RandomOps

Pseudo-Random Number Generator Source-Code Library for C#
The Manual
Revision 2.1

By

Magnus Erik Hvass Pedersen

November 2010

Contents

Contents 2
1. Introduction 3
1.1 Installation 3
1.2 License 3
2. Features 4
2.1 The RNG Object 4
2.2 Generators 5
2.3 Internet Retrieval 5
2.4 Manipulators 6
2.5 Thread-Safety 6
3. Tutorial 7
3.1 Global Object 7
3.2 Basic Usage 7
3.3 Index 7
3.4 Random Sets 8
3.5 Internet Retrieval 8
3.6 Array-Seeding 10
3.7 Thread-Safety (Locking) 11
3.8 Thread-Safety (Non-Locking) 12
Bibliography 13

1. Introduction

RandomOps is a source-code library in the C# programming language that implements several Pseudo-Random Number Generators (PRNGs) as well as retrieval of "true" random numbers through the internet. As the source-code is thoroughly documented, this manual merely lists the main ideas and features of RandomOps, as well as giving tutorials on its use. To obtain updates to the source-code or manual go to the webpage www.hvass-labs.org

1.1 Installation

To install RandomOps follow these simple steps:

1. Unpack the RandomOps archive to a convenient directory.
2. In MS Visual Studio open the Solution in which you will use RandomOps.
3. Add the RandomOps project to the solution.
4. Add a reference to RandomOps in all the projects which must use it.

1.2 License

2. Features

2.1 The RNG Object

The basic function any RNG must implement is the generation of a floating point number uniformly distributed in the exclusive range (0,1). In RandomOps the Random-class has an abstract method called Uniform() which must be implemented to do just that. A number of other methods have then been implemented using Uniform(), such as Gauss() for generating a Gaussian (or normally) distributed random number, Sphere() for generating uniformly distributed points on the n-dimensional hypersphere, and Index(n) for generating a random integer between zero and $n-1$. When adding a new RNG to RandomOps you will therefore only have to implement a minimum of functionality as common methods can be reused from the base-class. The class-hierarchy for RandomOps is shown in Figure 1.

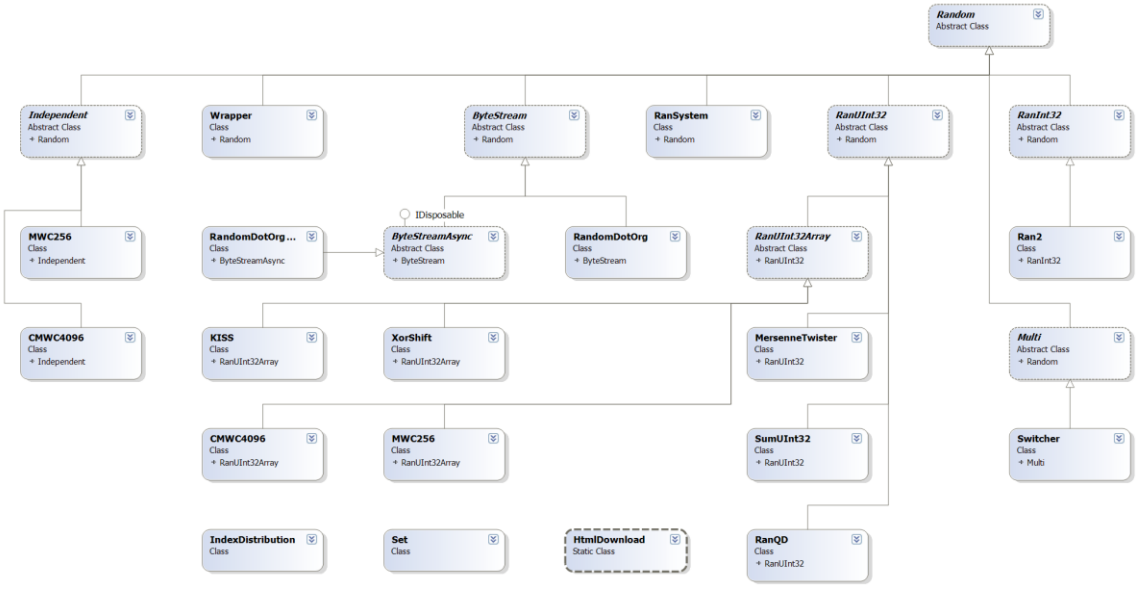

Figure 1 Class hierarchy for RandomOps.

2.2 Generators

Several PRNGs have been supplied with RandomOps:

- RanQD, based on the Quick-and-Dirty generator from (1).
- Ran2, based on the Ran2 generator from (1).
- RanSystem, which just uses the .NET implementation of a PRNG, whatever that may be (this could presumably change in the future.)
- MersenneTwister, which is the Mersenne Twister 19937 from(2).
- KISS, XorShift, MWC256 and CMWC4096 from (3).
- Parallel versions of MWC256 and CMWC4096.

2.3 Internet Retrieval

Perhaps unique to RandomOps is the ability to seamlessly retrieve random bits from the internet website www.random.org as if it was any other RNG. A fallback-PRNG is employed so that if the internet retrieval fails for whatever reason, the fallback-PRNG is used to generate a pseudo-random number instead. Two modes of retrieval are supported through the use of the RandomDotOrg- and RandomDotOrgAsync-classes. In both cases a buffer is maintained, but in the synchronous version the calls to get random numbers will stall if the buffer is empty and has to be refilled, and the fallback-PRNG will only be used if an error occurs, such as internet connection errors. In the asynchronous version the retrieving and buffering of random bits occurs in a separate worker-thread, and when the buffer is found to be empty the fallback-PRNG will be used immediately without waiting for the buffer to be refilled. This means the async-version is best used in time-critical scenarios.

2.4 Manipulators

Because of the object oriented framework of RandomOps it is easy to implement combinations of PRNGs. Included is a simple example called Switcher, which uses a PRNG to decided what other PRNG to use in generating the next random number.

2.5 Thread-Safety

Thread-safety is transparently supported so you can very easily extend your program to be multi-threaded without needing to rewrite all the source-code that made use of the RNG when only a single execution thread was being used.

3. Tutorial

3.1 Global Object

You will most likely use only one PRNG for your entire application. To achieve this in C# you must make a static class containing the PRNG object. Let us call the class Globals, implemented as follows:

```
namespace Test
{
    public static partial class Globals
    {
        public static RandomOps.Ran2 Random =
                                                new RandomOps.Ran2();
    }
}
```

3.2 Basic Usage

Once you have your global PRNG object defined, you can make calls to it, e.g.:

```
namespace Test
{
    static void Main(string[] args)
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(Globals.Random.Uniform());
        }
    }
}
```

This will print 10 uniformly distributed random numbers in the range $(0,1)$.

3.3 Index

The Index() method of an RNG object is used to draw random integers that are uniformly distributed in a given range. A probability distribution can also be used to

draw the integers non-uniformly, the probabilities should then sum to one. If repeated drawings are made from the same distribution then an IndexDistribution object can be created for better runtime performance, e.g:

```
double[] probabilities = {0.1, 0.2, 0.3, 0.4};
RandomOps.IndexDistribution indexDistribution =
   new RandomOps.IndexDistribution(Globals.Random, probabilities);
for (int i = 0; i < 1000; i++)
{
    Console.WriteLine(indexDistribution.DrawLinearSearch());
}
```

For large indices use DrawBinarySearch() which has logarithmic time-complexity. For small indices use DrawLinearSearch() which has linear time-complexity. Use the RNG-object's Index() method if you just need to draw one integer.

3.4 Random Sets

The Set-class provides a set of integers enumerated from zero and upwards that can be drawn at random until the set is empty. Note that an RNG object must be supplied at construction, and also note that the set must be initialized (or Reset), e.g.:

```
int RandSetSize = 8;
RandomOps.Set RandSet =
                  new RandomOps.Set(Globals.Random, RandSetSize);
RandSet.Reset();
for (int i = 0; i < RandSetSize; i++)
{
    Console.WriteLine(RandSet.Draw());
}
```

3.5 Internet Retrieval

To retrieve "true" random numbers through the internet you have several options. First you must decide upon which PRNG to use as a fallback generator in case the internet server is unavailable, let us say you want to use Ran2 for this. Then you

must decide whether to use sync or async retrieval from the internet; often you will use async for speed. Usage of this composite RNG is completely transparent as only the definition in the Globals-class needs to be changed:

```
namespace Test
{
    public static partial class Globals
    {
        static readonly int BufferSize = 16384;
        static readonly int NumFallback = 32768;
        static readonly int RetrieveTrigger = 4096;
        static RandomOps.Ran2 Fallback = new RandomOps.Ran2();
        public static RandomOps.RandomDotOrgAsync Random =
                  new RandomOps.RandomDotOrgAsync(BufferSize,
                      RetrieveTrigger, Fallback, NumFallback);
    }
}
```

Here we make use of a buffer containing 16384 bytes retrieved from the internet, and when the buffer gets depleted we will use the fallback PRNG for the next 32768 bytes. Whenever the remaining number of bytes available in the buffer drops below 4096 bytes, the buffer will be attempted filled again. You may have to call Dispose() to terminate the worker-thread after you are done using the RNG. If you need to ensure that the random numbers you draw are really the “true” random numbers retrieved through the internet and not from the fallback generator, you must do something like the following, and preferably use the sync-version of the RandomDotOrg-class with numFallback set to zero in the object construction:

```
// Wait for 'true' random bytes to become available.
while (!Globals.Random.IsAvailableUniform())
{
    // We could sleep the thread here for a while ...
}

double d = Globals.Random.Uniform();
```

The internet server that supplies the random bytes sets a limit on the number of bytes you can download per day (registered by the IP-address), so if you need many random numbers you may want to use the ones retrieved through the internet to seed another PRNG, e.g. the CMWC4096 as described next.

3.6 Array-Seeding

Some of the PRNGs must be seeded with an array of integers. This seed-array can be generated automatically by another PRNG, or it can be "true" random bytes downloaded through the internet, e.g.:

```
namespace Test
{
    public static partial class Globals
    {
        static RandomOps.CMWC4096 Random =
           new RandomOps.CMWC4096();
        static readonly int BufferSize = Random.SeedLength;
        static readonly int NumFallback = BufferSize;
        static RandomOps.Ran2 Fallback = new RandomOps.Ran2();
        public static RandomOps.RandomDotOrg RandInternet =
                  new RandomOps.RandomDotOrg(BufferSize,
                      Fallback, NumFallback);

        static void Initialize()
        {
            Random.Seed(RandInternet);
        }
    }
}
```

Call Initialize() somewhere in your code to seed the PRNG before using it.

3.7 Thread-Safety (Locking)

There are different ways to make an RNG object thread-safe. For infrequent access to the RNG object the easiest way is to use a lock and one way to do this is to explicitly lock the object every time you use it, e.g.:

```
double d1, d2, d3;

lock (Globals.Random)
{
    d1 = Globals.Random.Uniform();
    d2 = Globals.Random.Uniform(-2, 3);
    d3 = Globals.Random.Gauss();
}
```

This way of locking the RNG object is recommended if you need to draw several random numbers in succession, but it is cumbersome when just drawing single random numbers. There you would use ThreadSafe.Wrapper for the Globals.Random object, as follows:

```
namespace Test
{
    public static partial class Globals
    {
        static RandomOps.Ran2 RandNotTS = new RandomOps.Ran2();
        public static RandomOps.ThreadSafe.Wrapper Random =
                    new RandomOps.ThreadSafe.Wrapper(RandNotTS);
    }
}
```

This makes the locking transparent so you do not need to do it yourself every time you draw a number from the Globals.Random object, but if you need to draw several random numbers in succession, it is still recommended you lock the Globals.Random object as shown above. This works because C# supports nested locking. It is important to note that you must no longer call methods directly on the RandNotTS object because it is not locked, you must always call methods on the Globals.Random object.

3.8 Thread-Safety (Non-Locking)

If you need thread-safe and frequent access to the RNG object, for instance in a parallelized stochastic simulation where you draw millions of random numbers, then the use of locks is much too slow. There you need to use a specialized parallel PRNG, such as:

```
namespace Test
{
    public static partial class Globals
    {
        static RandomOps.ThreadSafe.CMWC4096 Random = new
                                    RandomOps.ThreadSafe.CMWC4096();
    }
}
```

This automatically and transparently creates a PRNG object for each thread that can be accessed without the use of locks for faster execution.

Bibliography

1. *Numerical Recipes in C, The Art of Scientific Computing.* **Press, William H., et al.** s.l. : Cambridge University Press, 1992.

2. *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator.* **Matsumoto, M. and Nishimura, T.** 1, January 1998, ACM Transactions on Modeling and Computer Simulation, Vol. 8, pp. 3-30.

3. *Random Number Generators.* **Marsaglia, G.** 1, s.l. : Journal of Modern Applied Statistics, 2003, Vol. 2, pp. 2-13.