Michael Cyran

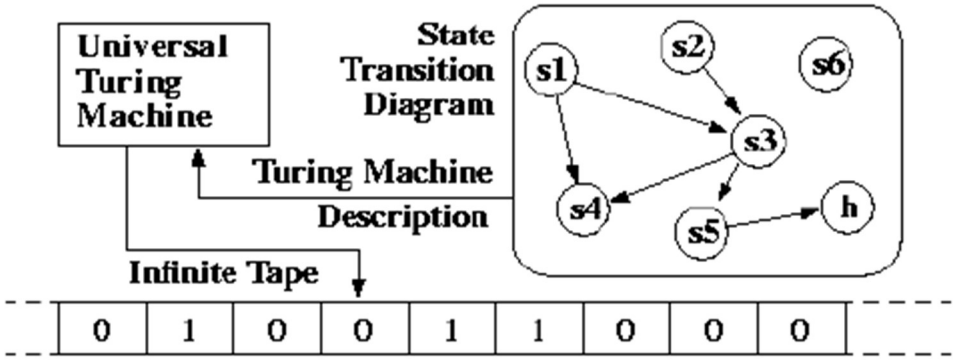
Randomness Testing of BitCycler

Abstract

Random number generation is an essential requirement for a broad range of applications, from simulations and modeling to cryptography and gaming. The increasing complexity of these applications has led to the development of novel random number generators that offer improved statistical properties, efficiency, and security. Many of the random number generators used today are based on pseudorandom algorithms that generate sequences of numbers that appear to be random but are deterministic and predictable. In this research paper, I introduce BitCycler, a new type of random number generator that attempts to mimic the behavior of particles in spin systems. BitCycler takes advantage of the principles of physics to generate asynchronous random numbers that are fast, secure, and non-predictable. Using 8 different randomness testing methods, including random walks, period, uniformity, chi-square, filling site, parking lot, hidden correlations, and short-term correlations, the performance is evaluated against other popular random number generators.

Background Information

The average computer conducts somewhere in the order of 1 giga-operations per second in order to maintain the operating system of the computer. We have been programming in the Java Virtual Machine inside a very closed environment that allows us to make reliable, deterministic simulations, and it is all made possible by Alan Turing, the father of computation. A Turing machine is a mathematical model of computation that consists of an infinite tape divided into cells that can either be 0 or 1, a read/write head that can read or write on the tape, and a set of rules that govern how the machine operates. The tape serves as the machine's memory, while the head moves back and forth along the tape, reading, writing, and changing its internal state according to the rules.



The set of rules that govern the behavior of the Turing machine is called the transition function. It specifies what the machine should do based on its current state and the symbol that it is currently reading. The transition function can be thought of as a set of instructions that tell the machine to perform a certain action, such as moving the head one cell to the left or writing a new symbol on the tape.

The behavior of the Turing machine is completely deterministic, meaning that if the initial state and transition function are known, the output of the machine can be predicted with certainty. However, the machine can perform any algorithmic computation, making it a powerful tool for modeling and analyzing complex systems. When it comes to random number generation, Turing machines are not capable of producing truly random numbers. This is because the set of rules that govern the behavior of the machine is deterministic and based on the seed value for the generator.

Therefore, to generate truly random numbers in practice, we need to rely on physical phenomena that are inherently random, such as thermal noise or radioactive decay. However, these physical sources of randomness can be difficult to access or impractical to use in many applications. As a result, pseudorandom number generators are often used as a substitute. These generators use deterministic algorithms that produce a sequence of numbers that approximate randomness but can be repeated and verified for consistency. While not truly random, they are useful for many applications where a source of true randomness is not available or practical.

While being useful for performing many applications, the Turing Machine model is more of a theoretical construct than what is implemented. On most modern computers, there are at least 2 cores, meaning that we have a Turing multi-machine model, with multiple different heads

traveling along the same tape. We've built rules inside the computers to prevent the heads from overlapping with each other because that ruins the certainty and determinism of computation along any single head. This is the concurrency problem in Computer Science. In computing, threads are separate execution contexts that can run concurrently within a program. Think of each thread as a world line moving along its individual reference frame according to its internal state. An example would be multiple tabs open on google chrome. A race condition occurs when multiple threads access shared resources or data in an uncoordinated way, leading to unpredictable behavior. This uncertain behavior is where the randomness in BitCycler is taken.

Algorithm Descriptions

In order to organize the testing process for each of the random number generators, I implemented an abstract Random_class which has two methods, getSeed() and next().

```
abstract public class Random_class {
    int seed;

    public Random_class(int a_seed) {
        seed = a_seed;
    }

    public int getSeed() {
        return seed;
    }

    abstract protected int next();
}
```

The next method is implemented inside of each class that inherits the Random_class, and this makes testing a lot simpler because it's only one class that the TestingApp has to deal with instead of three.

Linear Congruential

The first random number generator is the Linear Congruential (LC) method. It works by repeatedly applying a mathematical operation to the previous generated number in the sequence. The basic equation for generating the next number in the sequence is:

$$x_n = (ax_{n-1} + c) \bmod m$$

The constant "a" is referred to as the multiplier and determines the period or cycle length of the generator. The period is the number of unique values the generator can produce before it repeats itself. To achieve a full period, "a" should have certain properties. It should be greater than 0, relatively prime to m (meaning they share no common factors other than 1), and the difference between m and the largest prime factor of a should also be small. The constant "c" is called the increment and affects the starting point of the sequence. It should also be relatively prime to m. The constant "m" is referred to as the modulus and determines the range of the generated numbers. The generator will produce values between 0 and m-1.

To generate a random number using LC, you start with an initial seed value (X_0) and then repeatedly apply the equation to generate subsequent numbers in the sequence.

Here is the coding for the next() function in the LC.

```
@Override
public int next() {
    x = (a*x+c)%m;
    return x;
}
```

It uses the existing x value to perform a linear mathematical operation on it, and then keep it within the bounds of the modulus.

Generalized Feedback Shift Register

The next method is the Generalized Feedback Shift Register (GFSR). It involves the use of a shift register, which is a sequence of bits that can shift positions based on a feedback function. The feedback function combines the contents of specific register positions to determine the value of the next number in the sequence. By repeatedly shifting the register and applying the feedback function, a pseudo-random sequence of numbers is generated.

$$x_n = x_{n-p} \oplus x_{n-q}$$

Like the LC method, the GFSR has a period that can be maximized based on the parameters p and q. The GFSR method keeps a list of its previous numbers and generates a new number by combining two previous using XOR.

```
@Override
public int next() {
    n++;
    x[n%(p+1)] = x[((p+1)+n%(p+1)-p)%(p+1)]^x[((p+1)+n%(p+1)-q)%(p+1)];
    return x[n%(p+1)];
}
```

My algorithm was implemented with an array x[ ] which stores the p+1 most recent previous numbers. Each index is taken modulo p+1 to make sure that the generator index stays inside the array while n is allowed to increase with the number of steps. This normalizes the value of n within the array, and then on the left p is subtracted, on the right q is subtracted, and once again normalized to within the array. This efficiently stores the variables within a set range instead of computing it dynamically using arraylists. The reason I did this was because I've implemented it with an arraylist in the past, and I wanted the challenge of limiting the memory size.

BitCycler

Before I describe how it works, I wanted to give the motivation behind designing a random number generator as such. If you think about observing atomic particles, the act of observation changes the particle state, and by the time the photon leaves the particle and goes into your eye (or camera), the particle has already moved locations. This is uncertainty fundamental to quantum mechanics as we only have the past information that we see and not the present information. This separates the observed from the observer, two independent frames of reference where a measurement is taken relative to each other.

BitCycler recreates the observer and observed into two classes, the outer AsyncRand class, and the inner Spinner class. The outer class extends Rand_class, and the inner class extends Thread. The spinner class runs asynchronously from the main class so that their operations do not depend on each other. Inside the spinner, there are two static variables size and boundary, and one dynamic variable line.

```
public AsyncRand(int a_seed, int a_size) {
    super(a_seed);
    c = new Spinner(1, a_size);
    c.start();
}
```

```
public Spinner(int start, int a_size) {
    line = 1 << (start-1);
    size = a_size;
    boundary = 0;
    for(int i = 0; i < size; i++) {
        boundary += (1 << i);
    }
}
```

The BitCycler algorithm looks at the boundary variable and the line variable as a set of 1's and 0's. The variables don't represent the data as integers, instead they represent the data as a set of buckets which can have a value of 0 or 1. The boundary variable sets the range that the line variable can roam freely inside and the line variable changes with each step. The best way to imagine it is with an example.

If you want to create a distribution of numbers from 1-4, you could initialize a Spinner with start=1 and size=4. In the first line of the constructor, this would set the line variable to equal

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The for loop would set the boundary variable to equal

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

These two variables are set up in such a way that the '1' in the line variable will oscillate inside the first four positions of the bit array. It does so using various bit operations including shift registers, AND, and OR. The run( ) method is inherited from the Thread class and it is what is run inside the thread.

```
public void run() {
    while(running) {
        lock.writeLock().lock();
        try {
            line = ((line << 1) & boundary) | ((line >> (size-1)) & boundary);
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

At every iteration of the while loop, the line variable is adjusted. An atomic lock is used to make sure that the value assigned to line is consistent across both threads. On the left side of the OR (vertical bar |), we take line and shift it one to the left then AND it with boundary.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

On the right side of the equation, line is shifted to the right by 3, so that is just 0.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

When these two are taken together, the state of at t=0 goes from

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

To t=1

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

This is the encoding for a rotation of the single 1 by one position. If this algorithm is followed for multiple iterations greater than the size, the location as a function of time is as follows (with some columns removed for simplicity)

t=0	…	0	0	0	0	0	0	0	1
t=1	…	0	0	0	0	0	0	1	0
t=2	…	0	0	0	0	0	1	0	0
t=3	…	0	0	0	0	1	0	0	0
t=4	…	0	0	0	0	0	0	0	1
t=5	…	0	0	0	0	0	0	1	0
t=6	…	0	0	0	0	0	1	0	0
t=7	…	0	0	0	0	1	0	0	0

The position of the 1 in the line bit array is now a function of time. It no longer has a definite position, because it now has momentum. The spinner is allowed to oscillate this freely with time, and the main thread makes measurements of the spinner at non-deterministic times. When the main thread tries to read the position of the line variable, it needs to acquire a lock. This temporarily pauses the spinner because the lock is busy being used by the main thread.

```
public int getPoint() {
    lock.readLock().lock();
    try {
        for(int i = 0; i < size; i++) {
            if(1 == (line >> i))
                return i+1;
        }
        return -1;
    } finally {
        lock.readLock().unlock();
    }
}
```

By creating a condition where two threads are attempting to acquire a lock at the same time, it creates a race between the threads where the output will be different whether the spinner thread acquires it first or the main thread acquires it first. Since the behavior of race conditions are undefined, the output is uncertain, and the main class measures the uncertainty of the spinner class. This could possibly mean that this algorithm is non-deterministic, as has the potential to deliver more than just pseudorandom numbers.

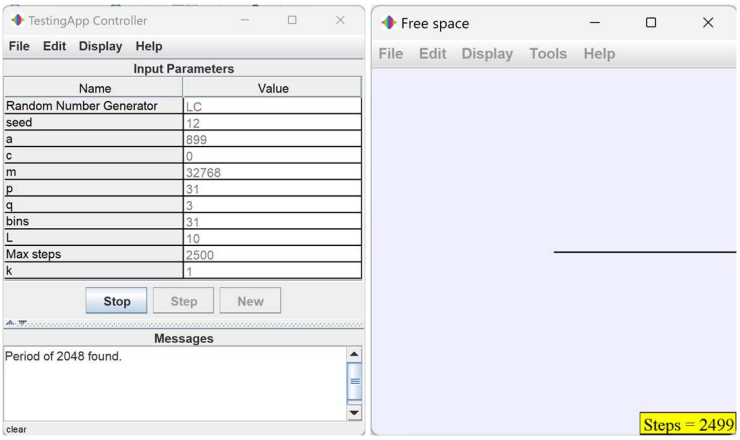
Testing Methods and Results

In determining the efficacy of the BitCycler, 8 different randomness testing methods were used, including random walks, period, uniformity, chi-square, filling site, parking lot, hidden correlations, and short-term correlations. For each method, I will describe how it is calculated and show results from each of the three random number generators. In designing the application to run all of these tests, I wanted to incorporate each graph to be calculated live.

Period and Random Walk

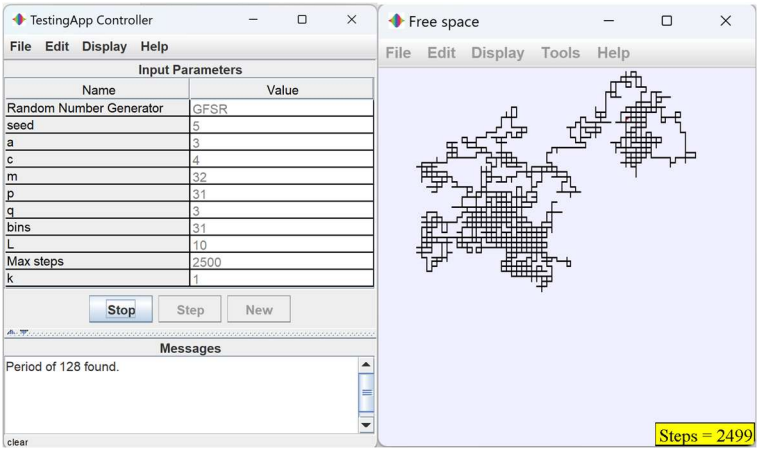
The first pattern to look for in a sequence of random numbers is the period. If a sequence repeats itself, it is not truly random. For most uses, we don't need that many random numbers so as long as the period is greater than the need, a pseudorandom generator is fine.

The period of a sequence of random numbers can be visualized as repetitions in the pattern of a random walk. Here is the random walk of the LC generator with a=899,c=0,m=32768, and x0=12.



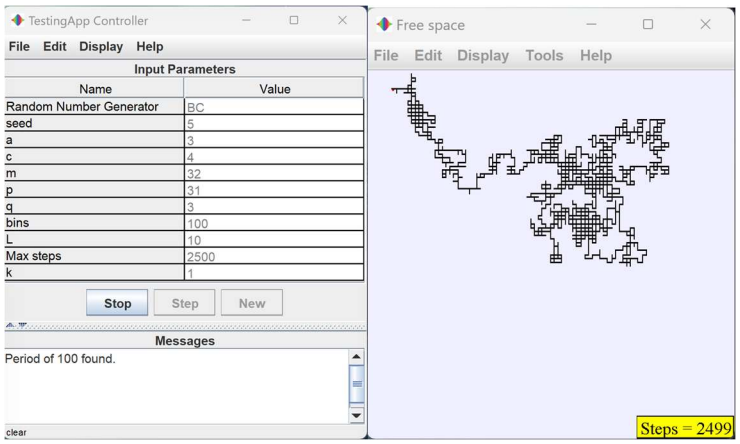
As we can see, every number in the sequence is mod4=1, so from the start you can see a pattern in the sequence. To find the period, each number in the sequence was added to a set. A set is a data structure that only allows one copy of each element, so this can be useful to find all the unique numbers that can be found in a sequence. The LC had a period of 2048, while its maximum range is 32768, so that is not a very good distribution of random numbers.

Here is a random walk of the GFSR method with p=31 and q=3. There is no obvious pattern from the random walk, but using the set data structure we find that there is a period of 128, and the range is 128 too, so that means that it is pretty good at filling out the range.

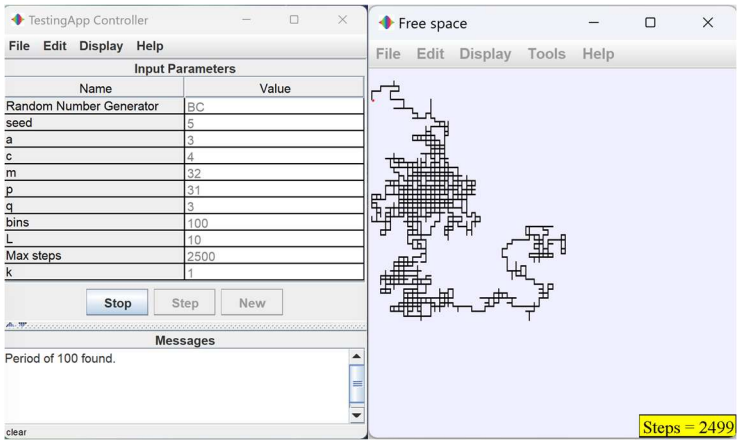


Here is a random walk for the BitCycler with a size of 100. Using the set, we find that it has a period of 100, meaning that every number in the range was found. This method of finding the period is not accurate for the BitCycler because once all the numbers are found, it doesn't repeat itself, that's just the time where all the numbers are found. With all the other numbers, the next

number depends on the previous number (or numbers), but the BitCycler doesn't use its previous numbers, it generates new numbers based on the current state of an oscillating number.



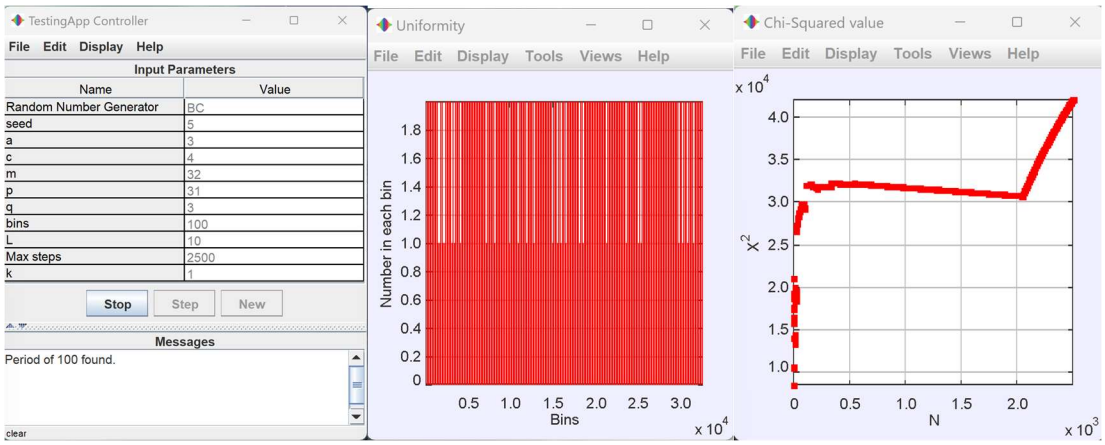
This can be shown by running the simulation again, with all the same parameters, it creates a different pattern yet still finds a period of 100, so there's something different going on. The sequence of the BitCycler is not predetermined, it's found during runtime.



Uniformity and Chi-Square

At the same time the random walks are found, a histogram is graphing the uniformity of the distribution of random numbers and the chi-square metric is found. With a random number generator, you would expect equal probability for each of the bins. Chi-Square is a metric for testing how close each bin is to the expectation of that bin. With a truly random number, statistics would tell us that chi-square should be of order 1, so the value should be near the range of values.

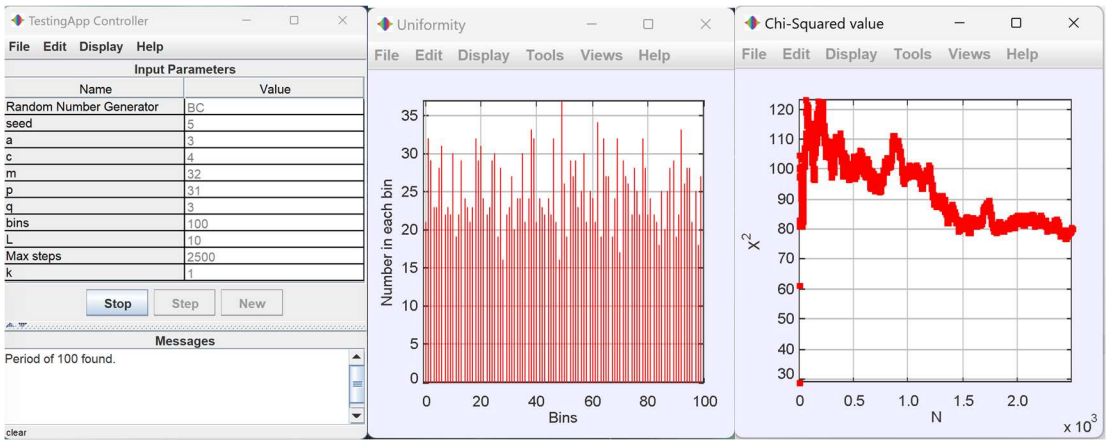
Here is the uniformity and chi-square of the same LC test as before, a=899, c=0, m=32768, x0=12. When the simulation was running, a single tick was found for 2048 numbers within the range of 32768, and it looked like a solid rectangle until it started to repeat and then every number was a repeat. In the chi-square graph, we can see the moment that it started to repeat. The sequence was looking good, approaching the range from above, and then as soon as it repeated once, it grew very fast, which is not a good sign.



Here is the uniformity and chi-square of the same GFSR test as before where p=31 and q=3. There is more variation to this method, and there isn't one sequence to the numbers, the sequence is more complex. On the right, you can see the chi-square metric approaching 128, the range of the random number generator.

Here is the uniformity and chi-square of the same BitCycler test as before with the size equal to 100. The uniformity looks similar to the GFSR method, but there is less of a spike in the chi-

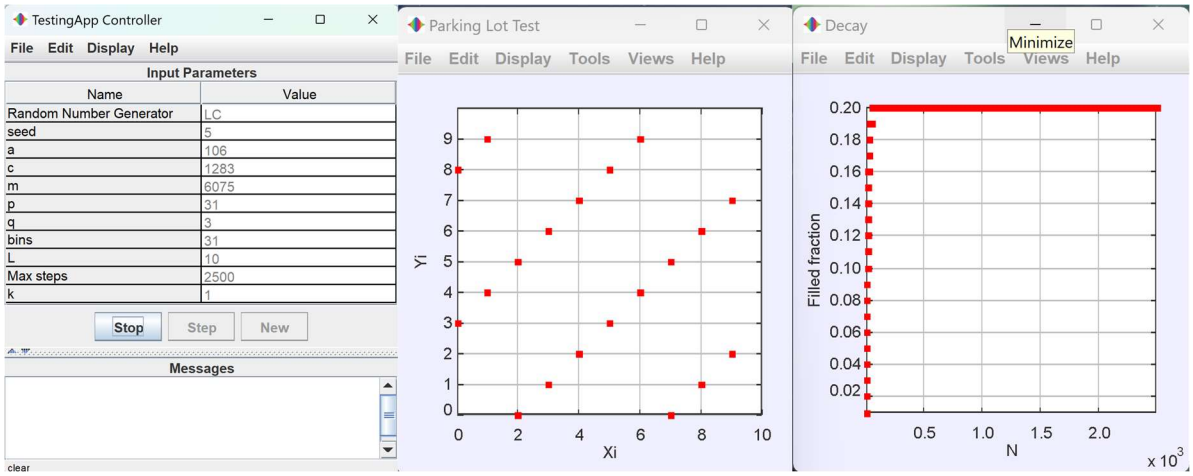
square metric. The chi square value actually goes below the range so that indicates a strong randomness in the sequence.



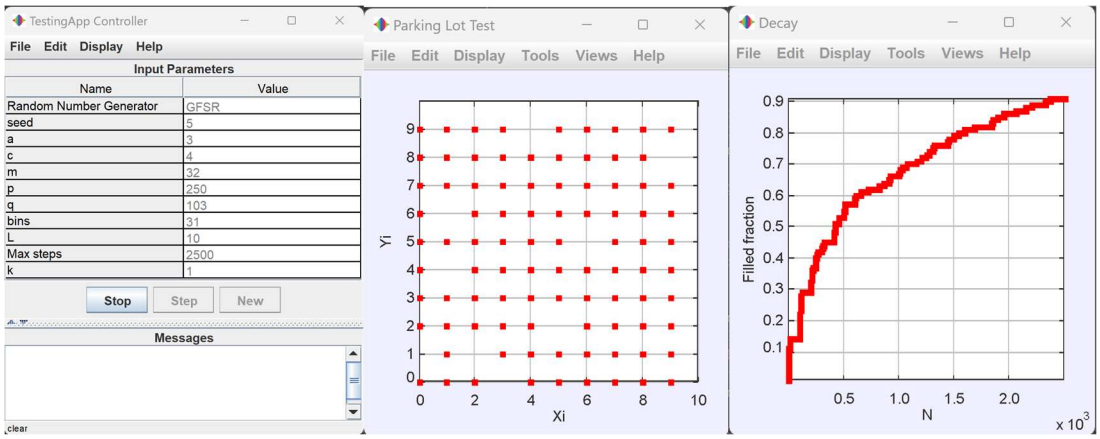
Parking Lot and Decay

The parking lot test is taking each pair of random numbers generated in a sequence and plot them as a car taking up a space in a parking lot. With a random distribution, you would expect every spot to eventually be filled up, and the rate at which they fill up to be e^{-t}. In order to demonstrate this, I created a plot frame that puts points on a plot as cars and plots the fraction of filled sites.

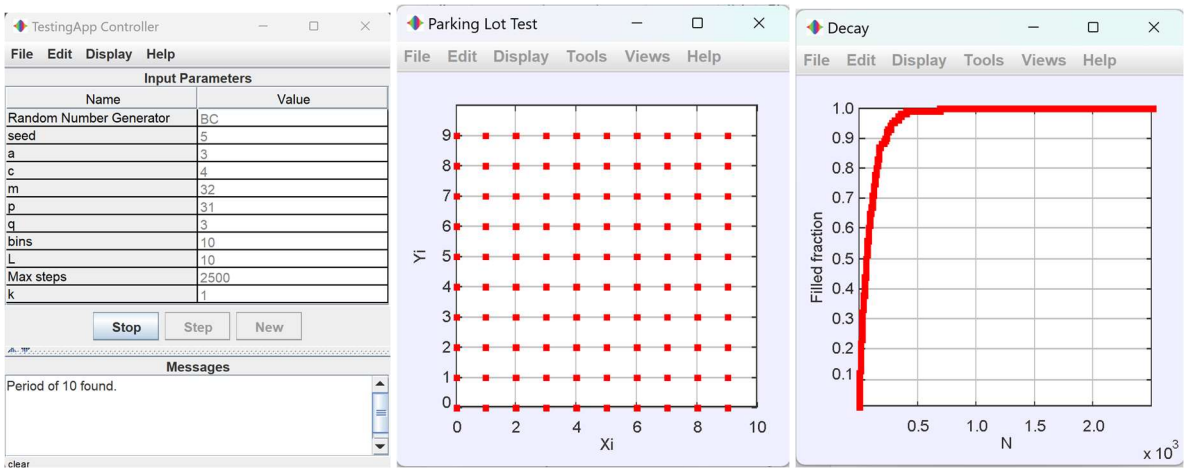
Here are the results for a new LC test with parameters a=106, c=1283, and m=6075. Only 20 spots are taken in the parking lot out of 100, so there is a pattern. We can see this in the decay graph because it maxes out at 20/100, so this does not follow the exponential decay that it should.



Here is the parking lot test for the GFSR method with parameters p=250, q=103. It was able to fill up a lot more spots in the parking lot, but it couldn't fill them all up. The decay function is not as steep as would be expected with e^{-t}, it is still exponential and approaching 1, but slower and with discrete chunks.



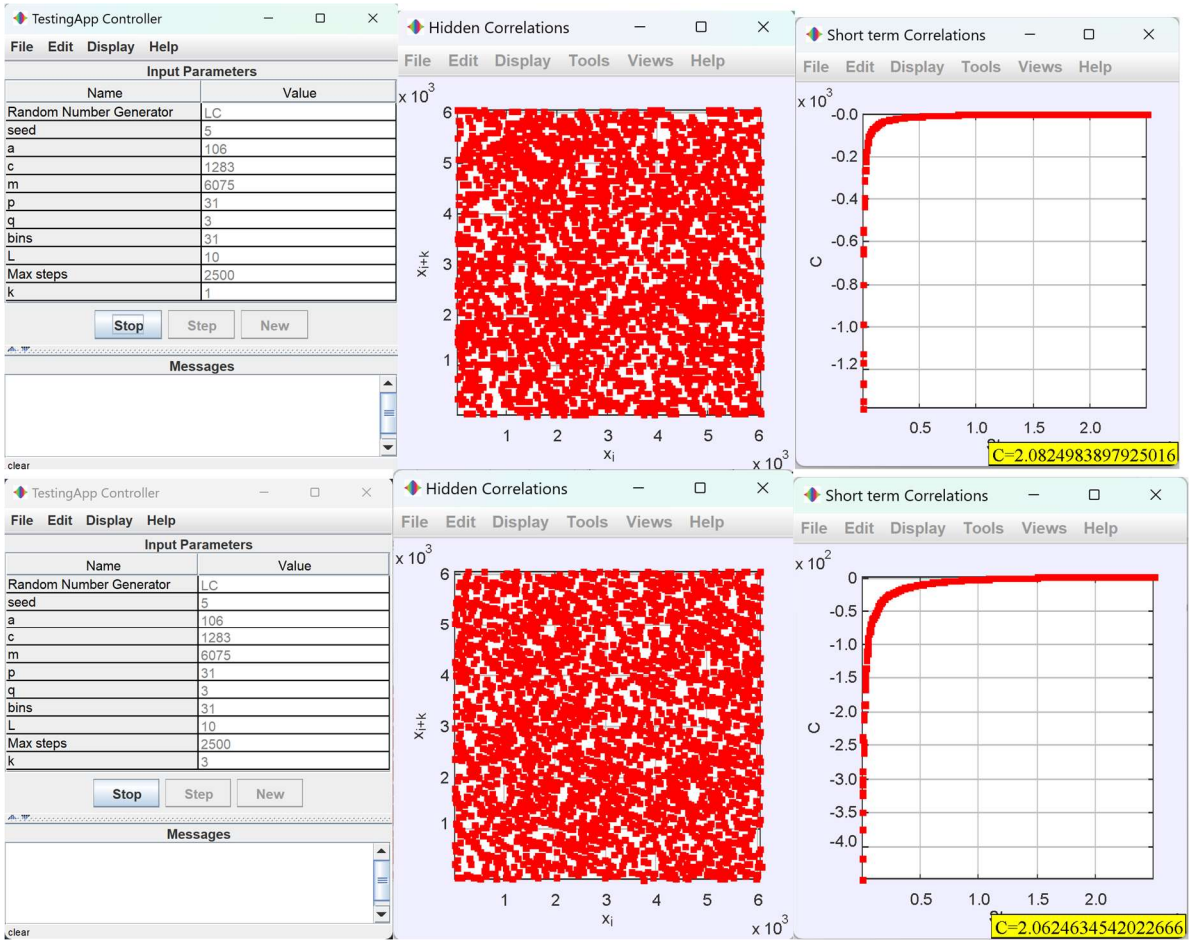
Here is the parking lot test for the BitCycler with size=10. This algorithm was significantly quicker than the others to fill up all the parking spaces, and the decay looks closer to exponential than the other two algorithms.



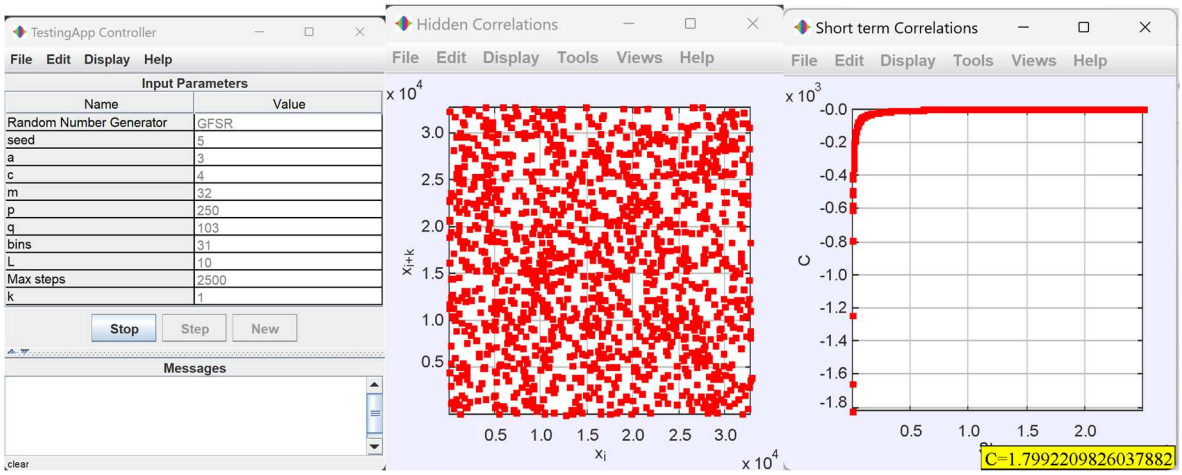
Hidden and Short-term correlations

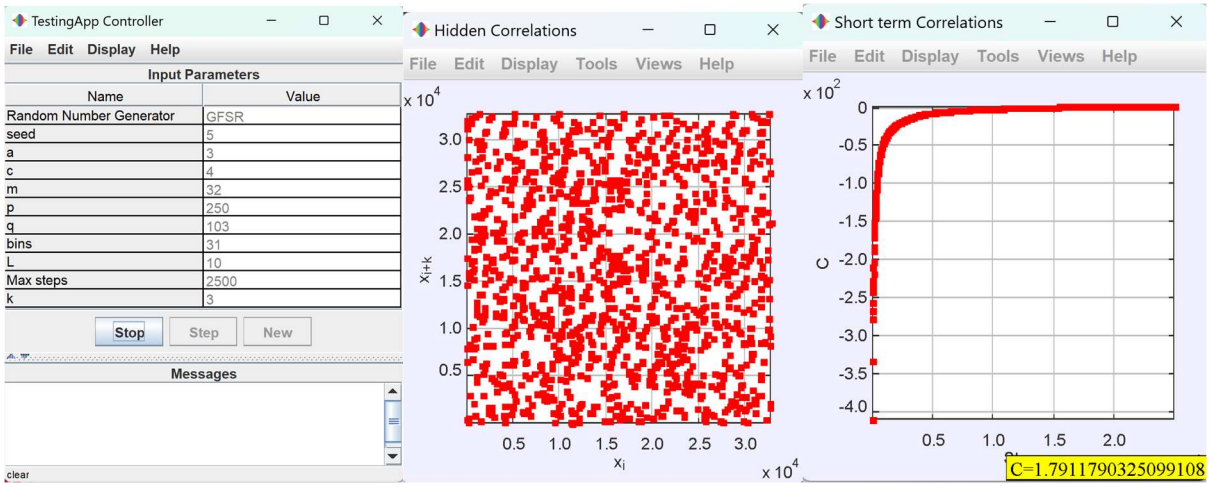
These two tests are conducted to find similarities between temporally close elements in the sequence of random numbers. It looks for pattern between the nth and n+kth element, and it can be useful to find patterns inside the randomness. The C metric is calculated to see the efficacy of these sequences. A C value closest to 0 indicates the most randomness.

Here is the hidden correlation and short term correlation test for LC with k=1 and k=3. There aren't many differences between the two results, but it's possible that with more k testing, some pattern could be found inside. The values of the C variable on the right are above 0, so this is not a good sequence of randomness.

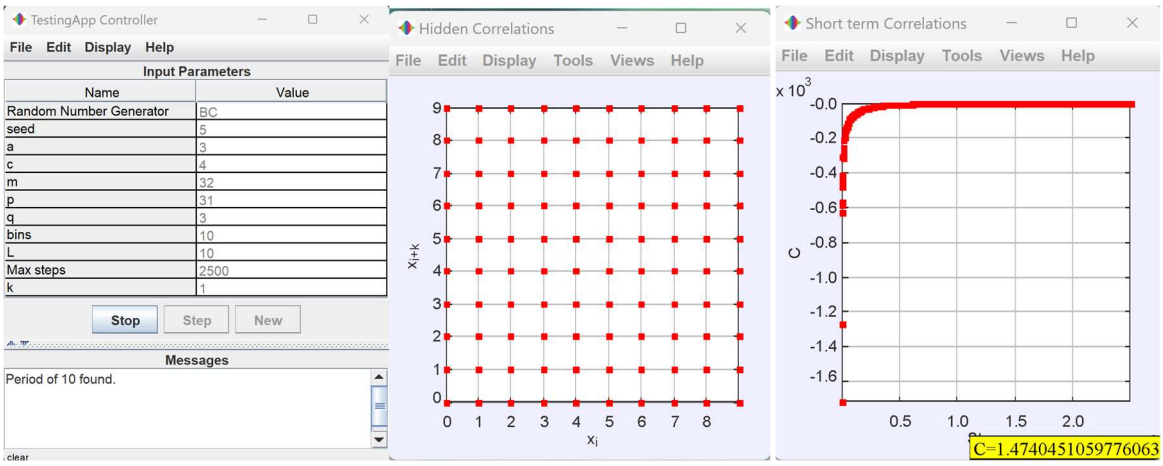


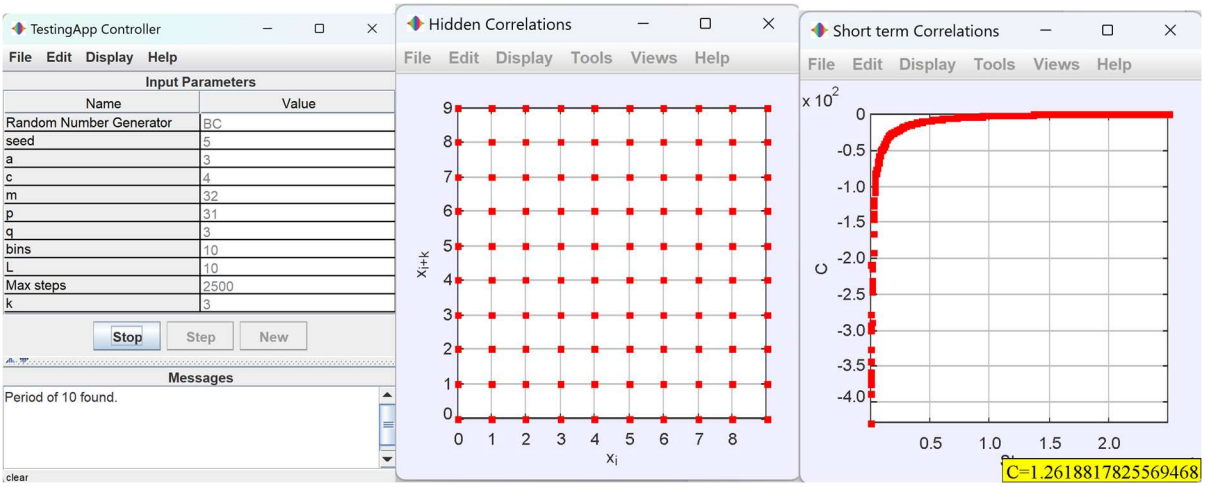
Here are the results for the GFSR method with k=1 and k=3. Once again, it is hard to tell a visual difference between the k=1 and k=3, but the C value is lower than the previous sequence so that means this is a better random number generator.





Here is the results of the BitCycler with k=1 and k=3. The hidden correlations are within a much smaller range because the size is set to 10, but the C value is lower than the previous 2 methods.





Discussion

After testing multiple algorithms using various tests of randomness, it's given me a greater appreciation for the fine-tuning that most random number generators use. If the LC or GFSR method are implemented in an application, very particular sets of parameters need to be found in order to find a standard of randomness that is necessary for their application. However, for certain applications, such as cryptography, these algorithms would not work because they are deterministic and easily predicted if the seed and parameters are known. One well known example is CloudFlare, an internet encryption company which uses a wall of lava lamps to generate random numbers.

BitCycler could be an alternative for applications which need certifiably random numbers. Since it works off the uncertainty between thread executions, every time the algorithm is run it will create a unique result. Although the uses for this application are small, I think it is important as a step in computer science as a non-deterministic Turing Machine.