LECTURE 4. RANDOM NUMBERS GENERATORS

4.1. Introduction

A degree of randomness is built into the fabric of reality. It is impossible to say for certain what a baby's personality will be, how the temperature will fluctuate next week, or which way dice will land on their next roll.

Because randomness is so inherent in everyday life, many researchers have tried to either harvest or simulate its effect inside the digital realm. Before accomplishing this feat, many important questions need to be answered.

What does it mean to be random?

How does a person go about creating randomness, and how can he capture the randomness he encounters?

How can someone know if an event or number sequence is random or not?

4.1.1. Defining Random

What does it mean to have random numbers?

Without understanding where a group of numbers came from, it is impossible to know if they were randomly generated. However, common sense claims that if the process to generate these numbers is truly understood, then the numbers could not be random.

It is impossible to appreciate a random number generator without firstly understanding *what it means to be random*.

Developing a well-rounded definition of randomness can be accomplished by studying a random phenomenon, such as a dice roll, and exploring what qualities makes it random. To begin, imagine that a family game includes a die to make things more interesting. In the first turn, the die rolls a five. By itself, the roll of five is completely random. However, as the game goes on, the sequence of rolls is five, five, five, and five. The family playing the game will not take long to realize that the die they received probably is not random.

To make sure the next die the family buys is random, they roll it 200 times. This time, the die did not land on the same face every time, but half of the rolls came up as a one. This die would not be considered random either, because it has a disproportionate bias toward a specific number.

To be random, the die should land on all possible values equally.

In a third scenario, the dice manufacturer guarantees that now all its dice land on all numbers equally. Cautious, a family roles this new die 200 times to verify. Although the numbers were hit uniformly, the family realized that throughout the entire experiment the numbers always followed a sequence: five, six, one, two, etc. Once again, the randomness of the die would be questioned.

For the die to be accepted as random, it could not have any obvious patterns in a sequence of dice rolls. If it can be predicted what will happen next, or anywhere in the future, the die cannot truly be random.

From the results of these dice illustrations a more formal definition of randomness can be constructed. A generally accepted and basic definition of a random number sequence is as follows: **a random number sequence is uniformly distributed over all possible values and each number is independent of the numbers generated before it** (Marsaglia, 2005).

A random number generator can be defined as any system that creates random sequences like the one just defined.

Unfortunately, time has shown that the requirements for a random number generator change greatly depending on the context in which it is used.

When a random number generator is used in cryptography, it is vital that the past sequences can neither be discovered nor repeated; otherwise, attackers would be able to break into systems (Kenny, 2005).

The opposite is true when a generator is used in simulations. In this context, it is actually desirable to obtain the same random sequence multiple times. This allows for experiments that are performed based on changes in individual values.

The new major requirement typical of simulations, especially Monte Carlo simulations, is that vast amounts of random numbers need to be generated quickly, since they are consumed quickly (Chan, 2009).

Random numbers are often used in digital games and in statistical sampling as well. These last two categories put very few requirements on the random numbers other than that they be actually random.

4.1.2 Types of Random Number Generators

With a description of randomness in hand, focus can shift to random number generators themselves and how they are constructed. Typically, whenever a random number generator is being discussed, its output is given in binary. Generators exist that have non-binary outputs, but whatever is produced can be converted into binary after the fact.

There are two main **types** of random number generators.

- *The first type attempts to capture random events in the real world to create its sequences*. It is referred to as a *true random number generator*, because in normal circumstances it is impossible for anyone to predict the next number in the sequence.
- *The second camp believes that algorithms with unpredictable outputs (assuming no one knows the initial conditions) are sufficient to meet the requirements for randomness*. The generators produced through algorithmic techniques are called *pseudo-random generators*, because in reality each value is determined based off the system's state, and is not truly random. To gain an understanding of how these generators work, specific examples from both categories will be examined.

4.2. True Random Number Generators

A true random number generator uses entropy sources that already exist instead of inventing them.

Entropy refers to the amount of uncertainty about an outcome. Real word events such as *coin flips* have a high degree of entropy, because it is almost impossible to accurately predict what the end result will be. It is the source of entropy that makes a true random number generator unpredictable. Flipping coins and rolling dice are two ways entropy could be obtained for a generator, although the rate at which random numbers could be produced would be restricted.

Disadvantages of random number generators:

- Low production rate
- These generators relie on some sort of hardware. Since they use real world phenomena, some physical device capable of recording the event is needed. This can make true random generators a lot more expensive to implement, especially if the necessary device is not commonly used.

- They are vulnerable to physical attacks that can bias the number sequences.
- Physical devices are typically vulnerable to wear over time and errors in their construction that can naturally bias the sequences produced (Sunar, Martin, & Stinson, 2006). To overcome bias, most true random number generators have some sort of post processing algorithm that can compensate for it.

Despite these disadvantages, there are many contexts where having number sequences that are neither artificially made nor reproducible is important enough to accept the obstacles. For security experts, there is a peace of mind that comes with knowing that no mathematician can break a code that does not exist.

Random.org. A widely used true random number generator is hosted on a website named Random.org. Random.org freely distributes the random sequences it generates, leading to a varied user base.

However, since the numbers are obtained over the Internet, it would be unwise to use them for security purposes or situations where the sequence absolutely needs to stay private. There is always the risk that the transmission will be intercepted.

The random number generator from this site *collects its entropy from atmospheric noise*. Radio devices pick up on the noise and run it through a postprocessor that converts it into a stream of binary ones and zeroes.

Scholars have pointed out that the laws governing atmospheric noise are actually deterministic, so the sequences produced by this generator are not completely random (Random.org, 2012).

The proponents of this claim believe that only quantum phenomena are truly nondeterministic. Random.org has countered this argument by pointing out that the number of variables that would be required to predict the values of atmospheric noise are infeasible for humans to obtain. Guessing the next number produced would mean accurately recoding every broadcasting device and atmospheric fluctuation in the area, possibly even down to molecules. It has been certified by several third parties that the number sequences on this site pass the industry-standard test suites, making it a free and viable option for casual consumers of random numbers.

HotBits (https://www.fourmilab.ch/hotbits/). This site generates its random number sequences *based on the radioactive decay*. Because this is a quantum-level phenomenon, there is no debate over whether the number sequences are truly non-deterministic. At the same time, the process involved in harvesting this phenomenon restricts HotBits to only producing numbers at the rate of 100 bytes per second (HotBits, 2012).

Although the HotBits server stores a backlog of random numbers, the rate at which random sequences can be extracted is still limited in comparison to other options.

As with Random.org, random numbers obtained from this generator are sent over the Internet, so there is always the possibility that a third party has knowledge of the sequence. This makes it unsuitable for security-focused applications, but Hotbits is useful when unquestionably random data is necessary.

Lasers. The use of lasers allows for true random number generators that overcome the obstacle of slow production. In laser-based generators, entropy can be obtained by several different means.

Having two photons race to a destination is one method that is currently implemented Another high-speed technique is measuring the varying intensity of a chaotic laser.

Prototype systems in this second category have been created that can produce random bits at rates of over ten Gigabits per second (Li, Wang, & Zhang, 2010). The prototypes exhibited a natural bias toward one value over the other, so a post processor needed to be applied to create truly random sequences. A commonly used tactic is to take several bits at a time and exclusive-or them together to remove the unwanted bias. The stream of bits that emerged from this process was able to pass the most stringent randomness tests that are used for generators dealing with cryptography.

Laser generators are capable of increased speeds, but they are complex to install and prohibitively expensive.

Care needs to be taken during construction and installation that no bias is introduced, and the natural wear of the laser could also lead to it subtly producing more biased results over time. It is difficult to imagine laser-based generators being used in practical applications.

Oscillators make use of basic hardware, resulting in more convenient installation.

An oscillator is a simple circuit obtained by placing an odd number of inverter gates in a loop. The final output of this configuration is undefined, since the current oscillates in a sine wave pattern over time.

However, manufacturing is never perfect and defects always cause a slight and random deviation from a sine wave. These deviations are referred to as jitter, which is a common source of entropy in simple random number generators (Sunar, Martin, & Stinson, 2006). Because oscillators rely on manufacturing defects that cannot be replicated, they are part of a broader group of physical unclonable functions, or PUF.

To increase the randomness of jitter, oscillators of different lengths can be combined and evaluated together. Oscillators are cheap to install and use in comparison to other types of physical devices since their components are commonly used.

Random number generators based off of oscillators are vulnerable to many types of attacks. Environmental effects such as temperature changes and power surges can influence the jitter of a system.

Attackers can change these variables intentionally to influence the random sequence for a limited time. These techniques are known as non-invasive attacks because they don't require direct contact to initiate.

Invasive attacks can also be launched against oscillators. These attacks attempt to inflict a permanent defect inside the oscillator system, which will break the circuit and force a nonrandom output (Sunar, Martin, & Stinson, 2006).

Fortunately, complexity can be added into oscillator-based generators that can thwart both types of attacks. When compared to pseudo-random generators that use tamper-proof algorithms, true random generators can appear to be very fragile.

The actual case is a tradeoff between passive and active attacks. A passive attack occurs when the attacker does not need to alter the system, and is much harder to detect then active attacks, which often leave a footprint. Although true random number generators can suffer from active attacks, the pseudo-random generators that will be described are all vulnerable to passive attacks where the entire sequence past and future can be predicted. In high-risk situations like cryptography, the potential setbacks of true generators are often preferable.

4.3. Pseudo Random Number Generators

Random number generators that do not rely on real world phenomena to produce their streams are referred to as pseudo random number generators.

These generators appear to produce random sequences to anyone who does not know the secret initial value.

In a minimalistic generator, the initial value will be the only time entropy is introduced into the system. *Unlike true random number generators that convert entropy sources directly into sequences, a pseudo random needs to find entropy to use to keep itself unpredictable*. Classic tactics for accomplishing this include taking the time of day, the location of the mouse, or the activity on the keyboard. Another way of explaining these sources is that they use the entropy of human interaction. This approach is frowned on in secure settings, because an attacker could purposely manipulate physical interactions to bias the system (Gutternman, Pinkas, & Reinman, 2006). If no human users interface with the hardware, generators are normally able to use other components on the system, such as hard drives, to generate entropy. *Regardless, pseudo random number generators are limited by the entropy in their host device*. These entropy sources determine the quality of the resulting sequences.

If the initial values of a pseudo random generator are known, every value in the sequence can be easily determined and even recalculated. This makes securing pseudo random generators against attackers of pivotal importance. The generator should be designed so that determining the internal variables at any given time is an infeasible task.

Going further, assuming that an attacker is able to determine the internal variables at a point in time, pseudo random generators should still be able to protect themselves.

Forward security is the term used to describe a generator where knowing the internal state of a generator at a point in time will not help an attacker learn about previous outputs (Gutternman, Pinkas, & Reinman, 2006). If random number generators are being used for purposes like password creation, keeping up forward security becomes vital.

Backward security denotes that an attacker who learns the state of the generator at a point in time will not be able to determine future numbers that will be produced. Backward security is only possible if the generator introduces some level of entropy into its equation.

True random number generators always have forward and backward security, because they have no deterministic components.

Outside attackers are not the only problem inherent in pseudo random generators. At times honest or malicious mistakes can render an entire generator insecure. For example, the National Institute of Standards and Technology, or NIST, periodically publishes a list of pseudo random generators it deems secure enough for cryptography. In their 2007 publication, one of the four generators listed was championed by the National Security Agency, or NSA. It was called Dual_EC_DRBG (Schneier, 2007). Independent researchers quickly discovered that this generator contained a backdoor. Theoretically, there exists a set of constant numbers that, when known, would allow an attacker to predict every value of NSA's generator after collecting thirty-two bytes of random output. Although it is impossible to tell if the NSA possessed that set of constants, or even knew that such a thing existed, this served as a reminder that all pseudo random generators should be thoroughly examined by experts before they are trusted.

Since random numbers are used in many important applications such as setting up secure Internet communications, there are many groups that desire to crack the generators involved.

Pseudo random number generators generally lack entropy after initialization, so once they are broken the attacker requires no additional effort to monitor the system. Additional complexity and thorough security checks need to be included in all pseudo random number generators to make them safe for public use.

Aside from vulnerability to attacks, all pseudo random generators share fundamental limitations. Without continual entropy, random generators can only create sequences based off of a

limited set of initial conditions. Sequences created this way can only last a limited amount of time before they reach their starting point and repeat themselves exactly.

The length a sequence can extend before it repeats itself is referred to as its period (Chan, 2009).

A major consideration in the choice of a pseudo random number generator is the size of its period, because this directly affects the frequency that a generator can be used.

Pseudo random generators are capable of producing sequences at high speeds, so in applications that use vast quantities of randomness, the threat of exceeding the period is not trivial.

The field of cryptography also contributes to the creation of pseudo random numbers. Good encryption techniques that make messages undecipherable can also be used to encrypt a starting value into seemingly random numbers. Most encryption techniques have a poor production rate however, so using encryption techniques in generators is generally a bad idea (Trappe & Washington, 2006).

In the next sections, several of the more common algorithms used in random number generation, namely linear congruential, lagged Fibonacci, and feedback shift registers, will be discussed.

- All the randomness required by the model is simulated by a random number generator (RNG)
- The output of a RNG is assumed to be a sequence of i.i.d. r.v. U(0, 1)
- These random numbers are transformed as needed to simulate r.v. from different probability distributions
- The validity of transformation methods strongly depend on i.i.d. U(0,1) assumption
- This assumption is false, since RNG are simple deterministic programs trying to fool the users by producing a deterministic sequence that looks random

Two important statistical properties of the random numbers:

- Uniformity Independence.
- Random Number R_i must be independently drawn from a uniform distribution with the probability density function (pdf) and CDF:

$$f(x) = \begin{cases} 1, & 0 \le x \le 1 \\ 0, & \text{otherwise} \end{cases}$$

$$F(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \le x \le 1 \\ 1, & x > 1 \end{cases}$$

$$E(X) = \frac{1}{2}, V(X) = \frac{1}{12}$$

Remarks

Consequences of the uniformity and independence properties:

- if [0, 1] is divided into n clases of equal length, the expected number of observations in each interval is N/n, where N is the total number of observations
- The probability of observing a value in a particular interval is independent of the previous value drawn

Problems or errors (departure from ideal randomness)

- generated numbers may not be uniformly distributed
- discrete instead of continuous values
- mean and/or variance too high or too low
- dependence

- autocorrelation
- number successively higher or lower than adjacent numbers
- several numbers above the mean followed by several numbers below the mean

Important considerations in RN routines:

- fast - a large number of RN is required - total cost maintained reasonable
- portable - to different computers, OS, and programming languages
- have sufficiently long period - period must be longer than the numbers of events to be generated
- replicable - given the starting point (or conditions) it should be possible to generate the same set of random numbers, independent of the system to be simulated; useful for debugging and comparison
- Closely approximate the ideal statistical properties of uniformity and independence.

Techniques to produce pseudo-random numbers:

- Linear Congruential Method (LCM).
- Combined Linear Congruential Generators (CLCG).
- Feedback Shift Register Generators (FSRG)
- Random-Number Streams.

4.3.1 Midsquare method

First arithmetic generator: Midsquare method (von Neumann and Metropolis in 1940s)

The Midsquare method has the following steps:

- Start with a four-digit positive integer Z_0
- Compute Z_0^2 to obtain an integer with up to eight digits
- Take the middle four digits for the next four-digit number
- Issue: Generated numbers tend to 0

i	Z_i	U_i	$Z_i \times Z_i$
0	7182	-	51581124
1	5811	0,5811	33767721
2	7677	0,7677	58936329
3	9363	0,9363	87665769
4	6657	0,6657	44315649
5	3156	0,3156	09960336
6	9603	0,9603	92217609
7	2176	0,2176	04734976
8	7349	0,7349	54007801
9	78	0,0078	00006084
10	60	0,006	00003600
11	36	0,0036	00001296
12	12	0,0012	00000144
13	1	0,0001	00000001
14	0	0	00000000
15	0	0	00000000

4.3.2 Linear congruential method (*LCG*)

- To produce a sequence of integers, $X_1, X_2, \ldots$ between 0 and $m-1$ by following a recursive relationship:

$$X_{i+1} = (aX_i + c) \bmod m, \qquad i = 0, 1, \ldots \tag{1}$$

  a multiplier, c increment, m modulus, X_0 seed, called *Linear Congruential Generator* (LCG)

- $c = 0$ *Multiplicative Congruential Generator* (MCG)
- The selection of the values for a, c, m, and X_0 drastically affects the statistical properties and the cycle length.
- The random integers are being generated $[0, m-1]$, and to convert the integers to random numbers:

$$R_i = \frac{X_i}{m}, \qquad i = 1, 2, \ldots$$

If the constants are selected in accordance with the well-established rules publicly available, such as choosing an m value that is prime, then this algorithm will produce every value zero through m exactly once in its period, which is m, and is uniformly distributed.

Examples:

1. Use $X_0 = 27$, $a = 17$, $c = 43$, and $m = 100$. The X_i and R_i values are:
   $X_1 = (17\times27+43) \bmod 100 = 502 \bmod 100 = 2 \quad \rightarrow R_1 = 0.02$
   $X_2 = (17\times2 +43) \bmod 100 = 77 \quad \rightarrow R_2 = 0.77$
   $X_3 = (17\times77+43) \bmod 100 = 52 \quad \rightarrow R_3 = 0.52$
   $X_4 = (17\times52+43) \bmod 100 = 27 \quad \rightarrow R_4 = 0.27$
   .....
2. Use $a = 13$, $c = 0$, and $m = 64$

- The period of the generator is very low
- Seed X_0 influences the sequence

i	X_i X_0=1	X_i X_0=2	X_i X_0=3	X_i X_0=4
0	1	2	3	4
1	13	26	39	52
2	41	18	59	36
3	21	42	63	20
4	17	34	51	4
5	29	58	23	
6	57	50	43	
7	37	10	47	
8	33	2	35	
9	45		7	
10	9		27	
11	53		31	
12	49		19	
13	61		55	
14	25		11	
15	5		15	
16	1		3	

Characteristics of a good generator

- Maximum Density

- Such that the values assumed by R_i , $i = 1, 2, \ldots$, leave no large gaps on [0, 1]
- Problem: Instead of continuous, each R_i is discrete
- Solution: a very large integer for modulus m

- Maximum Period

- To achieve maximum density and avoid cycling.
- Achieved by: proper choice of a, c, m, and X_0.

- Most digital computers use a binary representation of numbers. Speed and velocity are aided by a modulus, m, to be (or close to) a power of 2.

Theorem (Hull and Dobell,1962)

The LCG defined by (1) has the full period iff the following conditions hold:

(a) $(m, c) = 1$

(b) q *prime,* $q|m \Longrightarrow q|a-1$

(c) $4|m \Longrightarrow 4|a-1$

Corollary

$X_{i+1} = (aX_i + c) \bmod 2^n$ $(c, n > 1)$ *has the full period if* c *is odd and* $a = 4k + 1$ *for some* k.

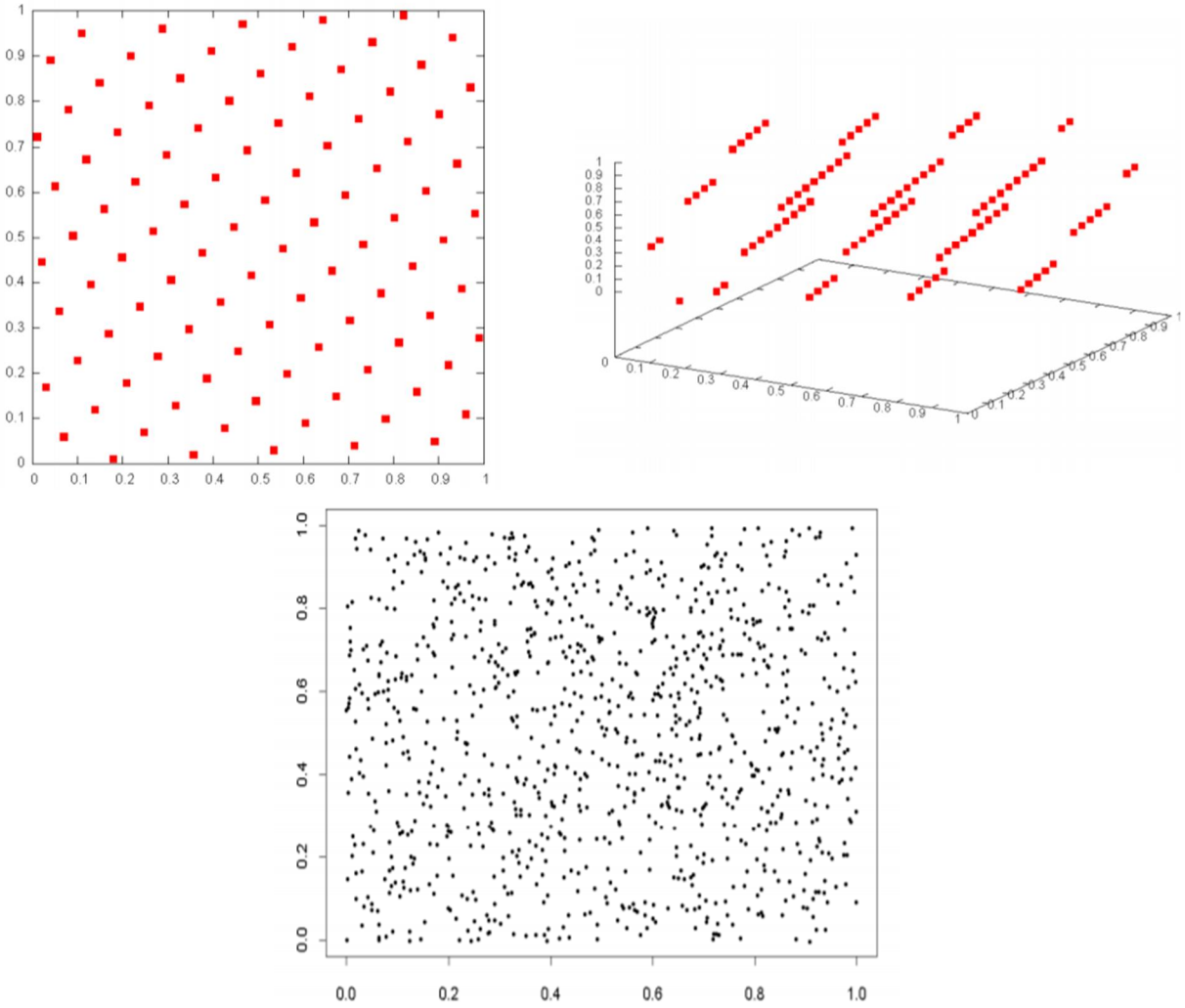
Theorem

$X_{i+1} = aX_i \bmod 2^n$ *had period at most* 2^{n-2}. *This can be achieved when* X_0 *is odd and* $a = 8k + 3$ *or* $a = 8k + 5$ *for some* k.

- For $m = 2^b$, and $c \neq 0$, maximum period is $P = m = 2^b$ – achieved when $(c, m) = 1$, $a = 4k + 1$, k integer
- For $m = 2^b$, and $c = 0$, maximum period is $P = m/4 = 2^{b-2}$ – achieved when X_0 odd and $a = 8k + 3$ or $a = 8k + 5$, k natural
- For m prime and $c = 0$, $P = m - 1$ – achieved when $\min\{k : m|a^k - 1\} = m - 1$

Remark: This algorithm demonstrates the point that pseudo-random generators are deterministic. The entropy for this generator is the initial $\mathbf{s}_0$ value, since a, c, and m will most likely

remain constant between uses. Once $\mathbf{s}_0$ is selected, the entire resulting sequence is solidified even before it is calculated. Although the sequence is unpredictable initially, once a seed value for the system is reused or an attacker recognizes the number sequence, then subsequent values can be predicted. The Boost library includes an implementation of a linear congruential generator (Chan, 2009).



4.3.3 General congruential generators

Linear Congruential Generators are a special case of generators defined by:

$$X_{i+1} = g(X_i, X_{i-1}, \dots) \bmod m .$$

where *g*() is a function of previous X_i's, $X_i \in [0, m-1], R_i = X_i / m$.

- *Quadratic congruential generator:* $g(X_i, X_{i-1}) = aX_i^2 + bX_i + c$.
- *Multiple recursive generators*: $g(X_i, X_{i-1}, \dots) = a_1 X_i + a_2 X_{i-1} + \cdots + a_k X_{i-k}$
- *Fibonacci generator*: $g(X_i, X_{i-1}) = X_i + X_{i-1}$.
- *Lagged Fibonacci generator*:

  A major problem with simple linear algorithms is that the period is limited by *m*. With the lagged Fibonacci algorithm, much larger periods can be obtained.

  The basic form of this algorithm is

$$X_i = X_{i-j} + X_{i-k} \ (mod\ m)$$

which, when *m* is prime and $k > j$, gives a period close to $m^k - 1$.

This algorithm can be used in conjunction with addition, subtraction, multiplication, or even exclusive-or (XOR). Because multiple past values are used instead of just the previous value as in the linear algorithms, values can be repeated inside the period without indicating a loop.

In the Boost library implementation of this algorithm, *p* is set to 44497, *q* is 21034, and the resulting period is around $2^{2300000}$ (Chan, 2009). A period of this size greatly reduces the likelihood of pattern recognition and secures the generator from the more basic forms of passive attacks.

- *Inversive congruential generators (due to Eichenauer Hermann).*

  This method produces uniform integers over [0, *m* − 1] by the relation

$$X_i = aX_{i-1}^{-1} + c \ (mod\ m)$$

where X^{-1} denotes the multiplicative inverse modulo *m* if it exists, or else is 0.

4.3.4 Combined linear congruential generators

Reason: Longer period generator is needed because of the increasing complexity of stimulated systems.

Approach: Combine two or more multiplicative congruential generators.

Let $X_{i,1}, X_{i,2}, \ldots, X_{i,k}$, be the i^{th} output from *k* different multiplicative congruential generators. The j^{th} generator $X_{\bullet j}$

- Has prime modulus m_j , the multiplier a_j and the period is m_j - 1
- Produced integer $X_{i,j}$ is approximately uniformly distributed on integers in [1,m_j - 1]
- $W_{i,j} = X_{i,j} - 1$ is approximately uniformly distributed on integers in [1,m_j - 2]
- Suggested form is

$$X_i = \left(\sum_{j=1}^{k} (-1)^{j-1} X_{i,j} \right) \bmod m_1 - 1 \qquad \text{Hence, } R_i = \begin{cases} \dfrac{X_i}{m_1}, & X_i > 0 \\ \dfrac{m_1 - 1}{m_1}, & X_i = 0 \end{cases}$$

The maximum possible period is

$$P = \frac{(m_1 - 1)(m_2 - 1)\ldots(m_k - 1)}{2^{k-1}}$$

Example:

For 32-bit computers, combining k = 2 generators with m_1 = 2147483563, a_1 = 40014, m_2 = 2147483399 and a_2 = 40692, the algorithm becomes:

Step 1: Select seeds:

- $X_{0,1}$ in the range [1, 2147483562] for the 1st generator,
- $X_{0,2}$ in the range [1, 2147483398] for the 2nd generator

Step 2: For each individual generator,

$$X_{i+1,1} = 40014 \times X_{i,1} \bmod 2147483563$$

$$X_{i+1,2} = 40692 \times X_{i,2} \bmod 2147483399$$

Step 3: $X_{i+1} = (X_{i+1,1} - X_{i+1,2}) \bmod 2147483562$

Step 4: Return

$$R_{i+1} = \begin{cases} \dfrac{X_{i+1}}{2147483563}, & X_{i+1} > 0 \\ \dfrac{2147483562}{2147483563}, & X_{i+1} = 0 \end{cases}$$

Step 5: Set *i* = *i*+1, go back to step 2.

Combined generator has period: $(m_1 - 1)(m_2 - 1)/2 \sim 2 \times 10^{18}$

4.3.5. Feedback shift register generator

Such a generator takes into consideration the binary representation of integers in registers of the computer. If a_i, $i = 1, ..., p,$ denote the binary digits of the random number, and c_i are given (not all zero) binary digits, then the digits a_i, of the new generated numbers are produced by

$$a_i = (c_p a_{i-p} + c_{p-1} a_{i-p+1} + ... + c_1 a_{i-1}) (mod\ 2).$$

This generator was introduced by Tausworthe.

In practice it has the form

$$a_i = (a_{i-p} + a_{i-p+q}) (mod\ 2) \qquad (*)$$

or, if we denote by $\oplus$ the binary exclusive-or operation, as addition of 0's and 1's modulo 2, the previous equation becomes

$$a_i = a_{i-p} \oplus a_{i-p+q}.$$

Note that this recurrence of bits a_i' s is the same as the recurrence of random numbers, (interpreted as l−tuples of bits), namely, $x_i = x_{i-p} \oplus x_{i-p+q}$.

If the random number has l binary digits ($l \le p$), and l is relatively prime to $2p - 1$, then the period of the l−tuples (i.e. of the sequence of generated numbers) is $2^p - 1$.

A variation of the Tausworthe generator, called *generalized feedback shift register* (GFSR), is obtained if we use a bit-generator in the form (*) to obtain an l−bit binary number and next bit-positions are obtained from the same bit-positions but with delay (by shifting usually to the left).

A particular GFSR is $x_i = x_{i-3p} \oplus x_{i-3q}$, $p = 521$, $q = 32$ which gives a period $2^{521}-1$.

Another generator of this kind is the so called *twisted GSFR generator*, which recurrently defines the random integers x_i as

$$x_i = x_{i-3p} \oplus A x_{i-p+q},$$

where A is a properly chosen p × p matrix .

A practical remark. Some other simple combinations could be used to produce "good" random numbers. Thus, if we use the following three generators

$x_i = 171x_{i-1} (mod\ 30269)$, $y_i = 172y_{i-1} (mod\ 30307)$, $z_i = 170z_{i-1} (mod\ 30323)$,

with positive initializations (seeds) $(x_0, y_0, z_0,)$ and take uniform (0, 1) numbers such as

$$u_i = (x_i /30269 + y_i/30307 + z_i /30323) (mod\ 1),$$

it can be shown that the sequence of u_i's has a period of order 10^{12} .

The Mersenne Twister is a very popular and widely used example of feedback shift registers in simulation and modeling (Nishimura, 2000). It is a good first choice when picking a pseudo random number generator. The Mersenne Twister can be classified as a *twisted generalized feedback shift register* (TGFSR), which has algorithms more tightly tied to matrices than strings. Adaptations exist both for making the generator faster and making it secure enough for cryptography. The benefits of this generator are rapid number generation, highly random sequences, and a large period. Because this generator is so popular, implementations and source code examples are easily available.

4.4. Testing a Random Number Generator

Many sequences that appear random may actually be easy to predict. For this reason, it is important to thoroughly test any generator that claims to produce random results. A fitting description of random in this context is as follows: a random sequence is one that cannot be described by a sequence shorter than itself (L'Ecuyer, 2007). Attempting to find these patterns by

intuition would be difficult if not impossible. Fortunately, many tests exist that can suggest if numbers in a sequence are random, and the algorithms in these tests are widely used outside of random numbers. Whenever professional forecasters make data-driven predictions, they apply formulas to determine the probability that the results were not random. These formulas can likewise be used to verify that a result was random, and the only thing that must change is the passing criteria. No test can conclusively prove randomness; the best that can be accomplished is that with enough testing, users of the generators can be confident that the sequence is random enough. There are scholars who believe that the source of a random sequence should dictate what tests to run. For example, true random generators tend to exhibit bias toward values, and this trait worsens as the hardware wears down. As a result, these scholars claim that if a random sequence comes from a true random generator, extra tests should be performed that check for bias (Kenny, 2005). Other scholars believe that random is random regardless of where it came from, and that it is appropriate to test all random sequences the same. In the following subsections several statistical and exploratory tests will be examined, as well as the major test suites NIST and Diehard.

Statistical Tests

One of the approaches to testing random number generators is leveraging the wide array of statistical formulas. With this approach, each test examines a different quality that a random number generator should have. For example, a random generator would go through a chi-squared test to ensure a uniform distribution, and then a reverse-arrangements test to see if the sequences containe any trends. Confidence in the generator's randomness is only gained after it passes an entire suite of tests which comes at it from different directions. These tests should be run on more than just a single sequence to ensure that the test results are accurate. Making the act of testing more difficult is the fact that failing a test does not indicate that a generator is not random. When outputs are truly random, then there will be some isolated sequences produced that appear non-random (Haahr, 2011). Tests need to be picked carefully and tailored to the context generators are needed in. The chi-squared, runs, next bit, and matrix based tests will be examined because of their popularity.

Chi-squared test. The chi-squared test is used to ensure that available numbers are uniformly utilized in a sequence. This test is easy to understand and set up, so it is commonly used (Foley, 2001). The formula for the test is: $x = \sum (\mathbf{O}_i - \mathbf{E}_i)^2 / \mathbf{E}_i$, where the summation is over all the available categories. ***O*** represents the actual number of entries in the category, and ***E*** is the expected number of entries. For example, if the random numbers were scattered one through six, then there would be six categories, and the number of times each value appeared in the sequence would become the values of O. When the resulting value is above the chosen significance level, then it can be said that the values in the sequence are uniformly distributed. Because this test is simple, it can be run many times on different sequences with relative ease to increase the chance of its accuracy (Foley, 2001).

The runs test. An important trait for a random sequence is that it does not contain patterns. The test of runs above and below the median can be used to verify this property (Foley, 2001). In this test, the number of runs, or streaks of numbers, above or below the median value are counted. If the random sequence has an upward or downward trend, or some kind of cyclical pattern, the test of runs will pick up on it. The total number of runs and the number of values above and below the median are recorded from the sequence. Then, these values are used to compute a z-score to determine if numbers are appearing in a random order. The formula for the score is: $z = ((u \pm 0.5) –$

MEAN$_u$) / σ_u, where σ denotes the standard deviation of the sequence. Just like the chi-squared test, a test for runs is easy to implement, and can be run frequently.

Next bit test. When testing pseudo random generators for cryptography applications, the next bit test is a staple. In its theoretical form, the next bit test declares that a generator is not random if given every number in the generated sequence up to that point there is an algorithm that can predict the next bit produced with significantly greater than 50% accuracy (Lavasani & Eghlidos, 2009). This definition makes the next bit test virtually impossible to implement, because it would require trying every conceivable algorithm to predict the next bit. Instead, it can be used after a pattern is discovered to cement the fact that a generator is insecure. Several attempts have been made to alter the next bit test so that it can be used as an actual test.

The universal next bit test developed in 1996 was the first to allow the next bit test to be administered, but it was shown that this test would pass non-random generators.

Later, **the practical next bit test** was developed and was shown to be as accurate as the NIST test suite at the time, if not more so (Lavasani & Eghlidos, 2009). However, this test required a large amount of resources to run, limiting its usefulness.

The next bit test remains relevant in cryptography because it has been proven that if a generator can pass the theoretical next bit test, then it will pass every other statistical test for randomness.

Matrix-based tests. Although the previous statistical test looks at the sequence of numbers linearly, a large portion of the more advanced testing techniques view it in terms of vectors and matrices. These tests will take the values of the generator, and sequentially add them into a hypercube, which is a cube with k dimensions (L'Ecuyer, 2007). From here the tests vary, although many of them conclude with a chi-squared calculation on their output (Chan, 2009). In a nearest pair test, Euclidean distances are computed and the nearest pairs are used to look for time-lagged patterns. It has been proposed that a large number of pseudo random generators based on circuits will fail the nearest pairs test, and it is one of the more stringent tests available. Multidimensional tests are often specifically tailored to look at random number generators, unlike the linear tests which are multipurpose. The calculations in matrix-based tests are complex, and will not be presented in this thesis.

Exploratory Analysis

Although it is true that determining randomness by intuition is a poor choice, visualizing random sequences is a good preliminary method to understand the data. There are several ways that sequences can be plotted in an attempt to bring out any oddities inside the random generator. By turning the sequence into a plot, it becomes more apparent that the given generator has some patterns in its sequences. In an ideal generator, the bitmap would seem like complete static, but in this example sections of black and white are clearly grouped. From this point, more specific tests can be decided on to determine just how severe the patterns actually are.

There is overlap between the problems that graphs can bring out and the problems that statistical techniques look for.

A run sequence plot is designed to look for trends in a sequence, much like the test for runs. To create this plot, actual sequence values on the y-axis are compared against the values' indexes in the sequence (Foley, 2001). If there are trends in the sequence, this plot will make them easier to see.

A histogram plot has a comparable purpose to the chi-squared test. It is composed of a bar graph, with the different categories on the x-axis plotted against the number of times that value appears on the y-axis. Any kind of non-uniformity will be brought out quickly this way, signaling that more tests concerning uniform distribution should be run.

Other exploratory graphs exist that look for unique types of problems.

A lag plot graphs a value on the y-axis against the value that came before it on the x-axis (Foley, 2001). The purpose of this plot is to expose outliers in the data. If the lag plot has too many outliers, there is most likely a problem with the generator.

Another unique exploratory tool is the **autocorrelation plot**. An autocorrelation plot examines the correlation of a value to the values that came before it at various intervals, called lags. If the plot displays no correlations between values at any lag, then the numbers are most likely independent of each other, which is a good indication of randomness. Using these exploratory plots allows analysts to get a feeling for the faults of a generator and better decide on which tests t run on the number sequences

NIST

Available at: https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final

Of the available suites for testing random number generators, the NIST suite reigns as the industry standard (Kenny, 2005). The NIST suite was designed to test bit sequences, with the idea that passing all NIST tests means that a generator is fit for cryptographic purposes. Even new true random number generators have their preliminary results run through the NIST battery to demonstrate their potential (Li, Wang, & Zhang, 2010). The NIST suite contains fifteen well-documented statistical tests (NIST.gov, 2008). Because cryptography has the most stringent requirements for randomness out of all the categories, a generator that passes the NIST suite is also random enough for all other applications. However when a generator fails the NIST suite, it could still be random enough to serve in areas such as gaming and simulation, since the consequences of using less than perfectly random information is small. NIST does not look at factors such as rate of production, so passing the NIST suite should not be the only factor when determining a generator's quality.

Diehard

Available at: https://webhome.phy.duke.edu/~rgb/General/dieharder.php

Another widely used suite of random number tests is known as Diehard. This suite was invented by George Marsaglia in 1995 (Kenny, 2005). It was made to be an update for the original random number test suite, Knuth. Knuth is named after Donald Knuth and was published in the 1969 book The Art of Computer Programming, Volume 2. Knuth's tests were designed before cryptography became a major industry, and the suite was later considered to be too easy to pass for situations where vast quantities of random numbers were needed. Diehard was designed to be more difficult to pass than Knuth's suite, fulfilling the role of a general-purpose battery for detecting non-randomness. All of the tests are available free online, so they can be easily used to test any number sequence (Marsaglia, 2005). The Diehard suite has not been updated since its inception in 1995, but is still a widely used test suite (Kenny, 2005).