Pseudo Random Number Generator

CS4050 – Assignment 4

Ryan Flageolle

**Description**

There are many different algorithms used to generate pseudo-random numbers, I looked into the Blum Blum Shub algorithm and compared it to the Mersenne twister algorithm which is used as the built-in random library for python. To test the “randomness” of each I compared mean, standard deviation and variance of each, and to test the sequences I tested each generator using the frequency and runs test as described in the NiST test suite.

**Blum Blum Shub**

The Blum Blum Shub is a one-way function derived from Michael O. Rabin.

xn+1 = x2n % M

In my implementation, I set a seed (x0) to the value of microseconds from the current time, then for some number n I calculate the next value of x and mod the answer by 2 to return a bit this occurs a dynamically set number of times producing a random string of bits. From this binary string I take a substring and return its integer representation. This number generator does not produce cryptographically safe random numbers however it does seem to produce similarly random numbers as python’s Mersenne twister algorithm.

A few things I noticed about my implementation, a shorter number of internal steps taken seems to add to unpredictability of the number generated, without affecting the overall runtime of the algorithm. Another step that I could take is by creating a method which returns a number between 0 and a given integer, by simply modding the returned random integer by the given integer.

**Tests Used**

I decided to test my sequences using the frequency test and runs test as they are explained in the NiST test suite. I chose these tests because they determine the “randomness” of a prng (pseudo-random number generator) by assessing the sequence of bits generated. The frequency test determines whether the sequence given has a frequency of 1s to 0s which is close to the expected rate of .5. The runs test determines whether a given sequence of bits is random by counting the number of runs, changes between 1s and 0s for example 0011011001 has 6 runs [00, 11, 0, 11, 00, 1], and comparing it to the number of expected runs.

**Results**

Mean value of my algorithm after 500000 runs is: 511.047

Mean value of built in python algorithm after 500000 runs is: 511.563

Standard Deviation of my algorithm after 500000 runs is: 295.552

Standard Deviation of built in python after 500000 runs is: 295.802

My algorithm passes the frequency test: 99 %

Built in python algorithm passes the frequency test: 99 %

My algorithm passes the runs test: 97 %

Built in python algorithm passes the runs test: 95 %

My algorithm takes 846.64 seconds

Built in python algorithm takes 1.36 seconds

**Conclusions**

These results lead me to believe that my interpretation of the Blum Blum Shub algorithm is at least as good as python’s implementation of the Mersenne twister algorithm. This is of course only the case for the randomness of the number generated. The algorithm I produced isn’t even in the same league when it comes to performance, I believe this is because produce 2 new prime number each time which are congruent to 3 % 4, as well as finding a small gcd is a very time-consuming process. While neither of these algorithms are cryptographically safe, they both seem to produce random-enough numbers.