RandomOps

Pseudo-Random Number Generator Source-Code Library for ANSI C

The Manual

Revision 1.2

By

Magnus Erik Hvass Pedersen

May 2008

Contents

Contents 2
1. Introduction 3
1.1 Manual Outline 3
1.2 License 3
1.3 Updates 4
2. Installation 5
2.1 Mirror Development Path 5
2.2 New Development Path 5
2.3 Direct Compiling 6
2.4 New Link-Library 6
3. Tutorial 7
3.1 Header-File 7
3.2 Initializing 7
3.3 Drawing Random Numbers 8
3.4 Probabilities 8
3.5 Random Sets 9
3.6 Shutting Down 10
4. Reference Manual 11
4.1 Error Handling 11
4.2 Basic Functions 11
4.3 Random Sets 14
Bibliography 16

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 ran2 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).

1.1 Manual Outline

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.
- Chapter 2 is an installation manual.
- Chapter 3 is a brief tutorial for getting started quickly.
- Chapter 4 is a reference manual giving more detailed information on implementation aspects of RandomOps as well as an account of its functions and datatypes.

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.

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. Installation

This chapter describes the various ways of installing and using the RandomOps source-code library. The chapter assumes you have already downloaded the latest source-code archive through the internet website: http://www.Hvass-Labs.org/
After you have got the source-code to compile see the tutorial in chapter 3 on how to include and use RandomOps in your own program.

2.1 Mirror Development Path

The easiest way to install and use RandomOps for Microsoft Visual C++ is to mirror the directory path of the development computer. All the ANSI C libraries from Hvass Laboratories are located in the following path on that computer:

C:\Users\Magnus\Documents\Development\Libraries\HvassLabs-C

To install the RandomOps source-code simply un-zip the archive to this path, and you should be able to open and use the MS Visual C++ projects as is.

2.2 New Development Path

If you do not wish to mirror the directory path of the development computer, but would still like to reuse the MS Visual C++ projects included with the RandomOps library, then you will have to manually alter all paths in the compilation projects. The compiler should inform you whenever you have an incorrect path that needs to be changed.

2.3 Direct Compiling

Another easy way of using RandomOps in your own source-code is to add all header- and source-files of RandomOps directly to your own project, to be compiled along with your own source-files. This however, is somewhat awkward if you have multiple projects where you need to use the RandomOps source-code, in which case you ought to build a separate RandomOps library, as described next.

2.4 New Link-Library

If you require RandomOps in several of your own projects but are unable to mirror the development path, or perhaps you're using a different ANSI C compiler, then you may wish to create a separate link-library yourself. How to do this differs from compiler to compiler, but after the project for the link-library has been created and compiled, you just include the appropriate RandomOps header-files (see the tutorial in chapter 3) and link with the library you just created. If you furthermore create a compilation dependency for the RandomOps link-library, then you are always ensured the RandomOps library is compiled first.

3. 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.

3.1 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>
```

3.2 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:

```
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.

3.3 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:

```
/* 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));
```

3.4 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 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.

3.5 Random Sets

A Random Set basically just contains a range of indices $\{0,\ldots,\text{size}-1\}$ 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:

```
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);
```

3.6 Shutting Down

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

4. Reference Manual

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

4.1 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.

4.2 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.

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 $\{1, .., \text{RO\_RandMax}()\}$. 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.

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 $\{0, .., n-1\}$ 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 $\{0, .., n-1\}$. 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).

4.3 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 $\{0, \ldots, \text{size} - 1\}$, 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 $\{0, \ldots, \text{size} - 1\}$ 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.

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.*