Pseudorandom Numbers: Generation, Statistical Measures, Monte Carlo Methods, and Implementation in C++
Abstract

Applied mathematicians are developing stochastic models for the spread of disease, the behavior of new drugs in the human body, protein folding, and predator-prey models. The stochastic models are set up with interaction rules between each agent in the model, and steps are determined using a random number generator: a process referred to as a Monte Carlo simulation. The random numbers are the fuel that drives the models, and the spark that gives them movement. As a result, stochastic models are only as good as the random numbers they use. But how do we generate random numbers? How do we test numbers to verify that they are indeed random? What does it even mean for a sequence of numbers to be random? The objective of my senior thesis is to survey the literature for answers these questions, and to implement the results in C++. Several generators of random numbers are discussed and programmed, tested by recommended methods, and then used in some simple Monte Carlo simulations: an estimation of π and e, and Monte Carlo integration. Original work in the thesis includes the C++ implementations, along with an applied definition of “randomness” which gives rise to a method for ranking purportedly random sequences. The proposed method uses concepts related to the Kolmogorov-Smirnov test and to regression analysis: a chi-square statistical analysis is performed repeatedly over a sequence, allowing for the development of a cumulative histogram which is compared to an ideal CDF. This test is valuable because it gives researchers the ability to condense the data from a personalized battery of statistical tests for randomness into a single ranking, allowing them to select the random sequences best suited for their project.
Contents

Abstract i

Preface iv

Acknowledgments v

Chapter 1. Motivation 1
   1. Practical Uses of Random Number Generators 1
   2. Definitions of Random Sequences 2
   3. Advice from the Literature 2

Chapter 2. Pseudorandom Number Generators 3
   1. Linear Congruential Generators 3
   2. The Middle-Square Method 6
   3. Fibonacci Generators 9
   4. Our Generators 10

Chapter 3. Statistical Tests for Sequences 11
   1. Definitions and Considerations 12
   2. Frequency Test 18
   3. Serial Test 19
   4. Poker Test 19
   5. Runs Test 20
   6. Local Randomness of Our Generators 21
   7. Global Randomness 22

Chapter 4. Monte Carlo Methods 29
   1. Estimation of \( \pi \) 29
   2. Estimation of \( e \) 31
   3. Monte Carlo Integration 33

Chapter 5. Concluding Remarks 35

Appendix A. Ranking Global Random Matrices 37
Appendix B. C++ Programs 40

Bibliography 59
Preface

The advent of computers matured the incipient science of random number generation from an unstudied process of blindly sampling data from census tables and urns filled with marbles into a class of statistically tested mathematical and physical generators. This allowed for the development of Monte Carlo methods – statistical methods for estimating numerical solutions using random numbers – which in turn created demand for more “random” random-number generators. The purpose of this paper is to provide an in-depth study of a handful of historical and contemporary methods used to generate random sequences of numbers, to test the generated sequences with a battery of statistics, and to explore the applications of random sequences. For a broad survey of such generators see [Gen03], and for an exhaustive discussion on the related programming issues see [Knu98].

I will include instructions for implementing these methods in C++. In particular, I will examine seven types of generators in detail and then choose a battery of four statistical tests to apply to the chosen generators. The results of these tests will be included in the text, while the source code for the generators and the tests can be found in the appendix.

The topic at hand is nontrivial. Monte Carlo methods are only as good as the random sequences they use, and many older generators included in programming libraries have subtle – yet fatal – flaws [Knu98, 188-89]. While many of these errors have been corrected, it is important to test a generator before using it.
Acknowledgments

I would like to thank Dr. Snow for the many hours he spent helping me develop this paper over the past year and a half. I am very grateful for all of the time he spent on this project, both in our meetings and during his own time. I would also like to thank my classmate Nicholas Battafarano for having introduced me to the notion of Monte Carlo methods immediately prior to my initial meeting with Dr. Snow.
CHAPTER 1

Motivation

We avoided doing perhaps difficult analytic calculations by sitting back and instructing a computer to generate random numbers.

- John Rice [Ric07, 267]

The topic of random number generation does not fit neatly into any classical field of mathematics. Still a nascent discipline, its study currently utilizes abstract algebra, probability, statistics, combinatorics, and number theory. Early studies into the theory of probability – spurred on by attempts to analyze casino games – provided the original need for random number generators. Further development, however, would have to wait until the creation of computers. Prior to this, trials had to be performed with dice, cards or roulette wheels. This process was cumbersome, but even worse: it was irreproducible.

Mathematical modeling is a rapidly developing field. Applied mathematicians are developing models for the spread of disease (both in specific cities and nationwide), the behavior of new drugs in the human body, protein folding, predator-prey models, and the development of embryonic structures. Some of these models are deterministic and the end-behavior is determined by analytic or numerical solutions to large systems of differential equations.

However, the newest and most interesting research is coming from stochastic models. The stochastic models are set up with interaction rules between each agent in the model, and steps are determined using a random number generator. Because the models depend on the random numbers, several executions of the model over different random sequences will give output that can be analyzed in terms of statistical sampling distributions.

1. Practical Uses of Random Number Generators

Before looking to the powerful Monte Carlo methods, it would be nice to consider some naïve applications of random number generators. Consider first of all CD and MP3 players. We take it for granted that we push a “randomize” button, and our music plays in a different order. It would be irritating if this randomization contained obvious patterns, or did not play every available song (instead cycling through only a few).

What about some screen savers? For example, a standard Windows screen saver works by flashing the Windows logo in different parts of the screen, in an unpredictable manner. How does it choose where to display the logo? Or what about computerized lotteries? If they followed a pattern, it would be possible to anticipate future results.

1
Some large companies or government agencies give their executives “random”
keys to access a mainframe off-site. Each user has a username and password, but
also has a key chain which displays a 10-digit number. To login, the user gives their
username, password, and the number appearing on their key chain at that time.
The number changes every 15 seconds, as it is synchronized with the mainframe.
How are the numbers generated? Furthermore, how are they generated in a way
such that, given knowledge of a long sequence of past numbers, future numbers
remain unpredictable?

We will address these questions later. First we give a theoretical definition of
randomness.

2. Definitions of Random Sequences

Intuitively, a “random sequence” is one such that each number in the sequence
is independent of and uncorrelated with the other numbers in the sequence. Given
the knowledge of any portion of the sequence, we should not be able to predict the
next value in the sequence. That is, working with a decimal system, the probability
of the next number of the sequence being any number should be $\frac{1}{10^n}$ no matter how
many of the previous numbers we know.

Knuth [Knu98, 151-159] defines a random sequence as follows:
The sequence $\langle U_n \rangle = U_0, U_1, U_2, \ldots$ of real numbers with $0 \leq U_n < 1$ is said to
be $k$-distributed if
\[
\Pr(u_1 \leq U_n < v_1, \ldots, u_k \leq U_{n+k-1} < v_k) = (v_1 - u_1)\cdots(v_k - u_k)
\]
for all choices of real numbers $u_j, v_j$, with $0 \leq u_j < v_j \leq 1$ for $1 \leq j \leq k$. The
sequence is $\infty$-distributed, or random if it is $k$-distributed for all positive integers $k$.

We may interpret this definition in the language of probability: Let $\langle X_n \rangle$ be
a collection of independent identically distributed random variables, with uniform
distribution over $[0,1)$. Let $\langle U_n \rangle = U_0, U_1, U_2, \ldots$ be a sequence of real numbers with
$0 \leq U_n < 1$. Then $\langle U_n \rangle$ is a random sequence if
\[
\Pr(u_1 \leq U_n < v_1, \ldots, u_k \leq U_{n+k-1} < v_k) = \Pr(u_1 \leq X_n < v_1, \ldots, u_k \leq X_{n+k-1} < v_k)
\]
for all choices of real numbers $u_j, v_j$, with $0 \leq u_j < v_j \leq 1$ for $1 \leq j \leq k$. Because
of the independence assumption, this is equivalent to the condition
\[
\Pr(u_1 \leq U_n < v_1, \ldots, u_k \leq U_{n+k-1} < v_k) = \Pr(u_1 \leq X_n < v_1)\cdots\Pr(u_k \leq X_{n+k-1} < v_k)
\]

3. Advice from the Literature

James Gentle bemoans the subtle programming issues relating to generating
random sequences, and orders that well-tested software for random number generation
should be used instead of software developed ad hoc [Gen03, 87]. However,
it is most prudent to run each Monte Carlo program at least twice using different
sources of random numbers, before taking the output of the program seriously. This
will give an indication of the precision of the results, and will protect the simulation
from relying on generators with hidden deficiencies. “Every random number
generator will fail in at least one application.” [Knu98, 189]. Thus, the responsible
researcher should use at least two well tested generators for Monte Carlo applications,
and should also run a trial with his own generator, lest the researcher gamble
the validity of his results on the strength of generators he is not familiar with.
Chapter 2

Pseudorandom Number Generators

Uniform pseudorandom numbers in the interval \([0,1]\) are basic ingredients of any stochastic simulation. Their quality is of fundamental importance for the success of the simulation, since the outcome of a typical stochastic simulation strongly depends on the structural and statistical properties of the underlying pseudorandom number generator. Therefore, the selection of a suitable one is a crucial task in any simulation project, which should not be left to chance...

- Jürgen Eichenauer-Herrmann, Eva Herrmann, and Stefan Wegenkittl [EHHW96, 66]

“Random numbers should not be generated with a method chosen at random” [Knu98, 6]. Good generators are delicate, and haphazard modifications will often undo the qualities that made them good in the first place.

Our output from these generators will lie in the interval \([0,1]\). Even though the algorithms used here generate output in the range \([0,m−1]\) for some \(m \in \mathbb{Z}_+\), we scale each \(x_i\) to \(\frac{x_i}{m}\) before it is recorded.

1. Linear Congruential Generators

The simplest and best studied random generators are of the form

\[ x_i \equiv (ax_{i−1} + c) \mod m, \]

where \(a, c \in \mathbb{Z}, 1 \leq a \leq m−1, 0 \leq c \leq m−1\). Such generators are called linear congruential generators (LCG), or multiplicative congruential generators (MCG) in the case that \(c = 0\). The input \(a\) is called the multiplier, \(m\) is called the modulus, the initial value \(x_0\) is called the seed, and \(c\) is the increment.

Now since there are finitely many values for the generator to return, and because each \(x_i\) is determined by \(x_{i−1}\) (because \(a, m,\) and \(c\) are fixed), the sequence of output from any LCG must repeat after some point. Define the period of a random number generator to be the value \(j\) such that \(x_k = x_{k+j} \forall k \geq N\) for some fixed, finite \(N\). In the case of any LCG, we may simply let \(N = 0\), but for some other types of generators this may not be the case. A LCG with modulus \(m\) may return any of the \(m\) values in the set \(\{0,1,...,m−1\}\) and thus has a maximum period of \(m\). However, if a MCG should ever return \(x_i = 0\), then every subsequent value will also be 0, giving the generator a period of 1 (this will never happen if \(m\) is chosen to be a prime number, making the domain of the generator into an integral domain). Or if \(x_i\) is a multiple of any divisor \(d\) of \(m\), then every subsequent value of the sequence will also be a multiple of \(d\) [Knu98, 19]. So we want every element of the sequence to be relatively prime to \(m\); thus a MCG with modulus \(m\) has a maximum period of \(\phi(m)\) – Euler’s totient function, the number of positive integers relatively prime to \(m\) – which is maximized when \(m\) is prime to give \(\phi(m) = m−1\).
Remark 1.1. A sequence generated by this process is a Markov chain.

Example 1.2. Consider the MCG with \( a = 7 \), \( m = 17 \) and \( x_0 = 5 \), giving us \( x_i \equiv 7x_{i-1} \mod 17 \). From this it follows that \( x_1 = 1 \), \( x_2 = 7 \), and so the first 21 numbers from this generator are

\[
[5, 1, 7, 15, 3, 4, 11, 9, 12, 16, 10, 2, 14, 13, 6, 8, 5, 1, 7, 15, 3]
\]

Notice that when \( x_{i-1} \) returns to the seed value of 5, the sequence repeats. So \( x_{i+j} = x_i \) if \( j = 16 \), so this generator has period 16.

Example 1.3. Take the same MCG as in Example 1.2, but this time set the multiplier \( a = 4 \), giving us \( x_i \equiv 4x_{i-1} \mod 17 \). The first 21 numbers from this sequence are

\[
[5, 3, 12, 14, 5, 3, 12, 14, 5, 3, 12, 14, 5, 3, 12, 14, 5]
\]

This generator only has period 4, meaning it does not do as good of a job “covering” the range \( \{1, 2, ..., 16\} \).

These examples lead to a natural question: “What multipliers will give a generator of maximum period, given any modulus \( m \)?” However, we must place the value of the answer to this question in perspective, and that is best accomplished with an example.

Example 1.4. The LCG with \( a = 1 \), \( m = 17 \), \( x_0 = 16 \) and \( c = 1 \) taking the form \( x_i \equiv (x_{i-1} + 1) \mod 17 \) gives us

\[
[16, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
\]

as its first 17 numbers. So this LCG has maximum period, but we can hardly call it a “random” number generator.

Indeed, we would like for our random number generators to have maximum period; however, as Example 1.4 illustrates, the property of having maximum period may be regarded as a necessary, but not a sufficient condition for being a good random number generator. We will now consider the properties of good multiplicative congruential generators.

We first maximize the period of our generator. Every computer has a physical limitation on the largest integer it can store: this value is called the word size. The word size \( \omega \) for a C++ user is effectively determined by the range of the unsigned int data type. Most personal computers have a capacity of 4 bytes = 32 bits, meaning the range of unsigned int is \( 2^{32} - 1 = 4,294,967,295 \). This value will hereafter be referenced as the word size \( \omega \).

Remark 1.5. It is imperative that the programmer use unsigned int, because int uses a bit to indicate the sign of the number, and as such will only have a positive range of \( 2^{31} = 2,147,483,648 \). Additionally, it is unwise to be greedy and attempt to use the double data type. This can result in undesirable rounding. It is easy to become desirous of the \( 10^{308} \) number range offered by double, but the caveat is that only the ten most significant digits of such numbers are stored. A cardinal rule of random number generation is that the calculations must not be approximated; if they are, our delicate theory is lost and the generator loses all of the properties we worked so hard to develop.
We should worry that \( \omega \) may be still too lofty a bound for \( m \). Suppose that the multiplier and the \( i \)th value of the sequence are both, as they should be, less than \( m \) but that their product is greater than \( m \). This is a problem; at first glance it appears as if we must choose \( m < \sqrt{\omega} \). However, there is a way to calculate \( ax \mod m \) without computing any numbers that exceed \( m \) in absolute value, given a restriction on \( a \). The method is given in [Knu98, 16].

**Theorem 1.6.** Assume \( 0 < a < m \), and that \( a^2 \leq m \). Then

1. If \( q = \lfloor m/a \rfloor \) then \( a(x - (x \mod q)) = \lfloor x/q \rfloor (m - (m \mod a)) \).
2. \( a \) makes it possible to evaluate \( ax \mod m \) without computing any numbers that exceed \( m \) in absolute value.

**Proof.** (a) Note that \( \frac{m}{a} = q + \frac{m \mod a}{a} \), meaning \( m = aq + (m \mod a) \). Thus the right-hand side of (a) becomes

\[ \frac{x}{q} = \lfloor x/q \rfloor + \frac{(x \mod q)}{q}, \]

meaning \( x = q \lfloor x/q \rfloor + (x \mod q) \), and the left-hand side of (a) becomes

\[ a(q \lfloor x/q \rfloor + (x \mod q)) = aq \lfloor x/q \rfloor \]

(b) Let \( r = (m \mod a) \) and define

\[ t := a(x \mod q) - r \lfloor x/q \rfloor \]

Notice that \( t > -m \) because \( a(x \mod q)q - 1 < m \) and

\[ r \lfloor x/q \rfloor \leq r \lfloor (m - 1)/q \rfloor = r \lfloor a + (r - 1)/q \rfloor = ra \leq qa < m \]

if \( 0 < r \leq q \). Finally, our assumption that \( a^2 \leq m \) gives us \( r < a \leq q \). So we may conclude that if \( a(x \mod q) \geq r \lfloor x/q \rfloor \) then

\[ ax \mod m = t \]

else if \( a(x \mod q) < r \lfloor x/q \rfloor \) then

\[ ax \mod m = t + m \]

So we may choose any \( m \leq \omega - 1 \) as long as we also choose \( a \leq \sqrt{m} \). Note that in the second case we get \( t < 0 \), which is a problem because we are using unsigned integer. A solution which is used in the programs in the appendix is

\[ \text{if}(r \lfloor x/q \rfloor > a(x \mod q)) \]

**then subtract** \( r \lfloor x/q \rfloor \) **from** \( m \) **and then add** \( a(x \mod q) \)

We saw earlier that the period of a MCG is maximized if \( m \) is chosen to be prime. This leads us to hypothesize that the optimal modulus for a MCG is the largest prime less than \( \omega \). And we may conclude the same for a LCG with \( c \neq 0 \).

**Theorem 1.7.** A linear congruential sequence defined by \( m, a, c \) and \( x_0 \) as above has period length \( m \) if and only if

1. \( c \) is relatively prime to \( m \);
2. \( b = a - 1 \) is a multiple of \( p \), for every prime \( p \) dividing \( m \);
3. \( b \) is a multiple of \( 4 \), if \( m \) is a multiple of \( 4 \).

For a proof, see [Knu98, 17].

The simplest approach to satisfying these requirements is to choose \( m \) prime and \( c = 1 \). We will finish this section with another theorem.
THEOREM 1.8 (C.F. Gauss, Disquisitiones Arithmeticae (1801), §90-92.). The maximum period possible when \( c = 0 \) and \( m \) is prime is \( \phi(m) = m - 1 \). This period is achieved if \( a \) is a primitive element modulo \( m \).

The original version of Theorem 1.8 orders that \( x_0 \) is relatively prime to \( m \), and it gives details for calculating the maximum period possible for arbitrary \( m \). Both of these steps are simplified by our assumption that \( m \) is prime.

To summarize our results, a MCG will achieve its maximal period \( m - 1 \) if we choose \( m \) to be the largest prime less than \( \omega \), and choose \( a \leq \sqrt{m} \) such that \( a \) is a primitive element modulo \( m \). Of course, if \( m \) is prime, then every integer less than \( m \) is primitive \( \mod m \). We require that \( a \leq \sqrt{m} \) to satisfy the hypothesis of Theorem 1.6. If we want a LCG with \( c \neq 0 \), then we may obtain a maximal period of \( m \) by obeying the same restrictions on \( a \) and \( m \) and by additionally choosing \( c \) relatively prime to \( m \). In the case mentioned above where \( \omega = 2^{32} \), the smallest prime less than \( \omega \) is \( 2^{32} - 5 = 4,294,967,291 \). We will have to wait until we have statistical tests at hand to test the generators before we can further refine these choices. In fact, the next step of the refinement process of a LCG is to compare the results from different primes with statistical tests. We will save time later by using a handful of recommended generators.

However, we may wonder how we would fare by constructing a LCG from the theory we have developed so far. To satisfy our curiosity, we will make our own generator and test it against others, recommended generators.

We’ll choose a modulus first. From Theorems 1.6, 1.7, 1.8, we know it would be best to choose a prime modulus, so we will choose the smallest prime less than the word size; that is, let \( m = 2^{32} - 5 = 4294967291 \).

Now to choose \( a \). Theorem 1.6 required that \( a < \sqrt{m} = \sqrt{4294967291} \approx 65535 \). Let’s use 12345.

The increment must be some integer less than \( m \). Let’s pick \( c = 3 \) and likewise for our seed \( x_0 = 3 \).

We are following the guidelines we have developed, but at a certain point we must make a choice from the options allowed. With no theory to guide us at this point, we choose on a whim. However, we must keep in mind our initial warning to not “make random number generators at random.” We will see how this generator behaves, just to satisfy our curiosity. The tests we develop could and should be used at some later time to find the optimal parameters for the modulus \( 2^{32} - 5 \), rather than just saying, “let’s use...”

2. The Middle-Square Method

A primitive but historically significant method, the middle square method (MSM) was presented by John von Neumann in 1946 as a purely arithmetical process for generating random numbers. The program we will use for the process is given in Appendix B, Section B on p. 41.

Here is the process. Suppose we want to generate 10 digit random numbers. Choose some 10 digit number, square it, and then remove the middle 10 digits from the square of the original number. This is the next number in the sequence. Repeat.

Besides having been shown to have undesirable statistical qualities, implementations of MSM generators in C++ are severely restricted by the word size \( \omega \), defined earlier. The “squaring” process means that the input may not exceed \( \sqrt{\omega} = 65536 \).
Without more creative techniques (such as those employed by Mathematica), it is not easy to create a MSM faithfully with more than 4 digits. As a result, this generator is of little value for creating good random sequences, but we won’t abandon it yet. We should be curious about the period of the generator, and while we do not have any theory to answer this question, we can still test all 9,999 possible seeds.

Using the program in the appendix just referenced, we test each seed from 1 to 9999, counting the length of the sequence each sequence before it begins to cycle. This length will be referred to as the Total Length. Some of the seeds lead to degenerate sequences of zeros, while others progress for a while until a small piece of the end of the sequence begins to cycle. Call the non-cycling segment of the sequence the Stem, and simply refer to the cycling bit as the Cycle. Notice from Figure 2 that every Cycle period turns out to be 1, 4, or 10. (The case of Cycle period = 0 represents the sequences that degenerate to zero.) Also note Table 2 shows the 25 longest-sequence producing seeds. The longest sequence is of length 126 and is generated by the seed 1781.
2. PSEUDORANDOM NUMBER GENERATORS

Table 1. Period For Seed Values 1 Through 25, MSM

<table>
<thead>
<tr>
<th>Number</th>
<th>T-Length</th>
<th>S-Length</th>
<th>C-Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>74</td>
<td>74</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>72</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>73</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>24</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>32</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>71</td>
<td>71</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>70</td>
<td>66</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>95</td>
<td>91</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>24</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>72</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>67</td>
<td>63</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>29</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>11</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>20</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>58</td>
<td>54</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>70</td>
<td>66</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>57</td>
<td>57</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2. Top 25 Performing Seed Values for MSM (Seeds 1-9999)

<table>
<thead>
<tr>
<th>Number</th>
<th>T-Length</th>
<th>S-Length</th>
<th>C-Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>1781</td>
<td>126</td>
<td>122</td>
<td>4</td>
</tr>
<tr>
<td>1719</td>
<td>125</td>
<td>121</td>
<td>4</td>
</tr>
<tr>
<td>6554</td>
<td>125</td>
<td>121</td>
<td>4</td>
</tr>
<tr>
<td>9549</td>
<td>124</td>
<td>120</td>
<td>4</td>
</tr>
<tr>
<td>1834</td>
<td>123</td>
<td>119</td>
<td>4</td>
</tr>
<tr>
<td>3635</td>
<td>122</td>
<td>118</td>
<td>4</td>
</tr>
<tr>
<td>6899</td>
<td>122</td>
<td>118</td>
<td>4</td>
</tr>
<tr>
<td>7707</td>
<td>122</td>
<td>118</td>
<td>4</td>
</tr>
<tr>
<td>2132</td>
<td>121</td>
<td>117</td>
<td>4</td>
</tr>
<tr>
<td>3978</td>
<td>121</td>
<td>117</td>
<td>4</td>
</tr>
<tr>
<td>5962</td>
<td>121</td>
<td>117</td>
<td>4</td>
</tr>
<tr>
<td>6620</td>
<td>121</td>
<td>117</td>
<td>4</td>
</tr>
<tr>
<td>7799</td>
<td>121</td>
<td>117</td>
<td>4</td>
</tr>
<tr>
<td>4873</td>
<td>120</td>
<td>116</td>
<td>4</td>
</tr>
<tr>
<td>5454</td>
<td>120</td>
<td>116</td>
<td>4</td>
</tr>
<tr>
<td>5913</td>
<td>120</td>
<td>116</td>
<td>4</td>
</tr>
<tr>
<td>8244</td>
<td>120</td>
<td>116</td>
<td>4</td>
</tr>
<tr>
<td>7461</td>
<td>119</td>
<td>115</td>
<td>4</td>
</tr>
<tr>
<td>7814</td>
<td>119</td>
<td>115</td>
<td>4</td>
</tr>
<tr>
<td>9635</td>
<td>119</td>
<td>115</td>
<td>4</td>
</tr>
<tr>
<td>585</td>
<td>118</td>
<td>114</td>
<td>4</td>
</tr>
<tr>
<td>665</td>
<td>118</td>
<td>114</td>
<td>4</td>
</tr>
<tr>
<td>4174</td>
<td>118</td>
<td>114</td>
<td>4</td>
</tr>
<tr>
<td>6665</td>
<td>118</td>
<td>114</td>
<td>4</td>
</tr>
<tr>
<td>8332</td>
<td>118</td>
<td>114</td>
<td>4</td>
</tr>
</tbody>
</table>
3. Fibonacci Generators

We can construct generators using the Fibonacci sequence,

\[ x_{n+1} = (x_n) + x_{n-1} \mod m \]

This generator was considered in the early 1950s, but tests have shown that the numbers produced by this sequence are not random. We include it as an example of a bad generator. Later, our statistical tests will expose some of its weaknesses.

More useful are the lagged Fibonacci sequences, first proposed in 1958 by G. J. Mitchell and D. P. Moore [Knuy98, 27]. These sequences take the form

\[ x_n = (x_{n-l} + x_{n-k}) \mod m. \]

Specifically,

\[ x_n = (x_{n-24} + x_{n-55}) \mod 2^e, \quad n \geq 55 \]

where \( e \) is an arbitrary positive integer and \( x_0 \ldots x_{55} \) are random integers, not all even. The constants 24 and 55 in this definition were not chosen at random: they result from a paper by R. Brent. Equation 3.1 has a period of length exactly \( 2^{e-1}(2^{55} - 1) \) [Knuy98, 28-40], [Bre94]. Notice the change to a modulus of \( 2^e \) from the prime used for the LCG. Brent’s results hold only for such moduli, and “there is no obvious generalization to odd moduli” [Bre94, 393]. Having a non-prime modulus is not a disadvantage. Recall that we needed a prime modulus for the LCG in order to give us a maximum period slightly less than \( \omega \). Equation 3.1 with \( e = 32 \), however, has a period of \( 2^{31}(2^{35} - 1) \approx 2^{69} \).

Notice that lagged Fibonacci sequences require many seeds. The generator given by Equation 3.1 requires 56 seeds. These should come from another random generator; ours will come from the generator gMCG, described at the end of this chapter.

It turns out that there are other values of \( l, k \) that work and, in fact, give rise to sequences with period \( 2^{e-1}(2^k - 1) \) where the modulus is \( 2^e \). The pairs \((l, k)\) are those such that \( f(x) = x^l + x^k + 1 \) is primitive over \( \mathbb{Z}/2 \). Some of the other available pairs are listed in Table 3, which is reproduced here verbatim from [Knuy98, 29]. The formula given by Brent for period length holds in general for these pairs \((l, k)\). That is,

\[ \text{period} = 2^{e-1}(2^k - 1) \]

To put this into perspective, recall that the maximum period length which we can obtain with a LCG in C++ is just under 4.3-billion. We will use a recommended Fibonacci generator [Knuy98, 186],

\[ x_j = (x_{j-100} - x_{j-37}) \mod 2^{30}, \]

and so we obtain a period of

\[ 2^{29}(2^{100} - 1) \approx 6.8 \times 10^{38} = 680\text{-undecillion} \]
These lagged Fibonacci sequences have not been well tested. We can prove that they have very long periods, but their randomness is still in question. Recall our earlier example which showed that a long period is not sufficient evidence to call a generator “good.”

4. Our Generators

We will use the following generators as subjects for statistical tests and will rank them at the end of the next chapter.

<table>
<thead>
<tr>
<th>Generator Name</th>
<th>Abbreviation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Random</td>
<td>NR</td>
<td>$x_0 = 0, x_i = x_{i-1} + 1 \mod 100$</td>
</tr>
<tr>
<td>Our own LCG</td>
<td>oLCG</td>
<td>$x_0 = 3, a = 12345, c = 3, m = 2^{32} - 5 = 4294967291$</td>
</tr>
<tr>
<td>“Good” MCG</td>
<td>gMCG</td>
<td>$x_0 = 1, a = 48271, m = 2^{31} - 1$</td>
</tr>
<tr>
<td>Fibonacci Sequence</td>
<td>FIB</td>
<td>$x_j = (x_{j-1} + x_{j-2}) \mod 2^{30}$</td>
</tr>
<tr>
<td>Lagged FIB</td>
<td>LGFIB</td>
<td>$x_j = (x_{j-100} + x_{j-37}) \mod 2^{30}$</td>
</tr>
<tr>
<td>C++ generator</td>
<td>rand()</td>
<td>machine dependent</td>
</tr>
<tr>
<td>$\pi$</td>
<td>$\pi$</td>
<td>3141,5926,...</td>
</tr>
</tbody>
</table>

- NR is simply .01,.02,... It should be thrown out by most of our statistical tests. We include it so that we may see how the tests react to such sequences.
- oLCG is the one we developed previously in this chapter.
- gMCG is recommended by Knuth at the end of his discussion of random numbers [Knu98, 185].
- FIB is simply the Fibonacci sequence, $\mod 2^{30}$.
- LGFIB is also recommended by Knuth in his summary [Knu98, 186]
- rand() is the sequence resulting from the rand() function in C++. The numbers used here are those from the author’s compiler, for which the source code is not known.
- $\pi$ comes from taking four numbers at a time from the decimal expansion of $\pi$

The implementations of these programs can be found in the appendix. To generate the oLCG sequence, use gMCG.cpp and enter the proper parameters. The expansion of $\pi$ comes from the function N[Pi,20000000] in Mathematica, and then taking every four numbers to represent a four digit decimal.
CHAPTER 3

Statistical Tests for Sequences

*If the numbers are not random, they are at least higgledy-piggledy.*

-George Marsaglia (1984)

The moral of the story of testing sequences for randomness is to think of combinatorial properties of a \( U(0,1) \) sequence, and to test properties of a given sequence against these with a chi-square test. Other methods exist, but these are the easiest to work with.

Sequences are tested for randomness by disaggregating the numbers of the sequence into discrete categories for which it is possible to calculate expected values, and then using the chi-square test to check the results. Recall that given \( n \) observations divided into \( k \) categories, corresponding to \( k-1 \) degrees of freedom, \( Y_i \) the observed number of elements in each category \( i \), and \( p_i \) the probability that any observation belongs to \( Y_i \), then

\[
\chi^2 = \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i}
\]

There is a continuous analog of chi-square, known as the Kolmogorov-Smirnov test, which will not be discussed here; however, we will introduce a similar method. For more information, see [Knu98, 48] or [Gen03, 74].

In addition to the chi-square test, we will make use of the chi-square distribution with \( k \) degrees of freedom, which is defined as the distribution of the sum of the squares of \( k \) independent, standard normal variables. That is, for standard normal variables \( Z_i \),

\[
\chi^2 = Z_1^2 + Z_2^2 + \cdots + Z_k^2
\]

This distribution with \( \chi \) degrees of freedom has probability density function

\[
f_{\chi}(x) = \frac{x^{k/2-1}e^{-x/2}}{2^{k/2}\Gamma(k/2)}, \quad x > 0.
\]

However, the cumulative distribution function of \( \chi^2 \) is not generally an elementary function [BL90, 284,85]. It is given by the following equation, as described by [GN96, 7].

\[
\alpha = \mathbb{P}(\chi^2_k \geq y) = \frac{1}{2^{(k/2)}\Gamma(k/2)} \int_{y}^{\infty} z^{k/2} e^{-z/2} dz, \quad y > 0
\]

There are several sources of free, well developed and well tested batteries of statistical tests for random sequences. The DIEHARD program, for example, gives
a battery of eighteen tests, for which the input is a binary file consisting of the 32-bit (unsigned) integers produced by the random number generator.

For practical purposes, it is advisable to use these pre-developed batteries of tests on generators in addition to any homemade battery. For our purposes, however, we do not want to apply pre-developed tests to the sequences we have worked to produce, only to check the output of someone else’s program against a chi-squared table and trust the results. It is instructive to program the tests and tailor them to our own needs, while at the same time discussing the theory which motivates their use.

The output of our generating programs gives numbers in the unit interval $U[0, 1)$. Our statistical tests will require integer input. As in Chapter 1, we will refer to our generated sequence in $U[0, 1)$ with the notation

\[(U_n) = U_0, U_1, U_2, \ldots\]

which we will convert into the integer sequence

\[(Y_n) = Y_0, Y_1, Y_2, \ldots\]

via the rule

\[Y_n \left\lfloor dU_n \right\rfloor\]

for some integer $d$. We want to test the assumption that these sequences are independently and uniformly distributed between 0 and $d - 1$. The number $d$ is chosen for convenience: it should be large enough so that the test is meaningful, but not so large that the test becomes impractical to carry out [Knu98, 61]. To this end, it is important to keep machine limitations in mind. For example, the computer will record no more than the 6 most significant digits (roughly the 20 most significant bits) when storing double type. This means that we should never choose $d$ to be larger than $2^{19}$. Otherwise, the least significant digits of elements of $Y_n$ will be 0 because of rounding. In most cases, $d$ need not and should not exceed 100. Additionally, we must be mindful of the rounding when making calculations for our tests so that we do not claim to produce results which are more precise than our initial data.

1. Definitions and Considerations

The process of testing random number generators requires a different mind-set than that used to develop the generators. We learned that the generator must not round its calculations, so that its output behaves as predicted. We worked hard to determine the best combinations of a multiplier $a$, modulus $m$, and increment $c$ for linear congruential generators. However, we do not have tools to optimize the parameters of our statistical test.

What value of $n$ should we choose? That is, how many elements of a sequence should we use when we calculate the chi-square statistic? What scaling factor $d$ should we use? It is tempting to believe that these statistical tests give continuous output with respect to their arguments: but do they?

These factors can muddle attempts to measure and rank the quality of generators. Consider the following definitions which will be explored in great detail. Note that [Knu98] sprinkles the phrases “local nonrandomness” and “global non-randomness” around (e.g. page 80), but our definitions here are a bit different. Though Knuth does not give an explicit definition, the following summarizes and mirrors his notion of local randomness. We abuse the notion of “global” properties
below, in that our definition does not require knowledge of an entire period; instead, it is a large collection of “local” measurements over the sequence. Nevertheless, this abuse seems valid, because it seems that our global measurements converge. I offer no proof of this, only some graphical evidence later. The method of global testing we will develop here is similar to the Kolmogorov-Smirnov test. However, instead of looking for the maximum deviation between empirical data and an ideal cumulative distribution function, we will observe the sum of squared differences between the finite points of the empirical distribution and the CDF.

**Remark 1.1.** Notation: If $S$ is a sequence of numbers in $U[0,1)$, then $S_{(i,n)}$ will denote the subsequence of $S$ of length $n$, beginning at the $i$-th position in the sequence, where $n \geq 1$ and $i \geq 1$. Notice that we begin indexing $i$ at 1. And if we wish to discuss a collection of $J$ different sequences, and furthermore the length $n$ subsequences starting at the $i$-th value of each sequence, we will index the different sequences by a superscript, so that the subsequence of the $j$-th sequence $S^j$ is given by $S^j_{(i,n)}$.

**Definition 1.2.** A *statistical test* is a process by which combinatorial properties of a subsequence $S_{(i,n)}$ are measured against those of a uniform, independent sampling of $U[0,1)$ via a chi-square analysis. The output of a statistical test on the subsequence is a number in $[0, \infty)$, and is the result of a chi-square test. The number of degrees of freedom depends on the design of the test. We will talk more about the statistical tests later.

**Definition 1.3.** A *battery* $B$ of statistical tests is a collection of $m$ statistical tests. Order them so that it is possible to talk about the $j$-th test of the battery $B$, where $j = 1, \ldots, m$.

**Definition 1.4.** The *local random vector* $\xi(S_{(i,n)}, B)$ of a sequence $S$ of numbers is a vector such that the $j$-th coordinate is the chi-square value resulting from the $j$-th statistical test in $B$ evaluated on the subsequence $S_{(i,n)}$. If there are $m$ tests in the battery, then $\xi(S_{(i,n)}, B)$ is an $m$-tuple.

The local random vector encodes the results of several statistical tests performed on the same $n$ numbers determined by $S_{(i,n)}$.

**Definition 1.5.** The *global random matrix* $\Xi_t(S_{(i,n)}, B)$ of a sequence $S$ of numbers measured by a battery $B$ of $m$ tests is a $t \times m$ matrix. The $j$-th row of $\Xi_t(S_{(i,n)}, B)$ is the local random vector $\xi(S_{(jn,n)}, B)$.

That is, each row of the matrix is the local random vector for consecutive, independent length $n$ subsequences of $S$. To ease our discussion, we will refer to $n$ as the *sample size*, and to $t$ as the *number of local samples*.

The local random vector encodes information about the “random” properties of a length $n$ subsequence, as measured by the $m$ statistical tests. The global random matrix with $t$ local samples is obtained by forming the rows out of the $t$ local random vectors measured over the subsequences

$$S_{(i,n)}, S_{(i+1,2n)}, \ldots, S_{(i+(t-1)n+1,tn)}$$

More explicitly,
Remark 1.6. The global random matrix $\Xi_1(S_{(i,n)}, B)$ is the local random vector $\xi(S_{(i,n)}, B)$.

Our definitions of local random vectors and global random matrices of a sequence do not tell us much by themselves. For example, if we have a collection of purportedly random sequences $\langle S^j \rangle_{j=0}^J$, then calculating the local random vector with constant sample size $n$ for each of them gives us a set

$$\xi(S^0_{(i,n)}, B), \ldots, \xi(S^J_{(i,n)}, B).$$

We now need to put an ordering on this set to say that a sequence is more “random” than another. In fact, this ordering will become our practical definition of local randomness. The motivation for this definition is taken from [Knu98, 47].

Recall that each coordinate of a vector is the chi-square result of a statistical test. Now, each chi-square value corresponds to a percentage, which depends on the number of degrees of freedom. This is a one to one correspondence; the local ordering will consider the percentages instead of the chi-square values themselves.

Given a collection of sequences $\langle S^j \rangle_{j=0}^J$, we will put an ordering $\theta$ on the collection of corresponding local random vectors $\left\{ \left( \xi(S^j_{(i,n)}, B) \right)_{j=0}^J \right\}$ by “penalizing” each vector for having extreme percentages, “close” to 0 or 100. If the percentage is within 1 of 0 or 100, that means that $S^j_{(i,n)}$ does not well represent a random sequence. We wish to order the vectors so that those with no or few outlier percentages appear first and are declared to be more random than the others.

Definition 1.7. An ordering $\theta$ on a collection of local random vectors is given by a mapping of the local random vectors into the non-negative integers. Each coordinate of a vector will map into the set $\{0, 1, 3, 5\}$, and the sum of these coordinate values will give the mapping of the vector. We design the ordering so that the best sequences are represented by local random vectors with small ordering values, 0 being the best. If $c_k$ is a coordinate of a vector, the mapping is given by

$$c_k \rightarrow 0 \text{ if } c_k \in (10, 95)$$
$$c_k \rightarrow 1 \text{ if } c_k \in [5, 10) \cup [90, 95]$$
$$c_k \rightarrow 3 \text{ if } c_k \in [1, 5) \cup (95, 99]$$
$$c_k \rightarrow 5 \text{ if } c_k \in [0, 1) \cup (99, 100]$$

For example, we want to say that the local random vector $(32.03, 87.32, 50.34, 14.33)$ stemming from 4 statistical tests represents a sequence that is more random than one giving the local random vector $(1.02, 20.6, 99.32, 60.68)$. Use this process to implement the ordering:

1. DECLARE SUM=0, MAX=0, PREDICTABILITY
(2) Recall that each coordinate of a local random vector is a chi-square value.
Let \( c_k \) be the percentile identified with the \( k \)-th coordinate of the random vector:
- If \( c_k \) is in the range 0–1 percent or 99–100 percent, then \( \text{SUM} = \text{SUM} + 5 \)
- If \( c_k \) is in the range 1–5 percent or 95–99 percent, then \( \text{SUM} = \text{SUM} + 3 \)
- If \( c_k \) is in the range 5–10 percent or 90–95 percent, then \( \text{SUM} = \text{SUM} + 1 \)

(3) Do (2) for \( k = 1, ..., m \), where each local random vector is an \( m \)-tuple.

(4) Let \( \text{PREDICTABILITY} = \text{SUM} \)

(5) Repeat this process for each local random vector in the collection.

If a particular sequence has a greater value of \text{PREDICTABILITY} than another, then it is less random. We will use the notation
\[
\theta : \xi \left( S_{(i,n)}, B \right) \rightarrow [0, \infty)
\]
to indicate this ordering process for a given sequence.

**Definition 1.8.** A sequence \( S \) is **maximally locally random** if
\[
\theta \left( \xi \left( S_{(i,n)}, B \right) \right) = 0
\]

Notice that, beyond the dependencies of our definition of a local random vector, our notion of randomness also depends on the ordering we place on the vectors. Randomness, amongst other things, lies in the eyes of the beholder. For example, we could change the percentile ranges given in the ordering, the corresponding additions to \( \text{SUM} \), or both. This ordering process is best done by hand. In practice, we will be using a battery of only 5 or 6 statistical tests, making \( \xi \) a 5-tuple or 6-tuple. The decision to define “maximal randomness” as opposed to “randomness” is not an attempt to confuse the reader and clutter the paper: for practical purposes, randomness is not a yes/no question. We were able to give a theoretical definition of a random sequence, but we can never truly know if we have our hands on one.

**Remark 1.9.** Notice that the more tests included in the battery \( B \), the more tests a sequence has to pass perfectly to be considered maximally locally random. If we were curious, we might allow ourselves to try to define “locally random” as follows. Let \( m \) be the number of tests in the battery \( B \). Then a sequence \( S \) is “locally random” if \( \theta \left( \xi \left( S_{(i,n)}, B \right) \right) = 0 \) as \( m \to \infty \). This definition is impractical, but still interesting. Now let
\[
\mathcal{S} = \{ S | \theta \left( \xi \left( S_{(i,n)}, B \right) \right) = 0 \}
\]
it is clear that \( |\mathcal{S}| \) is monotonically decreasing in \( m \). What is the cardinality of \( \mathcal{S} \) as \( m \to \infty \)? That is, how many sequences are there that pass all possible tests? This is left as an exercise to the reader.

A fair question to ask at this point is how sensitive the local random vector is to changes in its arguments: in particular, the sample size \( n \). That is, given a statistical test for a sequence, how does the chi-square value change as \( n \to \infty \)?

Figure 1 shows readings from the frequency test for the sequences Fibonacci and gLCG. Notice that the value does not converge, and furthermore that the values dip in and out of rejection ranges (as defined by our ordering \( \theta \)). For the frequency test, using the convention that \( np > 5 \) and with \( p = 31 \) categories, we get a minimum sample size value of \( n = 155 \). The readings in Figure 1 are well beyond that value. Why is this a problem? Look at the behavior of gLCG in Figure 1. We might
accept this sequence without hesitation at \( n = 1,000,000 \), be worried about it at \( n = 2,000,000 \), but again be happy with it at \( n = 3,000,000 \). What value of \( n \) are we to use? This problem motivates our need for a notion of a global random matrix (as defined above). The graphs of gLCG and Fibonacci in Figure 1 show no signs of converging over \( n = 1,2,\ldots,5000000 \), and we have no reason to believe that the situation improves as \( n \to \infty \). In addition, notice that the chi-square values given at any point of these graphs would constitute a single coordinate of a local random vector of the sequence gLCG (or Fibonacci), and that we may infer that the local random vector does not converge as \( n \to \infty \), because its coordinates do not converge. However, in the absence of a proof, this is just speculation.

Before we move on to developing an ordering for global random matrices, we should observe that local random vectors are not useless. If we are performing an experiment and we know that we are going to use \( n = N \) values of a random sequence, then \( \xi (S_{t, N}, B) \) provides a way of ranking the sequences for that particular experiment. In fact, the rankings we will eventually give from an ordering of global random matrices should only be a starting point. If we know how many numbers we are going to use for an experiment, we should also use the local measurements for that sample size \( n \).

Now, to motivate what will become our definition of an ordering on global random matrices, consider the mechanics of our statistical test. Pick a particular test, and apply the test independently to \( t \) independent subsequences of a sequence \( S \), where \( S \) is a collection of independent samples from \( U[0,1] \) (an ideal random sequence). This gives us \( t \) local samples. In the end, this test relies on a chi-square test with \( \chi \) degrees of freedom. Create a cumulative histogram of the \( t \) chi-square values. Because we assumed \( S \) to be a truly random sequence with elements in \( U[0,1] \), as the number of local samples increases, the plot of this histogram
should coincide with the cumulative distribution function (CDF) of the chi-square distribution with \( \chi \) degrees of freedom.

If we instead applied this method to a sequence that was not a random sampling of \( U[0,1] \), then the plot of the cumulative histogram would deviate drastically from the chi-square CDF. The ordering we wish to define will work by creating a cumulative histogram for a given statistical test on several sequences (from the numbers in a column of a global random matrix) and determining which sequence gives a histogram that best fits the chi-square CDF. Obviously, we must use the same partition for each histogram. We will define a “best fit” as the sequence which minimizes the squares of the difference between each histogram value and the corresponding CDF value. See Appendix A for an example.

However, our global random matrix is defined on a battery of tests: not just one. We want to make sure that our ordering gives equal weight to each test. Unfortunately, each statistical test in \( B \) will use chi-square tests with different degrees of freedom, and so we cannot expect to use the same partition for each test if the partition is based on chi-square values; instead, we will base it on percentiles.

For example, let the inverse of the chi-square CDF at 2% be denoted \( \chi^{-1}(0.02) \). Then our hypothesis is that 2% of the \( t \) local samples from the particular statistical test should be \( \leq \chi^{-1}(0.02) \). Suppose \( p \% \) of the \( t \) actually are \( \leq \) this value. Then \( (p\% - 2\%)^2 \) gives us a measure of the fit of the histogram to the chi-square CDF. See Figure 1 of Appendix A. The histogram used here is referred to as the empirical distribution function in the Kolmogorov-Smirnov test.

This process enables us to give equal weight to the sum of squared differences obtained from tests with different degrees of freedom.

Given a collection of sequences \( \langle S^j \rangle \) for \( j = 0, \ldots, J \), we will put an ordering \( \Theta \) on the collection of corresponding global random matrices (with the same number of local samples \( t \) for each) \( \left\{ ( \Xi_t \left( S^j_{(i,n)}; B \right) ) \right\}_{j=0}^J \) by sorting the columns of each matrix (each of which represents runs of a particular test over independent, consecutive sequences of \( S^j \)) into cumulative histograms, fitting the histogram data to ideal chi-square CDF’s, and assigning a value PREDICTABILITY to each matrix that corresponds to the summed squared difference between the points its the \( m \) histograms and the chi-square CDF’s.

If a particular sequence has a greater value of PREDICTABILITY than another, then it is less random. We will use the notation

\[
\Theta : \Xi_t \left( S^j_{(i,n)}; B \right) \longrightarrow [0, \infty)
\]

to indicate this ordering process for a given sequence.

**Definition 1.10.** An ordering \( \Theta \) on a collection of global random matrices is given by a mapping of the matrices into the non-negative real line. Each column of a matrix will map into the non-negative real line, and the sum of these column values will give the mapping of the matrix. We design the ordering so that the best sequences are represented by global random matrices with small ordering values, \( 0 \) being the best. If \( C_j \) is a column of a matrix, the mapping is given by arranging its entries \( c_{i,j} \) into a cumulative percentage histogram with a particular partition, and then summing the square of the differences between the cumulative percentage histogram and a chi-square CDF, whose degrees of freedom \( d \) is specified by the statistical test which the column represents. Let \( X \) be the set of chi-square values
with \(d\) degrees of freedom corresponding to the set of percentages \{.02, .04, ..., .98\}. Then \(X\) is to be the partition of the cumulative percentage histogram.

The implementation of the ordering is given by the following algorithm:

1. DECLARE \(\text{SUM} = 0\), \(\text{PREDICTABILITY}\)
2. Recall that each of the \(m\) columns of \(\Xi\) is the chi-square data for a given test, taken over independent, subsequent, length \(n\) subsequences of \(S\).
3. (a) For each statistical test (each column of a matrix), reference the degrees of freedom \(\chi\) used for that test.
4. (b) Using Mathematica or some other software, calculate the inverse value of the CDF of the chi-square distribution with \(\chi\) degrees of freedom for values .02, .04, ..., .98
5. (c) Use the inverse values obtained in (2) as the partition for your cumulative percentage histogram for the test. Use this partition for measuring every sequence against this particular statistical test. (That is, use this same partition for the other matrices to be measured).
6. (d) Sum the squares of the difference between the values .02, .04, ..., .98 and those given by your histogram of a particular sequence at the points just determined.
7. (e) Repeat (a) - (d) for each of the \(m\) columns of the matrix.
8. We now have \(m\) numbers (since there were \(m\) tests). Let the sums be denoted \(A_1, ..., A_m\).
9. (5) \(\text{SUM} = \sum_{i=1}^{m} A_i\)
10. (6) \(\text{PREDICTABILITY} = \text{SUM}\)

Again, see Appendix A for an example of this process.

**Definition 1.11.** A sequence \(S\) is maximally globally random if

\[
\Theta \left( \Xi_t \left( S_{(i,n)}, B \right) \right) = 0
\]

This definition makes sense. Why? \(\text{PREDICTABILITY}\) can only be 0 if all of the \(A_i\) are 0, meaning the histograms match the CDF’s. We cannot define a “minimally globally random” sequence because we map the sequences into \([0, \infty)\).

### 2. Frequency Test

The frequency (equidistribution) test is the weakest of the tests to be discussed. Nearly all generators pass it. [Knu98, 74]. However, the test is easy to understand and implement, so it is worth a moment to include it. We will use \(d = 31\), giving us 30 degrees of freedom. This test ensures that the numbers reasonably represent a uniform sampling of the unit interval by the following process, extracted from [Knu98, 61]

1. For each integer \(r\), \(0 \leq r < 31\), let \(\text{COUNT}[r]\) be the number of times that \(Y_j = r\) for \(0 \leq j < n\).
2. Apply the chi-square test using 31 categories and probability \(p_\ast = 1/31\) for each category.
This test checks the *granularity* of the sequence. The granularity refers to the distance between ordered members of a discrete set. For example, the representation of $[0, 100] \in \mathbb{R}$ given by the set $\{0, 1, 2, \ldots, 99, 100\}$ is more granular than that given by the set $\{0, 0.5, 1, \ldots, 99.5, 100\}$. In an intuitive sense, the greater the granularity of a sequence, the “farther” it is from being dense in $\mathbb{R}$. If a sequence fails the frequency test (e.g. if $p < 5\%$ or $p > 95\%$), then it does not faithfully represent a uniform distribution. For the other tests to be discussed, we might allow a sequence to come close to failing a test and still decide it to be worthwhile, but we should not be so forgiving with the frequency test. Such a large class of sequences passes it that it is not useful in picking *good* generators, but it is useful in weeding out *bad* generators.

See Appendix B for the C++ implementation.

3. Serial Test

Just as we expect to find an even distribution of the numbers $1, 2, \ldots, d - 1$ in our scaled sequence, we also expect to find an even distribution of the pairs $(q, r)$ for $0 \leq q, r \leq (d - 1)$. We carry out the test by counting the number of times that each pair $(q, r)$ occurs, for all $0 \leq q, r < d$. We use only non-overlapping pairs of our sequence. The chi-square test is applied to the $k = d^2$ categories with probability $1/d^2$ in each category. Choose $d$ such that $n \geq 5d^2$. We will use $d = 10$. This process is developed in [Knu98, 62], and our program can be found in Appendix 2.

4. Poker Test

As the name implies, this test appeals to concepts from poker. For example, if a person is handed a stack of many cards, how could they tell if the stack could work as a fair poker deck? (More specifically, how could they tell if the stack was taken randomly from collections of poker decks?) Either from work in an introductory probability class or from casino experience, the reader should recall that each type of poker hand has a certain probability (i.e. $P($full house$) = p_1$, $P($two pairs$) = p_2$, etc.) The person can then take blind samples of 5 card hands, enumerate the types of hands, and then use a chi-square analysis against the theoretical probabilities to determine if the stack represented a fair poker deck.

We will do something similar, but simpler, by counting the number of distinct values in each 5-tuple.

- 5 values = all different;
- 4 values = one pair;
- 3 values = two pairs, or three of a kind;
- 2 values = full house, or four of a kind;
- 1 values = five of a kind.

To calculate the probability of each category, we need the concept of a Stirling number from combinatorics; that is, a number $\left\{\frac{k}{r}\right\}$ gives the number of ways to partition a set of $k$ elements into exactly $r$ parts. This number is given by:

$$\left\{\frac{k}{r}\right\} = \sum_{j=1}^{r} (-1)^{r-j} \frac{j^{k-1}}{(j-1)! (r-j)!}$$
To calculate $p_r$, we must count how many of the $d^k$ $k$-tuples between 0 and $d - 1$ have exactly $r$ different elements, and divide the total by $d^k$. The numerator is given by the number of permutations $P(d, r) = \frac{d!}{(d-r)!}$, so the probability that there are $r$ different numbers in any $k$-tuple is

$$p_r = \frac{d(d-1)\ldots(d-r+1)\binom{k}{r}}{d^k}.$$  

The Stirling numbers grow rapidly as $k$ increases, and the calculation of $\binom{k}{r}$ for $k = 10$ already causes overflow if the calculation is not done properly. Even if one spends time looking for clever techniques, $\binom{17}{7}$ is too large to be calculated in a reasonable time.

For $k = 5$ [Knuth73, 66],

$$\left(\binom{5}{r}\right)_{r=1}^5 = \{1, 15, 25, 10, 1\}$$

**Remark 4.1.** Since $p_1 = .0001$ and $p_2 = 0.0135$ are small, we will count these categories of low probability together before applying the chi-square test. Note that our test will then have 3 degrees of freedom, not 4.

Here is our procedure for testing $n$ groups of five successive integers: Load a sequence of 5 numbers into an array, and count the number of distinct values. Add 1 to the appropriate counter. (There will be 4 counters: 5 values, 4 values, 3 values, and 1 or 2 values). Load the next 5 integers into the array, and repeat. At the end, calculate the percentage of the $n$ total repetitions corresponding to each counter, and run a chi-square analysis using the expected probabilities given by Equation 4.1.

### 5. Runs Test

The **runs test** measures the length of increasing ‘runs’ in a sequence; that is, the length $n$ of subsequences such that $a_1 < a_2 < \ldots < a_n$. The $a_i$ are supposed to be distinct to prevent the sequence from having equal, adjacent values. As long as we use the standard decimal output of our generators, in the interval [0, 1), then the event $a_1 = a_{i+1}$ should happen infrequently enough that we do not worry about it. If two adjacent entries are equal, we will use the strict inequality $<$ to interpret this as a break in the run. Consider as an example the sequence 1,2,9,8,5,3,6,7,0,4.

Separating the runs with a bar, we have

| 1 2 9 | 8 | 5 | 3 6 7 | 0 4 |

This gives a runs of length 3, 1, 1, 3, and 2.

We cannot immediately apply a chi-square test, because adjacent runs are not independent, since long and short runs tend to alternate. A solution: discard each element of the sequence which follows the conclusion of a run. In the above sequence, we would discard 8, 3, and 0. We may then proceed with a chi-square test. [Knuth98, 66,67].

We will use 6 categories, resulting in 5 degrees of freedom. Because of their low probability, we group all runs of length $\geq 6$. To perform the test, set a counter to 1, select an initial element and compare it with the subsequent number. If the second number is larger, then increase a counter by 1, and test again. If the second number is not larger, then record it by adding one to the appropriate bin (the bin
6. LOCAL RANDOMNESS OF OUR GENERATORS

Table 1. Frequency Test. \( n = 4500; d = 31 \)

<table>
<thead>
<tr>
<th>GENERATOR</th>
<th>CHI-SQUARE (( k = 30 ))</th>
<th>PERCENTILE</th>
<th>PREDICTABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIB</td>
<td>48.6369</td>
<td>98.29</td>
<td>3</td>
</tr>
<tr>
<td>( \pi )</td>
<td>35.0796</td>
<td>76.02</td>
<td>0</td>
</tr>
<tr>
<td>rand()</td>
<td>22.1836</td>
<td>15.27</td>
<td>0</td>
</tr>
<tr>
<td>gMCG</td>
<td>34.9831</td>
<td>75.67</td>
<td>0</td>
</tr>
<tr>
<td>LGFIB</td>
<td>20.42</td>
<td>9.49</td>
<td>1</td>
</tr>
<tr>
<td>oLCG</td>
<td>32.6684</td>
<td>66.29</td>
<td>0</td>
</tr>
<tr>
<td>NR</td>
<td>75.6</td>
<td>100.00</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2. Serial Test. \( n = 4500 \) (2250 pairs); \( d = 10 \)

<table>
<thead>
<tr>
<th>GENERATOR</th>
<th>CHI-SQUARE (( k = 99 ))</th>
<th>PERCENTILE</th>
<th>PREDICTABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIB</td>
<td>105.111</td>
<td>68.18308864</td>
<td>0</td>
</tr>
<tr>
<td>( \pi )</td>
<td>131.511</td>
<td>98.39571225</td>
<td>3</td>
</tr>
<tr>
<td>rand()</td>
<td>80.6667</td>
<td>8.929136255</td>
<td>1</td>
</tr>
<tr>
<td>gMCG</td>
<td>93.4667</td>
<td>36.19750246</td>
<td>0</td>
</tr>
<tr>
<td>LGFIB</td>
<td>83.9556</td>
<td>13.99237473</td>
<td>0</td>
</tr>
<tr>
<td>oLCG</td>
<td>106.711</td>
<td>71.95427713</td>
<td>0</td>
</tr>
<tr>
<td>NR</td>
<td>20250</td>
<td>100</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3. Poker Test. \( n = 4500 \) (900 hands of 5); \( d = 10 \)

<table>
<thead>
<tr>
<th>GENERATOR</th>
<th>CHI-SQUARE (( k = 3 ))</th>
<th>PERCENTILE</th>
<th>PREDICTABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIB</td>
<td>73.9258</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>( \pi )</td>
<td>4.76125</td>
<td>80.98620681</td>
<td>0</td>
</tr>
<tr>
<td>rand()</td>
<td>0.538818</td>
<td>8.971585008</td>
<td>3</td>
</tr>
<tr>
<td>gMCG</td>
<td>4.04966</td>
<td>74.38485843</td>
<td>0</td>
</tr>
<tr>
<td>LGFIB</td>
<td>0.616584</td>
<td>10.73744282</td>
<td>0</td>
</tr>
<tr>
<td>oLCG</td>
<td>0.903621</td>
<td>17.54461081</td>
<td>0</td>
</tr>
<tr>
<td>NR</td>
<td>65276.5</td>
<td>100</td>
<td>5</td>
</tr>
</tbody>
</table>

that counts the number of runs of length given by the counter and reset the counter to 1. Discard the next number in the sequence, and begin the process again. See Appendix B for the computer implementation.

The corresponding probabilities for a run of length \( r \) are, as given by [Knu98, 564]

\[
p_r = \frac{1}{r!} - \frac{1}{(r + 1)!}, \quad r < t
\]

\[
p_t = \frac{1}{t!} \quad \text{for runs of length } \geq t
\]

6. Local Randomness of Our Generators

The sequences chosen at the end of Chapter 2 and coded in Appendix B have been evaluated with the tests given in this chapter, and the local ordering process described just after Definition 1.7 is executed. The first four tables given here report the results, and the final local ordering combines them.
7. Global Randomness

Again, the sequences chosen at the end of Chapter 2 and coded in Appendix B have been evaluated with the tests given in this chapter. However, this time the tests were run over 1,100 consecutive subsequences of each sequence, creating global random matrices. Finally, the global ordering process described just after Definition 1.10 is executed. The first four tables given here report the results, and the final global ordering combines them. This final global ordering is to be taken more seriously than the final local ordering.

Table 6. Global Frequency Test. n=4500; d=31; t=1,100

<table>
<thead>
<tr>
<th>GENERATOR</th>
<th>PREDICTABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>gMCG</td>
<td>0.001179339</td>
</tr>
<tr>
<td>π</td>
<td>0.003357851</td>
</tr>
<tr>
<td>oLCG</td>
<td>0.004735537</td>
</tr>
<tr>
<td>rand()</td>
<td>0.0091</td>
</tr>
<tr>
<td>LGFIB</td>
<td>0.010271074</td>
</tr>
<tr>
<td>FIB</td>
<td>0.024847107</td>
</tr>
<tr>
<td>NR</td>
<td>16.17</td>
</tr>
</tbody>
</table>
Table 7. Global Serial Test. \( n=4500; d=10; t=1,100 \)

<table>
<thead>
<tr>
<th>GENERATOR</th>
<th>PREDICTABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>gMCG</td>
<td>0.00291157</td>
</tr>
<tr>
<td>rand()</td>
<td>0.006124793</td>
</tr>
<tr>
<td>( \pi )</td>
<td>0.008973554</td>
</tr>
<tr>
<td>LGFIB</td>
<td>0.010949587</td>
</tr>
<tr>
<td>oLCG</td>
<td>0.016014876</td>
</tr>
<tr>
<td>FIB</td>
<td>0.146438843</td>
</tr>
<tr>
<td>NR</td>
<td>16.17</td>
</tr>
</tbody>
</table>

Table 8. Global Poker Test. \( n=4500 \) (900 hands of 5); \( d=10; t=1,100 \)

<table>
<thead>
<tr>
<th>GENERATOR</th>
<th>PREDICTABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>gMCG</td>
<td>0.004826446</td>
</tr>
<tr>
<td>oLCG</td>
<td>0.005281818</td>
</tr>
<tr>
<td>( \pi )</td>
<td>0.007698347</td>
</tr>
<tr>
<td>LGFIB</td>
<td>0.012645455</td>
</tr>
<tr>
<td>rand()</td>
<td>0.020880165</td>
</tr>
<tr>
<td>FIB</td>
<td>16.08551157</td>
</tr>
<tr>
<td>NR</td>
<td>16.17</td>
</tr>
</tbody>
</table>

Table 9. Global Runs Test. \( n=4500; t=1,100 \)

<table>
<thead>
<tr>
<th>GENERATOR</th>
<th>PREDICTABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi )</td>
<td>0.006999804</td>
</tr>
<tr>
<td>gMCG</td>
<td>0.008659504</td>
</tr>
<tr>
<td>LGFIB</td>
<td>0.009000826</td>
</tr>
<tr>
<td>oLCG</td>
<td>0.012707483</td>
</tr>
<tr>
<td>rand()</td>
<td>0.019067769</td>
</tr>
<tr>
<td>FIB</td>
<td>16.17</td>
</tr>
<tr>
<td>NR</td>
<td>16.17</td>
</tr>
</tbody>
</table>

Table 10. **FINAL GLOBAL ORDERING**

<table>
<thead>
<tr>
<th>RANK</th>
<th>GENERATOR</th>
<th>PREDICTABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>gMCG</td>
<td>0.01757686</td>
</tr>
<tr>
<td>2</td>
<td>( \pi )</td>
<td>0.026989256</td>
</tr>
<tr>
<td>3</td>
<td>oLCG</td>
<td>0.038739669</td>
</tr>
<tr>
<td>4</td>
<td>LGFIB</td>
<td>0.042866942</td>
</tr>
<tr>
<td>5</td>
<td>rand()</td>
<td>0.055172727</td>
</tr>
<tr>
<td>6</td>
<td>FIB</td>
<td>32.42679752</td>
</tr>
<tr>
<td>7</td>
<td>NR</td>
<td>64.68</td>
</tr>
</tbody>
</table>

7.1. **Conclusion.** So it turns out that the MCG called gMCG defined at the end of Chapter 2 is the most random of our sequences, when considered over its first 4,950,000 values. It is reassuring to see that \( \pi \) did well, that the sequence called
“Not Random” came in last, and that the Fibonacci sequence (which every piece of literature claims to be bad) came in second to last. Notice that the standard C rand() function turned out to be the weakest of the serious contenders. And even though the lagged Fibonacci sequence only did so-so, it did well enough to use in case a simulation needs a sequence with a huge period. Finally, note that even though the LCG called oLCG which we constructed with arbitrary choices of parameters (within given requirements) did well in this trial, it may have only been a lucky choice. It would be worthwhile to extend this project by using this same ranking process on many sequences generated by different parameters in oLCG to find the optimal values.
7. GLOBAL RANDOMNESS

Figure 2. Global Frequency Analysis

Figure 3. Global Frequency Analysis
Figure 4. Global Serial Analysis

Figure 5. Global Serial Analysis
Figure 6. Global Poker Analysis

Figure 7. Global Poker Analysis
Figure 8. Global Runs Analysis

Figure 9. Global Runs Analysis
Simulation is a dangerous game, and great caution is required in interpreting the results. There are two major reasons for this. First, a computer simulation is limited by the degree to which its so-called ‘pseudo-random number generator’ may be trusted. It has been said for example that the summon-according-to-birthday principle of conscription to the United States armed forces may have been marred by a pseudo-random generator with a bias for some numbers over others. Secondly, in estimating a given quantity, one may in some circumstances have little or no idea how many repetitions are necessary in order to achieve an estimate within a specified accuracy. . . . These techniques were named in honour of Monte Carlo by Metropolis, von Neumann, and Ulam, while they were involved in the process of building atomic bombs at Los Alamos in the 1940s.

-Geoffrey Grimmett and David Stirzaker [GS01, 43]

1. Estimation of $\pi$

We will put our newly created sequences to work with a simple but entertaining example of a Monte Carlo method. We begin: consider the intersection of the square $[0,1] \times [0,1]$ and the unit disk $D^2$, centered at $(0,0)$. See Figure 2. The sector of the disk will have area $\pi/4$. The area of the square is 1, so the percentage of the area of the square covered by the disk is $\pi/4$. If we were to throw darts blindly at the square, we would hit the circle with probability $\pi/4$. From this, we conclude that if we throw a dart $N$ times and hit the disk $h$ times, we may approximate $\pi$ by the value $4h/N$. We will simulate the throwing of darts with a random sequence. If we take a pair of numbers $(x_1, x_2)$ from the sequence, then we have a random point in $[0,1] \times [0,1]$. The point $(x_1, x_2)$ is in the circle and we count a hit if and only if $(x_1)^2 + (x_2)^2 < 1$. Repeat this with non-overlapping pairs from the random sequence.

This process highlights the importance of having a low autolag correlation (see Chapter 5). If it is too high, this process might not cover $[0,1] \times [0,1]$ well, and our dart tosses may fall into lines. This may be remedied by using only every other number from the sequence. However, this will decrease the number of points a sequence can generate. The program used here, given in Appendix B section B uses pairs of the form $(x_{2n}, x_{2n+1})$ for $n = 0, 1, 2, \ldots$ Table 1 gives the results for this experiment.

Remark 1.1. The author is aware of the irony of estimating $\pi$ by digits taken from $\pi$.  

- 29
30 4. MONTE CARLO METHODS

**Figure 1.** Approximation of \( \pi \)

**Figure 2.** Estimation of \( \pi \) with “darts”

**Table 1.** Results for estimation of \( \pi \)

<table>
<thead>
<tr>
<th>Tosses</th>
<th>FIB</th>
<th>( \pi )</th>
<th>( \text{rand()} )</th>
<th>gMCG</th>
<th>lagFIB</th>
<th>NR</th>
<th>oLCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3.12</td>
<td>3.28</td>
<td>3.2</td>
<td>3.32</td>
<td>3.28</td>
<td>2.88</td>
<td>3.12</td>
</tr>
<tr>
<td>1,000</td>
<td>3.192</td>
<td>3.22</td>
<td>3.112</td>
<td>3.148</td>
<td>3.112</td>
<td>2.88</td>
<td>3.192</td>
</tr>
<tr>
<td>10,000</td>
<td>3.15</td>
<td>3.196</td>
<td>3.114</td>
<td>3.1344</td>
<td>3.1096</td>
<td>2.88</td>
<td>3.1232</td>
</tr>
<tr>
<td>100,000</td>
<td>3.14264</td>
<td>3.14616</td>
<td>3.13756</td>
<td>3.1474</td>
<td>3.13412</td>
<td>2.88</td>
<td>3.14164</td>
</tr>
<tr>
<td>1,000,000</td>
<td>3.143</td>
<td>3.1424</td>
<td>3.13928</td>
<td>3.14427</td>
<td>3.14154</td>
<td>2.88</td>
<td>3.139</td>
</tr>
</tbody>
</table>
We should note that this should not be considered an acceptable approximation of $\pi$ because of its slow convergence. For every extra digit we wish to approximate, we must perform 10 more tosses. This can also be seen in Table 1.

2. Estimation of $e$

Our second example of a Monte Carlo method will use a bit more probability to provide an approximation of $e$.

We have at hand sequences of random numbers from the unit interval $U(0, 1)$. Choose your favorite random sequence, and take the numbers in order from the beginning until their total exceeds 1. Since the sequence is random, this represents random samples taken from $U(0, 1)$. What is the expected number of picks? [Der04, 366-367] and [Wei08] offer some insight.

Let $x_i$ denote the $i$-th pick. Since the samples are from a sequence which we claim is random, the $x_i$ are independent random variables with uniform distributions on $U(0, 1)$. The joint density function for the first $n$ random variables is $f_n = 1$ on the unit $n$-cube and 0 elsewhere. Let $N$ denote the random variable such that

$$ N = \min \{ j : s_j = x_1 + \ldots + x_j > 1 \} $$

Let $dV_n$ be the volume element $dx_1 \cdots dx_n$. Then

$$ \mathbb{P}(N \leq n) = \mathbb{P}(s_n > 1) = 1 - \mathbb{P}(s_n \leq 1) = 1 - \int_{s_n \leq 1} f_n dV_n = 1 - \text{vol}(C_n) $$

**Figure 3.** Approximation of $e$
Figure 4. The plane $x + y + z = 1$

where $C_n$ is the ‘corner’ of the $n$-cube. See Figure 4. We will verify in a moment that $\text{vol}(C_n) = \frac{1}{n!}$. Given this,

$$P(N = n) = P(N \leq n) - P(N \leq n - 1) = \left(1 - \frac{1}{n}\right) - \left(1 - \frac{1}{(n-1)!}\right) = \frac{n-1}{n!}$$

and the expected value of $N$ is

$$E(N) := \sum_{n=1}^{\infty} nP(N = n) = \sum_{n=2}^{\infty} \frac{n-1}{n!} = \sum_{n=2}^{\infty} \frac{1}{(n-2)!} = \sum_{n=0}^{\infty} \frac{1}{n!} = e$$

With the following lemma, we are done.

**Lemma 2.1** (Volume of a Corner). Let $C_n$ be the corner of the unit $n$-cube defined by $s_n = x_1 + \cdots + x_n \leq 1$ where $x_1 \geq 0, \ldots, x_n \geq 0$. Then $\text{vol}(C_n) = \frac{1}{n!}$

**Proof.** For any positive integers $i$ and $j$,

$$\int_{C_{j}} (1-s_j)^i dV_j = \int_{C_{j-1}} \left[ \int_{0}^{1-s_{j-1}} (1-s_{j-1}-s_j)^i dx_j \right] dV_{j-1} = \frac{1}{i+1} \int_{C_{j-1}} (1-s_{j-1})^{i+1} dV_{j-1}$$

Thus,

$$\text{vol}(C_n) = \int_{C_{n}} 1dV_n = \int_{C_{n-1}} \left[ \int_{0}^{1-s_{n-1}} 1dx_n \right] dV_{n-1} = \int_{C_{n-1}} (1-s_{n-1})dV_{n-1}$$

$$= \frac{1}{2} \int_{C_{n-2}} (1-s_{n-2})^2 dV_{n-2} = \cdots = \frac{1}{(n-1)!} \int_{0}^{1} (1-x_1)^{n-1} dx_1 = \frac{1}{n!}$$

The simulation, found in Appendix B, Section B, sums numbers in order from a given file of random numbers and counts the number until the total exceeds one. The sum then resets and the program continues beginning with the next number in the file.
Table 2. Results for estimation of $e$

<table>
<thead>
<tr>
<th>Numbers Used</th>
<th>FIB</th>
<th>π</th>
<th>rand()</th>
<th>gMCG</th>
<th>lagFIB</th>
<th>NR</th>
<th>oLCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4.7619</td>
<td>2.8</td>
<td>2.82857</td>
<td>2.63158</td>
<td>2.85294</td>
<td>2.85714</td>
<td>2.77778</td>
</tr>
<tr>
<td>1,000</td>
<td>2.92962</td>
<td>2.6973</td>
<td>2.71196</td>
<td>2.7027</td>
<td>2.72207</td>
<td>2.85714</td>
<td>2.74725</td>
</tr>
<tr>
<td>10,000</td>
<td>2.84576</td>
<td>2.70536</td>
<td>2.69879</td>
<td>2.72702</td>
<td>2.69924</td>
<td>2.85714</td>
<td>2.72601</td>
</tr>
<tr>
<td>100,000</td>
<td>2.82941</td>
<td>2.72148</td>
<td>2.71061</td>
<td>2.71943</td>
<td>2.71773</td>
<td>2.85714</td>
<td>2.72084</td>
</tr>
<tr>
<td>1,000,000</td>
<td>2.82717</td>
<td>2.71678</td>
<td>2.71753</td>
<td>2.718</td>
<td>2.71868</td>
<td>2.85714</td>
<td>2.71673</td>
</tr>
<tr>
<td>5,000,000</td>
<td>2.82796</td>
<td>2.71908</td>
<td>2.71772</td>
<td>2.7193</td>
<td>2.71813</td>
<td>2.85714</td>
<td>2.71735</td>
</tr>
</tbody>
</table>

3. Monte Carlo Integration

Random numbers are often used to approximate high dimensional integrals that cannot be solved easily in closed form. For simplicity, we will give an example of integration in one variable. It turns out that this so called Monte Carlo integration is inefficient for only a few dimensions when compared with other methods, but is extremely effective for high dimensions. For example, given $N$ subdivisions of a $d$ dimensional space, the midpoint-method has error $O(N^{-1/d})$, while the Monte Carlo method has error $O(N^{-1/2})$ for $N$ samples, independent of dimension [Fit29].

Let’s consider a method developed in [Ric07, 179]. Suppose we wish to estimate

$$I(f) = \int_0^1 f(x)dx$$

Use our generators to represent independent uniform random variables $X_1, ..., X_n$ on $[0, 1]$, and calculate

$$\hat{I}(f) = \frac{1}{n} \sum_{i=1}^{n} f(X_i)$$

Notice that the formula for $\hat{I}$ is the same as a Riemann sum, except that the $X_i$ are chosen randomly. On one hand, we could use the theory of integration to tell us that $\hat{I}$ would converge to the integral of $f$ on $[0, 1]$ if we were to take a finer and finer uniform partition of $[0, 1]$. We achieve the result with even less effort, however, with a result from probability. As $n \to \infty$, this should approach $E[f(x)]$ by the law of large numbers, and

$$E[f(x)] = \int_0^1 f(x) \cdot 1dx = \int_0^1 f(x)dx = I(f)$$

Now suppose $f(x)$ is the p.d.f. for the standard normal distribution. Then we wish to approximate

$$I(f) = \frac{1}{\sqrt{2\pi}} \int_0^1 e^{-x^2/2}dx$$

Referencing a table for the standard normal distribution in any statistics book will tell us that this value is .3413. We approximate the integral with $n = 100, 1000, 1000000$ numbers from our generators by the equation

$$\hat{I}(f) = \frac{1}{n} \left( \frac{1}{\sqrt{2\pi}} \sum_{i=1}^{n} e^{-X_i^2/2} \right)$$

The results of this estimation are in Table 3.
Table 3. Results for estimation of $I(f)$

<table>
<thead>
<tr>
<th>Numbers Used</th>
<th>FIB</th>
<th>$\pi$</th>
<th>rand()</th>
<th>gMCG</th>
<th>lagFIB</th>
<th>NR</th>
<th>oLCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>.3662</td>
<td>.3473</td>
<td>.3422</td>
<td>.3399</td>
<td>.3499</td>
<td>.3421</td>
<td>.3419</td>
</tr>
<tr>
<td>1,000</td>
<td>.3422</td>
<td>.3415</td>
<td>.3412</td>
<td>.3411</td>
<td>.3426</td>
<td>.3421</td>
<td>.3414</td>
</tr>
<tr>
<td>1,000,000</td>
<td>.3414</td>
<td>.3413</td>
<td>.3413</td>
<td>.3414</td>
<td>.3413</td>
<td>.3421</td>
<td>.3412</td>
</tr>
</tbody>
</table>
Concluding Remarks

We finish by considering further ideas that could be explored in the subject. Two such concepts are feedback shift register generators, and auto-lag correlation statistical tests. The shift register generators operate using bitwise operations on a seeded table of random binary numbers. The auto-lag correlation refers to the statistical correlation between consecutive numbers in a sequence (or likewise between every second, third, ..., number). A sequence with high lag-1 correlation will fall into lines in the plane when each number is plotted with its successor.

The auto-lag correlation test has been used to observe flaws in several popular generators. See [Gen03, 14-20] for a discussion of the test. As a quick example, examine the lag-1 plot of $x_i \equiv 3x_{i-1} \mod 31$, Figure 1. The auto-lag correlation turns out to be about 0.29, and the graph shows the correlation. We would rather have the points scattered throughout the graph to cover it better.

Additionally, it is possible to mix output from several generators, and to shuffling generator output. These operations could be used as “add-ons” to our other generators, and we could use the theory we have developed to test the effectiveness of the modifications.

It would also be worthwhile to examine the relationship between the CDF matching process we developed with global random matrices and the Kolmogorov-Smirnov test. If we had used the KS test instead, would we have obtained the same final ranking? If not, how different would the rankings have been? Our method was developed independently of the KS test, but closely parallels Knuth’s

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{$x_i \equiv 3x_{i-1} \mod 31$}
\end{figure}
treatment of the empirical distribution function of the KS test. In particular, Figure 4 of [Knu98, 49] uses the same method of partitioning the domain of the histogram according to percentages instead of particular chi-square values. (And as an aside, the author of this paper at first disliked the notation used by Knuth in Figure 4 of [Knu98, 49], and wrote him asking if an axis was mislabeled. Dr. Knuth responded that his notation was correct, but admittedly unclear, and included with his letter a check for $0.32.)
APPENDIX A

Ranking Global Random Matrices

The following are the instructions to perform on a single statistical test.

1. The test has a set number of degrees of freedom. For our example, we will use 3 degrees of freedom and the poker test. This is fixed.
2. Using Excel (you may use any number of programs), the function CHIINV(probability, degrees of freedom) returns the chi-square value $\chi^2$ such that $P(X \leq \chi^2) = probability$, where $X$ is any of the chi-square values obtained from the test. (To be honest, the Excel function returns $1 - P(X \leq \chi^2)$, the right-tail of the distribution, but I wish to use left-tails.) The result:

<table>
<thead>
<tr>
<th>Probability</th>
<th>$\chi^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2%</td>
<td>0.184831819</td>
</tr>
<tr>
<td>4%</td>
<td>0.300151419</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>98%</td>
<td>9.837409286</td>
</tr>
</tbody>
</table>

3. Run the statistical test on a sequence $S$ of numbers. For example, we will work with the sequence produced by the gMCG generator. Specifically, we will compute

$$\Xi_{1000} \left( gMCG_{(1,2000)}, B \right)$$

where $B$ is only the poker test. In other words, we will run the poker test on the first 2000 numbers generated by our gMCG generator. This is the first sample. Then, we will run the poker test on the next 2000 numbers of the sequence. This is the second sample. Repeat this until we have 1000 samples. (Thus we will use $2000 \times 1000 = 2,000,000$ numbers from the sequence. The first 10 of the 1000 chi-square values obtained are
(4) Using software, in our case Excel, organize these 1000 chi-values into bins which are partitioned by the $\text{CHIINV}$ function. That is, the first bin contains all values from 0 to 0.184831819, the second contains values from 0 to 0.300151419, and so on. (Yes, there is overlap. Remember that this is a cumulative histogram.)

(5) When we create the histogram, we want it to be a cumulative percentile histogram. So if there are 100 values in a bin, we divide by the total number of samples to get $\frac{100}{1000} = .1$. Here are the values for the first five bins of the histogram:

<table>
<thead>
<tr>
<th>Sample #</th>
<th>$\chi^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.44956</td>
</tr>
<tr>
<td>2</td>
<td>1.3849</td>
</tr>
<tr>
<td>3</td>
<td>1.8821</td>
</tr>
<tr>
<td>4</td>
<td>1.23609</td>
</tr>
<tr>
<td>5</td>
<td>0.566157</td>
</tr>
<tr>
<td>6</td>
<td>9.58302</td>
</tr>
<tr>
<td>7</td>
<td>0.637644</td>
</tr>
<tr>
<td>8</td>
<td>1.69514</td>
</tr>
<tr>
<td>9</td>
<td>2.03752</td>
</tr>
<tr>
<td>10</td>
<td>2.79614</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The target percentage is the percentage corresponding to the chi-square distribution at the bin value. That is how we obtained the bin values.

(6) Sum the squared differences, and call the sum the PREDICTABILITY. This sum turns out to be 0.012107

(7) If there were more sequences, calculate the PREDICTABILITY for each of them, so that they can be compared.

(8) Glance at Figure 1 to see how close the histogram values are to the chi-square distribution.

(9) Sort the sequences from least to greatest by their values of PREDICTABILITY. The sequences at the beginning of the list (with the least value PREDICTABILITY) are the most random, relative to that statistical test.

(10) This example ends here, but if there were more statistical tests in the battery $B$ then we would go through the same procedure for each test, and at the end add the PREDICTABILITY value for every test for each generator, and sort the list according to this grand total.
Figure 1. Chi-square with 3 degrees of freedom vs. global analysis of gMCG.
APPENDIX B

C++ Programs

All of the generators given here write their output to files. For example, a file of 5 million numbers from gMCC takes 48 MB. The statistics programs work by reading from user-specified files. Dr. Snow has pointed out that these programs work much more efficiently when the output of the generators is linked directly into the statistical tests. This method uses a constant, small amount of system memory and takes far less time. However, I used the less efficient method because 1) it was what I was most familiar with and 2) I wanted to be able to apply the statistical tests to files of numbers from other sources (e.g. my file containing digits of π).

thesis.h. The following header should be included in each of programs executing the code included in the appendix.

```cpp
#ifndef THEESIS_H
#define THEESIS_H
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <fstream>
using namespace std;
#endif

LCG.cpp.
/* This is the good Multiplicative Congruential Generator */
The output can be found in the file specified by file_name
The parameters for this generator were recommended by Knuth */
#include "thesis.h"
void LCG(char file_name[1000], unsigned int howmany,
         unsigned int modulus, unsigned int seed, unsigned int multiplier, unsigned int increment)
{
    unsigned int counter;
    unsigned int x[2];
    fstream NUMBERS;
    unsigned int y=seed;
    unsigned int q=unsigned int(modulus/multiplier);
    NUMBERS.open(file_name, fstream::in | fstream::out | fstream::app);
    for (counter=0; counter<howmany; counter++){
        /* This next part looks funny. It is the trick used
to make sure we don’t have overflow. See Knuth, or reference in my thesis */
        /*
            x[0]=multiplier*(y%q);
            x[1]=modulus%multiplier+y/((s+1)/q);
            if(x[0]>=x[1]){
                y=x[0]-x[1];
            } else if(x[0]<x[1]){
                y=modulus-x[1];
        */
```
B. C++ PROGRAMS

```cpp
#include "thesis.h"

int main() {
    char filename[100] = "MSQUARE.txt";
    unsigned int howmany; // number of numbers to generate
    unsigned int seed;
    unsigned int counter;
double alpha, beta;
unsigned int y, x;
int digits, c;
    cout << "MiddleSquare GENERATOR" << endl;

    cout << "Enter the Modulus to use default or input new parameters." << endl;
    cin >> modulus;
    if (modulus > pow((double) modulus, 5)) {
        cout << "Invalid Modulus" << endl;
        return (0);
    }
    cout << "Enter the Multiplier to use default or input new parameters." << endl;
    cin >> multiplier;
    if (multiplier > 0) {
        cout << "Invalid Multiplier" << endl;
        return (0);
    }
    cout << "Enter the seed to use default or input new parameters." << endl;
    cin >> seed;
    LCG(filename, howmany, modulus, seed, multiplier, increment);
    return (0);
}

MiddleSquare.cpp.

/*
Middle Square Generator
*/
#include "thesis.h"

int main() {
    char filename[100] = "MSQUARE.txt";
    unsigned int howmany; // number of numbers to generate
    unsigned int seed;
    unsigned int counter;
double alpha, beta;
unsigned int y, x;
int digits, c;
    cout << "MiddleSquare GENERATOR" << endl;

    cout << "Enter the Modulus to use default or input new parameters." << endl;
    cin >> modulus;
    if (modulus > pow((double) modulus, 5)) {
        cout << "Invalid Modulus" << endl;
        return (0);
    }
    cout << "Enter the Multiplier to use default or input new parameters." << endl;
    cin >> multiplier;
    if (multiplier > 0) {
        cout << "Invalid Multiplier" << endl;
        return (0);
    }
    cout << "Enter the seed to use default or input new parameters." << endl;
    cin >> seed;
    LCG(filename, howmany, modulus, seed, multiplier, increment);
    return (0);
}
```

MiddleSquare.cpp.
cout << "How many numbers would you like to generate?" << endl;
cin >> howmany;
cout << endl << "Recommended filename is SQUARE.txt."
cout << endl << "Enter the filename":" << endl;
cin >> file_name;
cout << endl << "Enter the seed":" << endl;
cin >> seed;

fstream NAMES;
remove(file_name);
NAMES.open(file_name, fstream::out | fstream::app);
x = seed;
for (counter = 0; counter <= howmany - 1; counter++){
x = x*x;
c = 1;
alpha = x;
y = 1;
beta = y;
for (c = 10; c > 0; c --) { // this part determines the middle digits
  y = int(pow(10.0, double(c)));
  beta = y;
  if (alpha / beta < 1) {
    digits = c;
  }
}
// when this is done we know that our number is "digits" digits long
if ((digits == 8) || (digits == 7)) {
  x = unsigned int(floor(x/100));
  x = x%10000;
}
if ((digits == 6) || (digits == 5)) {
  x = unsigned int(floor(x/10));
  x = x%10000;
}
NAMES << double(x)/pow(10.4) << endl;
}
NAMES.close();
return(0);
}

Fibonacci.cpp.

/*
This program produces the Fibonacci sequence mod m
m is currently set to 2^30
output can be found in fibonacci.txt
The modulus should be \lt half of the word size
\to prevent overflow
*/
#include "thesis.h"
#include <cmath>

void FIB(char file_name[1000], unsigned int howmany, unsigned int modulus, unsigned int seed[2])
{
  unsigned int Y[2];
  unsigned int x;
  ifstream NAMES;
  remove(file_name);
  NAMES.open(file_name, fstream::in | fstream::out | fstream::app);
  for (int c = 0; c <= 1; c++){
    Y[c] = seed[c];
  }
  for (unsigned int counter = 0; counter <= howmany - 1; counter++){
    x = (Y[1] + Y[0])%modulus;
    NAMES << double(x)/double(modulus) << endl;
    Y[0] = Y[1];
    Y[1] = x;
  }
}

int main()
{
  unsigned int seed[2];
//file_name is the name of the file to be written
char file_name[1000];
unsigned int howmany;  //how many to generate
unsigned int def_modulus = 1073741824;  //2^30
cout<<"FIBONACCI GENERATOR"<<endl;
cout<<"How many numbers would you like to generate?"<<endl;
cin>>howmany;
cout<<endl;
cout<<"Recommended filename is fibonacci.txt."
<<"Note that you must insert .txt extension."<<endl;
cout<<"Enter the filename"<<endl;
cin>>file_name;
cout<<endl;
cout<<"Default modulus is 1073741824."<<endl;
cout<<"What modulus would you like to use?";
cin>>modulus;
if(modulus==0){
    modulus=def_modulus;
}
seed[0]=0;
seed[1]=1;
FIB(file_name, howmany, modulus, seed);
return(0);

LaggedFibonacci.cpp.
/*
This program produces the lagged Fibonacci sequence mod m
m is currently set to 2^30
output can be found in lagfib.txt
NOTE!!
In order to work, a file containing at least 50 numbers in U(0,1)
must be specified under
void getseed.
user must specify file to seed from in void getseed
and file to write to in int main()
*/
#include "thesis.h"
void getseed(unsigned int seed[100], unsigned int modulus){
    //this is the MCG
    //DO NOT CHANGE THESE VALUES

    unsigned int mcg_seed=1;  //seed
    unsigned int mcg_multiplier =48271;  //multiplier
    unsigned int mcg_increment =0;  //increment
    unsigned int counter;
    unsigned int mcg_howmany=100;  //how many to generate
    unsigned int mcg_modulus = 2147483647;  //2^31-1
    unsigned int x[2];
    unsigned int y=mcg_seed;
    unsigned int q=unsigned int(mcg_modulus/mcg_multiplier);
    //DO NOT CHANGE THESE VALUES

    for(counter=0;counter<mcg_howmany;counter++){
        /*
        This next part looks funny. It is the trick used
to make sure we don't have overflow. See Knuth, or
reference in my thesis.
        */
        x[0]=mcg_multiplier*(y%q);
        x[1]=(mcg_modulus%mcg_multiplier)*unsigned int (y/q);
        if(x[0]>x[1]){
            y=x[0]-x[1];
        }
        else if(x[0]<x[1]){
            y=mcg_modulus-x[1];
            y=y+x[0];
        }
    }
}
44 B. C++ PROGRAMS

```cpp
seed(counter)=int(modulus+double(y)/double(mcgg_modulus));
// this scales numbers from [0, mcgg_modulus) to [0, modulus)
// where modulus is the lagfib modulus
}

void FIB(char file_name[1000], unsigned int howmany, unsigned int modulus, unsigned int seed[100])
{
    unsigned int result;
    fstream NUMBERS;
    remove(file_name);
    NUMBERS.open(file_name, fstream::in | fstream::out | fstream::app);
    for(unsigned int counter=0; counter<howmany-1; counter++)
    {
        result=(seed[99]+seed[16])%modulus;
        NUMBERS<<double(result)/double(modulus)<<endl;
    }
    seed[0]=result;
}

int main()
{
    unsigned int seed[100];
    char file_name[1000]; // name of file to be written
    unsigned int howmany; // how many to generate
    unsigned int def_modulus = 1073741824; // 2^30
    unsigned int modulus;
    // Note that you must worry about overflow unless modulus <= (word size)/2, because of the addition
    cout<<"LAGGED_FIBONACCI_GENERATOR"<<endl;
    cout<<"How_many_numbers_would_you_like_to_generate?"<<endl;
    cin>>howmany;
    cout<<"Recommended_filename_is_lagfib.txt."<<endl;
    cout<<"Enter_the_filename"<<endl;
    cin>>file_name;
    cout<<"Default_modulus_is_1073741824."<<endl;
    cout<<"Modulus_should_be<==(word_size)/2"<<endl;
    cout<<"You may enter_all_to_use_default."<<endl;
    cin>>modulus;
    if (modulus==0){
        modulus=def_modulus;
    } else {
        getseed(seed, modulus);
    }
    FIB(file_name, howmany, modulus, seed);
    return(0);
}
```

frequency test (single reading).cpp.

/*
This is the frequency test. This does not write to a file.
The result is printed to the screen.
*/

#include "thesis.h"

int main()
{
    int d; // the scaling factor
    char file_name[100]; // file to read from
double u_value;
    unsigned int COUNT[10000];
    unsigned int TOTAL_COUNT=0;
    unsigned int TC;
    // TC is the number of sample points
    cout<<"FREQUENCY_TEST~SINGLE_READING"<<endl;
    cout<<endl<<"Note that you must insert txt extension."<<endl;
    cout<<endl<<"Name the file to analyze which contains decimals in [0,1]."<<endl;
    cin>>file_name;
    fstream NUMBERS;
```
B. C++ PROGRAMS

NUMBERS.open(file_name, fstream::in);
if (NUMBERS.fail()){
    cout << "File could not be opened" << endl;
    return(0);
}

cout << "How many numbers from file_name should be used?" << endl;
if (cin.fail())
    return(0);
cout << "Enter the scaling factor."
    cin >> T;
cout << "The recommended scaling factor is " << endl;
cin >> D;
cout << "The scaling factor is " << endl;
    for (int e = 0; e < 10000; e++) {
        // this sets all counts to 0
        COUNT[e] = 0;
    }
    while ((NUMBERS >> u_value) && (TOTAL_COUNT < TC)) {
        // a number is retrieved
        y_value = int((u_value / D) + TOTAL_COUNT);
        COUNT[y_value]++;
        TOTAL_COUNT++;
    }
    // at this point our COUNTERS are full with their information
    if (TOTAL_COUNT < TC) {
        cout << "ERROR!!! Not enough numbers in file."
            cout << "Only " << TOTAL_COUNT << " numbers were available."
    }
    int fail;
    cin >> fail;
    return(0);
}
double V;
    double sum = 0;
    for (int m = 0; m < 10000; m++)
        sum = sum + COUNT[m] * (TOTAL_COUNT - COUNT[m] + 1) / (TOTAL_COUNT); // above is the chi-square calculation
    V = sum;
    cout << "FREQUENCY TEST";
    cout << "TOTAL COUNT: " << TOTAL_COUNT;
    cout << "CHI-SQUARE VALUE FOR " << file_name << " with " << " degrees of freedom is: "
        cout << V;
    return(0);
}

frequency test (multiple readings).cpp.

/*
This is the global frequency test
change number_of_tests to the number of independent frequency tests you wish to run
*/

#include "thesis.h"
int main()
{
    int d; // scaling factor
    unsigned int number_of_tests; // the number of times
    // to run the frequency test
    char file_name[100]; // file to read from
    //
    unsigned int COUNT[100000];
    unsigned int TOTAL_COUNT = 0;
    double V = 0;
    double sum = 0;
    double u_value;
    int y_value, sample_size;
    unsigned int point;
B. C++ PROGRAMS

```cpp
#include <fstream>
#include <iostream>
#include <vector>

int main()
{
    std::ifstream RECORD, RECORD2;
    remove("GLOBAL_freq.txt");
    remove("INDEX_freq.txt");
    RECORD.open("GLOBAL_freq.txt", std::ios::in | std::ios::out | std::ios::app);
    RECORD2.open("INDEX_freq.txt", std::ios::in | std::ios::out | std::ios::app);
    cout << "FUNCTION TEST MULTIPLE READINGS" << endl;
    cout << "Name the file to analyze which contains decimals in [0, 1]." << endl;
    cout << "Note that you must insert.txt extension." << endl;
    cin >> file_name;
    std::ifstream NUMBERS, open(file_name, std::ios::in);
    if (NUMBERS.fail())
    {
        cout << "File could not be opened" << endl;
        return 0;
    }
    cout << "How many numbers should be used per reading?" << endl;
    cin >> samplesize;
    cout << "This is referred to in the paper as the sample size."
        << endl;
    cout << "The recommended scaling factor is 31."
        << endl;
    cout << "Enter the scaling factor."
        << endl;
    double d;
    cout << "How many individual tests should be run?"
        << endl;
    cin >> numberoftests;
    cout << "This is referred to in the paper as the number of local samples."
        << endl;
    for (int e = 0; e < 1000; e++)
    {
        COUNT[e] = 0;
    }
    for (unsigned int squ = 0; squ < numberoftests; squ++)
    {
        for (int e = 0; e < 1000; e++)
        {
            COUNT[e] = 0;
        }
        double V = 0;
        double sum = 0;
        double TOTAL_COUNT = 0;
        for (point = 0; point < samplesize; point++)
        {
            while (TOTAL_COUNT < point)
            {
                if (NUMBERS >> u_value)
                {
                    y_value = int (u_value * d);
                    COUNT[y_value]++;
                    TOTAL_COUNT++;
                }
                else
                {
                    cout << "Process halted! nEnd of file reached before completion of analysis. n"
                        << "Only " << squ << " local samples were obtained."
                        << endl;
                    cin >> t;
                    cout << "Process halted! nEnd of file reached before completion of analysis. n"
                        << "Only " << squ << " local samples were obtained."
                        << endl;
                    cin >> t;
                    return 0;
                }
            }
        }
        sum = 0;
        for (int m = 0; m < m + 1)
        {
            sum = sum + (COUNT[m] - TOTAL_COUNT * (1 / double(d))) * (COUNT[m] - TOTAL_COUNT * (1 / double(d))) * d / TOTAL_COUNT;
        }
        V = sum;
        RECORD << V << endl;
        RECORD2 << squ << endl;
        cout << "The chi-square readings with " << d - 1 << " degrees of freedom"
            << endl;
        return 0;
    }
    return 0;
}
```

serial test (single reading).cpp.
This is the frequency test. This does not write to a file. The result is printed to the screen.

```cpp
#include "thesis.h"

int main()
{
    int d; // the scaling factor
    char file_name[100]; // file to read from
    double u_value, v_value;
    int x_value, y_value;
    unsigned int COUNT[200][200];
    unsigned int TOTAL_COUNT=0;
    unsigned int TC;
    //TC is the number of sample points
    cout<<"SERIAL\TEST\"<<"SINGLE\READING"<<endl;
    cout<<endl<<"Name a file to analyze which contains decimals in [0, 1]."
<<endl<<"Note that you must insert .txt extension."<<endl;
    cin>>file_name;
    fstream NUMBERS;
    NUMBERS.open(file_name, fstream::in);
    if(NUMBERS.fail()){
        cout<<"File could not be opened"<<endl;
        return(0);
    }
    cout<<endl<<"How many pairs from "<<file_name<<" should be used?"
    <<endl<<"This is half of the sample size."
    <<endl;
    cin>>TC;
    cout<<endl<<"The recommended scaling factor is 10. The max is 200."
    <<endl<<"Enter the scaling factor."
    <<endl;
    cin>>d;
    cout<<endl;
    for(int e=0; e<200; e++){
        for(int i=0; i<200; i++){
            // this sets all counts to 0
            COUNT[e][i]=0;
        }
        while((NUMBERS>>u_value)&&(NUMBERS>>v_value)&&(TOTAL_COUNT<TC)){ // a number is retrieved
            x_value=(int)(u_value); // and scaled
            y_value=(int)(v_value);
            COUNT[x_value][y_value]++; // placed into a bin
            TOTAL_COUNT++;
        }
    }
}
```

**above is the chi-square calculation**
serial test (multiple readings).cpp.

/*
This is the frequency test. This does not write to a file.
The result is printed to the screen.
*/
#include "thesis.h"

int main()
{
    int d; // the scaling factor
    int global;
    char file_name[100]; // file to read from
    double u_value, v_value;
    int x_value, y_value;
    unsigned int COUNT[200][200];
    unsigned int TOTAL_COUNT;
    unsigned int samplesize, numberoftests;
    unsigned int MASTER_COUNT = 0;
    // samplesize is the number of sample points
    ifstream RECORD1, RECORD2;
    remove("GLOBAL_serial.txt");
    remove("INDEX_serial.txt");
    RECORD1.open("GLOBAL_serial.txt", fstream::in | fstream::out | fstream::app);
    RECORD2.open("INDEX_serial.txt", fstream::in | fstream::out | fstream::app);

cout << "SERIAL_TEST_MULTIPLE_READINGS" << endl;
    cout << endl;"" << Name_of_file_to_analyze_which_contains_decimals_in_the_range[0, 1]."" 
    << endl;"" Note that you must insert .txt extension."" << endl;
    cin >> file_name;
    ifstream NUMBERS;
    NUMBERS.open(file_name, fstream::in);
    if (NUMBERS.fail())
    {
        cout << "File could not be opened" << endl;
        return(0);
    }
    cout << endl;"" How many pairs should be used per reading?"" 
    << endl;"" This is half of the sample size."" 
    << endl;
    cin >> samplesize;
    cout << endl;"" The recommended scaling factor is 10."" 
    << endl;"" Enter the scaling factor."" 
    << endl;
    cin >> d;
    cout << endl;"" How many individual tests should be run?"" 
    << endl;"" This is referred to in the paper as the number of local samples."" 
    << endl;
    cin >> numberoftests;
    cout << endl;"" end mouth debugging."
for (global = 0; global < numberoftests; global++)
{
    for (int e = 0; e < 200; e++)
    {
        for (int i = 0; i < 200; i++)
        {
            // this sets all counts to 0
            COUNT[e][i] = 0;
        }
    }
    TOTAL_COUNT = 0;
    while ((NUMBERS >> u_value) & (NUMBERS >> v_value) & (TOTAL_COUNT < samplesize))
    {
        // a number is retrieved
        x_value = (int)u_value * d; // and scaled
        y_value = (int)v_value * d;
        COUNT[x_value][y_value]++;// placed into a bin
        TOTAL_COUNT++;
        MASTER_COUNT++;
    }
    // at this point our COUNTERS are full with their information
    double V;
    double sum = 0;
for (e=0;e<d;++e) {
    for (int i=0;i<d;i++) {
        sum = sum + (COUNT[e][i] - TOTALCOUNT*(1/(double)(d*d)))/(d*d);
    }
}

// above is the chi-square calculation

V = sum;
RECORD<<V<<endl;
RECORD<<global<<endl;
}

if ((sampleSize*NumberOfTests) != MASTERCOUNT) {
    cout << "ERROR! Not enough numbers in file."
        << endl << "Only " << MASTERCOUNT << " numbers available."
        << endl;
    int fail;
cin >> fail;
return(0);
}

cout << "The chi-square readings with " << d << " degrees of freedom"
    << endl << "have been written to GLOBAL_serial.txt." << endl;
return(0);
}

poker test (single reading).cpp.

#include "thesis.h"
int main() {
    unsigned int n; // number of "hands" to deal
    char file_name[100]; // the file to read from
    int t; // counts cards loaded into hand; breaks at 5
    unsigned int t_0; // the scaling number, must be at least 4 here
    double u; // value read from file
    unsigned int t=0; // used as dummy counter
    unsigned int t_0; // counts number of hands dealt
    double V=0, sum=0;
    double PROB[4]; // these will hold ideal category prob.
    unsigned int COUNT[4];
    unsigned int NUMBR[5];

    cout << "POKER_TEST : SINGLE_READING" << endl;
cout << "Name a file to analyze which contains decimals in [0,1]."
    << endl << "Note that you must insert .txt extension." << endl;
cin >> file_name;
FILE open(file_name, fstream::in);
If (FILE fail) {
    cout << "File could not be opened" << endl;
    return(0);
}
    cout << "How many " << n >> " numbers from " << file_name << " should be used?"
    << endl << "This is one-fifth the sample size." << endl;
cin >> n;
cout << "The recommended scaling factor is 10."
    << endl << "Enter the scaling factor."
    << endl;
cin >> d;
cout << endl;
// must set the arrays to 0;
for (t=0; t<4; t++) {
```cpp
PROB[t]=0;
COUNT[t]=0;
tracker=0;

for (t=0; t<5; t++) {
    NUMBER[t]=0;
}

// tracks five card hands
i=0;
while ((FILE>>x_value)&(i<5)) { // loads 400 hands
    int y_value= int((x_value)*d);
    NUMBER[ticker]=y_value;
    ticker++;

    if (ticker==5) {
        i++;
        ticker=0;

        for (int a=0; a<5; a++) {
            for (int b=0; b<5; b++) {
                if ((NUMBER[a]==NUMBER[b])&&(a!=b)) {
                    tracker++;
                }
            }
        }

        if (tracker==0) {
            COUNT[1]++;
            // all different
        } else if (tracker==2) {
            COUNT[2]++;
            // one pair
        } else if ((tracker==4)&&(tracker==6)) {
            COUNT[1]++;
            // two pairs, three of a kind
        } else if ((tracker==8)&&(tracker==12)&&(tracker==20)) {
            COUNT[0]++;
            // full house, four of a kind, five of a kind
        } else {
            cout << "FAIL" << endl;
        }
    }

    if (i==n) {
        cout << "ERROR!!! Not enough numbers in file."
        << endl << "Only " << i << " hands were computed."
        << endl;
        int fail;
        cin >> fail;
        return (fail);
    }

    for (t=0; t<4; t++) {
        sum=sum+(COUNT[t]-n*PROB[t])*(COUNT[t]-n*PROB[t])/(n*PROB[t]);
    }
}

V=sum;
cout << "PokerTest" << endl;
if (i==n) {
    cout << "ERROR!!! Not enough numbers in file."
    << endl << "Only " << i << " hands were computed."
    << endl;
    int fail;
    cin >> fail;
    return (fail);
}

cout << "Number_of_hands_computed:" << i << endl;
cout << "The chi-square value for file_name is: " << endl;
cout << V << endl;
```
poker test (multiple readings).cpp.

#include "thesis.h"

int main()
{
    //with the thesis.h file,
    //poker test emultiple readingsflcppl

    char file_name[100]; //the file to read from
    int ticker; //counts cards loaded into hand; breaks at 5
    unsigned int tracker; //tracker measures a combinatorial property of each hand
    double u-value; //value read from file
    unsigned int t=0; //used as dummy counter
    unsigned int l; //counts number of hands dealt
    double PROB[4]; //these will hold ideal category probs.
    unsigned int COUNT[4];
    unsigned int NUMBER[5];
    unsigned int numberoftests;
    fstream INDEX, RECORD;
    remove(\"GLOBAL_poker.txt\")
    remove(\"INDEX_poker.txt\")
    INDEX.open(\"INDEX_poker.txt\", fstream::in | fstream::out | fstream::app);
    RECORD.open(\"GLOBAL_poker.txt\", fstream::in | fstream::out | fstream::app);
    cout \"\"POWQ \"FILE:\n    FILE.open(\"name\", fstream::in);

    FILE.open(\"file name\", fstream::in);
    if(FILE.fail()){
        cout \"\"File could not be opened\"<<endl;
        return (0);
    }

cout << \"How many \"hands\" of 5 numbers from \"file name\" should be used per sample?\"
   ipro##(FILE.fail()){
        cout \"\"File could not be opened\"<<endl;
        return (0);
    }

cout << \"How many individual tests should be run?\";

cout << \"This is referred to in the paper as the number of local samples.\"

cout >> numberoftests;

count << numberoftests;

    //and we need to calculate our theoretical probabilities
    //notice how we manually insert the Stirling numbers
    PROB[0] = 1
    PROB[1] = (((d-1)\(d-2)/\(d+\)5))\(15\)
    PROB[2] = (((d-1)\(d-2)/\(d+\)5))\(10\)
    PROB[3] = (((d-1)\(d-2)/\(d+\)5))\(1\)

    for(unsigned int global=0, global<endnumberoftests, global++){
        //global is the number of samples of 400 hands to generate
        //must set the arrays to 0;

        for(t=0,t<4, t++)
            COUNT[t]=0;

        tracker=0;
        for(t=0, t<5, t++)
            NUMBER[t]=0;

        ticker=0; //counts five card hands

while ((FILE>>u_value)&(i<n)) {
    int y_value = int(u_value)*d;
    NUMBER[ticker]=y_value;
    ticker++;
    if (ticker==5) {
        i++;
        ticker = 0;
        for (int a=0; a<=5; a++) {
            for (int b=0; b<=5; b++) {
                if (((NUMBER[a]==NUMBER[b])&(a!=b))) {
                    tracker++;
                }
            }
        }
        if (tracker==0) {
            COUNT[3]++;
            // all different
        } else if (tracker==2) {
            COUNT[2]++;
            // one pair
        } else if (((tracker==4)&&(ticker==6)) {
            COUNT[1]++;
            // two pairs, three of a kind
        } else if (((ticker==8)&&(ticker==12)&&(ticker==20)) {
            COUNT[0]++;
            // full house, four of a kind, five of a kind
        } else {
            cout<<"FAIL"<<endl;
        }
    } else if (!n) {
        cout<<"ERROR!!! Not enough numbers in file."
            "<endl;"
        int fail;
        cin>>fail;
        return(0);
    }
    // now we compute V = chi square
    sum = 0;
    for (t=0; t<4; t++) {
        sum = sum + ((COUNT[t]-n*PROB[t])*(COUNT[t]-n*PROB[t]))/(n*PROB[t]);
    }
    V = sum;
    RECORD< V  <endl;
    INDEX<< global<<4<< endl;
} // end of big loop
    cout<< "The chi-square readings with 3 degrees of freedom" 
        "<endl;"
    return(0);
}

runs test (single reading).cpp.

/*
This is the frequency test. This does not write to a file.
The result is printed to the screen.
*/
#include "thesis.h"
int main() {
    char file_name[100]; // file to read from
    double U[2];
    int j,t;
    unsigned int COUNT[7], numberofruns;

double PROB[7];
unsigned int sample_size;

// TC is the number of sample points
cout << "How many numbers from file should be used?" << endl;
cin >> sample_size;
if (sample_size < 0) { return 0; }

for (int c = 0; c < 7; c++) { COUNT[c] = 0; }
}

int numberofruns = 0;
while (j < sample_size - 1) {
    r = 1;
    // this is written to throw out the number which breaks each run
    if (!(NUMBERS >> U[0])) {
        cout << "Error! Not enough numbers" << endl;
        return 0;
    } j++;
    if (!(NUMBERS >> U[1])) {
        cout << "Error! Not enough numbers" << endl;
        return 0;
    } j++;
    while ((U[0] < U[1]) && (r < 6)) {
        r++;
        U[0] = U[1];
        if (!(NUMBERS >> U[1])) {
            cout << "Error! Not enough numbers" << endl;
            return 0;
        } j++;
    }
    if (r > 6) {
        COUNT[6]++;
    } else {
        COUNT[r]++;
    }
}

for (c = 1; c < 7; c++) {
    numberofruns += COUNT[c];
}

// the probabilities were calculated in the paper
PROB[1] = 5.;
PROB[2] = 3.3333333333333333;
PROB[3] = 1.25;
PROB[4] = 0.3333333333333333;
PROB[5] = 0.06944444444444444;
PROB[6] = 0.001388888888888888;
double V;

double sum=0;

for(int m=1;m<TMP1;m++) //index begins at 1
    sum=sum + ((double)COUNT[m] - (double)numberofruns*PROB[m])
* ((double)COUNT[m] - (double)numberofruns*PROB[m])
    //above is the chi-square calculation
}

V=sum;

cout<<"RUNS_TEST"<<endl;
cout<<"Total number of runs= "<<numberofruns<<endl;
cout<<"The chi-square value for g= "<<file_name<<endl;
cout<<V<<endl;
return(0);

runs test (multiple readings).cpp.
/*
 This is the frequency test. This does not write to a file.
The result is printed to the screen.
*/
#include "thesis.h"

int main()
{
    char file_name[100]; //file to read from
    double U[2];

    int j;
    unsigned int COUNT[7], numberofruns, numberofsamples;
    double PROB[7];
    unsigned int sample_size;
    ifstream RECORD, RECORD2;
    remove("GLOBAL.runs.txt");
    remove("INDEX.runs.txt");
    RECORD.open("GLOBAL.runs.txt", ios::in | ios::out | ios::app);
    RECORD2.open("INDEX.runs.txt", ios::in | ios::out | ios::app);

    //TC is the number of sample points
    cout<<"MULTIPLE_READINGS"<<endl;
    cout<<"Note that you must insert txt extension."<<endl;
    cin>>file_name;
    ifstream NUMBERS;
    NUMBERS.open(file_name, ios::in);
    if (NUMBERS.fail()){
        cout<<"File could not be opened"<<endl;
        return(0);
    }
    cout<<endl;
    cout<<"How many numbers from "<<file_name<<" should be used?"
<<endl;
    cout<<"This is referred to in the paper as the sample size."
<<endl;
    cin>>sample_size;
    cout<<endl;
    cout<<"How many individual tests should be run?"
<<endl;
    cout<<"This is referred to in the paper as the number of local samples."
<<endl;
    cin>>numberofsamples;
    cout<<endl;

    for(int global=0;global<numberofsamples;global++)
    {
        for(int c=0;c<TMP1;c++)
        {
            COUNT[c]=0;
        }
        numberofruns=0;
        while(c<sample_size-1){
            r=1;
            //this is written to throw out the number which breaks each run
            if (NUMBERS>>U[0]){
                cout<<"Error! Not_enough_numbers"<<endl;
                return(0);
            }
        }
    }
    return(0);
}
j++;  
if(!(NUMBERS>U[1])){
    cout<<"Error! Not enough numbers"<<endl;
    return(0);
}  
j++;  
while((U[0]<U[1])&&(r<6)){
    r++;  
    U[0]=U[1];  
    if(!(NUMBERS>U[1])){
        cout<<"Error! Not enough numbers"<<endl;
        return(0);
    }  
}  

/* In the event that adjacent numbers are equal, which is unlikely because we are using decimals with 6 degrees of precision, this command abandons the current run and moves to the next */

}  
if(r>sample_size){
    break;
}  
if(r>=6){
    COUNT[r]++;  
}  
else{
    COUNT[r]++;  
}
}

for(c=1;c<7;c++){
    numberofruns+=COUNT[c];  
}

// these probabilities were calculated in the paper
PROB[1] = .5;  
PROB[2] = 3.3333333333;  
PROB[3] = .125;  
PROB[4] = 0.333333333;  
PROB[5] = .0069444444;  
PROB[6] = .00138888889;

double V;  
double sum=0;  
for(int m=1;m<7;m++)//index begins at 1
    sum=sum+((double)COUNT[m]-(double)numberofruns*PROB[m])  
    +(double)COUNT[m]-(double)numberofruns*PROB[m])  
/((double)numberofruns*PROB[m]);

// above is the chi-square calculation
V=sum;  
RECORD1<<V<<endl;  
RECORD2<<global<<endl;
}

cout<<"RUNS TEST"<<endl;

cout<<"The chi-square value for "<<file_name()<<endl<<"with 5 degrees of freedom"<<endl;

return(0);
}

pi Estimate.cpp.
/*
This program estimates pi by counting the number of points falling within a circle inscribed in a square
*/
#include "thesis.h"
int main()
{
    double x,y, norm, counter, pi;
    unsigned int numberofpairs;

unsigned int darts = 0;
char file_name [100];
cout << "ESTIMATION_of_e" << endl;
cout << endl << "Name_a_file_containing_decimals_in_0,1." << endl;
cout << "Note_that_you_must_insert_.txt_extension." << endl;
cin >> file_name;
fstream FILE;
FILE.open (file_name, fstream::in);
if (FILE.fail()) {
    cout << "File_could_not_be_opened" << endl;
    return (0);
}
cout << endl;
while ((FILE >> x) && (FILE >> y) && (darts < number_of_pairs)) { // reads the numbers from file
    norm = pow((x, 2)) + pow((y, 2));
    if (norm <= 1) {
        counter++;
        darts++;
    }
}
if (darts != number_of_pairs) {
    cout << "ERROR!!! Not_enough_numbers_in_file." << endl;
    cout << "Only _" << darts << " pairs were_available." << endl;
    int fail;
    cin >> fail;
    return (0);
}
pi = 4 * counter / double (number_of_pairs);
cout << Pi << "_" << pi << endl;
return (0);
}

Estimate.cpp.

/*
This program estimates e by counting the number of
numbers from 0,1 required to reach a sum > 1
*/
#include "thesis.h"
int main() {
    double SUM, value;
    unsigned int counter, TOTAL, number;
    char file_name [100];
    unsigned int howmany;
    unsigned int c = 0;
cout << "ESTIMATION_of_e" << endl;
cout << endl << "Name_a_file_containing_decimals_in_0,1." << endl;
cout << "Note_that_you_must_insert_.txt_extension." << endl;
cin >> file_name;
fstream FILE;
FILE.open (file_name, fstream::in);
if (FILE.fail()) {
    cout << "File_could_not_be_opened" << endl;
    return (0);
}
cout << endl;
while ((howmany) && (FILE >> value)) { // reads from file
    if (SUM = SUM + value) { // adds to sum
        counter++;
    }
}
SUM=0;
TOTAL=TOTAL+counter; // counts how many numbers have been used
number++; // number counts how many times I has been reached
counter=0;
}
c++;
}
if(c!=howmany){
    cout<<"ERROR!!! Not enough numbers in file."
        "Only "<<c<<" numbers were available."
        endl;
    int fail;
    cin>>fail;
    return(0);
}cout<<"Approximation of e: "<<double(TOTAL)/double(number)<<endl<<endl;
return(0);
Bibliography


