4 * 10^8$, $r = 0$, $d = 8$, $k = 8$.
27. `sknuth_SimpPoker` with $N = 1$, $n = 4 * 10^8$, $r = 27$, $d = 8$, $k = 8$.
28. `sknuth_SimpPoker` with $N = 1$, $n = 10^8$, $r = 0$, $d = 32$, $k = 32$.
29. `sknuth_SimpPoker` with $N = 1$, $n = 10^8$, $r = 25$, $d = 32$, $k = 32$.
30. `sknuth_CouponCollector` with $N = 1$, $n = 2 * 10^8$, $r = 0$, $d = 8$.
31. `sknuth_CouponCollector` with $N = 1$, $n = 2 * 10^8$, $r = 10$, $d = 8$.
32. `sknuth_CouponCollector` with $N = 1$, $n = 2 * 10^8$, $r = 20$, $d = 8$.
33. `sknuth_CouponCollector` with $N = 1$, $n = 2 * 10^8$, $r = 27$, $d = 8$.
34. `sknuth_Gap` with $N = 1$, $n = 5 * 10^8$, $r = 0$, `Alpha` $= 0$, `Beta` $= 1/16$.
35. `sknuth_Gap` with $N = 1$, $n = 3 * 10^8$, $r = 25$, `Alpha` $= 0$, `Beta` $= 1/32$.
36. `sknuth_Gap` with $N = 1$, $n = 10^8$, $r = 0$, `Alpha` $= 0$, `Beta` $= 1/128$.
37. `sknuth_Gap` with $N = 1$, $n = 10^7$, $r = 20$, `Alpha` $= 0$, `Beta` $= 1/1024$.
38. `sknuth_Run` with $N = 5$, $n = 10^9$, $r = 0$, `Up = FALSE`.
39. `sknuth_Run` with $N = 5$, $n = 10^9$, $r = 15$, `Up = TRUE`.
40. `sknuth_Permutation` with $N = 1$, $n = 10^9$, $r = 0$, $t = 3$.
41. `sknuth_Permutation` with $N = 1$, $n = 10^9$, $r = 0$, $t = 5$.
42. `sknuth_Permutation` with $N = 1$, $n = 5 * 10^8$, $r = 0$, $t = 7$.
43. `sknuth_Permutation` with $N = 1$, $n = 5 * 10^8$, $r = 10$, $t = 10$.
44. `sknuth_CollisionPermut` with $N = 20$, $n = 2 * 10^7$, $r = 0$, $t = 14$.
45. `sknuth_CollisionPermut` with $N = 20$, $n = 2 * 10^7$, $r = 10$, $t = 14$.

46. `sknuth_MaxOft` with $N = 40$, $n = 10^7$, $r = 0$, $d = 10^5$, $t = 8$.

47. `sknuth_MaxOft` with $N = 30$, $n = 10^7$, $r = 0$, $d = 10^5$, $t = 16$.

48. `sknuth_MaxOft` with $N = 20$, $n = 10^7$, $r = 0$, $d = 10^5$, $t = 24$.

49. `sknuth_MaxOft` with $N = 20$, $n = 10^7$, $r = 0$, $d = 10^5$, $t = 32$.

50. `svaria_SampleProd` with $N = 40$, $n = 10^7$, $r = 0$, $t = 8$.

51. `svaria_SampleProd` with $N = 20$, $n = 10^7$, $r = 0$, $t = 16$.

52. `svaria_SampleProd` with $N = 20$, $n = 10^7$, $r = 0$, $t = 24$.

53. `svaria_SampleMean` with $N = 2 * 10^7$, $n = 30$, $r = 0$.

54. `svaria_SampleMean` with $N = 2 * 10^7$, $n = 30$, $r = 10$.

55. `svaria_SampleCorr` with $N = 1$, $n = 2 * 10^9$, $r = 0$, $k = 1$.

56. `svaria_SampleCorr` with $N = 1$, $n = 2 * 10^9$, $r = 0$, $k = 2$.

57. `svaria_AppearanceSpacings` with $N = 1$, $Q = 10^7$, $K = 10^9$, $r = 0$, $s = 3$, $L = 15$.

58. `svaria_AppearanceSpacings` with $N = 1$, $Q = 10^7$, $K = 10^9$, $r = 27$, $s = 3$, $L = 15$.

59. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^7$, $r = 0$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/4$.

60. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^7$, $r = 20$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/4$.

61. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^7$, $r = 28$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/4$.

62. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^7$, $r = 0$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/16$.

63. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^7$, $r = 10$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/16$.

64. `svaria_WeightDistrib` with $N = 1$, $n = 2 * 10^7$, $r = 26$, $k = 256$, `Alpha` $= 0$, `Beta` $= 1/16$.

65. `svaria_SumCollector` with $N = 1$, $n = 5 * 10^8$, $r = 0$, $g = 10$.

66. `smarsa_MatrixRank` with $N = 10$, $n = 10^6$, $r = 0$, $s = 5$, $L = k = 30$.

67. `smarsa_MatrixRank` with $N = 10$, $n = 10^6$, $r = 25$, $s = 5$, $L = k = 30$.

68. `smarsa_MatrixRank` with $N = 1$, $n = 5000$, $r = 0$, $s = 4$, $L = k = 1000$.

69. `smarsa_MatrixRank` with $N = 1$, $n = 5000$, $r = 26$, $s = 4$, $L = k = 1000$.
70. `smarsa_MatrixRank` with $N = 1$, $n = 80$, $r = 15$, $s = 15$, $L = k = 5000$.
71. `smarsa_MatrixRank` with $N = 1$, $n = 80$, $r = 0$, $s = 30$, $L = k = 5000$.
72. `smarsa_Savir2` with $N = 10$, $n = 10^7$, $r = 10$, $m = 2^{20}$, $t = 30$.
73. `smarsa_GCD` with $N = 10$, $n = 5 * 10^7$, $r = 0$, $s = 30$.
74. `swalk_RandomWalk1` with $N = 1$, $n = 10^8$, $r = 0$, $s = 5$, $L_0 = L_1 = 50$.
75. `swalk_RandomWalk1` with $N = 1$, $n = 10^8$, $r = 25$, $s = 5$, $L_0 = L_1 = 50$.
76. `swalk_RandomWalk1` with $N = 1$, $n = 10^7$, $r = 0$, $s = 10$, $L_0 = L_1 = 1000$.
77. `swalk_RandomWalk1` with $N = 1$, $n = 10^7$, $r = 20$, $s = 10$, $L_0 = L_1 = 1000$.
78. `swalk_RandomWalk1` with $N = 1$, $n = 10^6$, $r = 0$, $s = 15$, $L_0 = L_1 = 10000$.
79. `swalk_RandomWalk1` with $N = 1$, $n = 10^6$, $r = 15$, $s = 15$, $L_0 = L_1 = 10000$.
80. `scomp_LinearComp` with $N = 1$, $n = 400000$, $r = 0$, $s = 1$.
81. `scomp_LinearComp` with $N = 1$, $n = 400000$, $r = 29$, $s = 1$.
82. `scomp_LempelZiv` with $N = 10$, $k = 27$, $r = 0$, $s = 30$.
83. `scomp_LempelZiv` with $N = 10$, $k = 27$, $r = 15$, $s = 15$.
84. `sspectral_Fourier3` with $N = 100000$, $r = 0$, $s = 3$, $k = 14$.
85. `sspectral_Fourier3` with $N = 100000$, $r = 27$, $s = 3$, $k = 14$.
86. `sstring_LongestHeadRun` with $N = 1$, $n = 1000$, $r = 0$, $s = 3$, $L = 10^7$.
87. `sstring_LongestHeadRun` with $N = 1$, $n = 1000$, $r = 27$, $s = 3$, $L = 10^7$.
88. `sstring_PeriodsInStrings` with $N = 10$, $n = 5 * 10^8$, $r = 0$, $s = 10$.
89. `sstring_PeriodsInStrings` with $N = 10$, $n = 5 * 10^8$, $r = 20$, $s = 10$.
90. `sstring_HammingWeight2` with $N = 10$, $n = 10^9$, $r = 0$, $s = 3$, $L = 10^6$.
91. `sstring_HammingWeight2` with $N = 10$, $n = 10^9$, $r = 27$, $s = 3$, $L = 10^6$.
92. `sstring_HammingCorr` with $N = 1$, $n = 10^9$, $r = 10$, $s = 10$, $L = 30$.
93. `sstring_HammingCorr` with $N = 1$, $n = 10^8$, $r = 10$, $s = 10$, $L = 300$.
94. `sstring_HammingCorr` with $N = 1$, $n = 10^8$, $r = 10$, $s = 10$, $L = 1200$.
95. `sstring_HammingIndep` with $N = 10$, $n = 3 * 10^7$, $r = 0$, $s = 3$, $L = 30$, $d = 0$.

96. `sstring_HammingIndep` with $N = 10$, $n = 3 * 10^7$, $r = 27$, $s = 3$, $L = 30$, $d = 0$.

97. `sstring_HammingIndep` with $N = 1$, $n = 3 * 10^7$, $r = 0$, $s = 4$, $L = 300$, $d = 0$.

98. `sstring_HammingIndep` with $N = 1$, $n = 3 * 10^7$, $r = 26$, $s = 4$, $L = 300$, $d = 0$.

99. `sstring_HammingIndep` with $N = 1$, $n = 10^7$, $r = 0$, $s = 5$, $L = 1200$, $d = 0$.

100. `sstring_HammingIndep` with $N = 1$, $n = 10^7$, $r = 25$, $s = 5$, $L = 1200$, $d = 0$.

101. `sstring_Run` with $N = 1$, $n = 2 * 10^9$, $r = 0$, $s = 3$.

102. `sstring_Run` with $N = 1$, $n = 2 * 10^9$, $r = 27$, $s = 3$.

103. `sstring_AutoCor` with $N = 10$, $n = 10^9$, $r = 0$, $s = 3$, $d = 1$.

104. `sstring_AutoCor` with $N = 10$, $n = 10^9$, $r = 0$, $s = 3$, $d = 3$.

105. `sstring_AutoCor` with $N = 10$, $n = 10^9$, $r = 27$, $s = 3$, $d = 1$.

106. `sstring_AutoCor` with $N = 10$, $n = 10^9$, $r = 27$, $s = 3$, $d = 3$.

```
void bbattery_RepeatBigCrush (unif01_Gen *gen, int rep[]);
```

Similar to `bbattery_RepeatSmallCrush` above but applied on `BigCrush`.

```
void bbattery_Rabbit (unif01_Gen *gen, double nb);
```

Applies the `Rabbit` battery of tests to the generator `gen` using at most `nb` bits for each test. See the description of the tests in `bbattery_RabbitFile`.

```
void bbattery_RabbitFile (char *filename, double nb);
```

Applies the `Rabbit` battery of tests to the first `nb` bits (or less, if `nb` is too large) of the binary file `filename`. For each test, the file is reset and the test is applied to the bit stream starting at the beginning of the file. The bits themselves are processed in nearly all the tests as blocks of 32 bits (unsigned integers); the two exceptions are `svaria_AppearanceSpacings`, which uses blocks of 30 bits (and discards the last 2 bits out of each block of 32), and `sstring_PeriodsInStrings` which uses blocks of 31 bits (and discards 1 bit out of 32). The parameters of each test are chosen automatically as a function of `nb`, in order to make the test reasonably sensitive. On a PC with an Athlon processor of clock speed 1733 MHz running under Linux, `Rabbit` will take about 5 seconds to test a stream of 2^{20} bits, 90 seconds to test a stream of 2^{25} bits, and 28 minutes to test a stream of 2^{30} bits. Restriction: `nb` ≥ 500.

1. `smultin_MultinomialBitsOver`

2. `snpair_ClosePairsBitMatch` in $t = 2$ dimensions.

3. `snpair_ClosePairsBitMatch` in $t = 4$ dimensions.
4. `svaria_AppearanceSpacings`
5. `scomp_LinearComp`
6. `scomp_LempelZiv`
7. `sspectral_Fourier1`
8. `sspectral_Fourier3`
9. `sstring_LongestHeadRun`
10. `sstring_PeriodsInStrings`
11. `sstring_HammingWeight` with blocks of $L = 32$ bits.
12. `sstring_HammingCorr` with blocks of $L = 32$ bits.
13. `sstring_HammingCorr` with blocks of $L = 64$ bits.
14. `sstring_HammingCorr` with blocks of $L = 128$ bits.
15. `sstring_HammingIndep` with blocks of $L = 16$ bits.
16. `sstring_HammingIndep` with blocks of $L = 32$ bits.
17. `sstring_HammingIndep` with blocks of $L = 64$ bits.
18. `sstring_AutoCor` with a lag $d = 1$.
19. `sstring_AutoCor` with a lag $d = 2$.
20. `sstring_Run`
21. `smarsa_MatrixRank` with 32×32 matrices.
22. `smarsa_MatrixRank` with 320×320 matrices.
23. `smarsa_MatrixRank` with 1024×1024 matrices.
24. `swalk_RandomWalk1` with walks of length $L = 128$.
25. `swalk_RandomWalk1` with walks of length $L = 1024$.
26. `swalk_RandomWalk1` with walks of length $L = 10016$.

`void bbattery_RepeatRabbit (unif01_Gen *gen, double nb, int rep[]);`

Similar to `bbattery_RepeatSmallCrush` above but applied on `Rabbit`.

`void bbattery_Alphabit (unif01_Gen *gen, double nb, int r, int s);`

Applies the `Alphabit` battery of tests to the generator `gen` using at most `nb` bits for each test. The bits themselves are processed as blocks of 32 bits (unsigned integers). For each block of 32 bits, the r most significant bits are dropped, and the test is applied on the s following bits. If one wants to test all bits of the stream, one should set $r = 0$ and $s = 32$. If one wants to test only 1 bit out of 32, one should set $s = 1$. See the description of the tests in `bbattery_AlphabitFile`.

`void bbattery_AlphabitFile (char *filename, double nb);`

Applies the `Alphabit` battery of tests to the first `nb` bits (or less, if `nb` is too large) of the binary file `filename`. Unlike the `bbattery_Alphabit` function above, for each test, the file is rewound and the test is applied to the bit stream starting at the beginning of the file. On a PC with an Athlon processor of clock speed 1733 MHz running under Linux, `Alphabit` takes about 4.2 seconds to test a file of 2^{25} bits, and 2.3 minutes to test a file of 2^{30} bits.

`Alphabit` and `AlphabitFile` have been designed primarily to test *hardware* random bits generators. The four `MultinomialBitsOver` tests should detect correlations between successive bits by applying a `SerialOver` test to overlapping blocks of 2, 4, 8 and 16 bits. The `Hamming` tests should detect correlations between the successive bits of overlapping blocks of 16 and 32 bits, and the `RandomWalk` tests consider blocks of 64 and 320 bits.

1. `smultin_MultinomialBitsOver` with $L = 2$.
2. `smultin_MultinomialBitsOver` with $L = 4$.
3. `smultin_MultinomialBitsOver` with $L = 8$.
4. `smultin_MultinomialBitsOver` with $L = 16$.
5. `sstring_HammingIndep` with blocks of $L = 16$ bits.
6. `sstring_HammingIndep` with blocks of $L = 32$ bits.
7. `sstring_HammingCorr` with blocks of $L = 32$ bits.
8. `swalk_RandomWalk1` with walks of length $L = 64$.
9. `swalk_RandomWalk1` with walks of length $L = 320$.

```
void bbattery_RepeatAlphabit (unif01_Gen *gen, double nb, int r, int s,
int rep[]);
```

Similar to `bbattery_RepeatSmallCrush` above but applied on `Alphabit`.

```
void bbattery_BlockAlphabit (unif01_Gen *gen, double nb, int r, int s);
void bbattery_BlockAlphabitFile (char *filename, double nb);
```

Apply the `Alphabit` battery of tests repeatedly to the generator `gen` or to the binary file `filename` after reordering the bits as described in the filter `unif01_CreateBitBlockGen`. `Alphabit` will be applied for the different values of $w \in \{1, 2, 4, 8, 16, 32\}$. If $s < 32$, only values of $w \leq s$ will be used. Each test uses at most `nb` bits. See the description of the tests in `bbattery_AlphabitFile`.

```
void bbattery_RepeatBlockAlphabit (unif01_Gen *gen, double nb, int r, int s,
int rep[], int w);
```

Similar to `bbattery_RepeatSmallCrush` above but applied on `BlockAlphabit`. The parameter w is the one described in `bbattery_BlockAlphabit`. Restrictions: $w \in \{1, 2, 4, 8, 16, 32\}$ and $w \leq s$.

Other Tests Suites

```
void bbattery_pseudoDIEHARD (unif01_Gen *gen);
```

Applies the battery PseudoDIEHARD, which implements most of the tests in the popular battery DIEHARD [106] or, in some cases, close approximations to them. **We do not recommend this battery as it is not very stringent** (we do not know of any generator that passes the batteries Crush and BigCrush, and fails PseudoDIEHARD, while we have seen the converse for several defective generators). It is included here only for convenience to the user. The DIEHARD tests and the corresponding tests in PseudoDIEHARD are:

1. The **Birthday Spacings** test. This corresponds to `smarsa_BirthdaySpacings` with $n = 512$, $d = 2^{24}$, $t = 1$ and $r = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ successively. The test with each value of r is repeated 500 times and a chi-square test is then applied.

2. The **Overlapping 5-Permutation** test. This test is not implemented in TestU01.

3. The **Binary Rank Tests for Matrices**. This corresponds to `smarsa_MatrixRank`.

4. The **Bitstream** test. Closely related to `smultin_MultinomialBitsOver` with `Delta` $= -1$, $n = 2^{21}$, $L = 20$.

5. The **OPSO** test. This corresponds to `smarsa_CollisionOver` with $n = 2^{21}$, $d = 1024$, $t = 2$ and all values of r from 0 to 22.

6. The **OQSO** test. This corresponds to `smarsa_CollisionOver` with $n = 2^{21}$, $d = 32$, $t = 4$ and all values of r from 0 to 27.

7. The **DNA** test. This corresponds to `smarsa_CollisionOver` with $n = 2^{21}$, $d = 4$, $t = 10$ and all values of r from 0 to 30.

8. The **Count-the-1's** test is not implemented in TestU01. It is a 5-dimensional overlapping version of `sstring_HammingIndep`.

9. The **Parking Lot** test is not implemented in TestU01.

10. The **Minimum Distance** test. Closely related to `snpair_ClosePairs` with $N = 100$, $n = 8000$, $t = 2$, $p = 2$, $m = 1$.

11. The **3-D Spheres** test. Closely related to `snpair_ClosePairs` with $N = 20$, $n = 4000$, $t = 3$, $p = 2$, $m = 1$.

12. The **Squeeze** test. Closely related to `smarsa_Savir2`.

13. The **Overlapping Sums** test is not implemented in TestU01.

14. The **Runs** test. This corresponds to `sknuth_Run`.

15. The **Craps** test is not implemented in TestU01.

The NIST test suite The NIST (National Institute of Standards and Technology) of the U.S. federal government has proposed a statistical test suite [147] for use in the evaluation of the randomness of bitstreams produced by cryptographic random number generators. The test parameters are not predetermined. The NIST tests and the equivalent tests in TestU01 are:

1. The **Monobit** test. This corresponds to `sstring_HammingWeight2` with $L = n$.

2. The **Frequency test within a Block**. Corresponds to `sstring_HammingWeight2`.

3. The **Runs** test. Is implemented as `sstring_Run`.

4. The test for the **Longest Run of Ones in a Block**. Is implemented as the test `sstring_LongestHeadRun`.

5. The **Binary Matrix rank** test. Is implemented as `smarsa_MatrixRank`.

6. The **Discrete Fourier Transform** test. Is implemented as `sspectral_Fourier1`.

7. The **Non-overlapping Template Matching** test. Is implemented as the test `smarsa_CATBits`.

8. The **Overlapping Template Matching** test. This test does not exist as such in TestU01, but a similar and more powerful test is `smultin_MultinomialBitsOver`.

9. **Maurer's Universal Statistical** test. This test is implemented as `svaria_AppearanceSpacings`.

10. The **Lempel-Ziv Compression** test. Is implemented as `scomp_LempelZiv`.

11. The **Linear Complexity** test. Is implemented as part of `scomp_LinearComp`.

12. The **Serial** test. Corresponds to `smultin_MultinomialBitsOver` with `Delta` = 1.

13. The **Approximate Entropy** test. Corresponds to `smultin_MultinomialBitsOver` with `Delta` = 0, and to `sentrop_EntropyDiscOver` or `sentrop_EntropyDiscOver2`.

14. The **Cumulative Sums** test. This test is closely related to the M statistic in `swalk_RandomWalk1`.

15. The **Random Excursions** test. This test does not exist in TestU01, but closely related tests are in `swalk_RandomWalk1`.

16. The **Random Excursions Variant** test. This test does not exist in TestU01, but a closely related test is based on the R statistic in `swalk_RandomWalk1`.

```
void bbattery_FIPS_140_2 (unif01_Gen *gen);
void bbattery_FIPS_140_2File (char *filename);
```

These functions apply the four tests described in the NIST document *FIPS PUB 140-2, Security Requirements for Cryptographic Modules*, page 35, with exactly the same parameters (see the WEB page at `http://csrc.nist.gov/rng/rng6_3.html`). They report the values of the test statistics and their p-values (except for the runs test) and indicate which values fall outside the intervals specified by FIPS-140-2. The first function applies the tests on a generator `gen`, and the second applies them on the file of bits `filename`. First, 20000 bits are generated and put in an array, then the tests are applied upon these. The tests applied are:

1. The **Monobit** test. This corresponds to `smultin_MultinomialBits` with $s = 32$, $L = 1$, $n = 20000$.

2. The "**poker**" test, which is in fact equivalent to `smultin_MultinomialBits` with $s = 32$, $L = 4$, $n = 5000$.

3. The **Runs** test, which is related to `sstring_Run`.

4. The test for the **Longest Run of Ones in a Block**, which is implemented as `sstring_LongestHeadRun`.

Chapter 5

FAMILIES OF GENERATORS

The tools described in this chapter are convenient for examining systematically the interaction between specific tests and certain families of RNGs. The framework is as follows. For each family, an RNG of period length near 2^i has been selected, on the basis on some theoretical criteria that depend on the family, for all integers i in some interval (from 10 to 40, for example). The parameters of the pre-selected instances are stored in text files in directory `param`.

Typically, for a given RNG that fails a test, when the sample size of the test is increased, the p-value would remain "reasonable" for a while, say for n up to some threshold n_0 (roughly), and will then converges to 0 or 1 exponentially fast as a function of n. It is interesting to examine the relationship between n_0 and i. The idea is to fit a crude regression model of n_0 as a function of i. For example, one may consider a model of the form

$$\log_2 n_0 = \gamma i + \nu + \delta, \tag{5.1}$$

where γ and ν are constants and δ represents the noise. The result may give an idea of what period length ρ of the RNG is required, within a given family, to be safe with respect to the test that is considered, for a given computer budget.

This methodology has been applied in [48, 90, 87, 93, 96, 89], using an earlier version of the present library, and gave surprisingly good results in many cases. For full-period LCGs with good spectral test behavior, for example, the relationships $n_0 \approx 16\,\rho^{1/2}$ for the collision test and $n_0 \approx 16\,\rho^{1/3}$ for the birthday spacings test have been obtained. This means that no LCG is safe with respect to these particular tests unless its period length ρ is so large that generating $\rho^{1/3}$ numbers is practically unfeasible. A period length of 2^{48} or less, for example, does not satisfy this requirement.

Common parameters.

The first argument of each testing function in the `f` modules is the family `fam` of random number generators to be tested (see module `ffam` for details). That family must be created

by calling the appropriate function in the module `fcong` or `ffsr`, and deleted when no longer needed.

The second argument (the third one in the `fmultin` module) of each testing function is a structure `res` that can keep the tables of p-values and other results (see module `fres` for details). This is useful if one wishes to do something else with the tables of results after all tests are ended. If one does not want to post-process or use the tables of results after a `f` test, it suffices to set the `res` argument to the `NULL` pointer. Then, the structure is created and deleted automatically inside the testing function. In any case, the tables of results will be printed automatically on standard output.

The third argument (the fourth one in the `fmultin` module) of each testing function is a structure `cho` that allows the user to choose varying sample sizes and other parameters of the tests as a function of the generators *lsize* and the fixed parameters (see module `fcho` for details).

For each of these three arguments (except possibly for `res` as explained above), one must call the appropriate `Create` function before using them, and call the corresponding `Delete` function when they are no longer needed.

The last four arguments of each testing function are the integers `Nr, j1, j2` and `jstep`. The test functions will be applied on the first `Nr` generators of the family `fam`. If there are less than `Nr` generators in the family, then `Nr` will be reset to the number of generators in the tested family. For each of the generator, tests will be applied with varying sample sizes determined by the parameter j for j varying from `j1` to `j2` by step of `jstep`.

An example: The collision test applied to a family of LCG's.

For a concrete illustration, Figure 5.1 shows a program applying the collision test (see tests `sknuth_Collision` and `smultin_Multinomial`) systematically to a family of LCGs. First, the call to `fcong_CreateLCG` creates the family of generators to which the tests are applied; i.e., each generator instance has its parameters predefined in the file `LCGGood.par`. These "good LCGs" have prime modulus m (the largest prime less than 2^i), full period length $m-1$, and perform well in the spectral test for up to dimension 8. They are taken from [81] and are listed in Table 5.1 for $10 \leq i \leq 30$.

The following instruction in the program, `smultin_CreateParam` with `NbDelta = 1` and `ValDelta[0] = -1`, specifies that only the collision test will be applied. The `par` parameter in the test could be replaced by the `NULL` pointer, in which case default values would be used and it would not be necessary to create a `par` structure. The next instruction `fmultin_CreateRes` creates a structure to hold the results of the test. If one does not want to postprocess the results after all the tests are ended, one may also pass a `NULL` pointer as argument `res` to the test, in which case it would not be necessary to create a `res` structure either. The instruction `fcho_CreateSampleSize` specifies that the sample size n (the number of points) will be chosen as $n = 2^{i/2+j}$. The next instruction `fmultin_CreatePer_DT` says that the number of cells k will be equal to the period length of the generator tested, and that the relation between k, the one-dimensional interval d, and the dimension t (here

Table 5.1: Some good LCGs according to the spectral test in up to dimension 8.

m	a
$2^{10} - 3 = 1021$	65
$2^{11} - 9 = 2039$	995
$2^{12} - 3 = 4093$	209
$2^{13} - 1 = 8191$	884
$2^{14} - 3 = 16381$	572
$2^{15} - 19 = 32749$	219
$2^{16} - 15 = 65521$	17364
$2^{17} - 1 = 131071$	43165
$2^{18} - 5 = 262139$	92717
$2^{19} - 1 = 524287$	283741
$2^{20} - 3 = 1048573$	380985
$2^{21} - 9 = 2097143$	360889
$2^{22} - 3 = 4194301$	914334
$2^{23} - 15 = 8388593$	653276
$2^{24} - 3 = 16777213$	6423135
$2^{25} - 39 = 33554393$	25907312
$2^{26} - 5 = 67108859$	26590841
$2^{27} - 39 = 134217689$	45576512
$2^{28} - 57 = 268435399$	31792125
$2^{29} - 3 = 536870909$	16538103
$2^{30} - 35 = 1073741789$	5122456

```
#include <testu01/fcong.h>
#include <testu01/ffam.h>
#include <testu01/fcho.h>
#include <testu01/fmultin.h>
#include <testu01/smultin.h>

int main (void)
{
   int NbDelta = 1;
   double ValDelta[] = { -1 };
   int t = 2;
   ffam_Fam *fam;
   smultin_Param *par;
   fmultin_Res *res;
   fcho_Cho *chon;
   fcho_Cho *chod;
   fcho_Cho2 *cho;

   fam = fcong_CreateLCG ("LCGGood.par", 10, 30, 1);
   par = smultin_CreateParam (NbDelta, ValDelta, smultin_GenerCellSerial, 2);
   res = fmultin_CreateRes (par);
   chon = fcho_CreateSampleSize (0.5, 1, 0, NULL, "n");
   chod = fmultin_CreatePer_DT (t, 1);
   cho = fcho_CreateCho2 (chon, chod);

   fmultin_Serial1 (fam, par, res, cho, 1, 0, t, TRUE, 21, 1, 5, 1);

   fcho_DeleteCho2 (cho);
   fmultin_DeletePer (chod);
   fcho_DeleteSampleSize (chon);
   fmultin_DeleteRes (res);
   smultin_DeleteParam (par);
   fcong_DeleteLCG (fam);
   return 0;
}
```

Figure 5.1: Applying the collision test to a family of LCGs.

2) is given by $k = d^t$. Then these two "choose" functions are set in the structure `cho` that will be passed as argument to the test. The call `fmultin_Serial1` launches the series of collision tests, with the parameters $N = 1$, $r = 0$, $t = 2$, `Sparse = TRUE`. LCGs with period lengths near 2^i will be tested for $i = 10, 11, \ldots, 30$, each with sample size $n = 2^{i/2+j}$ for $j = 1, 2, 3, 4, 5$. Finally, all the created structures are deleted to free the memory used by each of them.

Table 5.2: Expected number of collisions and observed number of collisions.

LSize	$j=1$		$j=2$		$j=3$		$j=4$		$j=5$	
10	1.93	2	7.62	2	29.39	11	108.94	58	376.52	570
11	1.95	1	7.81	0	30.44	6	115.16	17	413.38	474
12	1.96	0	7.81	2	30.65	13	117.87	44	436.20	216
13	2.00	0	7.95	2	31.37	13	121.97	55	461.02	211
14	1.98	1	7.90	5	31.31	11	122.77	52	471.77	191
15	1.99	1	7.93	1	31.51	7	124.27	41	483.04	132
16	1.99	1	7.95	0	31.65	5	125.35	20	491.26	75
17	1.99	0	7.97	1	31.75	11	126.11	23	497.29	101
18	2.00	0	7.98	4	31.83	15	126.66	45	501.47	214
19	2.00	0	7.98	1	31.88	3	127.07	19	504.60	61
20	2.00	0	7.99	0	31.91	4	127.33	14	506.69	74
21	2.00	0	7.99	1	31.94	12	127.55	48	508.33	178
22	2.00	0	7.99	3	31.96	14	127.66	59	509.34	220
23	2.00	0	8.00	0	31.97	5	127.78	16	510.21	45
24	2.00	0	8.00	0	31.98	1	127.83	15	510.67	37
25	2.00	1	8.00	4	31.99	8	127.91	44	511.16	142
26	2.00	2	8.00	1	31.99	7	127.92	24	511.33	71
27	2.00	1	8.00	1	31.99	7	127.94	38	511.55	152
28	2.00	0	8.00	3	31.99	17	127.96	41	511.67	193
29	2.00	1	8.00	0	32.00	0	127.98	8	511.78	25
30	2.00	0	8.00	3	32.00	10	127.98	29	511.83	189

Tables 5.2 and 5.3 give the results of this program. In Table 5.2, for each value of i and j, the expected number of collisions is given on the left, and the observed number of collisions on the right. We see that for $j \geq 3$, the observed number of collisions is generally much too small (there are exceptions at $j = 5$ and $i = 10$ and 11). The p-values written in Table 5.3 are those which fall outside the interval $[0.01, 0.99]$, which may be called suspect p-values. A p-value smaller than 10^{-300} is noted by ϵ. A p-value larger than $1 - 10^{-15}$ is noted by $-\epsilon_1$ (p-values close to 1 are written as $-p$ instead of $1-p$). We see that for $j \geq 3$, the p-values in Table 5.3 are very close to 1. This means that the two-dimensional points produced by these generators are too evenly distributed and the test starts detecting this when the sample size reaches $n_0 \approx 2^{i/2+3} \approx 8\sqrt{\rho}$. Very clear rejection occurs in all cases for $j = 4$, i.e., at sample size $n \approx 16\sqrt{\rho}$.

Table 5.3: The p-values of the collision test for the good LCGs, for $t = 2$ and $k \approx 2^i$.
Total CPU time: 00:00:06.69

LSize	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
10			–1.0e–5	–1.1e–12	ϵ
11		–2.5e–4	–6.3e–9	$-\epsilon_1$	1.1e–6
12			–1.1e–4	$-\epsilon_1$	$-\epsilon_1$
13			–9.0e–5	–6.0e–14	$-\epsilon_1$
14			–1.5e–5	–9.4e–15	$-\epsilon_1$
15		–2.9e–3	–9.0e–8	$-\epsilon_1$	$-\epsilon_1$
16		–3.2e–4	–3.5e–9	$-\epsilon_1$	$-\epsilon_1$
17		–3.0e–3	–1.6e–5	$-\epsilon_1$	$-\epsilon_1$
18			–6.6e–4	$-\epsilon_1$	$-\epsilon_1$
19		–3.0e–3	–7.0e–11	$-\epsilon_1$	$-\epsilon_1$
20		–3.3e–4	–6.0e–10	$-\epsilon_1$	$-\epsilon_1$
21		–3.0e–3	–4.7e–5	$-\epsilon_1$	$-\epsilon_1$
22			–2.9e–4	–7.1e–12	$-\epsilon_1$
23		–3.3e–4	–4.1e–9	$-\epsilon_1$	$-\epsilon_1$
24		–3.3e–4	–4.1e–13	$-\epsilon_1$	$-\epsilon_1$
25			–4.5e–7	$-\epsilon_1$	$-\epsilon_1$
26		–3.0e–3	–1.1e–7	$-\epsilon_1$	$-\epsilon_1$
27		–3.0e–3	–1.1e–7	$-\epsilon_1$	$-\epsilon_1$
28			–2.8e–3	$-\epsilon_1$	$-\epsilon_1$
29		–3.4e–4	–1.3e–14	$-\epsilon_1$	$-\epsilon_1$
30			–5.6e–6	$-\epsilon_1$	$-\epsilon_1$

Another example: The birthday spacings test applied to a family of LCG's.

The next example in Figure 5.2 gives a program applying the birthday spacings test (see `smarsa_BirthdaySpacings`) to a family of generators. First the family `LCGPow2` is created by calling function `fcong_CreateLCGPow2`. Each generator of the family has its parameters predefined in file `LCGPow2.par`. These generators have been chosen such that their modulus m is exactly equal to a power of 2, i.e. $m = 2^i$ for $10 \leq i \leq 30$, and their multiplier gives them a good structure according to the spectral test. The sample size of the test (the number of points n) is then chosen (by calling `fcho_CreateSampleSize`) to follow the law $n = 2^{i/3+j}$, for generator of modulus $m = 2^i$. The next instruction `fmarsa_CreateBirthEC` indicates that, given $N = 1$ replication of the test and dimension $t = 2$, the number of segments d on the interval $[0, 1)$ will be chosen so that the expected number of collisions is (approximately) equal to 1. These two "choose" functions are set in the structure `cho` by the call `fcho_CreateCho2` and will thus be passed as arguments to the tests. Then function `fmarsa_BirthdayS1` applies the birthday spacings test on the 21 selected generators of the family, for $1 \leq j \leq 5$ and with the other parameters determined by the above functions. After all the tests are completed, the following `Delete` functions free the resources used by the program.

```
#include <testu01/fcong.h>
#include <testu01/ffam.h>
#include <testu01/fcho.h>
#include <testu01/fmarsa.h>

int main (void)
{
   ffam_Fam *fam;
   fcho_Cho *chon;
   fcho_Cho *chod;
   fcho_Cho2 *cho;

   fam = fcong_CreateLCGPow2 (NULL, 10, 30, 1);
   chon = fcho_CreateSampleSize (1.0/3.0, 1, 0, NULL, "n");
   chod = fmarsa_CreateBirthEC (1, 2, 1.0);
   cho = fcho_CreateCho2 (chon, chod);
   fmarsa_BirthdayS1 (fam, NULL, cho, 1, 0, 2, 1, 21, 1, 5, 1);
   fcho_DeleteCho2 (cho);
   fmarsa_DeleteBirthEC (chod);
   fcho_DeleteSampleSize (chon);
   fcong_DeleteLCGPow2 (fam);
   return 0;
}
```

Figure 5.2: Applying the birthday spacings test to a family of LCGs.

Tables 5.4 and 5.5 give the results of this program. In Table 5.4, for each value of i and j, the expected number of collisions is given on the left and the observed number of collisions on the right. We see that for $j \geq 2$, the observed number of collisions is much too large and the more so as j increases. The right p-values written in Table 5.5 are those which fall outside the interval $[0.01, 0.99]$. A p-value smaller than 10^{-300} is noted by ϵ. For $j = 1$, the

number of collisions is close to the expected value and the corresponding p-values are in the interval $[0.01, 0.99]$. The generators pass the test also for $j \leq 1$. But for $j \geq 2$, the p-values in Table 5.5 becomes smaller as j increases. The tests signals catastrophic failures of the generators already for $j \geq 3$. This is because the two-dimensional points produced by these generators are too evenly distributed and the test starts detecting this when the sample size reaches $n \approx 2^{i/3+2}$. These are quite small sample sizes; for example, for the generator with $m = 2^{30}$, the tests start to fail for n as small as 4096.

Table 5.4: Birthday spacings test: Expected and observed number of collisions for each i and j.

LSize	$j=1$		$j=2$		$j=3$		$j=4$		$j=5$	
10	—	—	1.01	5	1.00	48	1.00	131	—	—
11	—	—	1.01	8	1.00	52	1.00	147	1.00	377
12	1.01	0	1.00	10	1.00	44	1.00	204	1.00	447
13	1.01	0	1.00	11	1.00	50	1.00	196	1.00	579
14	1.01	0	1.00	10	1.00	56	1.00	247	1.00	662
15	1.00	2	1.00	9	1.00	55	1.00	258	1.00	911
16	1.00	1	1.00	5	1.00	61	1.00	316	1.00	958
17	1.00	0	1.00	7	1.00	70	1.00	365	1.00	1172
18	1.00	1	1.00	10	1.00	73	1.00	385	1.00	1374
19	1.00	4	1.00	13	1.00	79	1.00	414	1.00	1600
20	1.00	0	1.00	7	1.00	72	1.00	449	1.00	1895
21	1.00	1	1.00	5	1.00	73	1.00	509	1.00	2176
22	1.00	4	1.00	8	1.00	78	1.00	505	1.00	2472
23	1.00	3	1.00	13	1.00	79	1.00	522	1.00	2797
24	1.00	1	1.00	16	1.00	59	1.00	580	1.00	3092
25	1.00	2	1.00	9	1.00	102	1.00	588	1.00	3392
26	1.00	0	1.00	10	1.00	79	1.00	629	1.00	3709
27	1.00	2	1.00	11	1.00	89	1.00	636	1.00	3915
28	1.00	1	1.00	8	1.00	83	1.00	691	1.00	4097
29	1.00	2	1.00	12	1.00	89	1.00	650	1.00	4500
30	1.00	3	1.00	16	1.00	80	1.00	649	1.00	4551

Table 5.5: The right p-values $P[Y \geq y]$ of the BirthdaySpacings test for the LCGPow2s.
Total CPU time: 00:00:00.07

LSize	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
10	—	3.8e–3	3.7e–62	4.9e–223	—
11	—	1.1e–5	5.2e–69	2.2e–257	ϵ
12		1.1e–7	1.4e–55	ϵ	ϵ
13		1.0e–8	1.3e–65	ϵ	ϵ
14		1.1e–7	5.3e–76	ϵ	ϵ
15		1.1e–6	3.0e–74	ϵ	ϵ
16		3.7e–3	7.4e–85	ϵ	ϵ
17		8.3e–5	3.2e–101	ϵ	ϵ
18		1.1e–7	8.5e–107	ϵ	ϵ
19		6.4e–11	4.2e–118	ϵ	ϵ
20		8.3e–5	6.1e–105	ϵ	ϵ
21		3.7e–3	8.3e–107	ϵ	ϵ
22		1.0e–5	3.3e–116	ϵ	ϵ
23		6.4e–11	4.2e–118	ϵ	ϵ
24		1.9e–14	2.7e–81	ϵ	ϵ
25		1.1e–6	3.9e–163	ϵ	ϵ
26		1.1e–7	4.2e–118	ϵ	ϵ
27		1.0e–8	2.3e–137	ϵ	ϵ
28		1.0e–5	9.4e–126	ϵ	ϵ
29		8.3e–10	2.3e–137	ϵ	ϵ
30		1.9e–14	5.2e–120	ϵ	ϵ

ffam

This module provides generic tools used in testing whole families of generators. There are many predefined families of generators of the same kind (see modules `fcong` and `ffsr` for some examples); their defining parameters are in files kept in directory `param` of TestU01. For each generator in a given family, we define a size (the period length) and a resolution. We shall define the *lsize* of a generator as the (rounded) base-2 logarithm of the period length. *Resolution* is a somewhat fuzzy notion. If we have a LCG with modulus $m = 2^b$, say, then each output number is a multiple of 2^{-b} and we say we have b bits of resolution in the output. More generally, if the number of *different* output values that can be produced by the generator is n (not necessarily a power of 2), we say that the "resolution" is (approximately) $\lfloor \log_2 n \rfloor$.

For the predefined families, each generator of the family has been chosen in such a way that its period length (the number of possible states of the generator) is very close to a power of 2. Thus, a given test may be applied with a variable sample size on generators of the same kind whose size varies as successive powers of 2. One may then observe some interactions between a test and the structure of the generators of a given kind, and this will appear as regularities in the results. The user may also define his own family of generators.

```
#include <testu01/unif01.h>
```

Families of generators

The following structure is used in the `f` modules to keep a family of generators upon which tests with variable sample sizes are applied. Such a structure must always be created, directly or indirectly, before calling a testing function. Usually, this structure will be created indirectly by one of the `Create` function in modules `fcong` and `ffsr`.

```
typedef struct {
unif01_Gen **Gen;
int *LSize;
int *Resol;
int Ng;
char *name;
} ffam_Fam;
```

Array element `Gen[i]` is a generator of the family, array element `Resol[i]` gives the (approximate) number of bits of resolution in the output values of generator `Gen[i]`. Array element `LSize[i]` gives the base-2 logarithm of the approximate period length of `Gen[i]`, i.e. if `LSize[i]` $= h$, then `Gen[i]` has a period that is very close to 2^h. There are `Ng` members in the family (and the size of the above arrays is `Ng`), so their elements are numbered from $0 \leq i \leq$ `Ng-1`. The string `name` gives the name of the family.

```
ffam_Fam * ffam_CreateFam (int Ng, char *name);
```

Creates and returns a structure to hold a family named `name` that can contains up to `Ng` generators.

```
void ffam_DeleteFam (ffam_Fam *fam);
```

Frees the memory allocated to `fam` by `ffam_CreateFam`.

```
void ffam_PrintFam (ffam_Fam *fam);
```

Prints all the fields of the family `fam`.

```
void ffam_ReallocFam (ffam_Fam *fam, int Ng);
```

Reallocs memory to the three arrays of `fam` so that they can contain up to `Ng` elements.

```
ffam_Fam * ffam_CreateSingle (unif01_Gen *gen, int resol, int i1, int i2);
```

Creates and returns a structure to hold a family that can contains up to `i2` $-$ `i1` $+ 1$ generators. All members of the family will be the same generator `gen` of resolution `resol`. The generator will imitate a family of generators of lsize in `i1` $\leq$ lsize $\leq$ `i2`. This is useful, amongst other things, to explore the domain of the approximation error in the distribution function of a test with the help of a high quality generator, or to find out the behaviour of a given generator with respect to a given test as the sample size increases.

```
void ffam_DeleteSingle (ffam_Fam *fam);
```

Frees the memory allocated to `fam` by `ffam_CreateSingle`.

fcong

The functions in this module creates whole families of generators of the same kind (such as LCGs, MRGs, inversive, cubic, ...), based on recurrences modulo some integer m, of different period lengths (the number of states) near powers of 2. Each `Create` function will return a family of generators whose *lsize* varies from `i1` to `i2` by step of `istep`, and whose parameters are taken from a file. The *lsize* of a generator is defined as the (rounded) base-2 logarithm of its period length.

There are predefined families of each kind whose parameters are given in files with extension `.par` in directory `param` of TestU01. If the file name in the `Create` functions below is set to `NULL`, a default predefined family will be used. Otherwise, the given file will be used to set the parameters of the generators of a family. The members of a predefined family usually have a *lsize* equal to successive integers in [`i1, i2`].

The user may want to define his own family of generators. If it is closely related to one of the predefined family, he may use one of the `Create` function in this module to create his family. In that case, his parameter file must contain the family parameters in the same order as the predefined family of the same kind (see the different cases of `fcong_CreateLCG` below for an example).

More information about each specific kind of generator considered can be found by looking at the corresponding function in modules `u...` For example, `fcong_CreateInvImpl2b` implements the same generators as in function `uinv_CreateInvImpl2b`.

```
#include <testu01/ffam.h>
```

The families of generators

```
ffam_Fam * fcong_CreateLCG (char *fname, int i1, int i2, int istep);
```

Creates a family of LCG generators whose parameters are defined in file named `fname`, beginning with the first generator whose *lsize* is `i1`, up to the last generator whose *lsize* is `i2`, taking every `istep` generator. The predefined LCG files are:

- **LCGGood.par**: the generators have a good lattice structure up to dimension 8 (at least), with a prime modulus m just below 2^i and period $m - 1$, for $10 \leq i \leq 60$. For each i, the LCG provided is the one with the best value of M_8 in [81]. Restrictions: $10 \leq i1 \leq i2 \leq 60$.
- **LCGBad2.par**: Similar to the LCGGood family, but with a lattice structure that is bad in dimension 2. The figure of merit S_2 in dimension 2 is approximately 0.05. Restrictions: $10 \leq i1 \leq i2 \leq 40$.
- **LCGWu2.par**: the generators have a good lattice structure as the LCGGood family, but with the restriction that the multiplier is a sum or a difference of two powers of 2, as suggested by P. C. Wu [177]. Restrictions: $10 \leq i1 \leq i2 \leq 40$.

- **LCGGranger.par**: the generators have a good lattice structure up to dimension 8 (at least), with a prime modulus m just below 2^i, for $10 \leq i \leq 31$. They were chosen so as to give a maximal period for the combined generators in modules `ugranger` and `tgranger`.

`ffam_Fam * fcong_CreateLCGPow2 (char *fname, int i1, int i2, int istep);`

Creates a family of LCG generators whose parameters are defined in file named `fname`. By default, uses a predefined family of generators with a good lattice structure as the LCGGood family, but with the modulus equal to 2^i and period length also equal to 2^i. (The recurrence has a nonzero constant term.) Restrictions: $10 \leq i1 \leq i2 \leq 40$.

`ffam_Fam * fcong_CreateMRG2 (char *fname, int i1, int i2, int istep);`

Creates the family of MRG (multiple-recursive generators) of order 2 whose parameters are defined in file named `fname`. By default, uses a predefined family of generators with a good lattice structure up to dimension 8 (at least), with a prime modulus m just below $2^{i/2}$ and period length $m^2 - 1$. The implementation is similar to that of the LCGs. Restrictions: $20 \leq i1 \leq i2 \leq 60$.

`ffam_Fam * fcong_CreateMRG3 (char *fname, int i1, int i2, int istep);`

Creates the family of MRG of order 3 whose parameters are defined in file named `fname`. By default, uses a predefined family of generators with a good lattice structure up to dimension 8 (at least), with a prime modulus m just below $2^{i/3}$ and period length $m^3 - 1$. Restrictions: $30 \leq i1 \leq i2 \leq 90$.

`ffam_Fam * fcong_CreateCombL2 (char *fname, int i1, int i2, int istep);`

Creates the family of combined LCG with two components, whose parameters are defined in `fname`. The combination uses the method of L'Ecuyer [72]. By default, uses a predefined family of generators with a good lattice structure up to dimension 8 (at least). The components have distinct prime moduli m_1 and m_2 just below $2^{i/2}$ and the period length is $(m_1 - 1)(m_2 - 1)/2$. The parameters are chosen to get an excellent value of M_8 where M_8 is defined as in [81]. Restrictions: $20 \leq i1 \leq i2 \leq 60$.

`ffam_Fam * fcong_CreateCombWH2 (char *fname, int i1, int i2, int istep);`

Same as `fcong_CreateCombL2`, except that the combination is of the Wichmann and Hill type (see [97]). Restrictions: $20 \leq i1 \leq i2 \leq 60$.

`ffam_Fam * fcong_CreateInvImpl (char *fname, int i1, int i2, int istep);`

Creates the family of implicit inversive generators whose parameters are defined in `fname`. By default, uses a predefined family of generators with prime modulus m slightly below 2^i and period length m. Restrictions: $10 \leq i1 \leq i2 \leq 30$.

`ffam_Fam * fcong_CreateInvImpl2a (char *fname, int i1, int i2, int istep);`

Creates a predefined family of implicit inversive generators whose parameters are fixed, with prime modulus m slightly below 2^{i+1} and period length $m/2$. Restrictions: $7 \leq i1 \leq i2 \leq 31$.

`ffam_Fam * fcong_CreateInvImpl2b (char *fname, int i1, int i2, int istep);`

Creates a predefined family of implicit inversive generators whose parameters are fixed, with prime modulus m slightly below 2^i and period length m. Restrictions: $7 \leq i1 \leq i2 \leq 32$.

`ffam_Fam * fcong_CreateInvExpl (char *fname, int i1, int i2, int istep);`

Creates the family of explicit inversive generators whose parameters are defined in `fname`. By default, uses a predefined family of generators with prime modulus m slightly below 2^i and period length m. Restrictions: $10 \leq i1 \leq i2 \leq 31$.

`ffam_Fam * fcong_CreateInvExpl2a (char *fname, int i1, int i2, int istep);`

Creates a predefined family of explicit inversive generators whose parameters are fixed, with prime modulus m slightly below 2^i and period length m. Restrictions: $7 \leq i1 \leq i2 \leq 32$.

`ffam_Fam * fcong_CreateInvExpl2b (char *fname, int i1, int i2, int istep);`

Creates a predefined family of explicit inversive generators whose parameters are fixed, with prime modulus m slightly below 2^i and period length m. Restrictions: $7 \leq i1 \leq i2 \leq 32$.

`ffam_Fam * fcong_CreateInvMRG2 (char *fname, int i1, int i2, int istep);`

Creates the family of inversive MRG of order 2 whose parameters are defined in file `fname`. By default, uses a predefined family of generators with a prime modulus m just below $2^{i/2}$ and period length $\approx m^2$. Restrictions: $20 \leq i1 \leq i2 \leq 60$.

`ffam_Fam * fcong_CreateCubic1 (char *fname, int i1, int i2, int istep);`

Creates the family of cubic congruential generator whose parameters are defined in file `fname`. By default, uses a predefined family of generators with modulus m slightly below 2^i and period length m. Restrictions: $6 \leq i1 \leq i2 \leq 18$.

`ffam_Fam * fcong_CreateCombCubic2 (char *fname, int i1, int i2, int istep);`

Creates the family of combined cubic congruential generators with 2 components whose parameters are defined in file `fname`. By default, uses a predefined family of generators with moduli m_1 and m_2 slightly below 2^{e_1} and 2^{e_2}, where $e_1 + e_2 = i$ and $e_2 = \lfloor i/2 \rfloor$. The period length should be near 2^i. Restrictions: $12 \leq i1 \leq i2 \leq 36$.

```
ffam_Fam * fcong_CreateCombCubLCG (char *fname, int i1, int i2, int istep);
```

Creates the family of combined generators with one component a cubic congruential and the other component an LCG whose parameters are defined in file `fname`. By default, uses a predefined family of generators with respective moduli of the components m_1 and m_2 slightly below 2^{e_1} and 2^{e_2}, where $e_1 + e_2 = i$ and $e_2 = \lfloor i/2 \rfloor$. Restrictions: $19 \leq i1 \leq i2 \leq 36$.

Clean-up functions

```
void fcong_DeleteLCG     (ffam_Fam *fam);
void fcong_DeleteLCGPow2 (ffam_Fam *fam);
void fcong_DeleteMRG2    (ffam_Fam *fam);
void fcong_DeleteMRG3    (ffam_Fam *fam);
void fcong_DeleteCombL2  (ffam_Fam *fam);
void fcong_DeleteCombWH2 (ffam_Fam *fam);
void fcong_DeleteInvImpl   (ffam_Fam *fam);
void fcong_DeleteInvImpl2a (ffam_Fam *fam);
void fcong_DeleteInvImpl2b (ffam_Fam *fam);
void fcong_DeleteInvExpl   (ffam_Fam *fam);
void fcong_DeleteInvExpl2a (ffam_Fam *fam);
void fcong_DeleteInvExpl2b (ffam_Fam *fam);
void fcong_DeleteInvMRG2   (ffam_Fam *fam);
void fcong_DeleteCubic1    (ffam_Fam *fam);
void fcong_DeleteCombCubic2  (ffam_Fam *fam);
void fcong_DeleteCombCubLCG  (ffam_Fam *fam);
```

Frees the dynamic memory allocated to `fam` by the corresponding `Create` function.

ffsr

This module operates in the same way as `fcong` (see the introduction in module `fcong`). It defines *linear feedback shift register* (LFSR) generators of different period lengths (or number of states) near powers of 2, and different kinds such as Tausworthe, GFSR, twisted GFSR (TGFSR), and their combinations. All these generators are based on linear recurrences modulo 2.

```
#include <testu01/ffam.h>
```

The families of generators

```
ffam_Fam * ffsr_CreateLFSR1 (char *fname, int i1, int i2, int istep);
```

Creates a family of simple LFSR (or Tausworthe) generators whose parameters are defined in file named `fname`. By default, uses a predefined family of generators with primitive characteristic trinomial of degree i (with period length $2^i - 1$) and the best equidistribution properties within its class. Restrictions: $10 \leq i1 \leq i2 \leq 60$.

```
ffam_Fam * ffsr_CreateLFSR2 (char *fname, int i1, int i2, int istep);
```

Creates a family of combined LFSR generators with two components, whose parameters are defined in file named `fname`. By default, uses a predefined family with each component based on a primitive characteristic trinomial. The combination generator has period length near $2^i - 1$ and the best possible equidistribution within its class. Restrictions: $10 \leq i1 \leq i2 \leq 36$.

```
ffam_Fam * ffsr_CreateLFSR3 (char *fname, int i1, int i2, int istep);
```

Creates a family of combined LFSR generators with three components, whose parameters are defined in file named `fname`. By default, uses a predefined family with each component based on a primitive characteristic trinomial. The combination generator has period length near $2^i - 1$ and the best possible equidistribution within its class. Restrictions: $14 \leq i1 \leq i2 \leq 36$.

```
ffam_Fam * ffsr_CreateGFSR3 (char *fname, int i1, int i2, int istep);
```

Creates a family of generalized feedback shift register (GFSR) generators with primitive characteristic trinomial of degree i (period length $2^i - 1$) and good equidistribution. ***** NOT YET IMPLEMENTED.

```
ffam_Fam * ffsr_CreateGFSR5 (char *fname, int i1, int i2, int istep);
```

Creates a family of generalized feedback shift register (GFSR) generators with primitive characteristic pentanomial of degree i (period length $2^i - 1$) and good equidistribution. ***** NOT YET IMPLEMENTED.

```
ffam_Fam * ffsr_CreateTGFSR1 (char *fname, int i1, int i2, int istep);
```

Creates a family of twisted GFSR generators with primitive characteristic trinomial of degree i (period length $2^i - 1$) and good equidistribution. ***** NOT YET IMPLEMENTED.

```
ffam_Fam * ffsr_CreateTausLCG2 (char *fname, int i1, int i2, int istep);
```

Creates a family of combined generators that adds the outputs of an LCG and an LFSR2, modulo 1, whose parameters are defined in file named `fname`. Each generator is created by calling the function `ulec_CreateCombTausLCG21`. By default, uses a predefined family with generators having a period near 2^i. Restrictions: $20 \leq i1 \leq i2 \leq 62$.

Clean-up functions

```
void ffsr_DeleteLFSR1 (ffam_Fam *fam);
void ffsr_DeleteLFSR2 (ffam_Fam *fam);
void ffsr_DeleteLFSR3 (ffam_Fam *fam);
void ffsr_DeleteTausLCG2 (ffam_Fam *fam);
```

Frees the dynamic memory allocated to `fam` by the corresponding `Create` function.

ftab

This module provides tools to manipulate and print tables of p-values and other results, when statistical tests with different sample sizes are applied to a whole family of generators of different sizes or different resolutions (or precisions). Each table contains a two-dimensional array of values (`Mat`), indexed by i and j. The row i of `Mat` is associated with a generator of the family that is being tested, while the column j is associated with the different sample sizes for the test being applied on the generator. Such a table can be created by `ftab_CreateTable`, printed by `ftab_PrintTable` or `ftab_PrintTable2`, and deleted by `ftab_DeleteTable`. The function `ftab_MakeTables` is used to run a series of tests on a whole family of generators and to fill up the tables of results.

```
#include <testu01/ffam.h>
#include <testu01/unif01.h>


typedef struct {
double **Mat;
int *LSize;
int Nr, Nc;
int j1, j2, jstep;
ftab_FormType Form;
char *Desc;
char **Strings;
int Ns;
} ftab_Table;
```

A structure that contains a two-dimensional matrix `Mat` with `Nr` rows and `Nc` columns, used to store the values of statistics, their p-values, or some other information depending on the format `Form`, though the numbers are always stored as `double`'s. The values are stored in matrix element `Mat[`i`][`j`]` for $0 \leq i <$ `Nr` and $0 \leq j <$ `Nc`. Row i of `Mat` is associated with a generator of size `LSize[`i`]`. The index j is used to select the different sample sizes of a test for a given generator. The character string `Desc` gives a short description of the table.

The array `Strings` points to the `Ns` possible messages that can be printed for each element `Mat[`i`][`j`]` when `Form` is `ftab_String`. In this case, `Mat[`i`][`j`]` is an integer giving the index s of the message `Strings[`s`]` to be printed. When `Form` is not `ftab_String`, `Strings` is set to the `NULL` pointer. The function `ftab_CreateTable` creates such a structure.

Functions to manipulate tables

```
ftab_Table *ftab_CreateTable (int Nr, int j1, int j2, int jstep,
char *Desc, ftab_FormType Form, int Ns);
```

Creates a structure `ftab_Table` and its matrix `Mat` so that it can store test results in format `Form` for a family of `Nr` generators. Each generator is subjected to tests with different sample sizes indexed by j, with j varying from `j1` to `j2` by step of `jstep`. The function initializes the description of the table to `Desc`. If `Form = ftab_String`, it also allocate an array of `Ns` pointers of `char` for the field `Strings` of the structure.

```
void ftab_DeleteTable (ftab_Table *T);
```

Frees all the memory allocated for `T` and deletes `T`.

```
void ftab_SetDesc (ftab_Table *T, char *Desc);
```

Sets the `Desc` field of `T` to `Desc`.

```
void ftab_InitMatrix (ftab_Table *T, double x);
```

Initializes all the values in `T->Mat` to `x`.

```
typedef void (*ftab_CalcType) (ffam_Fam *fam, void *res, void *cho,
void *par, int LSize, int j,
int irow, int icol);
```

Type of function called by `ftab_MakeTables` to fill up the entry (`irow`, `icol`) in one or more tables of results. Typically, it computes p-values to be put in the appropriate table. It tests one generator of the family `fam`, using `res` to keep the tables of results, `cho` is used to choose the values of the varying parameters of the test as a function of the generator size `LSize`, of `j` and the other parameters, while `par` holds the fixed parameters of the test. This function is used internally by the tests.

```
void ftab_MakeTables (ffam_Fam *fam, void *res, void *cho, void *par,
ftab_CalcType Calc,
int Nr, int j1, int j2, int jstep);
```

This function calls `Calc(fam, res, cho, par, LSize, j, irow, icol)` on each of the first `Nr` generators of family `fam`, for `j` going from `j1` to `j2` by step of `jstep` (thus varying the sample size for a given generator). It uses `res` to keep the tables of results after all the tests have been done on the family, `cho` is used to choose the values of the varying parameters of the test as a function of the generator size and the other parameters, while `par` holds the fixed parameters of the test. Normally, `Calc` calls a test and places the results (e.g., p-values) in the entries of the appropriate tables.

Printing the tables

```
typedef enum {
ftab_Plain,                /* To print tables in plain text */
ftab_Latex                 /* To print tables in Latex format */
} ftab_StyleType;
```

The possible styles in which the tables of this module can be printed.

```
extern ftab_StyleType ftab_Style;
```

This environment variable determines the style in which all the tables of this module will be printed. The default value is `ftab_Plain`.

```
typedef enum {
ftab_NotInit,              /* Uninitialized */
ftab_pVal1,                /* One-sided p-value */
ftab_pVal2,                /* Two-sided p-value */
ftab_pLog10,               /* Logarithm of p-value in base 10 */
ftab_pLog2,                /* Logarithm of p-value in base 2 */
ftab_Integer,              /* Integer number */
ftab_Real,                 /* Real number */
ftab_String                /* String */
} ftab_FormType;
```

Possible formats that can be used to print the table entries. An appropriate format must be chosen before printing. Here, `ftab_pVal1` stands for a one-sided p-value (a number in the interval [0, 1]), printed only when it is near 0; `ftab_pVal2` stands for a two-sided p-value, printed when it is near 0 or near 1 (p-values near 1 are printed as $-p$ instead of $1 - p$). The other formats are self-evident.

```
extern double ftab_Suspectp;
```

Environment variable used in `ftab_PrintTable` and `ftab_PrintTable2` to determine which p-values should be printed in the table. When the format `ftab_pVal2` is used, only the p-values outside the interval [`ftab_Suspectp`, $1 -$ `ftab_Suspectp`] will be considered suspect and printed. The default value is 0.01.

```
extern int ftab_SuspectLog2p;
```

Environment variable used in `ftab_PrintTable` and `ftab_PrintTable2` to determine which p-values should be printed in the table, when using the format `ftab_pLog2`. If `ftab_SuspectLog2p` $= \sigma$, the p-values outside the interval $[1/2^\sigma, 1 - 1/2^\sigma]$ are considered suspect and are printed. The default value is 6.

```
void ftab_PrintTable (ftab_Table *T);
```

Prints the values `T->Mat[`i`][`j`]`, one value of i per line, for $0 \leq i <$ `T->Nr` and $0 \leq j <$ `T->Nc`.

If `Form = ftab_pVal1`, prints the entries as p-values for one-sided tests (prints only the ones close to 0, i.e., less than `ftab_Suspectp`). If `Form = ftab_pVal2`, prints the entries as p-values for two-sided tests (prints only those close to 0 or 1, i.e., less than `ftab_Suspectp` or larger than $1 -$ `ftab_Suspectp`). If the p-value $p <$ `ftab_Suspectp`, then print p as is. unless $p <$ `gofw_Epsilonp`, in which case `eps` will be printed (`\eps` when `ftab_Latex` style is chosen). If $p > 1 -$ `ftab_Suspectp`, then print $p - 1$. unless $p > 1 -$ `gofw_Epsilonp`, in which case `-eps` will be printed (`\epsm` when `ftab_Latex` style is chosen).

In the case where `Form = ftab_pLog10`, if $p \leq$ `ftab_Suspectp` it prints $\text{Round}(-\log_{10} p)$, else if $p \geq 1-$ `ftab_Suspectp` it prints $-\text{Round}(-\log_{10}(1-p))$, otherwise it prints nothing. In the case where `Form = ftab_pLog2`, if $p \leq 2^{-\sigma}$, where $\sigma =$ `ftab_SuspectLog2p`, it prints $\text{Round}(-\log_2 p)$, else if $p \geq 1 - 1/2^\sigma$ it prints $-\text{Round}(-\log_2(1-p))$, and otherwise it prints nothing.

If `Form = ftab_Integer`, it prints them as (rounded) integers. If `Form = ftab_Real`, it prints them as `double`'s. If `Form = ftab_String`, prints the string `T->Strings[s]` where `s = Round(T->Mat[i][j])`.

`void ftab_PrintTable2 (ftab_Table *T1, ftab_Table *T2, lebool ratioF);`

Similar to `ftab_PrintTable`, but prints two tables simultaneously, using two columns for each entry, for purposes of comparison. If the flag `ratioF` is `TRUE`, it prints the numbers of the first table in a first column, and the ratio of the numbers from the second table over the corresponding numbers from the first table in the second column. This is done for each element of the tables. If the flag `ratioF` is `FALSE`, the numbers from the second table will be printed as is.

fres

This module defines common structures used to keep the results of tests in the `f` modules. They are described in the detailed version of this guide.

The argument `res` of each testing function is a structure that can keep the test results, i.e. tables of p-values, ... This is useful if one wishes to do something else with the results or the information generated during a test. If one does not want to post-process or use the results after a test, it suffices to set the `res` argument to the `NULL` pointer. Then, the structure is created and deleted automatically inside the testing function. In any case, the tables of results will be printed automatically on standard output.

fcho

This module provides tools to choose the value of some parameters of a test as a function of the generator's *lsize* and other parameters, when a sequence of tests is applied to a family of generators. The sample size n, for example, is normally chosen by the functions in this module.

Choosing one parameter

```
typedef struct {
void *param;
double (*Choose) (void *param, long, long);
void (*Write) (void *param, long, long);
char *name;
} fcho_Cho;
```

This structure is used to choose and keep some of the parameters of a test (e.g. the sample size) as a function of the *lsize* of the generator and some other parameters. The function `Choose` computes a parameter which appear as argument in a test function from a `s` module which is applied on a family of generators. The parameters of `Choose` itself are kept in `param`. The function `Write` writes some information about the parameters and the `Choose` function. The string `name` is the name of the parameter that is computed by `Choose`.

```
long fcho_ChooseParamL (fcho_Cho *cho, long min, long max, long i, long j);
```

This function chooses a parameter (most often the sample size) by calling `cho->Choose (cho->param, i, j)`. If the chosen parameter is smaller than `min` or larger than `max`, the function returns -1, otherwise it returns the chosen parameter.

Choosing two parameters

```
typedef struct {
fcho_Cho *Chon;
fcho_Cho *Chop2;
} fcho_Cho2;
```

```
fcho_Cho2 * fcho_CreateCho2 (fcho_Cho *Chon, fcho_Cho *Chop2);
```

This function creates and returns a structure to hold the two sub-structures `Chon` and `Chop2`. It will not create the memory for `Chon` or `Chop2` themselves, which must have been created before. These two are used when choosing the sample size n and another parameter $p2$ in the same test. For some tests, both n and the other parameter can be varied as the sample size; in this case, one of these two arguments may be a NULL pointer. For some other tests, both parameters must be chosen.

```
void fcho_DeleteCho2 (fcho_Cho2 *cho);
```

Frees the memory allocated by `fcho_CreateCho2`, but not the memory reserved for the two fields `Chon` and `Chop2`.

Choosing the sample size

```
typedef double (*fcho_FuncType) (double);
```

This kind of function is used to compute a parameter for a test depending on the *lsize* of the generator and the other parameters of the test.

```
double fcho_Linear (double x);
```

Returns x.

```
double fcho_LinearInv (double x);
```

Returns $1/x$.

```
double fcho_2Pow (double x);
```

Returns 2^x.

```
fcho_Cho * fcho_CreateSampleSize (double a, double b, double c,
fcho_FuncType F, char *name);
```

Creates and returns a structure `fcho_Cho` which is used normally to compute the sample size of a test. Given the two arguments i and j of the `Choose` function in `fcho_Cho`, the function will return the value of $F(a * i + b * j + c)$. The string `name` is the name of the variable that is being computed by `Choose`. One can choose both `F` and `name` as the `NULL` pointers, in which case these two will be set to default variables `fcho_2Pow` and "n". Then the sample size of the test will be chosen as $n = 2^{a*i+b*j+c}$, where i is the *lsize* of the generator being tested.

```
void fcho_DeleteSampleSize (fcho_Cho *cho);
```

Frees the memory allocated by `fcho_CreateSampleSize`.

Choosing the number of bits

Each generator returns a given number of bits of resolution in its output values. The resolution usually increases with the *lsize* of the generator. It would be meaningless to apply a test that requires more bits of resolution than the generator can deliver, since the extra bits will be akin to noise.

Many of the tests depend on the two arguments r and s. The argument r is the number of (most significant) bits dropped from each random number outputted by the generator, while s is the number of bits from each random number that are kept and used in the test.

`int fcho_Chooses (int r, int s, int resol);`

This function returns the number of bits s_1 that are effectively used in a test when the test function depends on s. `resol` is the resolution of the generator being tested. If $r + s \leq$ `resol`, then the function returns s unchanged. Otherwise, the function returns $s_1 =$ `resol` $- r$. When $s_1 \leq 0$, then obviously the test should not be done.

fmultin

This module applies multinomial tests, from the module `smultin`, to a family of generators of different sizes.

```
#include <testu01/gdef.h>
#include <testu01/ftab.h>
#include <testu01/ffam.h>
#include <testu01/fres.h>
#include <testu01/fcho.h>
#include <testu01/smultin.h>


extern long fmultin_Maxn;
```

Upper bound on n. A test is called only when n does not exceed this value. Default value: 2^{24}.

Choosing the parameters

Given the sample size n, the following functions determines how the number of cells k is chosen according to different criteria. This will determine the other varying parameter of the tests besides n. The returned structure must be passed as the second member in the argument `cho` of the test, the first member of `cho` being always the function that chooses the sample size n.

```
fcho_Cho * fmultin_CreateEC_DT (long N, int t, double EC);
fcho_Cho * fmultin_CreateEC_2HT (long N, int t, double EC);
fcho_Cho * fmultin_CreateEC_2L (long N, double EC);
fcho_Cho * fmultin_CreateEC_T (long N, double EC);
```

Given the number of replications N, the sample size n, these functions choose the number of cells k so that the expected number of collisions is approximately EC, i.e. $EC \approx Nn^2/2k$. Given the dimension t, the function `fmultin_CreateEC_DT` chooses the one-dimensional interval d so that $k = d^t$, and `fmultin_CreateEC_2HT` chooses $d = 2^h$ so that $k = 2^{ht}$ (d is a power of 2 with h integer). These two cases apply to the tests `fmultin_Serial1` and `fmultin_SerialOver1`. The function `fmultin_CreateEC_2L` chooses L so that $k = 2^L$. This case applies to the tests `fmultin_SerialBits1` and `fmultin_SerialBitsOver1`. The function `fmultin_CreateEC_T` chooses t so that $k = t!$. This case applies to the test `fmultin_Permut1`.

```
void fmultin_DeleteEC (fcho_Cho *cho);
```

Frees the memory allocated by the create functions `fmultin_CreateEC...`

```
fcho_Cho * fmultin_CreateDens_DT (int t, double R);
fcho_Cho * fmultin_CreateDens_2HT (int t, double R);
fcho_Cho * fmultin_CreateDens_2L (double R);
fcho_Cho * fmultin_CreateDens_T (double R);
```

Similar to `fmultin_CreateEC...`, but the parameters are chosen so that the density (the number of points per cell) is approximately $R \approx n/k$.

```
void fmultin_DeleteDens (fcho_Cho *cho);
```

Frees the memory allocated by `fmultin_CreateDens...`

```
fcho_Cho * fmultin_CreatePer_DT (int t, double R);
fcho_Cho * fmultin_CreatePer_2HT (int t, double R);
fcho_Cho * fmultin_CreatePer_2L (double R);
fcho_Cho * fmultin_CreatePer_T (double R);
```

Similar to `fmultin_CreateEC...`, but the parameters are chosen so that the number of cells k is approximately R times the period length of the generator, i.e., $k \approx R\,2^{lsize}$.

```
void fmultin_DeletePer (fcho_Cho *cho);
```

Frees the memory allocated by `fmultin_CreatePer...`

The tests

```
void fmultin_Serial1 (ffam_Fam *fam, smultin_Param *par,
fmultin_Res *res, fcho_Cho2 *cho,
long N, int r, int t, lebool Sparse,
int Nr, int j1, int j2, int jstep);
```

Applies the same tests as in `smultin_Multinomial` with `smultin_GenerCellSerial`, with parameters N and r, in dimension t and with the same parameter `Sparse`, for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. The sample size n is chosen by the function `cho->Chon` while the number of cells k is chosen by `cho->Chop2`. This last must have been initialized by one of the method for choosing cells described above. Whenever n exceeds `fmultin_Maxn` or k exceeds `smultin_Maxk`, the test is not run.

```
void fmultin_SerialOver1 (ffam_Fam *fam, smultin_Param *par,
fmultin_Res *res, fcho_Cho2 *cho,
long N, int r, int t, lebool Sparse,
int Nr, int j1, int j2, int jstep);
```

Similar to `fmultin_Serial1`, except that it applies the tests in the function `smultin_MultinomialOver`.

```
void fmultin_SerialBits1 (ffam_Fam *fam, smultin_Param *par,
fmultin_Res *res, fcho_Cho2 *cho,
long N, int r, int s, lebool Sparse,
int Nr, int j1, int j2, int jstep);
```

Similar to `fmultin_Serial1`, except that it applies the tests in the function `smultin_MultinomialBits`.

```
void fmultin_SerialBitsOver1 (ffam_Fam *fam, smultin_Param *par,
fmultin_Res *res, fcho_Cho2 *cho,
long N, int r, int s, lebool Sparse,
int Nr, int j1, int j2, int jstep);
```

Similar to `fmultin_SerialBits1`, except that it applies the tests in the function `smultin_MultinomialBitsOver`.

```
void fmultin_Permut1 (ffam_Fam *fam, smultin_Param *par,
fmultin_Res *res, fcho_Cho2 *cho,
long N, int r, lebool Sparse,
int Nr, int j1, int j2, int jstep);
```

Similar to `fmultin_Serial1` except that it uses `smultin_GenerCellPermut` to generate the cell numbers. Here, d is unused and t is chosen as a function of k by $k = t!$.

fnpair

This module applies close-pairs tests from the module `snpair` to a family of generators of different sizes.

```
#include <testu01/gdef.h>
#include <testu01/ffam.h>
#include <testu01/fres.h>
#include <testu01/fcho.h>
#include <testu01/snpair.h>


extern long fnpair_Maxn;
```

Upper bound on n. When n exceeds its limit value, the test is not done. Default value: $n = 2^{22}$.

Choosing the parameter m

One may choose to vary the parameter m as a function of the sample size in the test `fnpair_ClosePairs1`. In that case, one must create an appropriate Choose function, pass it as a member of the argument `cho` to the test, and call the test with a negative m to signal that this is the case. If one wants to do the test with a fixed m, one has but to give the given $m > 0$ as a parameter to the test.

```
fcho_Cho *fnpair_CreateM1 (int maxm);
```

Given the number of replications N and the sample size n, the parameter m in `snpair_ClosePairs` will be chosen by the relation $m = \min \left\{ \texttt{maxm}, \sqrt{n/\left(4\sqrt{N}\right)} \right\}$. If this method of choosing m is used, then the returned structure must be passed as the second pointer in the argument `cho` of the test and the argument m of the test `fnpair_ClosePairs1` must be negative.

```
void fnpair_DeleteM1 (fcho_Cho * cho);
```

Frees the memory allocated by `fnpair_CreateM1`.

Applying the tests

```
void fnpair_ClosePairs1 (ffam_Fam *fam, fnpair_Res1 *res, fcho_Cho2 *cho,
long N, int r, int t, int p, int m,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `snpair_ClosePairs` with parameters N and r, in dimension t, with L_p norm, sample size $n =$ `cho->Chon->Choose(param,` i`,` j`)`, for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. The parameters in `param` were set at the creation of `cho` and i is the lsize of the generator being tested. If $m > 0$, it will be used as

is in the test. If $m < 0$, it will be chosen by $m =$ `cho->Chop2->Choose(param, N, n)`. When n exceeds `fnpair_Maxn` or $n < 4m^2N^{1/2}$, the test is not done.

```
void fnpair_Bickel1 (ffam_Fam *fam, fnpair_Res1 *res, fcho_Cho *cho,
long N, int r, int t, int p, lebool Torus,
int Nr, int j1, int j2, int jstep);
```

Similar to `fnpair_ClosePairs1` but with `snpair_BickelBreiman`. There is no parameter m in this test.

```
void fnpair_BitMatch1 (ffam_Fam *fam, fnpair_Res1 *res, fcho_Cho *cho,
long N, int r, int t,
int Nr, int j1, int j2, int jstep);
```

Similar to `fnpair_Bickel1` but with `snpair_ClosePairsBitMatch`.

fknuth

This module applies tests from the module `sknuth` to families of generators of different sizes, and prints tables of the corresponding p-values.

```
#include <testu01/gdef.h>
#include <testu01/ffam.h>
#include <testu01/fres.h>
#include <testu01/fcho.h>
```

```
extern long fknuth_Maxn;
```

Upper bound on n. A test is called only when n does not exceed this value. Default value: 2^{22}.

Applying the tests

```
void fknuth_Serial1 (void);
```

This is equivalent to calling `fmultin_Serial1` with `Sparse = FALSE`, `NbDelta = 1`, and `ValDelta[0] = 1`.

```
void fknuth_SerialSparse1 (void);
```

This is equivalent to calling `fmultin_Serial1` with `Sparse = TRUE`, `NbDelta = 1`, and `ValDelta[0] = 1`.

```
void fknuth_Collision1 (void);
```

This is equivalent to calling `fmultin_Serial1` with `Sparse = TRUE, NbDelta = 1`, and `ValDelta[0] = -1`.

```
void fknuth_Permutation1 (void);
```

This is equivalent to calling `fmultin_Permut1` with `Sparse = FALSE, NbDelta = 1`, and `ValDelta[0] = 1`.

```
void fknuth_CollisionPermut1 (void);
```

This is equivalent to calling `fmultin_Permut1` with `Sparse = TRUE`, `NbDelta = 1`, and `ValDelta[0] = -1`.

```
void fknuth_Gap1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, double Alpha, double Beta,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `sknuth_Gap` with parameters `N`, n, `r, Alpha, Beta`, for sample size n chosen by the function `cho->Choose(param, i, j)`, for the first `Nr` generators of family

`fam`, for j going from `j1` to `j2` by steps of `jstep`. The parameters in `param` were set at the creation of `cho` and i is the lsize of the generator being tested. When n exceeds `fknuth_Maxn`, the test is not run.

```
void fknuth_SimpPoker1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, int d, int k,
int Nr, int j1, int j2, int jstep);
```

Similar to `fknuth_Gap` but with `sknuth_SimpPoker`.

```
void fknuth_CouponCollector1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, int d,
int Nr, int j1, int j2, int jstep);
```

Similar to `fknuth_Gap` but with `sknuth_CouponCollector`.

```
void fknuth_Run1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, lebool Up, lebool Indep,
int Nr, int j1, int j2, int jstep);
```

Similar to `fknuth_Gap` but with `sknuth_RunIndep` if `Indep = TRUE`, and `sknuth_Run` otherwise.

```
void fknuth_MaxOft1 (ffam_Fam *fam, fknuth_Res1 *res, fcho_Cho *cho,
long N, int r, int d, int t,
int Nr, int j1, int j2, int jstep);
```

Similar to `fknuth_Gap` but with `sknuth_MaxOft`.

fmarsa

This module applies tests from the module `smarsa` to a family of generators of different sizes.

```
#include <testu01/ffam.h>
#include <testu01/fres.h>
#include <testu01/fcho.h>
```

```
extern long fmarsa_Maxn, fmarsa_MaxL;
```

Upper bound on the sample size n in `fmarsa_BirthdayS1` and on the dimension $L \times L$ of the matrices in `fmarsa_MatrixR1`. A test is called only when n and L do not exceed their limit value. Default values: $n = 2^{24}$ and $L = 2^{12}$.

Choosing the parameters

```
fcho_Cho * fmarsa_CreateBirthEC (long N, int t, double EC);
```

Given the number of replications N, the dimension t and the sample size n, the parameter d in `smarsa_BirthdaySpacings` will be chosen so that the expected number of collisions is approximately EC, i.e. $EC = Nn^3/4d^t$. The returned structure must be passed as the second pointer in the argument `cho` of the test.

```
void fmarsa_DeleteBirthEC (fcho_Cho *cho);
```

Frees the memory allocated by `fmarsa_CreateBirthEC`.

The tests

```
void fmarsa_SerialOver1 (void);
```

This is equivalent to calling `fmultin_SerialOver1` with `Sparse = FALSE`, `NbDelta = 1`, and `ValDelta[0] = 1`.

```
void fmarsa_CollisionOver1 (void);
```

This is equivalent to calling `fmultin_SerialOver1` with `Sparse = TRUE`, `NbDelta = 1`, and `ValDelta[0] = -1`.

```
void fmarsa_BirthdayS1 (ffam_Fam *fam, fres_Poisson *res, fcho_Cho2 *cho,
long N, int r, int t, int p,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `smarsa_BirthdaySpacings` with parameters `N`, n, `r`, d, `t` and `p` for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. The sample

size n and the one-dimensional interval d are chosen from $n =$ `cho->Chon->Choose(param, i, j)` and $d =$ `cho->Chop2->Choose(param, n, 0)`. `Chon` and `Chop2` must have been created before. When n exceeds `fmarsa_Maxn` or $k = d^t$ exceeds `fmarsa_Maxk`, the test is not run.

```
void fmarsa_MatrixR1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho2 *cho,
long N, long n, int r, int s, int L,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `smarsa_MatrixRank` with parameters `N`, n, `r`, `s`, L for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. Either or both of n and L can be varied as the sample size, by passing a negative value as an argument of the function. One must then create the corresponding function `cho->Chon` or `cho->Chop2` before calling the test. One will have either $n =$ `cho->Chon->Choose(param, i, j)`, or $L =$ `cho->Chop2->Choose(param, i, j)`. A positive value for n or L will be used as is by the test. When n exceeds `fmarsa_Maxn` or L exceeds `fmarsa_MaxL`, the test is not run. Only square matrices of order $L \times L$ are considered.

```
void fmarsa_GCD1 (ffam_Fam *fam, fmarsa_Res2 *res, fcho_Cho *cho,
long N, int r, int s,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `smarsa_GCD` with parameters `N`, n, `r`, `s` for sample size n chosen by $n =$ `cho->Choose(param, i, j)`, for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. When n exceeds `fmarsa_Maxn`, the test is not run.

fvaria

This module applies tests from the module `svaria` to a family of generators of different sizes.

```
#include <testu01/ffam.h>
#include <testu01/fres.h>
#include <testu01/fcho.h>


extern long fvaria_MaxN;
extern long fvaria_Maxn;
extern long fvaria_Maxk;
extern long fvaria_MaxK;
```

Upper bounds on N, n, k and K. When N, n, k or K exceed their limit value, the test is not done. Default values: $N = 2^{22}$, $n = 2^{22}$, $k = 2^{22}$ and $K = 2^{22}$.

The tests

```
void fvaria_SampleMean1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long n, int r,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `svaria_SampleMean` with parameters N, `n` and `r` for sample size N chosen by the function `cho->Choose(param, i, j)`, for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. The parameters in `param` were set at the creation of `cho` and i is the lsize of the generator being tested. When N exceeds `fvaria_MaxN`, the test is not done.

```
void fvaria_SampleCorr1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, int k,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `svaria_SampleCorr` with parameters `N`, n, `r` and `k` for sample size n chosen by the function `cho->Choose(param, i, j)`, for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. The parameters in `param` were set at the creation of `cho` and i is the lsize of the generator being tested. When n exceeds `fvaria_Maxn`, the test is not done.

```
void fvaria_SampleProd1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, int t,
int Nr, int j1, int j2, int jstep);
```

Similar to `fvaria_SampleCorr1` but with `svaria_SampleProd`.

```
void fvaria_SumLogs1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r,
int Nr, int j1, int j2, int jstep);
```

Similar to `fvaria_SampleCorr1` but with `svaria_SumLogs`.

```
void fvaria_SumCollector1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, double g,
int Nr, int j1, int j2, int jstep);
```

Similar to `fvaria_SampleCorr1` but with `svaria_SumCollector`.

```
void fvaria_Appearance1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, int s, int L,
int Nr, int j1, int j2, int jstep);
```

Similar to `fvaria_SampleCorr1` but with `svaria_AppearanceSpacings` and with K as the varying sample size.

```
void fvaria_WeightDistrib1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho2 *cho,
long N, long n, int r, long k,
double alpha, double beta,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `svaria_WeightDistrib` with parameters `N`, `n`, `r`, `k`, `alpha`, and `beta` for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. Either or both of `n` and `k` can be varied as the sample size, by passing a negative value as argument of the function. One must then create the corresponding function `cho->Chon` or `cho->Chop2` before calling the test. One will have either `n` = `cho->Chon->Choose(param, i, j)`, or `k` = `cho->Chop2->Choose(param, i, j)` or both. A positive value for `n` or `k` will be used as is by the test. When `n` exceeds `fvaria_Maxn` or `k` exceeds `fvaria_Maxk`, the test is not done.

fwalk

This module applies random-walk tests from the module `swalk` to a family of generators of different sizes.

```
#include <testu01/ffam.h>
#include <testu01/fres.h>
#include <testu01/fcho.h>


extern long fwalk_Maxn;
extern long fwalk_MaxL;
extern double fwalk_MinMu;
```

Upper bounds on n, L and lower bound on Mu. When n, L or Mu exceed their limit value, the tests are not done. Default values: $n = 2^{22}$, $L = 2^{22}$ and Mu $= 2^{-22}$.

The tests

```
void fwalk_RWalk1 (ffam_Fam *fam, fwalk_Res1 *res, fcho_Cho2 *cho,
long N, long n, int r, int s, long L,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `swalk_RandomWalk1` with parameters `N`, `n`, `r`, `s`, and `L` for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. Either or both of `n` and `L` can be varied as the sample size, by passing a negative value as argument of the function. One must then create the corresponding function `cho->Chon` or `cho->Chop2` before calling the test. One will have either `n` = `cho->Chon->Choose(param, i, j)`, or `L` = `cho->Chop2->Choose(param, i, j)` or both. A positive value for `n` or `L` will be used as is by the test. When `n` exceeds `fwalk_Maxn` or `L` exceeds `fwalk_MaxL`, the test is not run.

```
void fwalk_VarGeoP1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho2 *cho,
long N, long n, int r, double Mu,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `swalk_VarGeoP` with parameters `N, n, r, Mu` for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. Either or both of n and `Mu` can be varied as the sample size, by passing a negative value as argument of the function. One must then create the corresponding function `cho->Chon` or `cho->Chop2` before calling the test. One will have either n = `cho->Chon->Choose(param, i, j)`, or `Mu` = `cho->Chop2->Choose(param, i, j)`. A positive value for n or `Mu` will be used as is by the test. When n exceeds `fwalk_Maxn` or `Mu` is less than `fwalk_MinMu`, the test is not done.

```
void fwalk_VarGeoN1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho2 *cho,
long N, long n, int r, double Mu,
int Nr, int j1, int j2, int jstep);
```

Similar to `fwalk_VarGeoP1` but with `swalk_VarGeoN`.

fspectral

This module applies spectral tests from the module `sspectral` to a family of generators of different sizes.

```
#include <testu01/ffam.h>
#include <testu01/fres.h>
#include <testu01/fcho.h>


extern long fspectral_MaxN;
```

Upper bound on N. When N exceeds its limit value, the tests are not done. Default value: $N = 2^{22}$.

The tests

```
void fspectral_Fourier3 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
int k, int r, int s,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `sspectral_Fourier3` with parameters N, `k`, `r`, and `s` for sample size N chosen by $N =$ `cho->Choose(param, i, j)`, for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. The parameters in `param` were set at the creation of `cho` and i is the lsize of the generator being tested. When N exceeds `fspectral_MaxN`, the test is not done.

fstring

This module applies tests from the module `sstring` to a family of generators of different lsizes. The results are placed in tables of p-values.

```
#include <testu01/ffam.h>
#include <testu01/fres.h>
#include <testu01/fcho.h>
```

```
extern long fstring_Maxn, fstring_MaxL;
```

Upper bound on n and L. A test is called only when n and L do not exceed their limit value. Default values: $n = 2^{22}$ and $L = 2^{20}$.

The tests

```
void fstring_Period1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, int s,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `sstring_PeriodsInStrings` with parameters `N`, `r`, `s` and sample size $n =$ `cho->Choose(param, i, j)`, for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. The parameters in `param` were set at the creation of `cho` and i is the lsize of the generator being tested. When n exceeds `fstring_Maxn`, the test is not run.

```
void fstring_Run1 (ffam_Fam *fam, fstring_Res2 *res, fcho_Cho *cho,
long N, int r, int s,
int Nr, int j1, int j2, int jstep);
```

Similar to `fstring_Period1` but with `sstring_Run`.

```
void fstring_AutoCor1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho *cho,
long N, int r, int s, int d,
int Nr, int j1, int j2, int jstep);
```

Similar to `fstring_Period1` but with `sstring_AutoCor`.

```
void fstring_LongHead1 (ffam_Fam *fam, fstring_Res1 *res, fcho_Cho2 *cho,
long N, long n, int r, int s, long L,
int Nr, int j1, int j2, int jstep);
```

This function calls the test `sstring_LongestHeadRun` with parameters `N`, `r`, `s` for the first `Nr` generators of family `fam`, for j going from `j1` to `j2` by steps of `jstep`. If `s` is greater than the resolution of the generator, it will be reset to the resolution of the generator. Either or both of n and L can be varied as the sample size, by passing a negative value as an argument of the function. One must then create the corresponding function `cho->Chon` or `cho->Chop2` before calling the test. One will have either $n =$ `cho->Chon->Choose(param, i, j)`, or $L =$

`cho->Chop2->Choose(param, i, j)` (or both). A positive value for n or L will be used as is by the test. When n exceeds `fstring_Maxn` or L exceeds `fstring_MaxL`, the test is not run.

```
void fstring_HamWeight1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho2 *cho,
long N, long n, int r, int s, long L,
int Nr, int j1, int j2, int jstep);
```

Similar to `fstring_LongHead1` but with `sstring_HammingWeight`.

```
void fstring_HamWeight2 (ffam_Fam *fam, fres_Cont *res, fcho_Cho2 *cho,
long N, long n, int r, int s, long L,
int Nr, int j1, int j2, int jstep);
```

Similar to `fstring_LongHead1` but with `sstring_HammingWeight2`.

```
void fstring_HamCorr1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho2 *cho,
long N, long n, int r, int s, long L,
int Nr, int j1, int j2, int jstep);
```

Similar to `fstring_LongHead1` but with `sstring_HammingCorr`.

```
void fstring_HamIndep1 (ffam_Fam *fam, fres_Cont *res, fcho_Cho2 *cho,
long N, long n, int r, int s, long L,
int Nr, int j1, int j2, int jstep);
```

Similar to `fstring_LongHead1` but with `sstring_HammingIndep`.

References

[1] N. S. Altman. Bit-wise behavior of random number generators. *SIAM Journal on Scientific and Statistical Computing*, 9(5):941–949, 1988.

[2] N. H. Anderson and D. M. Titterington. A comparison of two statistics for detecting clustering in one dimension. *Journal of Statistical Computation and Simulation*, 53:103–125, 1995.

[3] E. Barker and J. Kelsey. Recommendation for random number generation using deterministic random bit generators. SP-800-90, U.S. DoC/National Institute of Standards and Technology, 2006. See `http://csrc.nist.gov/publications/nistpubs/`.

[4] E. R. Berlekamp. *Algebraic coding theory.* Aegean Park Press, Laguna Hills, CA, USA, 1984.

[5] P. J. Bickel and L. Breiman. Sums of functions of nearest neighbor distances, moment bounds, limit theorems and a goodness of fit test. *The Annals of Probability*, 11(1):185–214, 1983.

[6] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation.* Springer-Verlag, New York, second edition, 1987.

[7] R. P. Brent. On the periods of generalized fibonacci recurrences. *Mathematics of Computation*, 63(207):389–401, 1994.

[8] R. P. Brent. Note on Marsaglia's xorshift random number generators. *Journal of Statistical Software*, 11(5):1–4, 2004. See `http://www.jstatsoft.org/v11/i05/brent.pdf`.

[9] D. G. Carta. Two fast implementations of the "minimal standard" random number generator. *Communications of the ACM*, 33(1):87–88, 1990.

[10] G. D. Carter. *Aspects of Local Linear Complexity.* PhD thesis, University of London, 1989.

[11] P. D. Coddington. Analysis of random number generators using Monte Carlo simulation. *International Journal of Modern Physics*, C 5:547, 1994.

[12] J. S. Coron and D. Naccache. An accurate evaluation of Maurer's universal test. In S. E. Tavares and H. Meijer, editors, *Selected Areas in Cryptography, Proceedings of the 5th Annual International Workshop, SAC'98*, volume 1556 of *Lecture Notes in Computer Science*, pages 57–71. Springer-Verlag, 1999.

[13] R. Couture and P. L'Ecuyer. On the lattice structure of certain linear congruential sequences related to AWC/SWB generators. *Mathematics of Computation*, 62(206):798–808, 1994.

[14] R. Couture and P. L'Ecuyer. Linear recurrences with carry as random number generators. In *Proceedings of the 1995 Winter Simulation Conference*, pages 263–267, 1995.

[15] R. Couture and P. L'Ecuyer. Distribution properties of multiply-with-carry random number generators. *Mathematics of Computation*, 66(218):591–607, 1997.

[16] N. Cressie. On the logarithms of high-order spacings. *Biometrika*, 63(2):343–355, 1976.

[17] N. Cressie. An optimal statistic based on higher-order gaps. *Biometrika*, 66:619–627, 1979.

[18] N. Cressie. Asymptotic distribution of the scan statistic under uniformity. *Annals of Probability*, 9:828–840, 1980.

[19] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer Verlag, New York, 2002. See `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/`.

[20] A. De Matteis and S. Pagnutti. A class of parallel random number generators. *Parallel Computing*, 13:193–198, 1990.

[21] G. E. del Pino. On the asymptotic distribution of k-spacings with applications to goodness-of-fits tests. *Annals of Statistics*, 7:1058–1065, 1979.

[22] L.-Y. Deng and D. K. J. Lin. Random number generation for the new century. *The American Statistician*, 54(2):145–150, 2000.

[23] L.-Y. Deng and H. Xu. Design, search, and implementation of high-dimensional, efficient, long-cycle, and portable uniform random variate generator. Technical report, Department of Statistics, University of California at Los Angeles, 2002. Preprint #327.

[24] E. J. Dudewicz and E. C. van der Meulen. Entropy-based tests of uniformity. *Journal of the American Statistical Association*, 76(376):967–974, 1981.

[25] E. J. Dudewicz, E. C. van der Meulen, M. G. SriRam, and N. K. W. Teoh. Entropy-based random number evaluation. *American Journal of Mathematical and Management Sciences*, 15:115–153, 1995.

[26] M. Dworkin. Recommandations for block cipher modes of operation. SP 800-38a, U.S. DoC/National Institute of Standards and Technology, 2001. See `http://csrc.nist.gov/CryptoToolkit/modes/`.

[27] J. Eichenauer and J. Lehn. On the structure of quadratic congruential sequences. *Manuscripta Mathematica*, 58:129–140, 1987.

[28] J. Eichenauer-Herrmann. Inversive congruential pseudorandom numbers: A tutorial. *International Statistical Reviews*, 60:167–176, 1992.

[29] J. Eichenauer-Herrmann. Statistical independence of a new class of inversive congruential pseudorandom numbers. *Mathematics of Computation*, 60:375–384, 1993.

[30] J. Eichenauer-Herrmann. Modified explicit inversive congruential pseudorandom numbers with power-of-two modulus. *Statistics and Computing*, 6:31–36, 1996.

[31] J. Eichenauer-Herrmann and H. Grothe. A new inversive congruential pseudorandom number generator with power of two modulus. *ACM Transactions on Modeling and Computer Simulation*, 2(1):1–11, 1992.

[32] J. Eichenauer-Herrmann and E. Herrmann. Compound cubic congruential pseudorandom numbers. *Computing*, 59:85–90, 1997.

[33] J. Eichenauer-Herrmann, E. Herrmann, and S. Wegenkittl. A survey of quadratic and inversive congruential pseudorandom numbers. In P. Hellekalek, G. Larcher, H. Niederreiter, and P. Zinterhof, editors, *Monte Carlo and Quasi-Monte Carlo Methods 1996*, volume 127 of *Lecture Notes in Statistics*, pages 66–97, New York, 1998. Springer.

[34] J. Eichenauer-Herrmann and K. Ickstadt. Explicit inversive congruential pseudorandom numbers with power of two modulus. *Mathematics of Computation*, 62(206):787–797, 1994.

[35] F. Emmerich. Equidistribution properties of quadratic congruential pseudorandom numbers. *Journal of Computational and Applied Mathematics*, 79(2):207–214, 1997.

[36] E. D. Erdmann. Empirical tests of binary keystreams. Master's thesis, Department of Mathematics, Royal Holloway and Bedford New College, University of London, 1992.

[37] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382–3384, 1992.

[38] G. S. Fishman. *Principles of Discrete Event Simulation*. Wiley, New York, 1978.

[39] G. S. Fishman. Multiplicative congruential random number generators with modulus 2^β: An exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$. *Mathematics of Computation*, 54(189):331–344, Jan 1990.

[40] G. S. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*. Springer Series in Operations Research. Springer-Verlag, New York, 1996.

[41] G. S. Fishman and L. S. Moore III. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM Journal on Scientific and Statistical Computing*, 7(1):24–45, 1986.

[42] A. Fog. Chaotic random number generators with random cycle lengths. available at `http://www.agner.org/random/theory/`, 2001.

[43] A. Földes. The limit distribution of the length of the longest head run. *Period Math. Hung.*, 10(4):301–310, 1979.

[44] M. Fushimi. Increasing the orders of equidistribution of the leading bits of the Tausworthe sequence. *Information Processing Letters*, 16:189–192, 1983.

[45] M. Fushimi. Random number generation with the recursion $x_t = x_{t-3p} \oplus x_{t-3q}$. *Journal of Computational and Applied Mathematics*, 31:105–118, 1990.

[46] I. J. Good. The serial test for sampling numbers and other tests for randomness. *Proceedings of the Cambridge Philos. Society*, 49:276–284, 1953.

[47] L. Gordon, M. F. Schilling, and S. Waterman. An extreme value theory for long head runs. *Probability Theory and Related Fields*, 72:279–287, 1986.

[48] J. Granger-Piché. Générateurs pseudo-aléatoires combinant des récurrences linéaires et non linéaires. Master's thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, 2001.

[49] R. E. Greenwood. Coupon collector's test for random digits. *Math. Tables and other Aids to Computation*, 9:1–5, 224, 229, 1955.

[50] L. J. Guibas and A. M. Odlyzko. Periods in strings. *Journal of Combinatorial Theory, Series A*, 30:19–42, 1981.

[51] P. Hall. On powerful distributional tests based on sample spacings. *Journal of Multivariate Analysis*, 19:201–224, 1986.

[52] P. Hellekalek and S. Wegenkittl. Empirical evidence concerning AES. *ACM Transactions on Modeling and Computer Simulation*, 13(4):322–333, 2003.

[53] B. L. Holian, O. E. Percus, T. T. Warnock, and P. A. Whitlock. Pseudorandom number generator for massively parallel molecular-dynamics simulations. *Physical Review E*, 50(2):1607–1615, 1994.

[54] W. Hörmann and G. Derflinger. A portable random number generator well suited for the rejection method. *ACM Transactions on Mathematical Software*, 19(4):489–495, 1993.

[55] F. James. RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher. *Computer Physics Communications*, 79:111–114, 1994.

[56] S. R. Jammalamadaka, X. Zhou, and R. C. Tiwari. Asymptotic efficiencies of spacing tests for goodness of fits. *Metrika*, 36:355–377, 1989.

[57] B. Jenkins. ISAAC. In Dieter Gollmann, editor, *Fast Software Encryption, Proceedings of the Third International Workshop, Cambridge, UK*, volume 1039 of *Lecture Notes in Computer Science*, pages 41–49. Springer-Verlag, 1996. Available at `http://burtleburtle.net/bob/rand/isaacafa.html`.

[58] Z. A. Karian and E. J. Dudewicz. *Modern Statistical Systems and GPSS Simulation*. Computer Science Press, W. H. Freeman, New York, 1991.

[59] M. G. Kendall and B. Babington-Smith. Randomness and other random sampling numbers. *Journal of the Royal Statistical Society*, 101:147–166, 1938.

[60] M. G. Kendall and B. Babington-Smith. Second paper on random sampling numbers. *Journal of the Royal Statistical Society Supplement*, 6:51–61, 1939.

[61] W. O. Kermack and A. G. Kendrick. Tests for randomness in a series of numerical observations. *Proc. of the Royal Society of Edinburgh*, 57:228–240, 1937.

[62] S. Kirkpatrick and E. Stoll. A very fast shift-register sequence random number generator. *Journal of Computational Physics*, 40:517–526, 1981.

[63] P. Kirschenhofer, H. Prodinger, and W. Szpankowski. Digital search trees again revisited: The internal path length perspective. *SIAM Journal on Computing*, 23:598–616, 1994.

[64] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., second edition, 1981.

[65] D. E. Knuth. *The Art of Computer Programming, Vol. 1*. Addison-Wesley, Reading, Mass., third edition, 1997.

[66] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., third edition, 1998.

[67] C. Koç. Recurring-with-carry sequences. *Journal of Applied Probability*, 32:966–971, 1995.

[68] M. Kuo and J. S. Rao. Limit theory and efficiencies for tests based on higher order spacings. In *"Proceedings of the Indian Statistical Institute Golden Jubilee International Conference on Statistics: Applications and New Directions*, pages 333–352. Statistical Publishing Society, Calcuta, 1981.

[69] J. C. Lagarias. Pseudorandom numbers. *Statistical Science*, 8(1):31–39, 1993.

[70] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, second edition, 1991.

[71] P. L'Ecuyer. Efficient and portable 32-bit random variate generators. In *Proceedings of the 1986 Winter Simulation Conference*, pages 275–277, 1986.

[72] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–749 and 774, 1988. See also the correspondence in the same journal, 32, 8 (1989) 1019–1024.

[73] P. L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.

[74] P. L'Ecuyer. Testing random number generators. In *Proceedings of the 1992 Winter Simulation Conference*, pages 305–313. IEEE Press, Dec 1992.

[75] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.

[76] P. L'Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):816–822, 1996.

[77] P. L'Ecuyer. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213):203–213, 1996.

[78] P. L'Ecuyer. Tests based on sum-functions of spacings for uniform random numbers. *Journal of Statistical Computation and Simulation*, 59:251–269, 1997.

[79] P. L'Ecuyer. Random number generators and empirical tests. In P. Hellekalek, G. Larcher, H. Niederreiter, and P. Zinterhof, editors, *Monte Carlo and Quasi-Monte Carlo Methods 1996*, volume 127 of *Lecture Notes in Statistics*, pages 124–138. Springer-Verlag, New York, 1998.

[80] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[81] P. L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.

[82] P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225):261–269, 1999.

[83] P. L'Ecuyer. Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the 2001 Winter Simulation Conference*, pages 95–105, Pistacaway, NJ, 2001. IEEE Press.

[84] P. L'Ecuyer and T. H. Andres. A random number generator based on the combination of four LCGs. *Mathematics and Computers in Simulation*, 44:99–107, 1997.

[85] P. L'Ecuyer, F. Blouin, and R. Couture. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, 1993.

[86] P. L'Ecuyer, A. Compagner, and J.-F. Cordeau. Entropy tests for random number generators. Manuscript, 1996.

[87] P. L'Ecuyer, J.-F. Cordeau, and R. Simard. Close-point spatial tests and their application to random number generators. *Operations Research*, 48(2):308–317, 2000.

[88] P. L'Ecuyer and R. Couture. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, 9(2):206–217, 1997.

[89] P. L'Ecuyer and J. Granger-Piché. Combined generators with components from different families. *Mathematics and Computers in Simulation*, 62:395–404, 2003.

[90] P. L'Ecuyer and P. Hellekalek. Random number generators: Selection criteria and testing. In P. Hellekalek and G. Larcher, editors, *Random and Quasi-Random Point Sets*, volume 138 of *Lecture Notes in Statistics*, pages 223–265. Springer-Verlag, New York, 1998.

[91] P. L'Ecuyer and R. Simard. Beware of linear congruential generators with multipliers of the form $a = \pm 2^q \pm 2^r$. *ACM Transactions on Mathematical Software*, 25(3):367–374, 1999.

[92] P. L'Ecuyer and R. Simard. *MyLib: A Small Library of Basic Utilities in ANSI C*, 2001. Software user's guide.

[93] P. L'Ecuyer and R. Simard. On the performance of birthday spacings tests for certain families of random number generators. *Mathematics and Computers in Simulation*, 55(1–3):131–137, 2001.

[94] P. L'Ecuyer and R. Simard. *ProbDist: A Software Library of Probability Distributions and Goodness-of-Fit Statistics in ANSI C*, 2001. Software user's guide.

[95] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

[96] P. L'Ecuyer, R. Simard, and S. Wegenkittl. Sparse serial tests of uniformity for random number generators. *SIAM Journal on Scientific Computing*, 24(2):652–668, 2002.

[97] P. L'Ecuyer and S. Tezuka. Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57(196):735–746, 1991.

[98] P. L'Ecuyer and R. Touzin. Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. In J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 683–689, Pistacaway, NJ, 2000. IEEE Press.

[99] H. Leeb and S. Wegenkittl. Inversive and linear congruential pseudorandom number generators in empirical tests. *ACM Transactions on Modeling and Computer Simulation*, 7(2):272–286, 1997.

[100] H. Levene and J. Wolfowitz. The covariance matrix of runs up and down. *The Annals of mathematical statistics*, 15:58–69, 1944.

[101] P. A. W. Lewis, A. S. Goodman, and J. M. Miller. A pseudo-random number generator for the system/360. *IBM System's Journal*, 8:136–143, 1969.

[102] M. Lüscher. A portable high-quality random number generator for lattice field theory simulations. *Computer Physics Communications*, 79:100–110, 1994.

[103] G. Marsaglia. A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface*, pages 3–10, North-Holland, Amsterdam, 1985. Elsevier Science Publishers.

[104] G. Marsaglia. Note on a proposed test for random number generators. *IEEE Transactions on Computers*, C-34(8):756–758, 1985.

[105] G. Marsaglia. Remarks on choosing and implementing random number generators. *Communications of the ACM*, 36(7):105–110, 1993.

[106] G. Marsaglia. DIEHARD: a battery of tests of randomness. See `http://stat.fsu.edu/~geo/diehard.html`, 1996.

[107] G. Marsaglia. The Marsaglia random number CDROM including the DIEHARD battery of tests of randomness. See `http://stat.fsu.edu/pub/diehard`, 1996.

[108] G. Marsaglia. A random number generator for C. Posted to the electronic billboard `sci.math.num-analysis`, September 30 1997.

[109] G. Marsaglia. Random numbers for C: The END? Posted to the electronic billboard `sci.crypt.random-numbers`, January 20 1999.

[110] G. Marsaglia. Good 64-bit RNG's. Posted to the electronic billboard `sci.crypt.random-numbers`, August 25 2002.

[111] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003. See `http://www.jstatsoft.org/v08/i14/xorshift.pdf`.

[112] G. Marsaglia, K. Ananthanarayanan, and N. Paul. How to use the McGill Random Number Package "SUPER-DUPER". Technical report, School of Computer Science, McGill University, Montreal, Canada, 1973.

[113] G. Marsaglia, B. Narasimhan, and A. Zaman. A random number generator for PC's. *Computer Physics Communications*, 60:345–349, 1990.

[114] G. Marsaglia and W. W. Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–9, 2002. See `http://www.jstatsoft.org/v07/i03/tuftests.pdf`.

[115] G. Marsaglia and L.-H. Tsay. Matrices and the structure of random number sequences. *Linear Algebra and its Applications*, 67:147–156, 1985.

[116] G. Marsaglia and A. Zaman. A new class of random number generators. *The Annals of Applied Probability*, 1:462–480, 1991.

[117] G. Marsaglia and A. Zaman. The KISS generator. Technical report, Department of Statistics, University of Florida, 1993.

[118] G. Marsaglia and A. Zaman. Monkey tests for random number generators. *Computers Math. Applic.*, 26(9):1–10, 1993.

[119] G. Marsaglia, A. Zaman, and W. W. Tsang. Towards a universal random number generator. *Statistics and Probability Letters*, 8:35–39, 1990.

[120] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000.

[121] J. L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theor*, IT-15:122–127, 1969.

[122] MathSoft Inc. *S-PLUS 6.0 Guide to Statistics*, volume 2. Data Analysis Division, Seattle, WA, 2000.

[123] M. Matsumoto. Simple cellular automata as pseudo-random m-sequence generators for built-in self test. *ACM Transactions on Modeling and Computer Simulation*, 8(1):31–42, 1998.

[124] M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3):179–194, 1992.

[125] M. Matsumoto and Y. Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation*, 4(3):254–266, 1994.

[126] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[127] U. M. Maurer. A universal statistical test for random bit generators. *Journal of Cryptology*, 5(2):89–105, 1992.

[128] C. Moler. *Numerical Computing with MATLAB*. SIAM, Philadelphia, 2004.

[129] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1992.

[130] NIST. Advanced encryption standard (AES). FIPS-197, U.S. DoC/National Institute of Standards and Technology, 2001. See `http://csrc.nist.gov/CryptoToolkit/tkencryption.html`.

[131] NIST. Secure hash standard (SHS). FIPS-186-2, with change notice added in february 2004, U.S. DoC/National Institute of Standards and Technology, 2002. See `http://csrc.nist.gov/CryptoToolkit/tkhash.html`.

[132] F. Panneton and P. L'Ecuyer. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, 2005.

[133] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[134] W. H. Payne, J. R. Rabung, and T. P. Bogyo. Coding the Lehmer pseudo-random number generator. *Communications of the ACM*, 12:85–86, Feb. 1969.

[135] D. E. Percus and M. Kalos. Random number generators for MIMD parallel processors. *Journal of Parallel and Distributed Computation*, 6:477–497, 1989.

[136] O. E. Percus and P. A. Whitlock. Theory and application of Marsaglia's monkey test for pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 5(2):87–100, 1995.

[137] W. H. Press and S. A. Teukolsky. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 1992.

[138] W. H. Press and S. A. Teukolsky. Portable random number generators. *Computers in Physics*, 6(5):522–524, 1992.

[139] The GNU Project. *R: An Environment for Statistical Computing and Graphics*. The Free Software Foundation, 2003. Version 1.6.2. See `http://www.gnu.org/directory/GNU/R.html`.

[140] R. Pyke. Spacings. *Journal of the Royal Statistical Society, Series B*, 27:395–449, 1965.

[141] J. S. Rao and M. Kuo. Asymptotic results on the greenwood statistic and some of its generalizations. *Journal of the Royal Statistical Society, Series B*, 46:228–237, 1984.

[142] T. R. C. Read and N. A. C. Cressie. *Goodness-of-Fit Statistics for Discrete Multivariate Data*. Springer Series in Statistics. Springer-Verlag, New York, 1988.

[143] W. J. J. Rey. On generating random numbers, with help of $y = [(a + x)sin(bx)]$ mod 1. presented at the 22nd European Meeting of Statisticians and the 7th Vilnius Conference on Probability Theory and Mathematical Statistics, August 1998.

[144] V. Rijmen, A. Bosselærs, and P. Barreto. Optimised ANSI C code for the Rijndael cipher (now AES), 2000. Public domain software.

[145] B. D. Ripley. Thoughts on pseudorandom number generators. *Journal of Computational and Applied Mathematics*, 31:153–163, 1990.

[146] B. D. Ripley and W. N. Venables. *Modern applied statistics with S-Plus*. Springer-Verlag, New York, 1994.

[147] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST special publication 800-22, National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, USA, 2001. See `http://csrc.nist.gov/rng/`.

[148] A. L. Rukhin. Distribution of the number of words with a prescribed frequency and tests of randomness. *Advances in Applied Probability*, 34(4):775–797, 2002.

[149] SAS. *SAS Language: Reference, Version 6*. Cary, North Carolina, first edition, 1990.

[150] L. N. Shchur, J. R. Heringa, and H. W. J. Blöte. Simulation of a directed random-walk model: The effect of pseudo-random number correlations. *Physica A*, 241:579–592, 1997.

[151] Y. S. Sherif and R. G. Dear. Development of a new composite pseudo-random number generator. *Microelectronics and Reliability*, 30:545–553, 1990.

[152] J. Soto. Statistical testing of random number generators. Available at `http://csrc.nist.gov/rng/`, 1999.

[153] M. A. Stephens. Statistics connected with the uniform distribution: Percentage points and application to testing for randomness of directions. *Biometrika*, 53:235–240, 1966.

[154] M. S. Stephens. Tests for the uniform distribution. In R. B. D'Agostino and M. S. Stephens, editors, *Goodness-of-Fit Techniques*, pages 331–366. Marcel Dekker, New York and Basel, 1986.

[155] S. J. Sullivan. Another test for randomness. *Communications of the ACM*, 36(7):108, 1993.

[156] K. Takashima. Last visit time tests for pseudorandom numbers. *J. Japanese Soc. Comp. Satist.*, 9(1):1–14, 1996.

[157] S. Tezuka. Random number generation based on the polynomial arithmetic modulo two. Technical Report RT-0017, IBM Research, Tokyo Research Laboratory, Oct. 1989.

[158] S. Tezuka. *Uniform Random Numbers: Theory and Practice*. Kluwer Academic Publishers, Norwell, Mass., 1995.

[159] S. Tezuka and P. L'Ecuyer. Efficient and portable combined Tausworthe random number generators. *ACM Transactions on Modeling and Computer Simulation*, 1(2):99–112, 1991.

[160] S. Tezuka, P. L'Ecuyer, and R. Couture. On the add-with-carry and subtract-with-borrow random number generators. *ACM Transactions of Modeling and Computer Simulation*, 3(4):315–331, 1994.

[161] G. Tindo. *Automates Cellulaires; Applications à la Modélisation de Certains Systèmes Discrets et à la Conception d'une Architecture Parallèle pour la Génération de Suites Pseudo-Aléatoires.* Dissertation (thesis), Université de Nantes, France, 1990.

[162] M. Tomassini, M. Sipper, M. Zolla, and M. Perrenoud. Generating high-quality random numbers in parallel by cellular automata. *Future Generation of Computer Systems*, 16:291–305, 1999.

[163] J. P. R. Tootill, W. D. Robinson, and D. J. Eagle. An asymptotically random Tausworthe sequence. *Journal of the ACM*, 20:469–481, 1973.

[164] R. Touzin. Des générateurs récursifs multiples combinés rapides avec des coefficients de la forme $\pm 2^{p_1} \pm 2^{p_2}$. Master's thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, 2001.

[165] G. Ugrin-Sparac. Stochastic investigations of pseudo-random number generators. *Computing*, 46:53–65, 1991.

[166] I. Vattulainen, T. Ala-Nissila, and K. Kankaala. Physical tests for random numbers in simulations. *Physical Review Letters*, 73(19):2513–2516, 11 1994.

[167] I. Vattulainen, T. Ala-Nissila, and K. Kankaala. Physical models as tests of randomness. *Physical Review E*, 52(3):3205–3213, 1995.

[168] S. R. Wallenstein and J. I. Naus. Probabilities for the size of largest clusters and smallest intervals. *Journal of the American Statistical Society*, 69:690–695, 1974.

[169] M. Z. Wang. Linear complexity profiles and jump complexity. *Information Processing Letters*, 61:165–168, 1997.

[170] H. Weyl. Über die Gleichverteilung von Zahlen mod. Eins. *Math. Ann.*, 77:313–352, 1916.

[171] B. A. Wichmann and I. D. Hill. An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190, 1982. See also corrections and remarks in the same journal by Wichmann and Hill, **33** (1984) 123; McLeod **34** (1985) 198–200; Zeisel **35** (1986) 89.

[172] B. A. Wichmann and I. D. Hill. Building a random number generator. *Byte*, 12(3):127–128, 1987.

[173] R. S. Wikramaratna. ACORN—a new method for generating sequences of uniformly distributed pseudo-random numbers. *Journal of Computational Physics*, 83:16–31, 1989.

[174] S. Wolfram. Statistical mechanics of cellular automata. *Rev. Modern Physics*, 55:601–644, 1983.

[175] S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7:123–169, 1986.

[176] S. Wolfram. *Theory and Applications of Cellular Automata*, volume 1 of *Advanced Series on complex systems*. World Scientific Publishing Co., Singapore, 1986.

[177] P.-C. Wu. Multiplicative, congruential random number generators with multiplier $\pm 2^{k_1} \pm 2^{k_2}$ and modulus $2^p - 1$. *ACM Transactions on Mathematical Software*, 23(2):255–265, 1997.

[178] R. M. Ziff. Four-tap shift-register-sequence random-number generators. *Computers in Physics*, 12(4):385–392, 1998.

[179] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–343, 1977.

[180] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory*, 24(5):530–536, 1978.

Index