

RandomOps

Pseudo-Random Number Generator
Source-Code Library for ANSI C
The Manual
Revision 1.1

By

Magnus Erik Hvass Pedersen

March 2008

Copyright © 2008, all rights reserved by the author.
Printing & distribution for personal and academic use allowed.
Commercial use requires written consent from the author.
Please see page 5 for license details.

RandomOps

Contents

Contents	2
1. Introduction.....	3
1.1 Installation.....	4
1.2 License	5
1.3 Updates.....	5
2. Tutorial	6
2.1 Header-File.....	6
2.2 Initializing	6
2.3 Drawing Random Numbers	7
2.4 Probabilities.....	7
2.5 Random Sets.....	8
2.6 Shutting Down.....	9
3. Reference Manual	10
3.1 Error Handling.....	10
3.2 Basic Functions	10
void RO_RandSeed(RO_Int seed)	10
void RO_RandSeedClock(RO_Int seed).....	11
RO_Int RO_Rand(void).....	11
RO_Int RO_RandMax(void).....	11
double RO_RandUni().....	11
double RO_RandBi().....	11
double RO_RandBetween(const double lower, const double upper)	12
double RO_RandGauss(double mean, double deviation)	12

RandomOps

size_t RO_RandIndex(size_t n)	12
void RO_RandIndex2(size_t n, size_t *i1, size_t *i2).....	12
int RO_RandBool()	12
3.3 Random Sets.....	13
struct RO_RandSet	13
struct RO_RandSet RO_RandSetInit(size_t size).....	13
void RO_RandSetFree(struct RO_RandSet *s).....	13
void RO_RandSetReset(struct RO_RandSet *s)	13
void RO_RandSetResetExclude(struct RO_RandSet *s, size_t index)	14
size_t RO_RandSetDraw(struct RO_RandSet *s, RO_FRandIndex fRandIndex)	14
Bibliography	15

1. Introduction

RandomOps is a small source-code library implementing a high quality Pseudo-Random Number Generator (PRNG) in the ANSI C programming language. The PRNG implemented in RandomOps is based on the rand2 algorithm from (1), but is implemented anew here with a number of modifications. Furthermore, the license allows RandomOps to be used and improved in both scientific and commercial settings (see below for license details).

This manual describes how to use RandomOps and offers a reference for all of its functions. The manual is divided into the following chapters:

- Chapter 1 is this introduction which includes an installation manual.
- Chapter 2 is a brief tutorial for getting started quickly.

- Chapter 3 is a reference manual giving more detailed information on implementation aspects of RandomOps as well as an account of its functions and datatypes.

1.1 Installation

To install RandomOps do the following:

- Download the RandomOps source-code from <http://www.hvass-labs.org/>
- Unpack the source-code to a suitable directory.

Now you can do either of two things to use RandomOps in your own source-code.

The easiest is:

- Simply add the header- and source-files of RandomOps directly to your own project, to be compiled along with your own source-files.
- Include the file Random.h in your own source-code.

Or if you require RandomOps in several of your own projects, you may wish to do the following:

- Build a RandomOps library. The RandomOpsLib project does this for MS Visual C++, although you may have to modify the project settings to match your own directory structure. If you use another compiler you must make a similar project yourself.
- Add the path where you unpacked the RandomOps source-code as an include-path in your own project.
- Add the output-path of RandomOpsLib as a library-path of your own project.
- Add the name of the RandomOps library-file (e.g. RandomOpsLib.lib) to the list of libraries that will be linked with your project.

- Include the file `<RandomOps/Random.h>` in your own source-code.

1.2 License

The RandomOps source-code is published under the GNU Lesser General Public License (2), which essentially means that you may distribute commercial programs that link with the RandomOps library, as well as make alterations to the RandomOps library itself. There are certain terms to be met though, but for those details please see the license included in the source-code distribution.

This manual may be downloaded, printed, and used for any personal purpose, be it commercial or non-commercial (provided the author(s) are not held responsible for your actions, or any damage caused by your use of the manual.) If you want to distribute the manual commercially, for example in a printed book, or on a web-page that requires payment, then you must obtain a license from the author(s).

1.3 Updates

To obtain updates to the RandomOps source-code library or to get newer revisions of this manual, go to the library's webpage (<http://www.Hvass-Labs.org/>).

2. Tutorial

This chapter offers a basic tutorial for getting started with using RandomOps. It assumes that you have already installed the library, as described above.

2.1 Header-File

First you need to include the Random.h header-file. This provides most of the functions of the RandomOps library. It is done by adding the following statement to the top of your own source-code file:

```
#include <RandomOps/Random.h>
```

2.2 Initializing

To initialize the PRNG in RandomOps you must call either of the RO_RandSeed() or RO_RandSeedClock() functions when your program starts. The former function is for when you wish to use the same seed for multiple runs of the program, so as to cause identical streams of random numbers to be created. And the latter function is for when you wish a new seed for each run of the program, and hence create different streams of random numbers for each run. A good place to call the seed-function is at the beginning of the main() function, something like this:

RandomOps

```
int main(int argc, char* argv[])
{
    const RO_Int kSeed = 39873985;
    RO_RandSeedClock(kSeed);
    /* Other code ... */
    return 0;
}
```

Note that you must supply a default seed for the clock-based seeding-function, in case the wall clock of the computer does not work. A 6-8 digit number that you type in at random will work fine.

2.3 Drawing Random Numbers

Now the PRNG is ready to be used and many people will only need to use the `RO_RandUni()`, `RO_RandGauss()`, and `RO_RandIndex()` functions, for example as follows:

```
/* Print a random number between zero and one. */
printf("%g\n", RO_RandUni());

/* Print a Gaussian or normal-distributed number. */
printf("%g\n", RO_RandGauss(0, 1));

/* Print a random integer between 0 and n-1. */
printf("%i\n", RO_RandIndex(0, n));
```

2.4 Probabilities

Many stochastic experiments need choices to be made depending on certain probabilities. Let $p \in [0,1]$ be such a probability, with $p = 0$ meaning that some event will never occur, $p = 1$ meaning the event always occurs, and, say, $p = 0.5$ means

RandomOps

the event occurs half the time, and $p = 0.1$ means the event occurs every tenth time. To simulate this using a PRNG we need to draw a random number between zero and one and compare it to p , so as to decide whether the event should occur or not. This can be done as follows:

```
/* The probability p. */
const double p = 0.1;

/* Other code ... */

if (RO_RandUni() < p)
{
    /* Do some event. */
}
else
{
    /* Don't do the event. */
}
```

Since the output of `RO_RandUni()` is uniformly distributed on the range $(0,1)$, this code behaves as expected because the condition is true approximately once every $1/p$ time it is executed.

2.5 Random Sets

A Random Set basically just contains a range of indices $\{0, \dots, \text{size} - 1\}$ that may be drawn at random until the set is finally empty. First the `RandomSet.h` header-file must be included:

```
#include <RandomOps/RandomSet.h>
```

Then the Random Set may be used as follows:

RandomOps

```
const size_t kRandSetSize = 8;
struct RO_RandSet randSet = RO_RandSetInit(kRandSetSize);
RO_RandSetReset(&randSet);
printf("%i\n", RO_RandSetDraw(&randSet, &RO_RandIndex));
printf("%i\n", RO_RandSetDraw(&randSet, &RO_RandIndex));
printf("%i\n", RO_RandSetDraw(&randSet, &RO_RandIndex));
RO_RandSetFree(&randSet);
```

2.6 Shutting Down

There is no code needed to shut down the PRNG, so the program can just be terminated without paying any special attention to the PRNG.

3. Reference Manual

This chapter documents various implementation aspects of RandomOps, along with a detailed account of its functions and datatypes.

3.1 Error Handling

The error handling in RandomOps is done primarily through use of the `assert()` function. Assertions are used to check internal states of the PRNG as well as its output, but also for checking various aspects of the computer that the PRNG is running on. The advantage of assertions is that they do not compile into the optimized release-build of the library, but only exist in the debug-version where they trigger exceptions if the assertions are not met. Since the use of a PRNG can normally be checked quite thoroughly during development, this approach is considered the best in terms of runtime efficiency and ease of development.

3.2 Basic Functions

The basic functions of RandomOps are all available when including the `Random.h` header-file. The functions are as follows:

`void RO_RandSeed(RO_Int seed)`

This is used to seed the PRNG with the given seed. You may use this function if you wish to be certain that the seed is the same for each time you call the function. Otherwise the `RO_RandSeedClock()` function is generally better to use.

RandomOps

void RO_RandSeedClock(RO_Int seed)

Tries to seed the PRNG with the current time of the wall clock. If this is not possible it reverts to automatically calling `RO_RandSeed()` with the given seed. This function is often preferred for seeding the PRNG, because it will generally give different seeds for each time the PRNG is initialized.

RO_Int RO_Rand(void)

Returns a random integer from the range $\{1, \dots, \text{RO_RandMax}()\}$. The number returned by `RO_Rand()` is supposed to be uniformly distributed on the entire range, although for the `rand2` algorithm which the `RandomOps` implementation is based on, the lower-order bits are not very random. Note that the internal datatype for the PRNG is `RO_Int`.

To obtain other kinds of random numbers, such as floating points in the range $[0,1]$, you should generally use the functions listed below, such as `RO_RandUni()`, `RO_RandGauss()`, and `RO_RandIndex()`.

RO_Int RO_RandMax(void)

Returns the largest number that may be returned by the `RO_Rand()` function.

double RO_RandUni()

Return a uniform random number in the range $(0,1)$, excluding the endpoints.

double RO_RandBi()

Return a uniform random number in the range $(-1,1)$, excluding the endpoints.

RandomOps

double RO_RandBetween(const double lower, const double upper)

Return a uniform random number in the range (*lower*, *upper*). Due to rounding errors of floating point numbers, the endpoints may be returned if *lower* and *upper* are very close. Similarly, if you experience a runtime error with this function in debug-mode, you should remove the assert() statements in the implementation in the file Derivations.c.

double RO_RandGauss(double mean, double deviation)

Return a normal or Gaussian distributed random number with the given mean and deviation.

size_t RO_RandIndex(size_t n)

Return a random number from the set $\{0, \dots, n - 1\}$ with uniform probability.

void RO_RandIndex2(size_t n, size_t *i1, size_t *i2)

Assigns two distinct and random numbers to the variables pointed to by *i1* and *i2*, from the set $\{0, \dots, n - 1\}$. The first one is picked by RO_RandIndex(*n*), and the second is picked by RO_RandIndex(*n*-1), and then basic arithmetic operations (addition and modulo) are used to ensure the two numbers are different.

int RO_RandBool()

Return a boolean value with equal probability for the values zero (false) and one (true).

3.3 Random Sets

RandomOps has the ability to draw numbers randomly from a set of indices. These Random Sets do not require the rest of the RandomOps library to work, and may therefore be used with another PRNG as well. To use Random Sets you must include the file `RandomSet.h` in your source-code, and use the functions listed below.

struct RO_RandSet

This is the internal datastructure for keeping track of the Random Set. The user is not supposed to manipulate this datastructure but merely pass it to various functions.

struct RO_RandSet RO_RandSetInit(size_t size)

Used to create the `RO_RandSet` datastructure and initialize it with the given set-size. The Random Set will then be able to contain the elements $\{0, \dots, \text{size} - 1\}$, but requires for one of the Reset-functions below to be called before it can be used.

void RO_RandSetFree(struct RO_RandSet *s)

Frees the internal elements of the `RO_RandSet` datastructure; but not the `RO_RandSet` datastructure itself.

void RO_RandSetReset(struct RO_RandSet *s)

Resets the Random Set and makes it ready for use. After this the Random Set contains all the elements $\{0, \dots, \text{size} - 1\}$ which may be drawn at random, until the Random Set is empty. Resetting may be done before the Random Set is entirely empty. This function has $O(\text{size})$ time complexity.

RandomOps

void RO_RandSetResetExclude(struct RO_RandSet *s, size_t index)

Same as RO_RandSetReset() but removes the given element from the set.

size_t RO_RandSetDraw(struct RO_RandSet *s, RO_FRandIndex fRandIndex)

Randomly draw a number and remove it from the set. The function has $O(1)$ time-complexity. The second parameter to this function is a pointer to a function similar to RO_RandIndex() above, in case you want to use Random Sets with your own PRNG.

Bibliography

1. **Press, William H., et al.** *Numerical Recipes in C, The Art of Scientific Computing*. s.l. : Cambridge University Press, 1992.
2. **Free Software Foundation.** *GNU Lesser General Public License*. URL <http://www.gnu.org/copyleft/lesser.html>.