Implementations and Statistical Testing of Random Number Generators

Kenny Ning

December 3, 2013

Abstract

This paper is an exposition and analysis of various random number generators (RNGs), or programs that output numbers that were seemingly randomly sampled from some known distribution. While there are several implementations of RNGs, this paper will focus on three: RANDU, Mersenne Twister (MT), and Marsaglia multiply-with-carry (MWC). A framework for testing the supposed "randomness" of these RNGs is developed and then applied to each of these implementations.

1 Introduction

The notion of "randomness" has profound implications in many applied fields including statistics, mathematics, and computer science. Oftentimes, we need to model randomness for purposes such as product testing, algorithm design, or simulation, and it would be useful to have some sort of black box that can provide a reliable source of random numbers. Fortunately, we can call on programs known as *random number generators* (RNGs) that output numbers that resemble random draws from some known distribution. There exist RNG implementations that simply sample some physical random phenomena (e.g. the 10th most significant digit on the computer clock, noise from an audio signal, radioactive decay) and return that as a result. However, the focus of this paper will be on *pseudorandom number generators*, which are RNGs that are based on deterministic logic and "appear" to be from some truly random source. While it may seem contradictory that a deterministic program is used to generate numbers that should be theoretically non-deterministic, we will see that most random generators have fairly straightforward implementations and can provide fairly robust results.

In the most general case, RNGs oftentimes work by doing some modular arithmetic operations on an initially provided *seed* value. Each iteration then performs some operation on the previous value in the sequence to get our current number. If we choose our constants and modulo well, we get a fairly good source of random numbers.

While anyone can write a quick and dirty RNG, the key difficulty in random number generation is determining how good the RNG actually was. There is no official standard for what random numbers should look like, and upon first glance of a sequence of numbers, it is far from intuitive to gauge how "random" that sequence is. Many researchers have thus developed several test suites and frameworks that are designed to solve this very problem. Examples of tests this paper will discuss include comparing the sequence to its supposed theoretical distribution and looking at the ordering of the sequence.

As a caveat, I will only be looking at uniform RNG's, which is the foundation for all other types of generators (e.g. normal, gamma). In addition, finding a very strong uniform RNG is arguably all one really needs, since a random draw from any distribution is theoretically obtainable with just a uniform RNG using the inverse-CDF method.

The rest of the paper will be organized as follows. Section 2 will discuss specific RNG implementations. Section 3 will discuss the randomness testing framework, as well as outline, in detail, the specific statistical tests being used. Section 4 will briefly define the experimental

setup with which we will test our RNGs. Section 5 will be the evaluation of the RNGs using our testing framework. Section 6 will provide applications of RNGs, including a specific example. Then, we will conclude and also attach all code used in this paper in the Appendix section.

2 Implementations

2.1 Linear Congruential Generators

A *linear congruential generator* is a very common model for random generators, where each number in the sequence is determined by taking the previous number and applying some linear function to it. For reference, most of the following definitions were taken from [2]. A simple formulation for a linear congruential generator is

$$x_i \equiv ax_{i-1} \bmod m, \quad \text{for } 0 \leq x_i < m$$

where a is a multiplicative constant and m is the modulus of our generator. This value for m also determines the maximum *period* of our RNG, or the maximum number of cycles we can have before the sequence starts repeating. This implies that our choices of m, as well as a, (generally very large values) determine the quality of our random number generator. Modulus values most commonly take on the form $2^p - 1$, also known as a *Mersenne prime*, where p is just some constant. a is then typically chosen based on m such that we maximize the period of the RNG.

This basic model has formed the foundation for some of the more popular implementations of RNGs that will be discussed in this section.

2.2 RANDU

RANDU is a notoriously bad random generator that dates back to the 1960s. Essentially, RANDU is just a linear congruential generator with the following chosen constants:

$$x_i \equiv 65539x_{i-1} \bmod 2^{31}$$

For reference, this was implemented in the C programming language as follows:

```
#include<stdio.h>
#include<math.h>

int main() {
    //constants used in for RANDU
    int seed = 1;
    int a = 65539;
    int mod = pow(2,31);

    //get our starting number
    int curr = (a * seed) % mod;

    //following steps print out sequence in series of triples
    int i;
    int counter = 0;
    int tuple_length = 3;
    int seq_length = 10000;

    for(i=0; i<seq_length; i++) {
        printf("%d\t", curr);
        curr = (a * curr) % mod;
        counter++;
```

```
        if(counter == tuple_length){
            counter = 0;
            printf("\n");
        }

    }
    printf("\n");

    return 0;
}
```

The reason for outputting numbers in tuples of 3 is for one of our graphical analysis tests in the evaluation section. We will be plotting these triples in a three-dimensional scatterplot, hopefully illuminating some of the flaws of this generator.

2.3 Mersenne Twister (MT)

The Mersenne Twister (MT) [4] is a popular modern implementation, used as the default uniform generator in current versions of R, the statistical programming language. It is referenced as one of the more state-of-the-art and most researched uniform RNGs [6]. It is known to be incredibly comprehensive in terms of covering the numerical space as well as incredibly fast, computationally. The "Mersenne" part of its name comes from the fact that the period of this RNG is an incredibly large Mersenne prime number, $2^{19937} - 1$. The acutal implementation of the MT algorithm is quite complex and involves several steps and operations such as bitshifts and XORs.

We will be using the R implementation of MT without worrying too much about the specific implementation details. However, for interested readers, the full details are covered in Matusmoto and Nishimura's paper, included in the References section.

2.4 Marsaglia Multiply-With-Carry (MWC)

The Marsaglia multiply-with-carry is another widely used RNG, and a fairly natural extension of the linear congruential model. It is another example of a quality RNG used in industry, such as in the implementation of dynamic analysis tool, DieHard [1]. In that specific case, it was used as a way to identify random blocks of memory to store objects as a way to probabilistically avoid runtime memory errors.

Marsaglia's MWC generator is an extension of the basic linear congruential model that incorporates a new "carry" parameter referred to as c_i:

$$x_i \equiv (ax_{i-1} + c_i) \bmod m$$

An implementation of this already exists in R, and the documentation claims that it has a period of more than 2^{60} and has passed all statistical tests. We will be using this implementation in our experiments.

3 Testing Framework

3.1 Runs Test

Suppose you are a high school math teacher, and you decide to give your students a simple homework assignment: go home and flip a coin 100 times and record the results. How can you tell which students actually did all the coin flips and which students cheated and made up results?

A mathematician should be able to tell the truly random results from the made-up random results fairly quickly using some simple inutition about probability. When humans are asked to make up the outcomes of 100 coin flips, their sequences oftentimes alternate between heads

and tails too much. A human would rarely, for example, write down a series of 7 heads in a row. However, if we are indeed looking at a random coin flip sequence of 100, we would actually expect longer, consecutive "runs" of heads or tails to show up in our sequence at some point. In other words, we can gain some understanding of randomness based on the different types and lengths of "runs" that we observe.

The *runs test* essentially quantifies this intuition into a statistical test. After runs theory was covered quite comprehensively by Mood in his 1940 paper [5], the runs test has become fairly ubiquitous in the RNG testing world. Essentially, it quantifies a notion of ordering in our random number sequence and attempts to identify improbable or unlikely "runs" of numbers. Much of the derivation and examples are derived from [3].

We first define the runs test by transforming the random number sequence in question into a sequence of two different symbols. For example, given some sequence of "random" uniform numbers between 0 and 1, replace all values less than the median (≈ 0.5 in our case) with A and values greater than the median with B. For example, consider the following sequence of 12 draws from $U(0,1)$:

$$0.3, 0.4, 0.2, 0.9, 0.2, 0.2, 0.9, 0.9, 0.1, 0.8, 0.8, 0.6$$

This gets transformed into the following sequence of $n = 6$ A's and $n = 6$ B's:

$$A, A, A, B, A, A, B, B, A, B, B, B$$

Define m_{Ai} as the count of i length runs of A, and m_{Bi} as the count of i length runs of B. Using this notation, we can see from our example that:

$$m_{A1} = 1, m_{A2} = 1, M_{A3} = 1, m_{Ai} = 0 \text{ for } i > 3$$

$$m_{B1} = 1, m_{B2} = 1, m_{B3} = 1, m_{Bi} = 0 \text{ for } i > 3$$

Our null hypothesis in this test is that the sequence is random, and we reject if we notice anything unusual about the observed runs. Intuitively, if our null hypothesis was true, we would observe "insignificant clumping" [3] of A runs and B runs.

Also, define $M_A = \sum_i m_{Ai}$ and $M_B = \sum_i m_{Bi}$. In other words, M_A is the total number of A runs and M_B is the total number of B runs. The number of ways to permute the runs of A and B is:

$$\frac{M_A!}{m_{A1}!m_{A2}!\ldots} \cdot \frac{M_B!}{m_{B1}!m_{B2}!\ldots} \tag{1}$$

Note that $M_A = M_B$ or they only differ by 1. They would differ by 1 in the case that the sequence both starts and ends with the same type of symbol. If $M_A = M_B$, then we actually have double the number of possible sequences, since the A's and B's can just swap places. Thus, we can define a fairly simple function $\alpha(M_A, M_B)$, which evaluates to 2 if $M_A = M_B$ and evaluates to 1 if $M_A \neq M_B$. Our equation from (1) becomes:

$$\frac{M_A!}{m_{A1}!m_{A2}!\ldots} \cdot \frac{M_B!}{m_{B1}!m_{B2}!\ldots} \cdot \alpha(M_A, M_B) \tag{2}$$

Finally, we must also recognize that there are $\frac{(2n)!}{n!n!}$ total ways to order n A's and n B's. Therefore, the probability of a given sequence with these values is:

$$\frac{M_A!}{m_{A1}!m_{A2}!\ldots} \cdot \frac{M_B!}{m_{B1}!m_{B2}!\ldots} \cdot \alpha(M_A, M_B) \cdot \frac{n!n!}{(2n)!} \tag{3}$$

However, this is not the exact probability distribution for M_A, M_B, since there are multiple values of m_{ji} that could satisfiy our fixed constraints of M_A, M_B, and n. To get the exact distribution, we must sum over all different possible values of m_{ji}. We will make the following claim about the exact distribution.

Claim:
Suppose we have n $A's$ and n $B's$, with M_A runs of A's and M_B runs of B's. Then, the exact

probability distribution for M_A and M_B is

$$P(M_A, M_B) = \frac{2\binom{n-1}{M_A-1}\binom{n-1}{M_B-1}}{\binom{2n}{n}} \tag{4}$$

Proof:
First, we will show that there are $\binom{n-1}{M_A-1}\binom{n-1}{M_B-1}$ number of arrangements of runs of A's and B's.

Let's start with the A's. Given n A's and M_A number of A runs, we have to choose M_A A's to serve as "starting points" for our runs, where a "starting point" is defined as the first symbol in a completely new run of that particular symbol. Without loss of generality, let us assume that the first run starts with an A. Thus, we need to choose the remaining $M_A - 1$ starting points for A, which can be done in $\binom{n-1}{M_A-1}$ ways. This same logic applies when ordering the B's.

Now that we have chosen our starting points for the A runs and B runs, we can merge these two orderings together by alternating every A run with a B run. In other words, take the first A starting point and write A's until we reach the next A starting point. Then from there, write the first B starting point and continue writing B's until we reach the next B starting point. Continue this alternating process for all A's and B's. This can be done in $\binom{n-1}{M_A-1}\binom{n-1}{M_B-1}$ ways. However, this is assuming that we start with an A sequence, when we could easily also start with a B sequence. So, the total number of orderings is actually $2\binom{n-1}{M_A-1}\binom{n-1}{M_B-1}$.

Finally, we know that there are $(2n!)/(n!n!)$ total ways to order $2n$ A's and B's. This gives us a final probability of

$$P(M_A, M_B) = \frac{2\binom{n-1}{M_A-1}\binom{n-1}{M_B-1}}{\binom{2n}{n}}$$

□

It may be more convenient to represent our probability distribution in terms of one variable instead of two. Let $M = M_A + M_B$, which is essentially the total count of all types of runs. Note that this barely alters our probabiliy formulation, since M_A either equals M_B or they only differ by 1. With this in mind, we can easily transform our probability distribution from (4) in terms of this single variable M to be:

$$P(M) = \begin{cases} \left[2\binom{n-1}{M/2-1}\right]^2 / \binom{2n}{n} & \text{if } M_A = M_B \\ 2\binom{n-1}{M/2-3/2}\binom{n-1}{M/2-1/2} / \binom{2n}{n} & \text{if } M_A \neq M_B \end{cases}$$

Now that we have the exact distribution of M, we can calculate the exact probability of seeing a particular M value. However, this formulation is quite complicated, and it would be fairly inconvenient to calculate the probability exactly. As it turns out, we can use this formulation to actually show that M can be normally approximated with mean and variance:

$$E(M) = n + 1$$
$$\text{Var}(M) = \frac{n(n-1)}{2n-1}$$

We will not go into the algebraic details of why this is the case, as it is beyond the scope of this paper, though I will direct interested readers to the Mood (1940) paper included in the References section.

3.2 Kolmogorov-Smirnov Test (K-S Test)

The K-S Test can be used to determine if some sample that we observe was drawn from some known distribution. It defines a statistic that quantifies a distance between an empircial distribution of the sample and the theoretical distribution of our known reference.

Before we define our K-S test statistic, we must first define an *empirical distribution function* as

$$F^*(x) = \frac{1}{n}\sum_{i=1}^{n} I_{x_i \leq x} \text{ for } 0 \leq x \leq 1$$

where $I_{x_i \leq x}$ is 1 if $x_i \leq x$ and 0 if $x_i \not\leq x$. In other words, $\sum I_{x_i \leq x}$ is essentially the count of values in our sequence less than x. Define our *theoretical distribution function* as $F(x)$, which is simply going to be the uniform CDF, so $F(x) = x$ for $0 \leq x \leq 1$.

Given these two functions $F^*(x)$ and $F(x)$, our K-S test statistic is simply

$$D = \max |F^*(x) - F(x)|$$

In other words, D is simply the maximum distance between our empirical distribution function and the theoretical distribution. Unfortunately, the K-S distribution for this statistic is not trivial, and there are several various formulations and implementations. Thus, it is not within the scope of this paper to go into the details of the K-S distribution and critical values for D.

Fortunately, the programming language, R, has a built-in K-S test function that we will simply use as a black box. We will be using this function to perform our K-S test.

3.3 Other Tests

While the K-S Test is good for testing the distribution of a sequence and the runs test is good for testing the order of a sequence, these tests only scratch the surface of the battery of tests available for thoroughly testing a RNG. For example, the National Institute of Standards & Technology (NIST) has a statistical testing suite [7] that is quite comprehensive and includes several more complicated tests that include looking at matrix rank, the discrete fourier transform, and entropy to name a few.

In this specific testing framework, we will also include a graphical analysis of a plot using some tuple-collection of numbers in a given random sequence. This is sometimes referred to as the *spectral test*. For example, if RANDU gave us a sequence $x_1, x_2, x_3, \ldots$, we could plot tuples $(x_1, x_2, x_3), (x_4, x_5, x_6), \ldots$, and attempt to identify a distinct "pattern". While this test is somewhat less quantitative than the runs test and the K-S test, it provides a different geometric perspective on random numbers and is useful in illustrating the inherent flaws with RANDU.

4 Experimental Design

With our statistical tests properly defined, we now define a way to apply these tests to our RNGs. For RANDU, we will simply run a graphical analysis kind of test to assess the structure in the generated numbers and hopefully identify a non-trivial pattern. For Mersenne Twister and MWC, we will generate 100 random sequences, each of size 100. We will then run those sequences through the K-S test and the runs test and return the proportion of sequences that passed each test (i.e. the proportion of tests that had non-significant p-values).

For the runs test, we will use the normal approximation to calculate our p-values, with mean and standard deviation as specified in Section 3. Since we are generating sequences of size 100 (therefore $n = 50$ in our notation), the theoretical mean and standard deviation values will be

$$E(M) = n + 1 = 51$$

$$\text{StDev}(M) = \sqrt{\frac{n(n-1)}{(2n-1)}} \approx 4.97$$

The normal approximation method for the runs test is more computationally straightforward than finding exact probability values. Additionally, histograms of the different M values as well

as their normal probability plots will be included in the evaluation for the purposes of arguing normality.

It is also important to note that Mersenne Twister and MWC will be generating $U(0, 1)$ draws. For RANDU, we will generate 10,000 random numbers using the implementation provided in Section 2. This groups successive numbers in pairs of 3, which will then be normalized and used to create a 3D scatterplot in R.

All code used in these experiments will be attached at the bottom of this paper in the Appendix section.

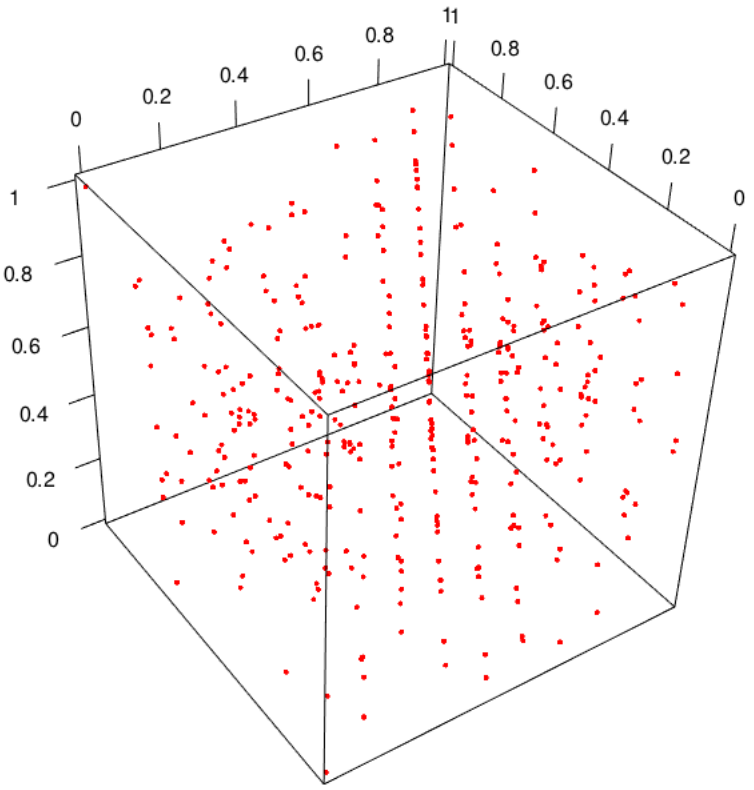
5 Evaluation

The following table gives a summary of the results of each test for each RNG implementation. Note that the framework with which RANDU was tested is fairly different than the framework used for MT and MWC. Values for the runs test and the K-S test are the propotions of sequences generated that passed.

	RANDU	MT	MWC
Runs Test	-	.97	.94
K-S Test	-	.96	.96
Graphical Test	FAIL	-	-

5.1 RANDU

The triples that RANDU generated were graphed into R, and we can clearly see from the attached figure that if the plot is rotated correclty, one can see that the triples disturbingly only map to a distinct number of linear planes.



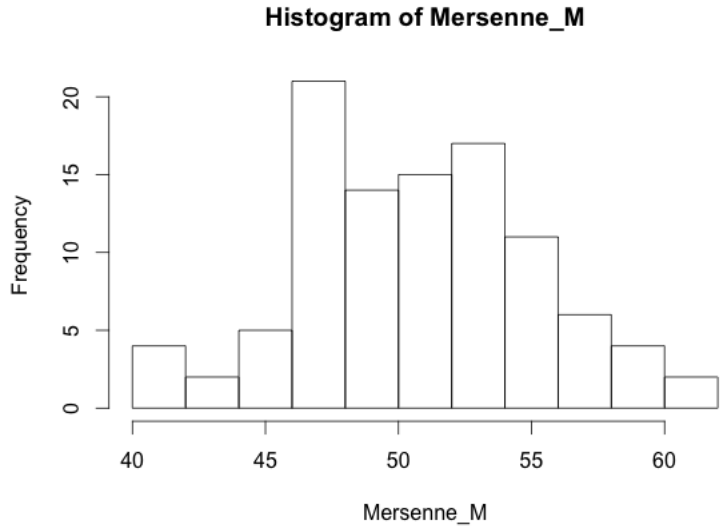
The most visible parallel planes occur in the middle of the cube, and other planes become visible as well with a bit of rotation. It is surprising to realize that although RANDU should technically have a high period (given its high value for the modulo constant), it ends up still only mapping to a handful of planes, as opposed to uniformly filling the space of the cube.

This result suggests that knowing something about the first two numbers in a sequence generated by RANDU has some correlation with what the next number in the sequence is going to be. This is clearly not desirable in an RNG, since ideally each number in the sequence should be completely independent of all other items in the sequence. An adversary could potentially use this information and exploit an RNG, a flaw that could have severe repercussions in applications such as cryptography.

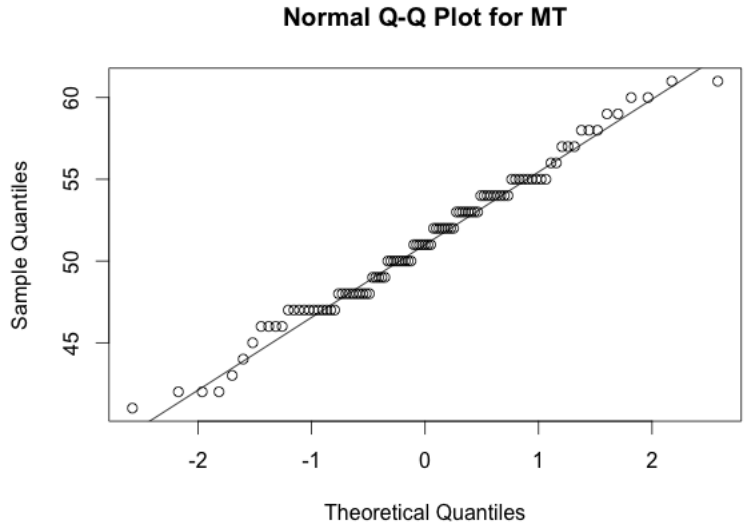
This evaluation of RANDU helps to illustrate the crucial role that constants play in the effectiveness of an RNG. It also provides insight into how the geometric structure of a sequence of numbers can sometimes be more illuminating than a more quantitative statistical test.

5.2 Mersenne Twister

After calculating 100 different values for M, our statistic for the total number of runs, we get the following histogram for M:



This histogram seems to be fairly well approximated by a normal distribution with mean 51 and standard deviation ≈ 4.97. The normal probability plot also provides evidence that the distribution is fairly normal, despite the fact that the disribution is discrete:



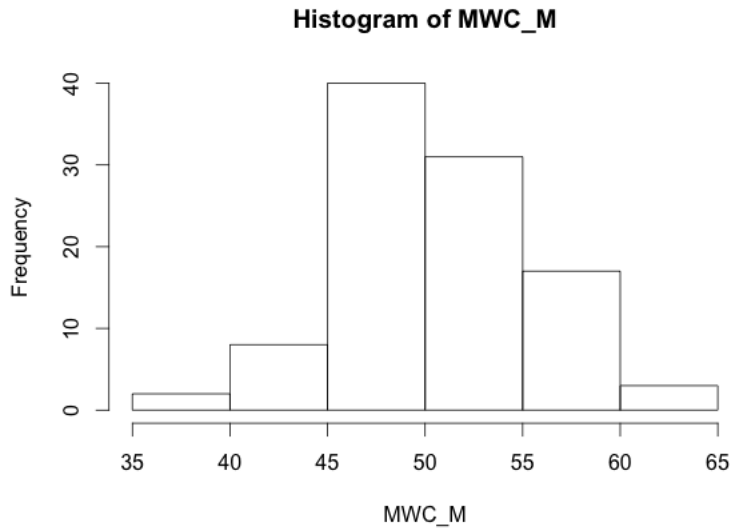
Thus, we can argue that the normal approximation method is acceptable for this test.

For the 100 values of M, we conducted a two-sided p-value test for each one, generating 100 different p-values. For Mersenne Twister, the proportion of p-values that passed (i.e. had a p-value $> .05$) was .97.

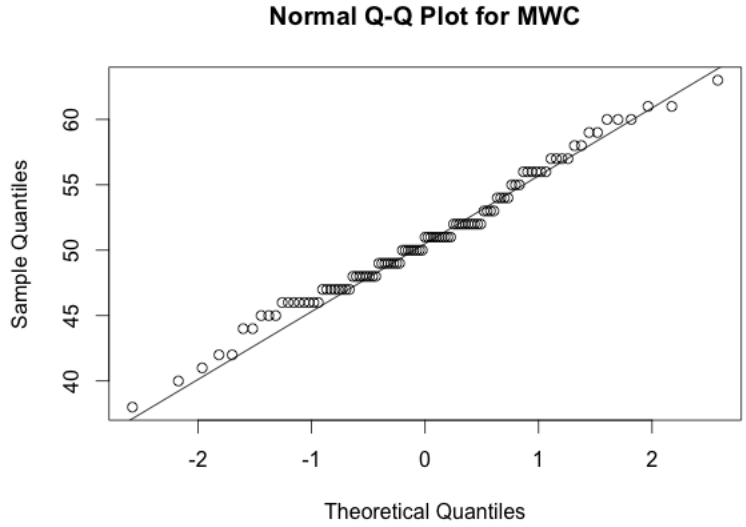
For the K-S test, 96 of the 100 generated sequences passed with non-significant p-values.

5.3 Marsaglia Multiply-With-Carry

The following is the histogram for the 100 different M statistics for MWC:



The histogram for MWC seems a little thinner than the one for MT, but it still seems fairly close to a normal distribution with mean 51 and standard deviation ≈ 4.97. The corresponding normal probability plot also supports the normal approximation:



So, we can still use the normal approximation method to achieve acceptable p-values.

For the 100 values of M, we conducted a two-sided p-value test for each one, generating 100 different p-values. The proportion of p-values that passed was .94, slightly less than the proportion achieved by MT.

For the KS-test, 96 of the 100 generated sequences passed, equivalent to the proportion for Mersenne Twister.

6 Applications - Randomized Algorithms

As alluded to briefly in the introduction, RNGs have immense applications in numerous fields outside of mathematics. There are some immediately obvious examples, such as modeling natural, random phenomena. Cryptography is a significantly growing research issue, especially in the field of computer security, and nearly all of those protocols rely on the performance of a good RNG. The specific example of DieHard presented in the implementation section of MWC also illustrates how the functionality of industry-grade software is completely dependent on the quality of RNGs. As a specific case study, consider the application of RNGs in the field of algorithms, and more specifically, *randomized algorithms*.

Consider the somewhat toy problem of finding the *median* of some set S of size n, where we define the median as the kth smallest element where $k = n/2$ if n is even and $k = (n+1)/2$ if n is odd. An immediately obvious solution would be to sort the array and return the element at index $n/2$ or $(n+1)/2$, depending on whether n is odd or even. However, this algorithm seems overzealous; we only want the median element, yet we end up having to sort the entire set to do this. Additionally, for those familiar with sorting algorithms and Big-O notation, the best sorting algorithms still require $O(n \log n)$ runtime.

Fortunately, one can actually design an algorithm based on randomization to reduce this expected runtime. In this randomized algorithm, we select some index $i \sim U(1, n)$ using some RNG and choose the element at that index (call it a_i) to be a "pivot". Then, we compare every element in S with this pivot element and place it in some new set S^- if it is less than the pivot or some new set S^+ if it is greater than the pivot. We can then examine the sizes of S^- and S^+ to determine our next step.

Let S^- be the set of elements less than our current pivot and S^+ be the set of elements greater than our current pivot. Then, there are three cases:

- $|S^-| = k - 1$: a_i is our median and we are done.
- $|S^-| \geq k$: our pivot was too large, so recursively look for the kth smallest element in S^-.
- $|S^-| < k - 1$: our pivot was too small, so recursively look in S^+. However, in this call, the true median won't be the kth smallest element in S^+. We actually want to find the $(k - 1 - |S^-|)$th smallest element, since we already know the median is greater than $|S^-|$ elements.

We can show that this randomized algorithm of finding the median only has *expected running time* of $O(n)$. However, the analysis of this expected runtime is completely dependent on the fact that choosing the random pivot truly comes from a uniform distribution (i.e. the probability of choosing any particular element as a pivot is equally likely). Clearly, this implies that if the RNG we are using to select our pivots is flawed, then the runtime of this algorithm could be very seriously affected.

7 Conclusion

We have presented some examples of both poor and robust implementations of random number generators. Then, a custom testing framework was developed out of the several different types of RNG tests available in the literature. We found that RANDU has clear flaws in the geometric structure of its random numbers, while Mersenne Twister and Marsaglia's multiply-with-carry stand up to the tests fairly well, each passing the K-S test and runs test around 95% of the time. Finally, we closed with a quick discussion of the application of good RNGs which included a specific case study in the field of randomized algorithms.

Overall, this paper has framed the general process that random number generator programs must go through. First, they are implemented, typically based upon a fairly straightforward model. Then, they are subject to a battery of statistical tests, where designing said tests has always been and will continue to be a heated topic of research. Finally, the acceptable RNGs are sent into the field, upon which numerous research and software projects base much of their results.

8 Acknolwedgments

Thanks to Professor Phil Everson for his constant guidance in selecting a topic, finding resources, and working through proofs. Thanks to Professor Deb Bergstrand for her help in formulating an elegant counting argument for the runs test. Thanks to Professor Joshua Brody for his lecturing in CS41: Algorithms class, which provided the inspiration for the randomized algorithms case study in the Applications section.

9 Appendix (R Code)

```
N <- 100
m <- 100
#store the M statistics for the runs test
Mersenne_M <- c()
MWC_M <- c()

#store the K-S statistics for the K-S test
MT_ks <- c()
MWC_ks <- c()

#generate 100 sequences of length 100
for(j in 0:m) {

set.seed(j, kind="Mersenne")
r1 <- runif(N)
set.seed(j, kind="Marsaglia")
r2 <- runif(N)
### K-S TEST ###
MT_test <- ks.test(r1,punif)
MWC_test <- ks.test(r2, punif)
MT_ks <- c(MT_ks, MT_test['p.value'] )
MWC_ks <- c(MWC_ks, MWC_test['p.value'])
### RUNS TEST ###
r1_runs <- c()
r2_runs <- c()
for(i in 1:N) {
  if (r1[i] < median(r1)) {
    r1_runs <- c(r1_runs, 'A')
  }
  else {
    r1_runs <- c(r1_runs, 'B')
  }
  if (r2[i] < median(r2)) {
    r2_runs <- c(r2_runs, 'A')
  }
  else {
    r2_runs <- c(r2_runs, 'B')
  }
}
#populate M for Mersenne
curr <- r1_runs[1]
Mersenne_total_runs <- 1
for(i in 1:N) {
  if (r1_runs[i] != curr) {
    Mersenne_total_runs <- Mersenne_total_runs + 1
```

```
    curr = r1_runs[i]
  }
}
#populate M for MWC
curr <- r2_runs[1]
MWC_total_runs <- 1
for(i in 1:N) {
  if (r2_runs[i] != curr) {
    MWC_total_runs <- MWC_total_runs + 1
    curr = r2_runs[i]
  }
}
#add to our array of M statistics
Mersenne_M <- c(Mersenne_M, Mersenne_total_runs)
MWC_M <- c(MWC_M, MWC_total_runs)
}

#expected parameters
n = N/2
M_mean = n+1
M_var = n*(n-1)/(2*n-1)
M_stdev = sqrt(M_var)

#p-value arrays for Mersenne and MWC
Mersenne_p <- c()
MWC_p <- c()
#conduct 2-sided p-values
for(i in 1:m){
  if(Mersenne_M[i] < M_mean){
    Mersenne_p <- c(Mersenne_p, 2*pnorm(Mersenne_M[i],M_mean,M_stdev))
  }
  else {
    Mersenne_p <- c(Mersenne_p, 2*(1-pnorm(Mersenne_M[i],M_mean,M_stdev)))
  }
  if(MWC_M[i] < M_mean) {
    MWC_p <- c(MWC_p, 2*pnorm(MWC_M[i],M_mean,M_stdev))
  }
  else {
    MWC_p <- c(MWC_p, 2*(1-pnorm(MWC_M[i],M_mean,M_stdev)))
  }
}

#proportion of p-values for runs test
MT_prop <- 0
for(i in Mersenne_p){
  if(i>0.05) {
    MT_prop <- MT_prop + 1
  }
}
MWC_prop <- 0
for(i in MWC_p){
  if(i>0.05) {
    MWC_prop <- MWC_prop + 1
  }
}
```

```
#propotion of p-values for K-S test
MT_prop <- 0
for(i in MT_ks){
  if(i>0.05) {
    MT_prop <- MT_prop + 1
  }
}
MWC_prop <- 0
for(i in MWC_ks){
  if(i>0.05) {
    MWC_prop <- MWC_prop + 1
  }
}
```

References

[1] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. *ACM*, June 2006.

[2] James E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, New York, 1998.

[3] Oscar Kempthorne and Leroy Folks. *Probability, Statistics, and Data Analysis*, chapter 9.7: Tests of Randomness of a Sequence. Iowa State University Press, 1971.

[4] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1), January 1998.

[5] A.M. Mood. The distribution theory of runs. *The Annals of Mathematical Statistics*, 1940.

[6] Tim Riley and Adam Goucher, editors. *Beautiful Testing*, chapter 10: Testing a Random Number Generator. O'Reilly Media, Inc., 2010.

[7] Andrew Rukhin and et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. National Institute for Standards and Technology, April 2010.