Randomness and Pseudorandom Number Generators

Arjun Bhamra

November 7, 2023

Contents

1 Introduction

Before we begin, I'd like to provide some reasoning behind my topic selection. Throughout this course, Dr. McCuan has placed some emphasis on technology and simulations; indeed, we've had some homeworks on simulation components that have emphasized, for example, the Bernoulli and Binomial measures and distributions. In order to even create these simulations, we have to rely on software packages such as Python, R, Mathematica, etc., which all need to be able to generate random numbers. First, we will begin by exploring what it means to be a random number and what it means to be a pseudorandom number. Then, we will extend this idea to sequences of pseudorandom numbers and some examples. My end goal is to give all of you a greater appreciation for how a pseudorandom number generator (PRNG) works, and some insight into the nature of randomness as we know it.

2 Randomness, Pseudorandomness, and Physically Random Systems

2.1 What is a truly random number?

A guiding question: First and foremost, what is a random number? Attempt to define one for yourself right now before I attempt one myself.

Now that you have an idea of what a random number may mean, allow me to give you my definition of a truly random number. A **random number** is a number chosen as if by chance from some specified distribution such that selection of a large set of these numbers reproduces the underlying distribution. Almost always, such numbers are also required to be **independent**, so that there are no correlations between successive numbers. This last statement implies some sort of requirement relating to sequences, which we will get into later. For the time being, note that in order to "evaluate", in some way, the randomness of some system, we must get some non-singleton set of trials from said system.

Here we define a **physically random system** as a system that has no deterministic method of predicting a given output value at some point in measurement or after some action is performed.

The main distinction between a truly random number, and a "fake" or pseudorandom number, is that truly random numbers are defined uniquely by physical systems and entropy (as a measure of unpredictability or surprise of the number generation process), whereas a pseudorandom number is defined (necessarily) by some mathematical formula that can repeatably give the same number, if given the same inputs [3]. We will talk about this later.

2.2 Examples of Physically Random Systems (with caveats)

An interesting note: Consider how this topic and what I've said relates to paper 2. As a refresher, the (paraphrased) prompt was

> Given someone flips a coin and you do not see the outcome, does the number $\frac{1}{2}$ give you any information?

This idea is relevant because it showcases, quite clearly, the clearest example of a physically random system. I want to be clear that there are actually some implicit assumptions in the phrasing of this question that make it interesting: we *assume* that the person witnessing the coin flip has no knowledge of the wind speed, angle of flip, method used to flip, etc.

The Caveat: Given **all** of this information, it is theoretically possible to reliably ascertain the side of the coin that is face up, however, we **consider this system physically random** because it is **near impossible** to accurately get all of these initial conditions for a coin flip from someone's hand. In a similar way, we will also consider dice as physically random systems as well.

A final interesting example is that of Unix's /dev/random [4], which actually uses the physical systems of your device (such as device driver data, heat outputs, etc.) in order to generate a random number.

More examples: Try to think of other examples of physically random systems!

3 Mathematical Primers

Before we begin actually delving into Pseudorandom Number Generators (PRNGs), it makes sense to review a few key tools that are used frequently to define PRNGs. We will introduce these concepts here, and then explain why they are useful in the next section.

3.1 Sequences of numbers

3.1.1 What is a sequence?

A sequence of numbers is an **ordered list of numbers**, where each number in the sequence is called a **term**. We are going to be focusing on sequences with terms in $\mathbb{R}$. Sequences can be finite or infinite, but because we are discussing random number generation for chiefly practical purposes, we will focus on finite sequences. Some common notations for sequences are to use parentheses and comma separated terms such as

$$(a_1, a_2, a_3, \ldots)$$

A sequence can either start with a term a_0 indexed by 0 or a term a_1 indexed by 1, and the ellipses are commonly used to signify that the rest of the sequence

can be clearly ascertained from the given elements. This hints at the idea of using a current number to generate a subsequent number in some way, which we will discuss later. Finite sequences are usually given some starting number or numbers, which we will call a **seed** (or seeds), as well as some constraint on the upper bound of the sequence, which can either be a maximum allowed value, a set length of the sequence, etc. The way to notate this is as follows:

$$(a_k)_{k=0}^{n} = (a_0, a_1, \ldots, a_n)$$

for some upper iteration bound n. A well known example of a sequence is $(0, 1, 1, 2, 3, 5, 8, \ldots)$, which is exactly the infinite Fibonacci Sequence.

The most important way (for us) in which we can generate sequences is through...

3.1.2 Recurrence Relations

A **recurrence relation** is an equation that expresses each element of a sequence as a **function** of the preceding elements. Mathematically, this can be described as

$$\omega_n = \varphi(n, \omega_{n-1}, \ldots, \omega_{n-k}) \text{ for } n \geq k$$

This function has signature $\varphi : \mathbb{N} \times X \to X$, where the n encodes that we are generating the nth term in the sequence, and there are dependencies on previous terms $\omega_{n-1}, \ldots, \omega_{n-k}$.

To reiterate, a recurrence relation can have multiple seed/initial values, and can have zero to multiple dependencies on previous entries. Two valid sequences are

$$(a_n = 1)_{n=1}^{10}$$

$$(a_n = a_{n-1} + a_{n-2})_{n=2}^{5} \text{with } a_0 = 0, a_1 = 1$$

where the second sequence is, once again, the Fibonacci Sequence, this time defined up to 6 terms. The **order** of a recurrence is the difference between the highest and lowest subscripts of the equation.

3.2 Modular Arithmetic

Modular Arithmetic is a system of arithmetic for integers, where a number may "wrap around" when reaching a certain value called the **modulus**. A common usage of modulus is the standard 12 hour clock; if we go over 12, we simply wrap back around to 0 after every 12 hours. Let's go over some key concepts.

3.2.1 Congruence

Given an integer $n > 1$ called a **modulus**, two integers a and b are **congruent modulo** n if n is a divisor of their difference; that is, if there is an integer k such that

$$a - b = kn$$

This is denoted as

$$a \equiv b \pmod{n}$$

Note that $a - b = kn$ can be rewritten as $a = kn + b$, which should remind you of the Euclidean Division Algorithm from CS2050/51 and MATH 3012.

Important Note: Be sure to understand the difference between $a \equiv b \pmod{n}$ and $a \equiv b \mod n$. The 2nd equation applies the modulus only to b.

3.2.2 Other Useful Properties

The congruence relation satisfies all conditions of an **equivalence relation**, which guarantees certain properties. These are:

1. Reflexivity: $a \equiv a \pmod{n}$
2. Symmetry: $a \equiv b \pmod{n}$ if $b \equiv a \pmod{n}$
3. Transitivity: If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$

There are, of course, many other properties of the modulus, but these equivalence relation properties are key to know.
Note: Also recall that the congruence relation is compatible with addition, subtraction, and multiplication.

3.2.3 How does the modulus play with sequences?

To be brief: given some sequence where a modulus is present at the end, like $a_n = 2 \cdot a_{n-1} + 1 \pmod{n}$, we can see clearly that any values that are generated by iterating through this sequence will be at **most** $n - 1$ (because $0 \equiv n \pmod{n}$). This idea of a "cap" of sorts on the generable numbers is useful if we are trying to generate numbers within a specific interval (hint hint wink wink...)

4 Pseudorandom Number Generators

Since we have already mentioned pseudorandom numbers in some capacity, this section will be brief but will aim to formalize some of what we've talked about. As previously mentioned, we can use physically random systems as some form of a **Physically True Random Number Generator (PTRNG)**; this is an ideal situation for anyone that wants to get a truly random number as an output. However, the more common alternative is a Deterministic Random Number Generator (DRNG) or Pseudorandom Number Generator (PRNG), which extends a **seed** value to a (possibly) very long output sequence of random numbers in a deterministic way. Interestingly enough, while outside the scope of this discussion, there are "hybrid" PRNGs that use a Physically True Random Number Generator (i.e. some physically random system that can be sampled) to provide either a seed value or an intermittent modification to the generated sequence [3].

4.1 Why are sequences and recurrence relations relevant to random number generation?

I'd argue that this is probably one of the key takeaways of this entire paper; **all deterministic pseudorandom number generators are dependent on a recurrence relation**.

Note that, by definition, all DRNGs/PRNGs are **deterministic**; i.e. given some initial conditions, there needs to be a 100% reproducible way to get to the nth term of some sequence of some PRNG. This guarantee of reproducibility and state transition can be efficiently represented through, you guessed it, a recurrence relation; this can be easily shown for simple examples such as a counter and Linear Congruential Generator PRNGs, which we will see soon, but is even true for more complicated and industry standard algorithms such as the Mersenne Twister algorithm, as seen here.

4.2 Aside: Targeting a Distribution

As a very brief aside, we've gone through quite some content without really mentioning what distribution we are trying to randomly generate *from*. Here, the industry standard is, intuitively, the uniform distribution. It is particularly easy to work with, is convenient to engineer new PRNGs (and other tooling) around, and while outside the scope of this presentation, it is also easy (relative to other distributions) to convert the uniform distribution to any other distribution of your choice.

5 Period m of a PRNG

5.1 What is the period of a sequence of pseudorandom numbers?

One of the most important pieces of data we can get about any PRNG is its **period**, which is the length m of the sequence generated by a PRNG before it begins to repeat itself. As an example, lets say I have a PRNG that generates the ordered set $\{1, 2, 3, 5, 8\}$ and then repeats. We say this PRNG (which is comically bad) has a period of 5.

5.2 Why do we want to maximize the period?

A thought experiment: First, I'd like for you to think for yourself why this may be true. See if you can come up with any compelling reasons.

Hopefully, you've realized that the shorter a period, the quicker a sequence repeats itself. In the real world, this is undesirable for numerous reasons, most notable of which are reduced statistical quality and reliability. In this way, having a maximal period m is crucial, but we can't (usually) directly control the period of a PRNG. Instead, depending on the PRNG, we can carefully

choose inputs that guarantee a maximal period. We can see an example of this later on with Linear Congruential Generators.

6 More Concrete Examples of PRNGs

6.1 The Counter

The counter PRNG is exactly what it sounds like; a PRNG that uses a constant increment to traverse over some subset of the values between the seed X_0 and the modulus m. The form of this PRNG is

$$X_n = X_{n-1} + c \pmod{m}$$

for some parameters c and m. Here,

1. m is the modulus
2. c such that $(0 \leq c < m)$ is the increment
3. X_0 such that $(0 \leq X_0 < m)$ is the seed

Note that after generating this sequence, we have to divide by the modulus m to normalize our sequence to emulate the uniform distribution.

It is clear to see that the counter does not generate very good random numbers; with a short sequence, we can ascertain the increment c quite easily.

Maximizing the sequence length: For what inputs c and m do we have a maximal period? Do these values have to depend on each other?

The answer is actually $c = 1$ for any m; this essentially guarantees that you hit every single integer from X_0 to $m-1$, and then back around until you reach X_0 again.

6.2 The Linear Congruential Generator

The Linear Congruential Generator (LCG) is one of the easiest popular PRNGs that's still in use today for certain applications, and it is easy to teach owing to its simple mathematical representation using only addition, multiplication, and modulus. The most important thing to know about an LCG is its form, which can be represented simply by the recurrence relation

$$X_n = (aX_{n-1} + c) \mod m$$

for some parameters a, c, and m. Here,

1. m is the modulus
2. a such that $(0 < a < m)$ is the multiplier

3. c such that $(0 \leq c < m)$ is the increment

4. As before, X_0 such that $(0 \leq X_0 < m)$ is the seed

Note again that after generating this sequence, we have to divide by the modulus m to normalize our sequence to emulate the uniform distribution.

If $c = 0$, then this is a special case of the LCG that is called the Multiplicative Congruential Generator or Lehmer RNG. Special consideration should be made for the parameters a and m; the goal is to find an a and m such that the period is long and sufficiently random.

Brief Interlude: Could you come up with some 3-tuple (a, c, m) that creates an obviously and intentionally poor LCG? Consider the case where you are generating 100 numbers (this can play a part in your decision!)
Example: A possible answer could be $(1, 0, 101)$, which results in the sequence $1, 1, \ldots, 1$, which is simply 100 "1"s. Note how because the m I chose is above the length of the sequence I want to generate, it does not even come into play.

6.3 The Hull-Dobel Theorem for maximizing an LCG's period

The Hull-Dobel theorem is a theorem that posits 3 constraints on the 3-tuple (a, c, m) for an LCG to have a maximal period. We will not delve into the proof of this theorem as it is somewhat involved, but I'm going to attempt to justify the three conditions in an easier-to-understand manner.

Theorem 1 (Hull-Dobel Theorem)**.** *Let* $X_n = (aX_{n-1} + c) \mod m$ *be the recurrence that defines an LCG. Then, the sequence defined by this recurrence has full period* m *provided that*

1. m *and* c *are relatively prime, or coprime*

2. $a \equiv 1 \mod p$ *for all prime factors* p *of* m

3. $a \equiv 1 \mod 4$ *if* 4 *is a factor of* m

The proof in full can be found here, in the original paper [1]. It is somewhat involved and is outside the scope of this discussion.

6.4 Example of the Hull-Dobel Theorem

Let's now try using the Hull-Dobel Theorem with $a = 165, c = 3, m = 2^5$, which are three contrived choices for numbers to demonstrate an application of this theorem. The three important cases of the Hull-Dobel Theorem are:

1. m and c are relatively prime, or coprime; here, $3|168$ (where $a|b$ denotes that b is divisible by a), so the condition fails. $3 \cdot 56 = 165$.

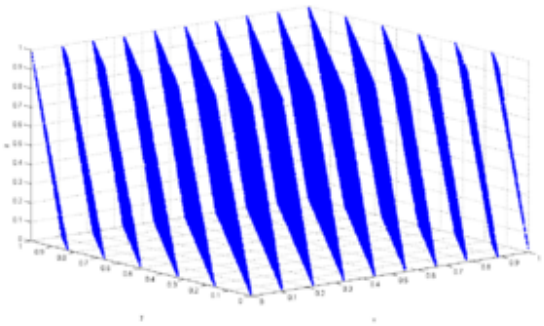

Figure 1: We can see that 15 distinct hyperplanes are created, telling us that IBM's RANDU fails to generate sufficiently random numbers in $n \geq 3$ dimensions.

2. $a \equiv 1 \mod p$ for all prime factors p of m: We can refactor this condition into the question, "is $a-1$ divisible by p?" Then, we can see that $2 \nmid 167$, so this condition fails.

3. $a \equiv 1 \mod 4$ if 4 is a factor of m: We can refactor this condition into the question, "is $a-1$ divisible by 4?" Then we can see that $4 \nmid 167$, so this condition fails.

As an aside, a portion of the original paper does recognize that the frequency with which people relied on m values of the form $2^a + b$ for some $a, b \in \mathbb{N}$, especially because of the nature of binary operating systems which are still in use today. This is partly the reason why the bottom two conditions are explored in quite some detail in the paper.

6.5 IBM RANDU and Poor Input Selection

A very famous example of a PRNG implementation that suffered due to poor inputs is IBM's RANDU [5]. The inputs to the LCG are the 3-tuple $(a = 65539, c = 0, m = 2^{31})$, and these inputs produce exclusively odd integers as outputs. **We can show that these inputs are quite lackluster.**

6.5.1 Modular Arithmetic for IBM RANDU

However, we're going to show with some modular arithmetic that it gets much worse! Note here that $65539 = 2^{16}+3$, and with this we're now going to attempt to randomly generate a point in 3D space. Lets use (i_x, i_y, i_z) to denote the 3 starting points. Given some seed for i_x, we get i_y and i_z by

$$i_y = (2^{16} + 3)i_x,$$

$$i_z = (2^{16} + 3)i_y = (2^{16} + 3)^2 i_x = (2^{32} + 6 \cdot 2^{16} + 9)i_x$$

$$= (2^{32} + 6 \cdot (2^{16} + 3) - 9)i_x = 6i_y - 9i_x \pmod{2^{32}}$$

We recall that we have to include the modulus operation, and $2^{32} \pmod{2^{31}} = 0$. Then, we have some point $9i_x - 6i_y + i_z$. Finally, we divide by 2^{31} as it is our modulus and thus our normalization constant. With this operation, note that $\frac{9i_x - 6i_y + i_z}{2^{31}} = 9x - 6y + z$ is divisible evenly by 2^{31}, so $9x - 6y + z$ is an *integer*. In fact, with some further effort it can be shown that actually this integer is limited to values of -5 and 9, which results in 15 clearly defined 2D planes in 3D space, as shown in 1.

7 Exercises

1. Solve some exercises from the Python file provided. For picking parameters, this paper may be a good resource [2]. The goal of this file is to create a series of examples that rely on the students creating their own simple Python implementations of an LCG for a variety of purposes and applications. Finally, they will try to find good input parameters (a, c, m) such that their LCG can pass the "die-hard" tests, which are a famous testing suite for statistical randomness. There would be zip file provided with a file to write their code in, a file to run for testing, and a README.md file for instructions.

2. Use a software tool and explore the random number generators it has to offer (R, for example, has documentation regarding which PRNGs it uses). Then, compare these PRNGs on a sample task such as randomly populating a cube of side length n.

3. If you agree with my interpretation of the distinction between a random number and pseudorandom number, try to come up with other interesting examples of physically random systems that exist naturally, and argue that they generate truly random numbers.

4. If you disagree with my interpretation of the distinction between a random number and pseudorandom number, try to come up with a counterexample that does not work with my definition.

For help with the last two, feel free to consult this link [3].

References

[1] T. E. Hull and A. R. Dobell. Random number generators. *SIAM Review*, 4(3):230–254, 1962.

[2] Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.

[3] Werner Schindler. A proposal for: Functionality classes for random number generators, Dec 1999.

[4] Sagar Sharma. What are /dev/random and /dev/urandom in linux?, Feb 2023.

[5] Peter Young. Physics 115/242 randu: A bad random number generator, Apr 2013.