Some Random Advice

Perrin Westrich

Executive Summary:

Going through life there one may notice that things happen that seem unpredictable. These events may even appear to be random (gasssp hisss). But for those of us who live in shells where all interaction to the outside world is through their good friend the Computer (myself… maybe?) may notice things to be a bit more predictable; maybe even deterministic you might say. Well now we say we want change; change that puts a bit of unpredictability, but from the confines of our own shells mind you. At about this time you might hear the seeds of the pseudo random number generators hit the ground and the footsteps of their numbers slowly lumber into the scene that I have (so eloquently?) painted (probably typed is a more accurate word as I am not one for physically doing stuff…) for all of (a single if that) the readers. Now that your minds have been seeded with a mind boggling attempt to give a hint as to what this all about (as if this incoherent banter had a point!), lets get down to business.

In the world around us there are things that we consider to be random. Random that individual elements of a set are unpredictable by any deterministic algorithm, but as a whole they form a probability distribution. As an example imagine a setting with no gravity, and a water balloon exploding. Droplets of water shooting out in all directions. Can't predict where individual droplets of water will end up striking any surfaces, however the arrangements of the droplets will likely approximate a distribution that looks something like: $\frac{1}{distanceFromExplosion^2}$ Meaning essentially the further from the explosion the less droplets of water will hit a fixed size surface.

Trying to create random numbers on a computer presents a problem, because a computer is a deterministic machine. Any method that strictly uses algorithms on a computer to attempt to generate random numbers will not succeed. Using this type of method would have to approximate random numbers, these approximations are called pseudo random numbers. Must of the time the type of distribution that is approximated by the computer is a uniform distribution; every point has an equal probability of occurring. A mechanism that generates pseudo random numbers is called a pseudo random number generator (PRNG). A PRNG would make the numbers that it generates as unpredictable as possible. Note that just because something is unpredictable does not mean that it is random. The length of sequence a PRNG produces before it starts to repeat is called its period. Generally a larger period is better for a PRNG.

A good PRNG (Uniform distribution) should have the following properties:

1. Stands up to statistical analysis and remains unpredictable.
2. Has a long period.
3. Stands up to attacks that attempt to calculate future values in the sequence.
4. Stands up to attacks to determine the inner state of the generator.

There are statistical tests to test how well a PRNG approximates random or in other words tests to see how unpredictable it is. One series of tests is called the 'Diehard Battery of Tests of Randomness'. The Diehard test sweet contains 15 tests for randomness. The test requires a rather large sequence of numbers to be generated; on the order of 2.9 million. Each test produces a p-value, which is between 0 and 1. According to the program when the p-value is 0 or 1 to more than 6 figures than the PRNG has really failed the test.

I used the Diehard tests to test three different PRNGs. The first of which was a Linear Congruential Generator that is supplied with compilers for C/C++. The second which is an Inversive generator, and another called the Mersenne Twister. The technical details of each are gone over in analysis section of the report. Each of the generators were run through the Diehard tests multiple times with various seed values. To ensure that the results were correct. The results are listed below. The benefits are listed with a ☺ and the deficiencies are listed with a ☹.

Linear Congruential Generator supplied with C/C++ standard libraries

1. ☺It is supplied with the standard libraries, which makes it very accessible.
2. ☺It doesn't require a very large seed, so it is quick to initialize.
3. ☹It doesn't produce very high quality pseudo random numbers.
4. ☹Shorter period than most generators. It can only hit each number in the range that it produces at most once for a given seed.

Inversive Generator

1. ☺Small seed quick to start up.
2. ☹Harder to obtain the code.
3. ☹Takes much longer than most generators due to finding the inverse of numbers in a given mod system.

Mersenne Twister

1. ☺Produces much higher quality random numbers than the built in generator for C/C++
2. ☺Very very long period.
3. ☺Fairly fast at producing numbers.
4. ☹Requires a large seed, takes longer to initialize.
5. ☹More complex to understand, although this doesn't really matter to an end user.

If the quality of the pseudo random number doesn't matter too much then it is probably easier to just use the built in C/C++ rand function. However, if higher quality pseudo random numbers are required, then I would recommend using a better algorithm. Of the ones that I looked at I would recommend the Mersenne Twister for this situation. I would not recommend the Inversive Generator for most situations, because it is slower and from my analysis produces slightly less quality pseudo random numbers than Mersenne Twister.

Problem Description:

On occasion it is useful to have random numbers generated for a project. Why not just use the random number generator that is built into the language that is being used? What it really comes down to is what is the generator needed for. More randomness required for security purposes. Is speed of the generator important? These are some questions that might need to be asked before choosing a random number generator. This paper will try to explore various random number generators, and some test specifically the diehard tests to determine the level of 'randomness' for a pseudo random number generator.

Analysis Technique:

There are many different uses for random numbers in computer applications such as: simulations, games, cryptography/computer security, are just a few examples. But what exactly are random numbers? Most everyone has an idea as to what they are, but is it the correct idea. A random sequence does not any deterministic pattern however, it follows probability distribution. <Randomness> This means that predicting individual elements in the sequence should be impossible, but as a whole the sequence tends to have a given structure. Generally a uniform distribution is desired, where every point is equally probable.

Trying to generate a random number sequence on a computer, a deterministic machine, becomes a problem due to the basic definition of randomness. Attempts at creating a random sequence, one which doesn't have any deterministic patterns, with mathematical functions on a deterministic machine will ultimately fail, because from a given state the functions will produce the same results, which would give the sequence a deterministic pattern.<PRNG> Any method which attempts to create random looking sequence through use of functions is called pseudo random, because it is not truly random for reasons stated above. A pseudo random number generator (PRNG) is the mechanism for creating a pseudo random sequence of numbers. The goal of a PRNG is to make sequences that appear random to any frame of reference outside the generator. The sequence produced should be unpredictable to the outside frames of references. Note that unpredictable and random are not the same. In this case the PRNG 'knows' what will come next, but the method of generating the numbers is hopefully sufficiently obfuscated so that the sequence is unpredictable to anyone else. In order to be a good PRNG it should be able to stand up to statistical analysis of the output and still remain unpredictable. The length of the sequence that the PRNG produces, called its period, should be sufficiently large so that it doesn't start to repeat. At which point the sequence would become quite predictable.

A good PRNG (Uniform distribution) should have the following properties:

5. Stands up to statistical analysis and remains unpredictable.
6. Has a long period.
7. Stands up to attacks that attempt to calculate future values in the sequence.
8. Stands up to attacks to determine the inner state of the generator.
   <PRNG>

Types of PRNGs

2. Linear Congruential generator
   The formula for this type of generator is:
   $$X_{n+1} = (aX_n + c) \bmod m$$

The linear congruential generator uses a few parameters and the previously generated number to create the next number. The parameter 'a' is a multiplier and 'c' is the value used to increment. Used in a mod m system to have the new value wrap back around. If 'c' and 'm' are relatively prime then the period of the generator is 'm'. In that case each number 0—m would be generated exactly once. This can detract from the value of this type of generator, because of the short period, and the fact that as elements are produced there are less possibilities for future elements.

   1. RANDU
      RANDU is an example of a linear congruential generator with a=65539, c=0, and m=2^31. This generator in particular is considered to be an exceptionally terrible generator that was used for years.<RANDU>
      $65539 = 2^{16} + 3$.
      Right below shows the relationship between three consecutive points.
      $x_{k+2} = 6x_{k+1} - 9x_k$

3. Inversive Generator

$$X_{n+1} = (X_n^{-1} + c) \bmod m$$

This generator is considered to be 20 – 50 times slower than faster types of generators. <Diehard>

4. Generalized Feedback Shift Register
   Have two defining terms a length and offset.
   1. Mersenne Twister
      Mersenne Twister(MT) is a type of feedback shift register PRNG called a twisted generalized feedback shift register. MT produces numbers fairly fast, and has a rather large period which is the length of a Mersenne prime (usually $2^{19937} - 1$ is used).<MT>
      However, the algorithm is criticized by the creator of the Diehard tests of random numbers (which this paper will look into further in just a bit), George Marsaglia, for being slower, more complicated, and having a far shorter period than other PRNGs <MT_Marsaglia>

Statistical tests on PRNGs

George Marsaglia devised several tests for random numbers that were published as the 'Diehard Battery of Tests of Randomness'. The Diehard test sweet contains 15 tests for randomness. The test requires a rather large sequence of numbers to be generated; on the order of 2.9 million. Each test produces a p-value, which is between 0 and 1. According to the program when the p-value is 0 or 1 to more than 6 figures than the PRNG has really failed the test. I will describe in some detail a few of the tests that Diehard performs.

Permutation Test: This test looks at five consecutive numbers in the sequence and compares the orderings from smallest to largest number can be ordered in 1 of 120 permutations. This test repeats this process many times accumulating how many times each particular permutation appears. Each of the possible permutations should be uniformly distributed or have the same probability of occuring.<Diehard><Diehard_Tests>

Birthday Spacings: Several numbers at different spots in the sequence are selected from a specific range. The distance between every possible pair is taken, and these distances are then compared to see if any of them are the same. The distances that are the same should fall in a certain distribution. <Diehard><Diehard_Tests>

Craps Test: Plays a craps repeatedly with using the numbers in the sequence to as rolls. It does this by using the bits as a float value between 0 and 1, multiplying by 6 and adding 1. The number of wins, and number of throws to win each game are tabulated. Upon finishing 200,000 games the wins and number of throws for each game should approximate a certain distribution if the sequence appears random. <Diehard><Diehard_Tests>

I also looked at other tests besides the Diehard tests, however I was not able to acquire source code in order to run experiments. A description of one of the other tests:

The GCD Test: The GCD test involves using the Euclidean algorithm for determining the GCD. Two numbers are selected from the sequence. The Euclidean algorithm is applied, and the GCD is recorded and also the number of iterations of the algorithm to determine the GCD. After repeating this procedure many times the distribution of the GCD and the number of iterations needed should have follow specific distributions. <GCD_Test>

Analyze PRNGs

Since these tests in a sense give a probability of failing, I decided to test each of the PRNGs multiple times with different seed values. The three PRNGs I tested are: the rand function that comes with the c/c++ standard libraries, an inversive generator, and the Mersenne Twister. The Mersenne Twister in this case uses the c/c++ libraries rand function (a linear congruential generator<LCG>) to initialize its buffer.

I had to do a few steps to get the numbers from the generator in the proper format. I output the sequence to a text file in a hex base. I had to ensure that 40 characters appeared per line. Each number was 32 bits, which means 4 hex characters. I also had to ensure that each number was printed as four characters for example printing 0001 instead of just 1. I then sent the text file and ran it through a program that converted it to the proper binary format required for the Diehard tests.

Assumptions:

The Battery of Tests of Randomness works correctly as intended. This assumption is required to be true since pretty much all of the analysis relies on this piece of software.

I properly converted the output of the PRNG's to the proper format. This assumption is also very crucial to the integrity of the analysis.

Results:

The built in PRNG for c/c++ produces low quality pseudo random numbers. This PRNG failed essentially every test in Diehard. This surprised me when I first tested this, so I immediately ran it again, and got the same results. I tried with a different sequence of numbers, and again got the same results; failure on every test. As I did more research I found that the generator that is supplied with the standard libraries by my compiler vendor is a linear congruential generator. Most of which do not do very well on the Diehard tests. I also tried the running the Mersenne Twister, which passed a large number of the tests. This gave weight to the fact that my assumption that the method I used for converted the PRNG's output to the correct format for the program to use was successful.

The Mersenne Twister I had a small problem with. The majority of the time Diehard would crash while running the first test, the Birthday Spacings test. This forced me to omit this test from the results as I was only getting half of that test for MT.

Both the Mersenne Twister and Inversive generator passed a majority of the tests. The results from running the tests multiple times, but with different seed values was fairly consistent as far as which tests the PRNGs had trouble passing. The Mersenne Twister however passed more of the tests than did the Inversive generator.

As far as choosing which PRNG to use it is good to know the weaknesses of each generator in order to compare with the requirements of the program. Then it becomes much easier to choose the appropriate generator. As a recap of some of the strengths and weaknesses of the PRNGs that I looked at:

Linear Congruential Generator supplied with C/C++ standard libraries

1. ☺It is supplied with the standard libraries, which makes it very accessible.
2. ☺It doesn't require a very large seed, so it is quick to initialize.
3. ☹It doesn't produce very high quality pseudo random numbers.

4. ☹Shorter period than most generators. It can only hit each number in the range that it produces at most once for a given seed.

Inversive Generator

4. ☺Small seed quick to start up.
5. ☹Harder to obtain the code.
6. ☹Takes much longer than most generators due to finding the inverse of numbers in a given mod system.

Mersenne Twister

6. ☺Produces much higher quality random numbers than the built in generator for C/C++
7. ☺Very very long period.
8. ☺Fairly fast at producing numbers.
9. ☹Requires a large seed, takes longer to initialize.
10. ☹More complex to understand, although this doesn't really matter to an end user.

If the quality of the pseudo random number doesn't matter too much then it is probably easier to just use the built in C/C++ rand function. However, if higher quality pseudo random numbers are required, then I would recommend using a better algorithm. Of the ones that I looked at I would recommend the Mersenne Twister for this situation. I would not recommend the Inversive Generator for most situations, because it is slower and from my analysis produces slightly less quality pseudo random numbers than MT.

Issues:

The Mersenne Twister always seemed to crash the Diehard program on the Birthday Spacings test. I was required to omit that test for the MT. However, the parts I could see from the log showed that it was passing this particular test.

References:

Diehard
Diehard Program. George Marsaglia

Diehard_Tests
http://en.wikipedia.org/wiki/Diehard_tests

GCD_Test
http://www.jstatsoft.org/v07/i03/paper

MT_Marsaglia
http://groups.google.com/group/sci.crypt/browse_thread/thread/305c507efbe85be4

LCG
http://en.wikipedia.org/wiki/Linear_congruential_generator

Randomness
http://en.wikipedia.org/wiki/Randomness

PRNG
http://en.wikipedia.org/wiki/Pseudo-random_number_generator

RANDU
http://en.wikipedia.org/wiki/RANDU

MT
http://en.wikipedia.org/wiki/Mersenne_twister