

Instructions for the random number generator libraries on www.agner.org

By Agner Fog. © 2008 - 2010. GNU General Public License.
Version 2.01. 2010-08-03.

Contents

1	Introduction	1
2	Randomc package of random number generators.....	2
2.1	Generators included in the randomc library.....	2
3	Randoma package of random number generators.....	5
3.1	Generators included in the randoma library	5
3.2	Other functions included in the randoma library	7
3.3	Execution speed	9
3.4	Library versions	10
4	Stocc package of non-uniform generators	10
5	Frequently asked questions.....	15
5.1	Getting started	15
5.2	Is the random number generator that comes with my compiler good enough?.....	15
5.3	Which random number generator should I choose?.....	15
5.4	How do I change the code to use a different random number generator?	16
5.5	Choosing a seed.....	17
5.6	C++ version or binary library?	17
5.7	Multi-threading	18
5.8	Calling from other programming languages	18
5.9	Position-independent code.....	19
5.10	IRandom or IRandomX?	19
5.11	Why is the floating point interval half-open?.....	20
5.12	Generating events with a specific probability.....	21
5.13	Generating non-uniform random numbers	22
5.14	Monte Carlo simulation applications.....	22
5.15	Simulating evolution.....	23
5.16	Games and entertainment applications.....	23
5.17	Gambling applications.....	23
5.18	Security applications	23
5.19	Error conditions.....	24
6	Theoretical considerations.....	24
6.1	Using multiple streams.....	27
6.2	Deciding the cycle length	28
7	File lists	29
8	License conditions.....	30
9	No support.....	31
10	Literature.....	31

1 Introduction

This manual describes three packages of uniform and non-uniform random number generators:

`randoma.zip`: uniform random number generators as binary libraries
`randomc.zip`: uniform random number generators as C++ class libraries
`stocc.zip`: non-uniform random number generators as C++ class libraries

The random number generators in standard function libraries are not always of the best quality. With today's fast computers it is possible to make larger computer simulations than

what has been common previously. Larger "Monte Carlo" simulations require better random number generators. Furthermore, today's multi-kernel microprocessors require random number generators with support for multithreading. The present random number generator libraries are designed for the purpose of meeting these high demands. The advantages of these packages are:

- Very good randomness
- Very long cycle length
- High resolution
- Support for multiple threads and multiple streams
- Very fast and efficient
- Allow seeds of any length
- Includes Mersenne Twister, Mother-Of-All generator, SFMT generator and combinations of these
- Discrete uniform distribution over arbitrary interval is exact where other implementations have rounding errors
- Continuous distributions supported: Uniform and normal
- Discrete distributions supported: Uniform, Poisson, binomial, hypergeometric and various noncentral hypergeometric distributions
- Open source
- Support for Windows, Linux, BSD, Mac, etc.

The following three packages of random number generator libraries are available here:

- `randomc.zip`. A C++ class library of uniform random number generators.
- `randoma.zip`. The same random number generators as optimized binary code libraries (`*.lib`, `*.dll`, `*.a`) to link into a software project. Includes support for almost all x86 and x86-64 platforms, including 32-bit and 64-bit Windows, Linux, BSD and Intel-based Mac.
- `stocc.zip`. Non-uniform random number generators, including the following probability distributions: Normal, Poisson, Binomial, Hypergeometric, Fisher's Noncentral Hypergeometric, Wallenius' Noncentral Hypergeometric, etc.

The latest versions of these packages are always available from www.agner.org/random.

2 Randomc package of random number generators

2.1 Generators included in the randomc library

The `randomc.zip` package is a C++ class library containing various random number generators. Single-threaded applications need only one instance of the desired class while multi-threaded applications need one instance for each thread.

The following classes are included

`CRandomMersenne`:

Header: `randomc.h`

Source file: `mersenne.cpp`

Constructor: `CRandomMersenne(int seed);`

This is the "Mersenne Twister", a random number generator which has become very popular in recent years because of its long cycle length.

`CRandomMother:`

Header: `randomc.h`

Source file: `mother.cpp`

Constructor: `CRandomMother(int seed);`

This is the "Mother-Of-All" generator invented by George Marsaglia. Uses less memory than the Mersenne Twister.

`CRandomSFMT0:`

Header: `sfmt.h`

Source file: `sfmt.cpp`

Constructor: `CRandomSFMT0(int seed);`

The "SIMD-oriented Fast Mersenne Twister" (SFMT) invented by Mutsuo Saito and Makoto Matsumoto. This is an improvement of the Mersenne Twister with better randomness and higher speed. Designed specifically for processors with Single-Instruction-Multiple-Data (SIMD) capabilities, such as the SSE2 and later instruction set. This instruction set is available in almost all modern PC's and Mac computers.

`CRandomSFMT1:`

Header: `sfmt.h`

Source file: `sfmt.cpp`

Constructor: `CRandomSFMT1(int seed);`

Combines the SFMT generator (see above) and the Mother-Of-All generator to get the best possible randomness.

See page 24 for a theoretical discussion of these random number generators.

Member functions (methods)

`void RandomInit(int seed);`

Initialize or re-initialize the random number generator. The value of `seed` can be any integer. The same seed always gives the same sequence of random numbers. A different seed gives a different sequence.

`void RandomInitByArray(int const seeds[], int NumSeeds);`

Available only in Mersenne Twister and SFMT generator.

Initialize or re-initialize the random number generator. Allows any number of seeds.

`seeds[]` is an array with `NumSeeds` seeds. A different value of any of the seeds gives a different sequence of random numbers.

`int IRandom(int min, int max);`

Generates a random integer n with uniform distribution in the interval $\min \leq n \leq \max$.

The distribution may be slightly biased due to rounding errors if the interval length ($\max - \min + 1$) is large and not a power of 2.

Use `IRandomX` instead if the highest precision is required.

Restrictions: $\max \geq \min$ and $\max - \min + 1 < 2^{32}$.

If $\max - \min + 1 = 2^{32}$, i.e. if you are using the whole range of integers, then use `BRandom()` instead. You may type-cast the output of `BRandom` to a signed integer if needed.

Error conditions: Returns 0x80000000 if $\max < \min$.

`int IRandomX(int min, int max);`

Available only in Mersenne Twister and SFMT generator.

Same as `IRandom`. The distribution is always exact. See page 19 below for a detailed explanation.

`IRandomX` takes a little extra time every time the length of the interval ($\max - \min + 1$) is different from the last value.

```
uint32_t BRandom();
```

Gives a random 32-bit integer. May be used as 32 random bits.

```
double Random();
```

Gives a random floating point number x with uniform distribution in the interval $0 \leq x < 1$.

Resolution:

32 bits in Mersenne Twister and Mother-Of-All generator, 52 bits in SFMT generator.

Overview of member functions

Function	Generator		
	Mersenne Twister	Mother-Of-All	SFMT or combined
Initialize with new seed	CRandomMersenne::RandomInit	CRandomMother::RandomInit	CRandomSFMT::RandomInit
Initialize by array	CRandomMersenne::RandomInitByArray		CRandomSFMT::RandomInitByArray
Integer random	CRandomMersenne::IRandom	CRandomMother::IRandom	CRandomSFMT::IRandom
Integer random, exact	CRandomMersenne::IRandomX		CRandomSFMT::IRandomX
Random bits	CRandomMersenne::BRandom	CRandomMother::BRandom	CRandomSFMT::BRandom
Floating point 32 bits resolution	CRandomMersenne::Random	CRandomMother::Random	
Floating point 52 bits resolution			CRandomSFMT::Random
Floating point 63 bits resolution			(only in randoma library)

Compiler requirements

Class	C++ compiler	Compiler support for 64-bit integers	Compiler support for SSE2 and intrinsic functions
CRandomMersenne	X	-	-
CRandomMother	X	X	-
CRandomSFMT	X	X	X

You may use the libraries in the [randoma.zip](#) package (see page 5) if your compiler doesn't meet these requirements.

Hardware requirements

CRandomMersenne and CRandomMother will work on any microprocessor for which a suitable compiler is available.

CRandomSFMT works only on Intel or AMD computers with the SSE2 or later instruction set. Use the [randoma.zip](#) library instead if it is not certain that the computer supports SSE2.

Randomness qualities

Class	Cycle length	Passes tests for randomness	Bifurcation / diffusion	Resolution of continuous uniform distribution
CRandomMersenne	$2^{19937}-1$	most	low	32 bits
CRandomMother	$\approx 2^{158}$	all	high	32 bits
CRandomSFMT	$\geq 2^{11213}-1$	all if Mother	high	52 or 63 bits

		included	
--	--	----------	--

See page 24 for a more detailed discussion of the randomness of these generators.

Execution time

The execution times vary a lot depending on the compiler and the optimization possibilities. The execution times are usually a little longer than for the `randoma` library versions listed on page 9.

3 Randoma package of random number generators

3.1 Generators included in the randoma library

The `randoma.zip` package is a binary code library containing carefully optimized versions of the same random number generators as the `randomc.zip` package. Several different versions are included for compatibility with different compilers and operating systems. The appropriate version of the library is linked into the code project that uses it. Single-threaded and multi-threaded C and C++ versions are included.

The header file `randoma.h` defines all functions and classes in the `randoma` library package. This header file must be used when calling these function from a C or C++ program.

Single threaded C or C++ functions, static linking (`*.lib, *.a`)

Function	Generator		
	Mersenne Twister	Mother-Of-All	SFMT or combined
Initialize	MersenneRandomInit	MotherRandomInit	SFMTgenRandomInit
Initialize by array	MersenneRandom-InitByArray		SFMTgenRandom-InitByArray
Integer random	MersenneIRandom	MotherIRandom	SFMTgenIRandom
Integer random, exact	MersenneIRandomX		SFMTgenIRandomX
Random bits	MersenneBRandom	MotherBRandom	SFMTgenBRandom
Floating point 32 bits resolution	MersenneRandom	MotherRandom	
Floating point 52 bits resolution			SFMTgenRandom
Floating point 63 bits resolution			SFMTgenRandomL

The single-threaded versions are simple functions that need no class object. This makes the programming simpler. The necessary storage is provided internally in the library.

Example:

```
#include <time.h>           // Define time()
#include <stdio.h>          // Define printf()
#include "randoma.h"       // Define library functions
int main() {
    int seed = time(0);    // Use time as random seed
    double r;             // Declare variable
    MersenneRandomInit(seed); // Must initialize first
    r = MersenneRandom();  // Get a random number
    printf("\n%f", r);    // Print the random number
    return 0;            // Finished
```

}

Single threaded C or C++ functions, dynamic linking (*.dll)

Function	Generator		
	Mersenne Twister	Mother-Of-All	SFMT or combined
Initialize	MersenneRandomInitD	MotherRandomInitD	SFMTgenRandomInitD
Initialize by array	MersenneRandom-InitByArrayD		SFMTgenRandom-InitByArrayD
Integer random	MersenneIRandomD	MotherIRandomD	SFMTgenIRandomD
Integer random, exact	MersenneIRandomXD		SFMTgenIRandomXD
Random bits	MersenneBRandomD	MotherBRandomD	SFMTgenBRandomD
Floating point 32 bits resolution	MersenneRandomD	MotherRandomD	
Floating point 52 bits resolution			SFMTgenRandomD
Floating point 63 bits resolution			

The dynamic linking versions are intended mainly for calling from other programming languages such as Pascal, Fortran, C#, Basic, managed C++.NET, etc. where static linking may be difficult or impossible. These functions use the `stdcall` calling conventions rather than the `cdecl` calling convention.

Multi-threaded C++ functions, static linking (*.lib, *.a)

Function	Generator			
	Mersenne Twister	Mother-Of-All	SFMT	SFMT combined with Mother-Of-All
Initialize with new seed	CRandomMersenneA::RandomInit	CRandomMotherA::RandomInit	CRandomSFMTA0::RandomInit	CRandomSFMTA1::RandomInit
Initialize by array	CRandomMersenneA::Random-InitByArray		CRandomSFMTA0::Random-InitByArray	CRandomSFMTA1::Random-InitByArray
Integer random	CRandomMersenneA::IRandom	CRandomMotherA::IRandom	CRandomSFMTA0::IRandom	CRandomSFMTA1::IRandom
Integer random, exact	CRandomMersenneA::IRandomX		CRandomSFMTA0::IRandomX	CRandomSFMTA1::IRandomX
Random bits	CRandomMersenneA::BRandom	CRandomMotherA::BRandom	CRandomSFMTA0::BRandom	CRandomSFMTA1::BRandom
Floating point 32 bits resolution	CRandomMersenneA::Random	CRandomMotherA::Random		
Floating point 52 bits resolution			CRandomSFMTA0::Random	CRandomSFMTA1::Random
Floating point 63 bits resolution			CRandomSFMTA0::RandomL	CRandomSFMTA1::RandomL

The multi-threaded C++ versions need one instance (object) of the class for each thread. The object must be declared locally inside the thread. Only one object is needed in single-threaded applications. Example:

```
#include <time.h> // Define time()
#include <stdio.h> // Define printf()
#include "randoma.h" // Define library classes
int main() {
    int seed = time(0); // Use time as random seed
```

```

    CRandomMersenneA RanObject(seed); // Declare generator object
    double r = RanObject.Random();    // Get a random number
    printf("\n%f", r);                // Print the random number
    return 0;                          // Finished
}

```

Multi-threaded C functions, static linking (*.lib, *.a)

Function	Generator		
	Mersenne Twister	Mother-Of-All	SFMT or combined
Initialize	MersRandomInit	MotRandomInit	SFMTRandomInit
Initialize by array	MersRandom-InitByArray		SFMTRandom-InitByArray
Integer random	MersIRandom	MotIRandom	SFMTIRandom
Integer random, exact	MersIRandomX		SFMTIRandomX
Random bits	MersBRandom	MotBRandom	SFMTBRandom
Floating point 32 bits resolution	MersRandom	MotRandom	
Floating point 52 bits resolution			SFMTRandom
Floating point 63 bits resolution			SFMTRandomL

A multi-threaded application in C language cannot encapsulate the random number generator in a class as you can in C++. Therefore, you must explicitly define a buffer in each thread for containing the internal variables of the generator. The necessary size of the buffer is defined in `randoma.h` as `MERS_BUFFER_SIZE`, `MOTHER_BUFFER_SIZE` and `SFMT_BUFFER_SIZE` respectively. The buffer must be passed to all the library functions. The size of the buffer is passed to the initialization function. Example:

```

#include <time.h> // Define time()
#include <stdio.h> // Define printf()
#include "randoma.h" // Define library functions
int main() {
    int seed = time(0); // Use time as random seed
    double r; // Declare variable
    // Declare buffer of necessary size:
    char MersenneBuffer[MERS_BUFFER_SIZE];
    // Must initialize random number generator first:
    MersRandomInit(MersenneBuffer, sizeof(MersenneBuffer), seed);
    r = MersRandom(MersenneBuffer); // Get a random number
    printf("\n%f", r); // Print the random number
    return 0; // Finished
}

```

3.2 Other functions included in the randoma library

PhysicalSeed

C++ prototype

```
extern "C" int PhysicalSeed(int seeds[], int NumSeeds);
```

Description

This function uses a physical random number generator which is available on VIA processors to generate one or more random numbers. The physical random number generator is too slow to use for generating a long series of random numbers, but it is useful

for generating the seed for a pseudo random number generator because the random value is not deterministic.

This function will use the internal clock counter (see ReadTSC below) if running on a processor without a physical random number generator, i.e. Intel or AMD. In this case it will generate at most two integer values, the remaining values will be zero. The first value (`seeds[0]`) will be the most random while the second value (`seeds[1]`) contains the upper bits of the counter which are less random. The internal clock counter is a good source of randomness if it is called at a time that depends on user input.

Parameters

`seeds` An array that will receive `NumSeeds` random 32-bit integers.
`NumSeeds` The desired number of random integers.

Return value

0 Failure. The processor has no random number generator and no internal clock counter (earlier than Pentium)
1 The processor has no physical random number generator. The internal clock counter is used.
2 A physical random number generator is used.

The return value will indicate the availability of a physical random number generator even if `NumSeeds` is zero.

The function will generate a general protection exception (interrupt) in the rare case that the RDTSC instruction is disabled by the operating system. There is no way to check for this condition without generating an interrupt.

Example of use:

```
#include <stdio.h> // Define printf()
#include "randoma.h" // Define library functions
int main() {
    const int NSEEDS = 4; // Size of seeds array
    int seeds[NSEEDS]; // Array of seeds
    CRandomMersenneA RanObject(0); // Declare generator object
    PhysicalSeed(seeds, NSEEDS); // Generate random seeds
    RanObject.RandomInitByArray(seeds, NSEEDS); // Initialize RNG
    double r = RanObject.Random(); // Get a random number
    printf("\n%f", r); // Print the random number
    return 0; // Finished
}
```

ReadTSC

C++ prototype

```
extern "C" uint64_t ReadTSC (void);
```

Description

This function returns the value of the internal clock counter in the microprocessor. A 32-bit value is returned if the compiler doesn't support 64-bit integers.

The length of a clock cycle is the reciprocal of the CPU clock frequency. For example, on a 2 GHz computer the length of a clock cycle is 0.5 ns (nanosecond). The value may vary since many microprocessors can change their clock frequency depending on the workload.

To count how many clock cycles a piece of code takes, call `ReadTSC` before and after the code to measure and calculate the difference. See www.agner.org/optimize/asmlib.zip for details.

`ReadTSC()` is useful for generating a seed for a random number generator because it has a very high resolution. The internal clock counter is a good source of randomness if it is called at a time that depends on user input. Example:

```
#include <time.h>           // Define time()
#include <stdio.h>          // Define printf()
#include "randoma.h"        // Define library functions
int main() {
    int seed = (int)ReadTSC(); // Use CPU clock as random seed
    CRandomMersenneA RanObject(seed); // Declare generator object
    double r = RanObject.Random(); // Get a random number
    printf("\n%f", r); // Print the random number
    return 0; // Finished
}
```

The function will generate a general protection exception (interrupt) in case the `RDTSC` instruction is disabled by the operating system. It will also fail on the old 80386 and 80486 processors that don't have this instruction.

InstructionSet

C++ prototype

```
int InstructionSet(void);
```

Description

This function detects which instruction set is supported by the microprocessor it is running on. It is used internally in the `randoma` library for selecting the optimal version of each function. See www.agner.org/optimize/asmlib.zip for details.

3.3 Execution speed

Random number generator	32 random bits <code>BRandom()</code>	Random integer <code>IRandom(min,max)</code>	Random floating point number <code>Random()</code>
Mersenne Twister	15	17	15
Mother-Of-All	17	23	21
SFMT	14	22	16
SFMT combined with Mother-Of-All	26	38	27

The execution times are given in core clock cycles measured on an Intel Core 2 processor running a single thread. The length of a clock cycle is the reciprocal of the clock frequency, e.g. 0.5 ns on a 2 GHz processor.

All the execution times are so small that they can be regarded as insignificant in most cases. Small differences in hardware, compilers, code environment, etc. can cause much bigger differences in execution time. For example, a single cache miss can add more than a hundred clock cycles to the time.

3.4 Library versions

The `randoma` library has many versions for compatibility with different platforms and compilers. Use the tables below to select the right version for a particular application.

Library version selection guide: Windows			
Compiler/language	File format	32 bit	64 bit
MS C++ unmanaged, Intel, Gnu	COFF	randomacof32.lib	randomacof64.lib
Borland C++, Digital Mars, Watcom	OMF	randomaomf32.lib	
MS C++.net, C#, VB	DLL	randomad32.dll	randomad64.dll
Borland Delphi	DLL	randomad32.dll	randomad64.dll
Other languages	DLL	randomad32.dll	randomad64.dll

Library version selection guide: Linux and BSD (x86 and x86-64)				
Compiler/language	File format	32 bit executable	32 bit shared object	64 bit
Gnu, Intel C++	ELF	randomaelf32.a	randomaelf32p.a	randomaelf64.a

Library version selection guide: Mac (Intel based)				
Compiler/language	File format	32 bit executable	32 bit shared object	64 bit
Gnu, Intel C++	MachO	randomamac32.a	randomamac32p.a	randomamac64.a

Explanation of the column headings:

Compiler/language: The compiler and programming language used. Different compilers may use different object file formats.

File format: It is necessary to select a library in the right object file format, or a dynamic link library if static linking is not possible.

32 bit / 64 bit: Use the appropriate version when compiling for 32-bit mode or 64-bit mode.

32 bit executable: Use this version when making an executable binary file.

32 bit shared object: Use this version when position-independent code is needed. Position-independent code is needed when building a shared object for 32-bit mode in Linux, BSD and Mac. The position-independent version is slightly slower.

4 Stocc package of non-uniform generators

The `stocc.zip` package is a C++ class library defining various non-uniform random number generators with various distributions. The non-uniform generators can be based on any of the uniform generators in the `randommc` and `randoma` packages, which are used as a C++ base class.

The following classes are included

`StochasticLib1:`

Header: `stocc.h`

Source file: `stoc1.cpp`

Constructor: `StochasticLib1(int seed);`

Base class: Any of the classes in `randomc.h` or `randoma.h`. Set `STOC_BASE` to the desired base class.

Defines generators for the following distributions: Bernoulli, Binomial, Hypergeometric, multivariate Hypergeometric, Multinomial, Normal, Poisson, and a shuffling function.

`StochasticLib2:`

Header: `stocc.h`

Source file: `stoc2.cpp`

Constructor: `StochasticLib2(int seed);`

Base class: `StochasticLib1`.

Defines alternative generators for the following distributions: Binomial, Hypergeometric, Poisson.

The implementations in `StochasticLib2` are faster than `StochasticLib1` if the parameters are constant but slower if the parameters are changing.

`StochasticLib3:`

Header: `stocc.h`

Source files: `stoc3.cpp`, `fnchyppr.cpp`, `wnchyppr.cpp`, `erfres.cpp`

Constructor: `StochasticLib3(int seed);`

Base class: `StochasticLib1` (or `StochasticLib2`).

Defines various noncentral hypergeometric distributions. These distributions are useful for simulating biased sampling and genetic models of evolution.

Member functions (methods) in `StochasticLib1`:

`int StochasticLib1::Bernoulli(double p);`

Bernoulli distribution with probability parameter p .

Returns 1 with probability p , or 0 with probability $1-p$.

Error conditions:

Gives error message if $p < 0$ or $p > 1$.

`int32_t Binomial (int32_t n, double p);`

Binomial distribution with parameters n and p .

This is the distribution of the number of red balls you get when drawing n balls with replacement from an urn where p is the fraction of red balls in the urn. Definition:

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

Error conditions:

Gives error message if $n < 0$ or $p < 0$ or $p > 1$.

`int32_t StochasticLib1::Hypergeometric(int32_t n, int32_t m, int32_t N);`

Hypergeometric distribution with parameters n , m , N . (Note the order of the parameters).

This is the distribution of the number of red balls you get when drawing n balls without replacement from an urn where the urn contains N balls, where m balls are red and $N-m$ balls are white. Definition:

$$f(x) = \frac{\binom{m}{x} \binom{N-m}{n-x}}{\binom{N}{n}}$$

Error conditions:

Gives error message if any parameter is negative or $n > N$ or $m > N$.

```
void StochasticLib1::MultiHypergeometric(int32_t * destination,  
int32_t * source, int32_t n, int colors);
```

Multivariate hypergeometric distribution. This is the distribution you get when drawing n balls from an urn without replacement, where there can be any number of colors. This is the same as the hypergeometric distribution when $colors = 2$. The number of balls of each color is returned in *destination*, which must be an array with $colors$ places. *source* contains the number of balls of each color in the urn. *source* must be an array with $colors$ places.

Error conditions:

Gives an error message if any parameter is negative or if the sum of the values in *source* is less than n . The behavior is unpredictable if *source* or *destination* has less than $colors$ places.

```
void StochasticLib1::Multinomial(int32_t * destination, int32_t *  
source, int32_t n, int colors);  
void StochasticLib1::Multinomial(int32_t * destination, double *  
source, int32_t n, int colors);
```

Multivariate binomial distribution. This is the distribution you get when drawing n balls from an urn *with replacement*, where there can be any number of colors. This is the same as the binomial distribution when $colors = 2$. The number of balls of each color is returned in *destination*, which must be an array with $colors$ places. *source* contains the number or fraction of balls of each color in the urn. *source* must be a *double* or *int* array with $colors$ places.

The sum of the values in *source* does not have to be 1, but it must be positive. The probability that a ball has color i is $source[i]$ divided by the sum of all values in *source*.

Error conditions:

Gives an error message if any parameter is negative or if the sum of the values in *source* is zero. The behavior is unpredictable if *source* or *destination* has less than $colors$ places.

```
double StochasticLib1::Normal(double m, double s);
```

Normal distribution with mean m and standard deviation s . This distribution simulates the sum of many random factors. Definition:

$$f(x) = \frac{1}{s\sqrt{2\pi}} e^{-\frac{(x-m)^2}{2s^2}}$$

Error conditions: None.

```
double StochasticLib1::double NormalTrunc(double m, double s,  
double limit);
```

Truncated normal distribution with mean m and standard deviation s . This is the normal distribution with the tails cut off at $m \pm limit$. Values outside the interval $(m-limit) \leq x \leq (m+limit)$ are rejected.

Error conditions: Gives error message if $limit < s$.

```
int32_t StochasticLib1::Poisson(double L);
```

Poisson distribution with mean L .

This is the distribution of the number of events in a given time span or a given geographical area when these events are randomly scattered in time or space. Definition:

$$f(x) = \frac{L^x}{x!} e^{-L}$$

Error conditions: Gives error message if $L < 0$ or $L > 2 \cdot 10^9$.

```
void StochasticLib1::Shuffle(int * list, int min, int n);
```

Shuffling a list. This function makes a list of the n numbers from min to $min+n-1$ in random order. The result is returned in `list`, which must be an array with n elements. The array index goes from 0 to $n-1$. If you want to shuffle something else than integers then use the integers in `list` as an index into a table of the items you want to shuffle. Error conditions: none. The behavior is unpredictable if the size of the array `list` is less than n .

Member functions (methods) in StochasticLib2:

```
int32_t StochasticLib2::Hypergeometric(int32_t n, int32_t m,
int32_t N);
```

```
int32_t StochasticLib2::Binomial(int32_t n, double p);
```

```
int32_t StochasticLib2::Poisson(double L);
```

This is an alternative implementation of the similar functions in `StochasticLib1`. `StochasticLib2` is faster than `StochasticLib1` if the functions are called many times with the same parameters, but slower than `StochasticLib1` if the parameters are changed. See the file `sampmet.pdf` for a description of the sampling methods.

Member functions (methods) in StochasticLib3:

```
void StochasticLib3::SetAccuracy(double accur);
```

Set the desired accuracy of the subsequent function calls. The default value is 10^{-8} .

```
int32_t StochasticLib3::FishersNCHyp (int32_t n, int32_t m, int32_t
N, double odds);
```

The **Fisher's noncentral hypergeometric distribution** is the distribution of two binomial variates conditional upon their constant sum. See the file `distrib.pdf` for a definition. Execution may be slow and inexact when N is high and `odds` is far from 1. Error conditions:

Gives error message if any parameter is negative or $n > N$ or $m > N$.

```
int32_t StochasticLib3::WalleniusNCHyp (int32_t n, int32_t m,
int32_t N, double odds);
```

The **Wallenius noncentral hypergeometric distribution** is similar to the hypergeometric distribution, but with bias. The bias can be seen as an odds ratio. `odds` > 1 will favor the red balls, and `odds` < 1 will favor the white balls. When `odds` = 1 we have the hypergeometric distribution. Error conditions:

Gives error message if any parameter is negative or $n > N$ or $m > N$.

```
void StochasticLib3::MultiFishersNCHyp (int32_t * destination,
int32_t * source, double * weights, int32_t n, int colors);
```

The **multivariate Fisher's noncentral hypergeometric distribution** is the distribution of multiple binomial variates conditional upon their constant sum. See the file `distrib.pdf` for a definition. This function may be inexact, but uses an approximation with an accuracy that is better than 1% in most cases. The precision can be tuned at the expense of higher calculation times.

Error conditions:

Gives an error message if any parameter is negative or if the total number of balls with nonzero weight is less than `n`. The behavior is unpredictable if any of the arrays has less than `colors` places.

```
void StochasticLib3::MultiWalleniusNCHyp (int32_t * destination,  
int32_t * source, double * weights, int32_t n, int colors);
```

Multivariate Wallenius noncentral hypergeometric distribution. This is the distribution you get when drawing colored balls from an urn without replacement, with bias. See the file `distrib.pdf` for a definition. `weights` is an array with `colors` places containing the weight or odds for each color. The probability of drawing a particular ball is proportional to its weight. This function may be inexact, but uses an approximation with an accuracy that is better than 1% in almost all cases.

Error conditions:

Gives an error message if any parameter is negative or if the total number of balls with nonzero weight is less than `n`. The behavior is unpredictable if any of the arrays has less than `colors` places.

```
void StochasticLib3::MultiComplWalleniusNCHyp (int32_t *  
destination, int32_t * source, double * weights, int32_t n, int  
colors);
```

Multivariate complementary Wallenius noncentral hypergeometric distribution. This is the distribution of the balls that remain in the urn when drawing `N-n` colored balls from an urn without replacement, with bias. (`N` is the sum of `source`). See the file `distrib.pdf` for a definition.

Other functions

```
void FatalError(const char *ErrorText);
```

Header: `randomc.h`

Source file: `userintf.cpp`

Used internally to generate error messages. There is no portable way of writing error messages. Systems with a graphical user interface (e.g. Windows) need a pop-up message box, while console mode programs and other line oriented systems need output to the standard error output. Therefore, you may have to modify the function `FatalError` in the file `userintf.cpp` to fit your system. This function is called by the library functions in case of illegal parameter values or other fatal errors. Experience shows that these error messages are very useful when debugging a program that uses the non-uniform random number generators. You may even enhance the `FatalError` function to output additional debug information about the state of your program.

```
void EndOfProgram(void);
```

Header: `randomc.h`

Source file: `userintf.cpp`

Program exit used in the program examples. Windows-like environments may require that the program waits for the user to press a key before exiting, in order to prevent the output screen image from disappearing. Therefore, you may have to modify the function `EndOfProgram` in `userintf.cpp` to fit your system if you experience this problem.

5 Frequently asked questions

5.1 Getting started

The best way to get started is to try some of the example programs included in `randomc.zip`.

Try to compile the file `ex-ran.cpp` with your C++ compiler and run it. The example program runs in console mode. It will output a list of random integer numbers, a list of random floating point numbers, and a list of random 32-bit numbers in hexadecimal representation.

You may modify the example file to make it do what you want. Use the class member function `IRandom(min,max)` to get a random integer in the interval from min to max.

Use the class member function `Random()` to get a floating point number in the interval from 0 to 1.

Use the class member function `BRandom()` to get random bits.

The package `stocc.zip` includes the following examples for generating non-uniform distributions:

`ex-cards.cpp`: Shuffle a deck of cards.

`ex-lotto.cpp`: Picks six random numbers in the interval from 1 to 36 so that no number occurs more than once.

`ex-stoc.cpp`: Generates random numbers with various different distributions: Uniform, normal distribution, Poisson, Binomial and Hypergeometric.

5.2 Is the random number generator that comes with my compiler good enough?

Historically, the random number generators in standard function libraries have a very bad reputation. Many function libraries have been improved in recent years, but it is still recommended to check the quality of a random number generator before using it for demanding applications.

Almost any random number generator is good enough for small entertainment applications. You need only be concerned if you are making large and demanding applications.

You need to check the documentation for the random number generator in your standard function library. If there is little or no documentation then it is probably not very good.

Few standard libraries have multi-threaded random number generators. You need to check this as well as various features of randomness to decide if a particular random number generator is good enough for your application.

5.3 Which random number generator should I choose?

All the random number generators in the present packages are very good. In most cases it doesn't matter which one you choose.

If portability is important then choose the Mersenne Twister. This generator has become very popular in recent years due to its long cycle length and high dimensions of

equidistribution. It is available in many different function libraries from different sources and in many different programming languages. The C++ code in `randomc.zip` can be compiled with almost any C++ compiler. Disadvantages: Fails a few of the most stringent tests for randomness. Poor bifurcation or diffusion, as explained in paragraph 6 page 24 below. Uses more memory than the other generators.

If memory use is important then use the Mother-Of-All generator. It uses less memory than the other generators. It has passed stringent tests for randomness. Disadvantages: Shorter cycle length. Lower dimensions of equidistribution.

If speed is important then use the SFMT generator alone. This is the fastest of the generators in the package and it performs better than the Mersenne Twister on some criteria of randomness. Disadvantages: Portability to other platforms is limited. Fails a few of the most stringent tests for randomness.

If high resolution is important then use the SFMT generator alone or combined with the Mother-Of-All generator. The code provides floating point random numbers with a resolution of 52 bits where the other generators have only 32 bits. The library version in `randoma.zip` also provides 63 bits resolution in a `long double`. Note that Microsoft compilers don't support `long double` precision.

If it is important to get the best possible randomness then use the SFMT generator combined with the Mother-Of-All generator. This is recommended for the largest multi-threaded Monte Carlo simulations and Monte Carlo integrations.

You may combine any two random number generators. The file `rancombi.cpp` shows an example of how to combine the Mersenne Twister and the Mother-Of-All generator.

See page 24 for a theoretical discussion of the difference between the different generators.

5.4 How do I change the code to use a different random number generator?

For uniform random number generators: See the example file `ex-ran.cpp`. This file makes an object of the class `CRandomMersenne` and includes the source file `mersenne.cpp` to make a Mersenne Twister random number generator. Use a different class name to get a different random number generator. The class names are listed below.

For non-uniform random number generators: Edit the file `stocc.h` and change the line `#define STOC_BASE CRandomMersenne`

Change the name of the random number generator class to the desired base class, for example:

```
#define STOC_BASE CRandomSFMT0
```

For the sake of simplicity, I have chosen not to use C++ templates for choosing the base class.

To make a Mother-Of-All generator, change the class name to `CRandomMother` and include the source file `mother.cpp`.

To make a SFMT generator, change the class name to `CRandomSFMT0` and include the header file `sfmt.h` and the source file `sfmt.cpp`. Compile with the SSE2 instruction set enabled (option `-msse2` required on 32-bit Gnu compiler).

To make a SFMT generator combined with the Mother-Of-All generator, change the class name to `CRandomSFMT1` and include the header file `sfmt.h` and the source file `sfmt.cpp`. Compile with the SSE2 instruction set enabled (option `-msse2` required on 32-bit Gnu compiler).

To use the binary library version of one of the random number generators, use the class name `CRandomMersenneA`, `CRandomMotherA`, `CRandomSFMTA0` or `CRandomSFMTA1` and include the header file `randoma.h` from `randoma.zip`. Link in the appropriate version of the library according to the table on page 10.

5.5 Choosing a seed

The seed is a start value that is needed for the random number generators. If you run a program again with the same seed then you will get exactly the same sequence of random numbers. If you use a different seed then you will get a different sequence of random numbers.

If you want the sequence to be different every time then you may use the time as seed. The example `ex-ran.cpp` uses the time in seconds as seed.

You can get better resolution by using the function `ReadTSC` which is provided in the binary libraries in `randoma.zip`. See the example on page 9. The `ReadTSC` function has a resolution of less than a nanosecond. If the `ReadTSC` function is called inside a task that is started by user input, e.g. by the user clicking a button, then you have a very good seed because it is extremely unlikely that two different users can press a button at the same time within less than a nanosecond.

In the case of Monte Carlo simulation or Monte Carlo integration applications it is desirable to have reproducible results as explained below in section 5.14 page 22. Let the user input the seed, for example 1 in the first simulation, 2 in the second simulation. etc.

It may be useful to have multiple sources of randomness for the seed, for example in security applications. The `RandomInitByArray` function allows multiple seeds. Useful sources of random seeds are various time functions, thread ID, user ID, various hardware parameters such as IP number, MAC address, hard disk ID, etc. and all sorts of sound and video.

Multi-threaded applications must have a different seed for each thread as explained below in section 5.7 page 18.

5.6 C++ version or binary library?

The C++ class library in `randoc.zip` and the binary library in `randoma.zip` provide the same random number generators. It is mainly a matter of convenience which one you choose. The main differences are:

- The binary library is faster than the C++ library in most cases, but not all. However, both libraries are so fast that they contribute little to the total execution time in most applications.
- The binary library works only on x86 platforms (including Windows, Linux, BSD and Intel-based Mac computers). The C++ implementations of the Mersenne Twister and the Mother-Of-All generator work on any platform for which a suitable C++ compiler is available.
- The C++ implementation of the SFMT generator works only on computers with the SSE2 or later instruction set. The binary library includes support for old computers without SSE2. The appropriate version of the code is selected automatically in the binary library so that the programmer doesn't have to care about compatibility.

- The C++ implementation is only for applications coded in the C++ language. The binary library also works with C and several other programming languages.
- The binary library allows a simple C-style function call interface without class objects for single-thread applications.
- The binary library implementation of the SFMT generator can provide long double random numbers with a very high resolution of 63 bits.
- The non-uniform generators in `stocc.zip` can use either the C++ or the binary library as base.

5.7 Multi-threading

Very time-consuming applications can often take advantage of computers with multiple microprocessor cores by dividing the work between the microprocessor cores. This requires that it is logically possible to divide the job into independent sub-jobs that can run in parallel threads. There should be no communication between the threads. The number of threads should preferably not be higher than the number of microprocessor cores.

Use any of the random number generator classes in `randomc.zip` or `randoma.zip` and make one object of the class in each thread.

The threads must use different seeds to make sure that each thread has a unique sequence of random numbers. It is convenient to use the `RandomInitByArray` function with two seeds where one of the seeds is the same for all the threads and the second seed is the thread number.

If the code is written in C rather than C++ then use the binary library and make a local buffer in each thread for the internal variables of the random number generator as shown in the example on page 7.

The non-uniform random number generators in `stocc.zip` do not work with the C-style functions because they need a C++ base class.

5.8 Calling from other programming languages

The `randoma` library is designed for calling from C and C++. It is possible to call the library from other languages if the compiler supports binary library calls.

Some compiled languages such as Fortran may allow static linking of libraries. Other languages such as Delphi Pascal, C#, Visual Basic and managed C++.NET work better with dynamic link libraries (DLL). Use the DLL version under Windows if the compiler doesn't support static linking or if the static link library is incompatible.

A DLL uses the `stdcall` calling convention by default. The `stdcall` versions of the functions in `randoma.zip` have a `D` suffix on the name.

Linking with Java is particularly difficult. It is necessary to use the Java Native Interface (JNI).

It is preferred to use the single-thread functions when calling from other languages than C or C++. If you consider using multiple threads for the sake of speed then you should be aware that the program will probably run faster if coded in C++.

If you nevertheless decide to make a multi-threaded program in a language other than C or C++ then you have to use the multi-threaded C functions mentioned on page 7 with a local array as buffer. Note that arrays are represented differently in different programming languages. You need to transfer the raw C-style array to the function rather than an array descriptor. See the manual for the specific compiler for how to link with C-style arrays.

5.9 Position-independent code

Shared objects (*.so) in 32-bit Linux, BSD and Mac require position-independent code. Special position-independent versions of asmlib are available for building shared objects. Use the position-independent libraries `randomaelf32p.a` or `randomamac32p.a` according to the list on page 10. Position-independent code is not an issue in 64-bit systems and in Windows.

5.10 IRandom or IRandomX?

The functions `IRandom` and `IRandomX` both give a random integer with uniform distribution in the interval from `min` to `max`, inclusive. `IRandom` is a little faster and a little less accurate than `IRandomX`.

The slight error in `IRandom` can be explained as follows. We have a random number generator that makes a random integer X in the interval $[0, B-1]$ so that there are B possible values for X . In our case we have a 32-bit output, so that $B = 2^{32}$. We want to convert this to an integer Y in the interval $[\text{min}, \text{max}]$ so that we have $L = \text{max} - \text{min} + 1$ different possible values of Y . The simplest way of conversion is

$$Y = \text{min} + \left\lfloor \frac{XL}{B} \right\rfloor$$

If B is divisible by L then there will be B/L different values of X for each value of Y . If B is not divisible by L then there will be $(B \bmod L)$ values in excess so that some of the Y values will have $\left\lfloor \frac{B}{L} \right\rfloor$ corresponding X values and some Y values have $\left\lfloor \frac{B}{L} \right\rfloor + 1$ corresponding X values.

In other words, some Y values have higher probability than others.

The Y values that have higher probability are evenly spread over the interval from `min` to `max` so that the mean will still be close to $(\text{min} + \text{max})/2$.

Most random number generator packages have this inaccuracy. It doesn't help to generate an integer random number from a floating point random number. The floating point number still has B possible values so that the problem is the same. The error can be reduced by using a high resolution (high value of B) but there is still a theoretical error.

If B is much bigger than L then the difference in probability between the Y values will be small. If L is a power of 2 then B is divisible by L and there will be no error.

The `IRandomX` function eliminates the inaccuracy by rejecting the excess $(B \bmod L)$ values of X . If X happens to have one of the excess values then it is rejected and a new value of X is generated. This method makes it certain that all values of Y have exactly the same probability.

The difference between `IRandom` and `IRandomX` is illustrated in the example program `testirandomx.cpp`. This test program compares the results for `IRandom` and `IRandomX` in the worst case where the value of L is 3/4 of B . The result in this case shows that every third Y value has a frequency that is double the frequency of the other Y values when `IRandom` is used. All Y values have the same frequency when `IRandomX` is used.

The `IRandomX` function needs extra time to calculate $(B \bmod L)$ every time L is changed.

In conclusion, it is recommended to use `IRandomX` when L is very large and not a power of 2 and you need high precision.

If L is less than around 1000 then the error is so small that it doesn't show even in large statistical tests. In this case it is acceptable to use `IRandom`.

5.11 Why is the floating point interval half-open?

The `Random` function generates random floating point numbers Y in the interval $0 \leq Y < 1$. This has to do with quantification. There is a finite number of possible Y values and we want these values to be equally spaced. The spacing is always a negative power of 2, for example 2^{-32} , because of the binary representation. If the spacing between possible Y values is 2^{-b} then there are 2^b possible values in the interval $0 \leq Y < 1$, but 2^b+1 possible values in the interval $0 \leq Y \leq 1$. The floating point value Y is generated from a b -bit integer X in the interval $0 \leq X < 2^b$ by calculating $Y = X / 2^b$. It would be difficult to generate 2^b+1 different Y values from 2^b different X values.

However, the choice of a half-open interval for X is not only a matter of making the generation simple. It is also desirable for certain reasons. Assume that we want to generate an event with a specific probability p , for example $p = 0.5$. In the C++ code we can write:

```
CRandomMersenne RanGen(time(0));
double Y = RanGen.Random();
double p = 0.5;
if (Y < p) {
    // This will happen with probability p
}
```

The above code generates an event with probability p because the fraction of possible Y values less than p is equal to p . If Y had 2^b+1 possible values in the interval $0 \leq Y \leq 1$ then the probability of $Y < p$ would be approximately $p - 2^{-b-1}$, and the probability of $Y \leq p$ would be approximately $p + 2^{-b-1}$. In other words, it is impossible to generate an event with exactly the probability p if we use the closed interval $0 \leq Y \leq 1$, but easy if we use the half-open interval $0 \leq Y < 1$. Generating events with specific probabilities is perhaps the most important application of random number generators. Therefore it is desirable to have the half-open interval $0 \leq Y < 1$.

On the other hand, the mean of Y is not exactly $\frac{1}{2}$ as we would like the mean of a uniform distribution to be, but $\frac{1}{2} - 2^{-b-1}$. If we have a symmetric interval, i.e. $0 \leq Y \leq 1$ or $0 < Y < 1$ then the mean will be exactly $\frac{1}{2}$.

So, as long as the random number is quantified we cannot have exactness in both the generation of events with specific probabilities and exactness in the mean at the same time.

Fortunately, the error is very small. With a resolution of $b = 32$ bits the error in the mean will be so small that we cannot detect it in a statistical test with a realistic sample size. In other words, the error is theoretical only and has no practical significance. And we can reduce the error further by choosing a higher resolution. The present packages offer random numbers with resolutions as high as 52 or 63 bits.

The difference between the probabilities of $Y < p$ and $Y \leq p$ may look confusing to a mathematician because math books usually assume infinite precision so that the two probabilities are the same. Theoretical math books rarely make this kind of distinctions and it may be quite arbitrary whether a book specifies open or closed intervals for uniform random numbers. Half-open intervals for random numbers are more likely to be seen in

computer books than in math books. You should not be concerned about a math book specifying e.g. a closed interval when in fact you have a half-open interval.

It is possible to manipulate the generators to get different output intervals, but this should be done only if there is a very specific reason to do so. Examples:

```
CRandomSFMTA1 RanGen(time(0));
double r1, r2, r3, r4, r5;

// Random number in interval 0 <= r1 < 1
r1 = RanGen.Random();

// Random number in interval 0 < r2 <= 1
r2 = 1.0 - RanGen.Random();

// Random number in interval 0 < r3 < 1
do {
    r3 = RanGen.Random();
} while (r3 == 0); // Reject r3 if 0

// Random number in interval 0 <= r4 <= 1
// (round from long double to double)
r4 = (double)RanGen.RandomL();

// Random number in interval a <= r5 < b
const double a = 5., b = 8.;
r5 = a + (b-a)*RanGen.Random();
```

In the `r4` example we are rounding from `long double` to `double` so that values very close to 1 will be rounded to 1. This is not necessary for the sake of precision, but it may be useful if you want to make sure that the exact values 0 and 1 can actually both occur (though very rarely). Note that the `long double` function `RandomL` is available only in the binary library SFMT generator and that Microsoft compilers don't support `long double` precision.

5.12 Generating events with a specific probability

There are various ways to generate an event with a specific probability. If you want to generate an event with probability `p`, where `p` is a floating point number, then use for example:

```
CRandomMersenne RanGen(time(0));
double p = 0.5;
if (RanGen.Random() < p) {
    // This will happen with probability p
}
```

There are faster ways if the probability is a rational number. For example, to generate an event with probability `P/Q` where `P` and `Q` are positive integers:

```
CRandomMersenne RanGen(time(0));
const int P = 7, Q = 20;
if (RanGen.IRandom(1,Q) <= P) {
    // This will happen with probability P/Q
}
```

This method is exact if you use `IRandomX`.

If `Q` is a power of 2 then it is even faster to use random bits:

```

CRandomMersenne RanGen(time(0));
const int P = 5, Q = 16;
if ((RanGen.BRandom() & (Q-1)) < P) {
    // This will happen with probability P/Q if Q is a power of 2
}

```

Special cases:

```

CRandomMersenne RanGen(time(0));
if (RanGen.BRandom() & 1) {
    // This will happen with probability 1/2
}
if ((RanGen.BRandom() & 3) == 0) {
    // This will happen with probability 1/4
}
if (RanGen.BRandom() & 3) {
    // This will happen with probability 3/4
}

```

5.13 Generating non-uniform random numbers

Monte Carlo simulations may require random variates with a specific probability distribution, such as normal or Poisson distribution. The variate generators are in `stocc.zip`. These variate generators convert a random number with uniform distribution to the desired distribution, e.g. a Poisson distribution. The uniform distribution generator can be any of the random number generators in `randomc.zip` or `randoma.zip`. This is specified in the file `stocc.h`, where the line

```
#define STOC_BASE CRandomMersenne
```

defines that the class `CRandomMersenne` is used as base class for the variate generators so that they will be based on the Mersenne Twister. You may change this line to specify the desired base class as random number generator.

The program example `ex-stoc.cpp` makes random numbers with uniform, normal, Poisson, binomial and hypergeometric distributions.

See the file `distrib.pdf` for definition of the distributions and the file `sampmet.pdf` for a description of the sampling methods used.

5.14 Monte Carlo simulation applications

The name Monte Carlo is traditionally used for computer simulation of processes that include random events. Depending on the application, you may need uniform or non-uniform random numbers.

The value of the seed for the random number generator should be input by the user. If the simulation shows a particularly interesting behavior then the user can repeat the simulation with the same seed to study this behavior in more detail, or repeat with a different seed to see if the unusual behavior disappears.

The seeds can be simple numbers such as 1, 2, 3. They don't have to be particularly "random".

Very time-consuming simulations may be split up into multiple threads on a computer with multiple microprocessor cores. See chapter 5.7 on page 18 above. Use `RandomInitByArray` with two seeds: the seed input by the user and the thread number. A repeated simulation will give the same results only if the seed is the same and the number of threads is the same.

Monte Carlo integration can be implemented in the same way.

5.15 Simulating evolution

The non-central hypergeometric distributions are useful for simulating Darwinian models of evolution. This is illustrated in the example programs `ex-evol1.cpp` and `ex-evol2.cpp`.

`ex-evol1.cpp` simulates evolution based on competition and selective survival of individuals with different phenotypes.

`ex-evol2.cpp` simulates evolution based on differential breeding where the breeding success depends on the phenotypes of both parents.

See also www.agner.org/evolution for a complete simulation program with graphical representation of the results.

These distributions are also available in the R-language package [BiasedUrn](#).

5.16 Games and entertainment applications

The quality of the random number generator is not very critical for entertaining games etc. Use the time as seed to get unpredictable randomness.

5.17 Gambling applications

Personally, I consider gambling an unethical exploitation of human psychological weaknesses. Consequently I do not endorse the use of this software for commercial gambling applications.

5.18 Security applications

The random number generators in the present packages are not intended for cryptographic applications. On most random number generators it is possible to reconstruct all past and future numbers in the random sequence from a subsequence of a certain length. This also applies to the generators used in the present packages. On the "linear feedback shift register" types (Mersenne Twister and SFMT generator) this is possible even without knowing the structure of the generator.

The reconstruction of a random sequence becomes much more difficult when two or more different random number generators of fundamentally different design are combined. Both generators should have a cycle length too long for trying all possible combinations. You may use the combination of the SFMT and the Mother-Of-All generator or the Mersenne Twister and the Mother-Of-All generator.

A single 32-bit seed can be a security problem because an attacker with a powerful computer can try all possible seeds in a reasonable amount of time. Use `RandomInitByArray` with multiple seeds of different origin. See chapter 5.5 on page 17 above.

These precautions should be taken if the random number generators are used for generating random passwords etc., but keep in mind that the present random number generators are not constructed specifically for security applications.

5.19 Error conditions

There is no standardized and portable way of generating error messages in a function library. I have not used structured exception handling because this could slow down the execution of error-free programs and because of compatibility problems across diverse platforms, compilers and programming languages.

The `randomc.zip` package has no specific error reporting. It is assumed that constructors are always called so that class data are initialized properly. The `IRandom` and `IRandomX` functions return 0x80000000 if $\max < \min$ or $\max - \min + 1 \geq 2^{32}$.

The `randoma.zip` package reports errors simply by provoking a divide-by-zero error. This may happen if a random number generator is not initialized before it is used or if the specified buffer size is insufficient. The `IRandom` and `IRandomX` functions return 0x80000000 if \max and \min are out of range as specified above.

The `stocc.zip` package reports errors by calling the `FatalError` function in `unserintf.cpp`. This happens if any parameter is out of range. You may modify the `FatalError` function to fit the user interface of your program.

6 Theoretical considerations

I will here describe the theoretical considerations behind the choice of random number generators in the present packages. The main criteria are: good randomness as evaluated by theoretical as well as experimental tests, long cycle length, and fast generation of random numbers.

A random number generator is typically based on a recursion of the form:

$$X_n = f(X_{n-1}, X_{n-2}, \dots, X_{n-k})$$

where the transition function f calculates each new value X_n from the k preceding values. The transition function f can use either integer (Euclidian) algebra modulo some value m , or *finite field* algebra.

The generators based on integer algebra use addition and/or multiplication. For example, a simple linear congruential generator has the form:

$$X_n = (aX_{n-1} + b) \bmod m$$

The modulo operation is easy to implement if $m=2^b$, where b is the number of bits in a computer word. To take a value modulo 2^b is simply to ignore the carry and use only the lower b bits of the result.

In some cases, a better randomness can be obtained by making m a prime. However, this implies a rounding error which most theorists have ignored. Assume, for example, that we have chosen $m = 2^{31}-1$, which is a prime. We want to convert X_n to a floating-point uniform random number $U_n = X_n/m$ in the interval $[0, 1)$. The representation of floating-point numbers (IEEE 754 standard) is quantified so that the maximum number of equidistant points in the interval $[0, 1)$ is a power of 2 (2^{24} , 2^{53} or 2^{64} , depending on the precision). To get equidistant values of U_n , we will need to space the values by 2^{-31} . There are only $2^{31}-1$ possible values of X_n , so one of the values in the interval $[0, 1)$ will be missing and the distribution will not be perfectly uniform. For this reason, I have chosen not to use any random number generator where m is not a power of 2.

If the transition function f contains only simple algebraic operations such as addition and multiplication then there is information flow from the least significant bits of the X values to the most significant bits through the carries, but no information flow in the opposite direction. The consequence of this is that the least significant bits of each X forms an independent random number generator with inferior randomness. The most significant bits are more random than the least significant bits. This is unacceptable since some applications may rely on the least significant bits. For example, it is quite common to have an application that tests whether X_n is odd or even, which is determined only by the least significant bit. To avoid this problem, it is necessary to establish a feedback from the most significant bits to the least significant bits. This is done in the multiply-with-carry generator, which adds the upper bits of a multiplication result to the lower bits. A multiply-with-carry generator with more than one factor is the Mother-Of-All generator invented by George Marsaglia:

$$S_n = a_1 X_{n-1} + \dots + a_k X_{n-k} + C_n$$

$$X_n = \text{lower } b \text{ bits of } S_n$$

$$C_n = \text{upper } b \text{ bits of } S_n$$

This generator has very good randomness and passes all tests in the powerful TestU01 battery of tests for randomness⁴. A minor drawback of this generator² is that it has a slight bias in the upper bits of X_n . This bias is too small to show in any experimental tests.

The present package uses a Mother-Of-All generator with $k = 4$ and $b = 32$. The transition function involves four multiplications of 32-bit factors into 64-bit products and addition of these 64-bit products. It is required that the compiler supports such 64-bit operations, or it must be coded in assembly language.

Some measures of randomness are better determined by theoretical analysis than by experiment. Most importantly, the cycle length should preferably be so long that it cannot be determined experimentally. The cycle length is the number of random numbers that can be generated before the sequence is repeated. The highest possible cycle length is equal to the number of different possible states in the state vector = 2^{kb} . In many cases, the cycle length is less than this value.

The theoretical analysis of a good random number generator can be very difficult. This is a serious dilemma: The random number generators that are easy to analyze theoretically tend to have poor randomness. If the random number generator has a simple mathematical structure that is easy to analyze, then it is also possible to construct a test that explores this structure and the generator will fail this test.

The best generators based on integer algebra with feedback from the high bits to the low bits are difficult to analyze theoretically. Thus, the theoretical properties of the Mother-Of-All generator have not been analyzed as thoroughly as one may wish.

Another class of random number generators that are easier to analyze theoretically are based on *finite field algebra*. Addition and multiplication in the finite field F_2^b are done simply by bitwise XOR and AND operations on b -bit integers (C operators \wedge and $\&$). These generators are known as Linear Feedback Shift Registers (LFSR). The much used *Mersenne Twister* belongs to this class of random number generators⁶. The transition function f consists of only XOR and AND operations, and shift operations for shuffling the bits. This type of generators can be constructed with extremely long cycle lengths.

The fact that LFSR generators have a relatively simple mathematical structure also means that it is possible to construct tests that they cannot pass. The *linear complexity test* can easily defeat any LFSR generator. This test is based on the Berlekamp-Massey algorithm which is an algorithm that detects the structure of an LFSR generator from any bit sequence it has generated⁴. It is no wonder that the LFSR generators fail this test because the

Berlekamp-Massey algorithm is in fact used during the construction of some LFSR generators⁹. An arguably more relevant test is the *binary matrix rank test*. All LFSR generators fail the binary matrix rank test when a sufficiently large matrix is used⁴. The bigger the state vector in the generator, the bigger a matrix is needed in the test before it fails. The standard Mersenne Twister (MT19937) has such a large state vector that it takes hours to execute a binary matrix rank test large enough to defeat it.

The Mersenne Twister has an output function $Y_n = g(X_n)$ where Y_n is used as the random number output. The output function g of the Mersenne Twister is called *tempering*⁶. The tempering function g simply shuffles and XOR's the bits in X_n with each other. You may say that this redistributes randomness rather than generate randomness because the value of Y_n is not fed back into the state vector. A Mersenne Twister without the tempering algorithm fails the important gap test. The tempering algorithm takes a significant part of the total execution time because it has a dependency chain that prevents parallel execution.

A good random number generator should have chaotic behavior¹. The degree of chaos is measured by a term which is called *bifurcation* in chaos theory. This is almost similar to the concept of *diffusion* in cryptology^{3,8}. The bifurcation is the divergence of two trajectories that differ in their starting point by only one bit in the state vector. The standard Mersenne Twister has very poor bifurcation. For example, it takes many steps to recover from a state where most of the bits are zero³.

Certain improvements have been made since the invention of the original Mersenne Twister. Two improved generators based on the same principle as the Mersenne Twister are the *WELL generator*⁸ and the *SFMT generator*⁹. Both have better randomness, better bifurcation/diffusion and higher speed than the original Mersenne Twister, and some versions don't require the tempering function.

While the WELL generator has better bifurcation/diffusion than the SFMT generator, I have chosen the latter for the sake of efficient implementation. The impressive speed of modern computers are to a considerable degree due to their ability to do multiple operations simultaneously. The amount of parallelism that can be obtained in the software implementation is limited by the shortest feedback path in the transition function f . The shortest feedback path is 32 bits in the WELL generator, but 128 bits in the SFMT generator. This makes it possible to do parallel operations in 128-bit SIMD (Single Instruction Multiple Data) registers with the SFMT generator, but not the WELL generator. None of these generators can fully use the 256-bit registers of the new AVX instruction set or further extensions that are expected to be available in future computers. The SFMT is among the fastest random number generators that satisfy our high requirements for randomness.

The SFMT generator is specifically designed to take advantage of the SIMD capabilities of modern computers. Such capabilities are standard in modern PC's (SSE2 or later instruction set), but absent in older PC's and some mainframe computers. The portability of the SFMT generator is therefore limited. The C++ implementation in the `randommc` class library requires that the computer has the SSE2 instruction set and that the compiler supports it. The implementation in the `randoma` library includes a branch for supporting old computers without SIMD/SSE2. On computers with non x86 instruction sets you need to use the original C implementation by Mutsuo Saito⁹. It would be possible to improve the bifurcation of the SFMT generator by using the new carry-less multiplication instruction, but nobody has explored this possibility yet.

A very efficient method of improving randomness is to combine the outputs of two or more different random number generators^{4,5}. In fact, you can get a good random number generator out of two or more bad ones, especially if they are very different. The philosophy behind this method is quite simple. Combining something non-random with something random produces something random. Any "non-randomness" that one of the generators

may have is eliminated by the other generator as long as the latter doesn't have the same type of weakness. Only if both generators have the same type of weakness will it show in the combined output. The combination of two random number generators can be as simple as generating a b -bit integer from each generator and adding these two numbers modulo 2^b . A particular random number generator is suitable for a particular application if there is no undesired interaction between generator and application. The risk of an undesired 3-way interaction between application, generator 1 and generator 2 is much smaller than the risk of an undesired 2-way interaction between the application and a single generator if the two generators are different. I have not been able to find any experimental evidence of undesired interactions between two random number generators, even if they were very similar in design.

I have implemented this principle by allowing the combination of the SFMT generator and the Mother-Of-All generator. These two generators are based on different kinds of algebra and are therefore very different. A generator based on integer algebra may fail certain tests based on integer algebra; and a generator based on finite field algebra is known to fail certain tests based on finite field algebra. But each generator eliminates the weaknesses of the other one so that the combined generator is as good as we can wish for. The advantages of the SFMT generator are long cycle length and high-order equidistribution⁹. The weaknesses are a relatively low bifurcation and the failure to pass certain tests based on finite field algebra. The advantages of the Mother-Of-All generator are a very high bifurcation and the fact that it passes the most stringent experimental tests for randomness. The disadvantages are a slight bias in the output and the fact that it is difficult to analyze theoretically so that it may have undetected theoretical weaknesses. All of these weaknesses are eliminated by combining the two generators. The fact that the two generators are based on fundamentally different algebras makes it unlikely that they have any noticeable weakness in common.

6.1 Using multiple streams

Many modern computers have multiple cores and the trend goes towards an increasing number of cores. Time-consuming applications can take advantage of this by dividing the work between multiple threads, with each core running one thread.

No random number generator is inherently thread-safe. This means that you cannot access it from more than one thread simultaneously without running the risk of messing up the internal state. Using a mutex is a very inefficient solution. It is much better to have one random number generator for each thread so that each thread has its own stream of random numbers. This can be done in C++ by making one instance (object) of the random number generator class in each thread.

Obviously, the multiple streams of random numbers should be different without any correlation between them. Four different ways of avoiding correlation between the streams have been proposed^{3,7}:

1. Use fundamentally different random number generators for each stream.
2. Use similar generators but with different values for various parameters in the generator algorithm, such as multiplication factors, shift counts and AND masks.
3. Use identical generators with a jump-ahead feature. If the first stream is expected to use at most L random numbers, then the second stream can jump ahead from the same starting point (seed) and skip the first L numbers.
4. Use identical generators with different seeds. The probability that the streams have overlapping sequences can be reduced to a negligible value if the cycle length is sufficiently long.

I have decided to use method 4 for the following reasons. Method 1 is not realistic because we have a limited number of random number generator algorithms with known good quality. Method 2 requires a computerized search for good values of the parameters in the generator algorithm. This search is too slow to be carried on online. Instead it is necessary to store a table with as many parameter sets as the maximum number of streams. This is actually feasible, but as the number of microprocessor cores may grow exponentially in the future according to Moores law, the program will be burdened with quite big tables in order to be suitable for future computers. Method 3 is only feasible if a fast jump-ahead method is available. Unfortunately, the fast jump-ahead feature comes at the cost of slowing down the basic generation of random numbers³.

Method 4 requires that the cycle length is very long. The probability that there is an overlap when you have s streams, each of length L , out of a total cycle length ρ is approximately

$$p = \frac{s(s-1)L}{2\rho}.$$

For example, if we make 100 streams of 10^{10} random numbers each from an SFMT generator with cycle length $\rho = 2^{11213}$, we have a probability of overlap $p \approx 10^{-3362}$. This probability is so small that we can rely on overlaps never happening. There is even plenty of room for future increases in the number of streams and their lengths.

A Mersenne Twister or a combined generator has even longer period, hence lower probability of overlap. A Mother-Of-All generator has a shorter cycle length so that it cannot be considered completely safe to generate multiple streams from a Mother-Of-All generator unless it is combined with some other generator with a long cycle length.

The above calculations are based on the assumption that each stream starts at a random point in the cycle of length ρ and that the starting points are independent. This requires a good seeding procedure. The seeding procedure fills the state vector with random numbers based on a seed which is typically a single integer. The seeding procedure used in the present software uses a separate random number generator of a different design in order to avoid any interference. An extra feature is the `RandomInitByArray` function which makes it possible to initialize the random number generator with multiple seeds. We can make sure that the streams have different starting points by using the thread id as one of the seeds.

6.2 Deciding the cycle length

There is no practical limit to how long we can make the cycle length. The advantages of a long cycle length are:

- The probability of overlapping subsequences is reduced.
- It is possible to obtain high-order equidistribution on generators with long cycle lengths.

The disadvantages of a long cycle length are:

- The search for good parameters becomes more difficult.
- The state vector becomes bigger. This takes more space in memory and cache and slows down cache-hungry applications.

The cycle length of 2^{19937} for the standard Mersenne Twister is actually excessive for most purposes. I have chosen to implement this cycle length nevertheless for the sake of

portability. Many software packages have a Mersenne Twister with this cycle length. I have chosen a somewhat shorter cycle length for the SFMT generator, but still long enough for even very demanding applications. The code can easily be changed to get a different cycle length.

7 File lists

Files in randomc.zip

ran-instructions.pdf	This file
randomc.h	Header file for the random number generator classes
sfmt.h	Header file for the SFMT generator
mersenne.cpp	Source code for Mersenne Twister generator
mother.cpp	Source code for Mother-Of-All generator
sfmt.cpp	Source code for SFMT generator
rancombi.cpp	Code for combining two generators
userintf.cpp	Functions that depends on user interface
ex-ran.cpp	Example program generating random numbers
testirandomx.cpp	Test difference between IRandom and IRandomx
license.txt	Gnu general public license
readme.txt	Short introduction

Files in randoma.zip

ran-instructions.pdf	This file
randoma.h	Header file for library functions
randomc.h	Header file for the random number generator classes
randomacof32.lib	Binary function library, 32-bit COFF format
randomacof64.lib	Binary function library, 64-bit COFF format
randomaomf32.lib	Binary function library, 32-bit OMF format
randomaelf32.a	Binary function library, 32-bit ELF format
randomaelf32p.a	Binary function library, 32-bit ELF format, position independent
randomaelf64.a	Binary function library, 32-bit ELF format
randomamac32.a	Binary function library, 32-bit Mach-O format
randomamac32p.a	Binary function library, 32-bit Mach-O format, position indepd.
randomamac64.a	Binary function library, 32-bit Mach-O format
randomad32.dll	Binary function library, 32-bit Windows DLL
randomad32.lib	Import library for randomad32.dll
randomad64.dll	Binary function library, 64-bit Windows DLL
randomad64.lib	Import library for randomad64.dll
testrandomac.cpp	Test comparing randomc.zip and randoma.zip libraries
license.txt	Gnu general public license
readme.txt	Short introduction
randomasrc.zip	Source code for binary libraries, assembly language

Files in randomasrc.zip

randomac.make	Makefile for all libraries and zip-files
MakeRandomac.bat	Batch program to make all libraries and zip-files
randoma.h	Header file for library functions
randomah.asi	Header file for assembly codes
mersenne32.asm	Source code for Mersenne Twister, 32 bit
mersenne64.asm	Source code for Mersenne Twister, 64 bit
mother32.asm	Source code for Mother-Of-All generator, 32 bit

mother64.asm	Source code for Mother-Of-All generator, 64 bit
sfmt32.asm	Source code for SFMT generator, 32 bit
sfmt64.asm	Source code for SFMT generator, 64 bit
instrset32.asm	Source code for InstructionSet function, 32 bit
instrset64.asm	Source code for InstructionSet function, 64 bit
rdtsc32.asm	Source code for ReadTSC function, 32 bit
rdtsc64.asm	Source code for ReadTSC function, 64 bit
physseed32.asm	Source code for PhysicalSeed function, 32 bit
physseed64.asm	Source code for PhysicalSeed function, 64 bit
randomad32.asm	Source code for DLL entry function, 32 bit
randomad64.asm	Source code for DLL entry function, 64 bit
randomad32.def	Module definition file for DLL, 32 bit
randomad64.def	Module definition file for DLL, 32 bit
randomad32.exp	Export definition for DLL, 32 bit
randomad64.exp	Export definition for DLL, 32 bit

Files in stocc.zip

ran-instructions.pdf	This file
distrib.pdf	Description of statistical distributions
sampmet.pdf	Description of sampling methods used
stocc.h	Header file for non-uniform random number generators
randomc.h	Header file for uniform random number generators
stoc1.cpp	Source code for Bernoulli, Binomial, Hypergeometric, Normal, Poisson, Multinomial, MultiHypergeometric and Shuffle
stoc2.cpp	Alternative source code for Binomial, Hypergeometric, Poisson
stoc3.cpp	Source code for noncentral hypergeometric distributions
wncypr.cpp	Code for Wallenius noncentral hypergeometric distribution
fnchypr.cpp	Code for Fisher's noncentral hypergeometric distribution
erfres.cpp	Auxiliary tables for Wallenius distribution
erfresmk.cpp	Program for making erfres.cpp
ex-stoc.cpp	Example program showing different distributions
ex-cards.cpp	Example program shuffling a deck of cards
ex-lotto.cpp	Example program producing random numbers without duplicates
ex-evol1.cpp	Example program simulating evolution with selective survival
ex-evol2.cpp	Example program simulating evolution with differential fertility
testbino.cpp	Test program for binomial distribution
testhype.cpp	Test program for hypergeometric distribution
testpois.cpp	Test program for Poisson distribution
testfnch.cpp	Test program for Fisher's noncentral hypergeometric distribution
testmfnc.cpp	Test program for multivariate Fisher's noncentral hyp. distrib.
testwnch.cpp	Test program for Wallenius' noncentral hyp. distrib.
testmwnc.cpp	Test program for multivariate Wallenius' noncentral hyp. distrib.
license.txt	Gnu general public license
readme.txt	Short introduction

8 License conditions

These software libraries are free: you can redistribute the software and/or modify it under the terms of the GNU General Public License as published by the [Free Software Foundation](#), either version 3 of the License, or any later version.

Commercial licenses are available on request to www.agner.org/contact.

This software is distributed in the hope that it will be useful, but *without any warranty*, without even the implied warranty of merchantability or fitness for a particular purpose. See the file `license.txt` or www.gnu.org/licenses for the license text.

The copyright shall be terminated when I die or no later than year 2056. No claim to copyright ownership of this software and documentation shall be valid thereafter.

9 No support

Note that this is free software provided without any warranty or support. It is intended for skilled programmers only, and it may not be compatible with all compilers and linkers. If you have problems using it, then don't.

I am sorry that I don't have the time and resources to provide support for this software. If you ask me to help with your programming problems then you will not get any answer.

10 Literature

1. Cernak, J: Digital Generators of Chaos. *Physics Letters A*, vol. 214, 1996, pp. 151-160.
2. Couture, R; L'Ecuyer, P: Distribution properties of Multiply-With-Carry Random Number Generators. *Mathematics of Computation*, Vol. 66, p. 591-607, 1997.
3. L'Ecuyer, P; Panneton, F: Fast Random Number Generators based on Linear Recurrences Modulo 2: Overview and Comparison. *Proceedings of the 2005 Winter Simulation Conference*. M. E. Kuhl et. al. eds. 2005. www.iro.umontreal.ca/~lecuyer/papers.html
4. L'Ecuyer, P; Simard, R: TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Transactions on Mathematical Software*, vol. 33, no. 4, 2007. www.iro.umontreal.ca/~simardr/testu01/tu01.html
5. Marsaglia, G: A Current View of Random Number Generators. *Proc. Computer Science and Statistics: 16th Symposium on the Interface, Atlanta 1984*. Elsevier Press.
6. Matsumoto, M; Nishimura, T: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, 1998, pp. 3-30.
7. Matsumoto, M; Nishimura, T: Dynamic Creation of Pseudorandom Number Generators. In: Niederreiter, H; Spanier, J., eds: *Monte Carlo and Quasi-Monte Carlo Methods 1998*. Springer, 2000, pp 56-69. www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html
8. Panneton, F; L'Ecuyer, P; Matsumoto, M: Improved Long-Period Generators Based on Linear Recurrences Modulo 2. *ACM Transactions on Mathematical Software*, vol. 32, no. 1, 2006, pp. 1-16.
9. Saito, M; Matsumoto, M: SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In: Keller, A; et. al., eds: *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Springer, 2008, pp. 607-622. www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/.