IMPLEMENTATION OF TAUSWORTHE, L'ECUYER AND MERSENNE TWISTER PSEUDO-RANDOM NUMBER GENERATORS

Utsab Mitra
Group 293

ISyE 6644: Simulation and Modeling for Engineering and Science
Georgia Institute of Technology, USA

ABSTRACT

Pseudo-random number generators play a fundamental role in a wide range of computational applications, including stochastic simulation and modeling techniques such as the Monte Carlo method, as well as in areas like computer gaming, online gambling, and cryptographic systems. This project focuses on the implementation of three widely studied PRNGs—Tausworthe, L'Ecuyer's combined multiple recursive generator (MRG32k3a), and the Mersenne Twister—within a Python programming environment. To evaluate the suitability of these generators, a number of statistical tests were conducted to assess whether their outputs exhibit characteristics of being independently and identically distributed (i.i.d.) with a uniform distribution over the interval (0,1). The tests performed include both visual diagnostics (e.g., histograms, 2D and 3D scatter plots) and formal statistical procedures (e.g., Chi-square test, Kolmogorov-Smirnov test, runs tests, serial correlation analysis, and the Von Neumann ratio test). Additionally, an application-based test was carried out using a Monte Carlo simulation to estimate the value of π, providing a practical benchmark for evaluating the performance of the generators.

1. BACKGROUND & DESCRIPTION

A Pseudo-Random Number Generator (PRNG) is an algorithm that produces a sequence of numbers that appear to be random but are actually generated deterministically by a computer. While true random numbers are said to be nondeterministic, since they are impossible to determine in advance, pseudo-random numbers are not truly random because they are based on a predictable process and hence the term 'pseudo'.

A typical PRNG operates using a defined structure involving a finite set of internal states, denoted by S, and a state transition function $f : \mathrm{S} \to \mathrm{S}$. Alongside this, there is an output function $g : \mathrm{S} \to (0,1)$ that maps each internal state to a corresponding real number in the interval (0,1), which represents the generated random number. The process begins with an initial state S_0, commonly referred to as the seed, which is usually supplied by the user [1].

The generator produces a sequence of states and corresponding outputs recursively as follows[1]:

$$\begin{aligned} S_n &= f(S_{n-1}), \quad n = 1, 2, 3, \ldots \\ U_n &= g(S_n) \end{aligned} \tag{1}$$

An important aspect of this mechanism is its deterministic nature: if the same seed is used to initialize the generator, the sequence of outputs will always be the same. Due to the finite nature of the state space, the generator is guaranteed to eventually revisit a previously encountered state, causing the sequence to repeat. The smallest positive integer p for which the sequence returns to a former state after p steps is referred to as the period of the generator. While a longer period is generally desirable, it alone does not guarantee high-quality of randomness. Other statistical properties must also be evaluated to ensure the generator produces outputs suitable for simulation and modeling applications.

Some desirable properties of a generator are as follows [2]:

1. **Uniformity:** The numbers generated appear to be distributed uniformly on (0, 1);
2. **Independence:** The numbers generated show no correlation with each other;
3. **Replication:** The numbers should be replicable (e.g., for debugging or comparison of different systems);
4. **Cycle length:** It should take long before numbers start to repeat, in other words long period;
5. **Speed:** The generator should be fast to produce numbers;
6. **Memory usage:** The generator should not require a lot of storage.

2. THEORY

2.1 Tausworthe Pseudo-Random Number Generator

The Tausworthe pseudo-random number generator is a type of linear feedback shift register (LFSR)-based generator known for its efficient bit-level operations and suitability for software implementation [3]. It was introduced by Robert C. Tausworthe in 1965 and gathered significant interest due to cryptographic applications. The algorithm operates directly on bits to form random numbers.

Define a sequence $b_1, b_2, ...$ of binary digits by the recurrence

$$b_i = (c_1 b_{i-1} + c_2 b_{i-2} + ... + c_q b_{i-q}) (\text{mod } 2) \tag{2}$$

where $c_1, c_2, ..., c_{q-1}$ are constants that are equal to 0 or 1 and $c_q = 1$. In most applications of *Tausworthe generators*, for computational simplicity, only two of the c_j coefficients are nonzero [3], in which case **Eq. (2)** becomes

$$b_i = (b_{i-r} + b_{i-q}) (\text{mod } 2) \tag{3}$$

for integers r and q satisfying $0 < r < q$. Further evaluation of **Eq. (3)** reveals that the addition modulo 2 is equivalent to the *exclusive-or* operation on bits. Thus, **Eq. (3)** can be expressed as

$$b_i = \begin{cases} 0 & \text{if } b_{i-r} = b_{i-q} \\ 1 & \text{if } b_{i-r} \neq b_{i-q} \end{cases} \tag{4}$$

which can be denoted by $b_i = b_{i-r} \oplus b_{i-q}$ [3]. The generator is initialized with a sequence of bits $\{b_i\}$: $b_1, b_2, ..., b_q$. After the operation, a sequence of binary integers $W_1, W_2, ...$ are formed by stringing together l consecutive b_i's and transforming as a number in base 2. The parameter l is the *bit grouping parameter*. Thus,

$$W_i = b_{(i-1)l+1} b_{(i-1)l+2} ... b_{il} \quad \text{for } i = 2, 3, .. \tag{5}$$

The ith U(0,1) random number U_i is then defined by [3]:

$$U_i = \frac{W_i}{2^l} \quad \text{for } i = 1, 2, ... \tag{6}$$

The theory behind the Tausworthe generator is related to irreducible primitive polynomials over GF(2). A polynomial over Galois field of order 2 (GF(2)) is a polynomial whose coefficients are either 0 and 1. Consider the characteristic polynomial of recurrence given by

$$f(x) = x^q + c_1 + x^{q-1} + ... + c_{q-1} x + 1 \tag{7}$$

Such a polynomial is irreducible primitive if it does not have nontrivial factors like 1 and has order of $2^n - 1$ for some n. The order of a polynomial $f(x)$ over GF(2) is the smallest integer e for which $f(x)$

divides x^e+1. For example, x^2+x+1 is irreducible primitive with order $3 = 2^2-1$, while $x^2+1 = (x+1)^2$ is not even irreducible [4]. If l is relatively prime to 2^q-1, then the period of the W_i's, and also the U_i's will be 2^q-1. Thus, for a computer with 31 bits of data storage, the maximum period is $2^{31}-1$.

A Tausworthe sequence can be easily generated in hardware using Linear-Feedback Shift Register (LFSR). For example, the polynomial x^5+x^3+1 results in the generator $b_n = b_{n-1} \oplus b_{n-5}$. This can be implemented using LFSR as shown in **Figure 1**. The circuit consists of six registers, each holding one bit. On every clock cycle, each register's content is shifted out, and the new content is determined by the input to the register.

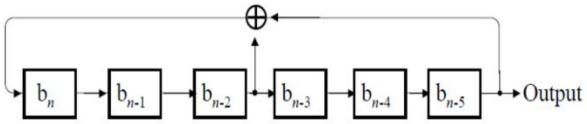

Figure 1: Linear Feedback Shift Register

The behavior and quality of a Tausworthe generator depend heavily on the choice of the parameters chosen. Some key considerations are:

- Seed: A binary vector of initial states. The seed must not be all zeros to avoid getting stuck in a degenerate state.
- Lag q and r: Determines the memory of the generator. Larger values typically lead to longer periods.
- Bit Grouping length l: The number of bits from the binary stream used to produce each uniform real number.

While the Tausworthe generator can be fast and efficient due to its use of simple bit-wise operations, and the sequence may produce good results over a complete cycle, it may not have satisfactory local behavior. It is also known to perform negatively on runs up and down test. Although the first-order serial correlation is almost zero, it is suspected that some primitive polynomials may give poor high-order correlations.

2.2 L'Ecuyer's Combined Multiple Recursive Generator

Linear congruential random number generators (LCGs) with prime moduli smaller than 2^{31} have the merit of being easily implemented on 32-bit computers, but has become increasingly unsatisfactory in modern computer intensive simulations. Even if these RNGs have periods several orders of magnitude, good structural properties are also needed [5]. A *multiple recursive generator* (MRG) of *order k* is defined by the linear recurrence:

$$\begin{aligned} x_n &= (a_1 x_{n-1} + ... + a_k x_{n-k}) \bmod m; \\ u_n &= x_n/m \end{aligned} \tag{8}$$

where m and k are positive integers, and each a_i belongs to $\mathbb{Z}_m = \{0, 1, ..., m-1\}$ [6]. Consider the characteristic polynomial for recurrence

$$P(z) = z^k - a_1 z^{k-1} - ... - a_k \tag{9}$$

The maximum period m^k-1 can be achieved if and only if m is prime and the polynomial is primitive, that is, the powers of z, modulo *P(z)* and m, run through all nonzero polynomials of degree less than k with coefficients in $\mathbb{Z}_m$. This can be achieved with only two nonzero coefficients, say a_r and a_k with $1 \le r < k$ [5]. When the coefficients are small, it is easier to implement the recurrence, however, a necessary condition for a good figure of merit is that $\sum_{i=1}^{k} a_i^2$ be large [6]. L'Ecuyer proposed combining MRGs to resolve these conflicting requirements. The algorithm will carefully select components so that the combined generator has good structural properties. The recurrence of such a *combined multiple recursive generator* (CMRG) can have many large coefficients even if there are only two small nonzero coefficients. The generic form of a CMRG described by L'Ecuyer is as follows:

$$x_{j,n} = (a_{j,1}x_{j,n-1} + ... + a_{j,k}x_{j,n-k}) \bmod m_j \tag{10}$$

for $j = 1, ..., J$, where the m_j are distinct primes and the jth recurrence has order k and period length $m_j^k - 1$. If $\delta_1, ..., \delta_J$ be arbitrary integers such that δ_j is relatively prime to m_j for each j, then define

$$Z_n = \left(\sum_{j=1}^{J} \delta_j x_{j,n} \right) \bmod m_1 \tag{11}$$

$$\tilde{U}_n = z_n / m_1 \tag{12}$$

k	m		
3	$2^{31} - 21069$,	$2^{31} - 43725$,	$2^{31} - 43845$
3	$2^{32} - 209$,	$2^{32} - 22853$,	$2^{32} - 30833$
3	$2^{32} - 32969$,	$2^{32} - 33053$	
3	$2^{63} - 21129$,	$2^{63} - 275025$	
3	$2^{64} - 239669$,	$2^{64} - 525377$,	$2^{64} - 539069$
3	$2^{127} - 601821$		
3	$2^{128} - 233633$		
5	$2^{31} - 22641$,	$2^{31} - 46365$,	$2^{31} - 59601$
5	$2^{32} - 18269$,	$2^{32} - 32969$,	$2^{32} - 56789$
5	$2^{32} - 88277$,	$2^{32} - 127829$	
5	$2^{63} - 19581$,	$2^{63} - 594981$,	$2^{63} - 745281$
5	$2^{64} - 460589$,	$2^{64} - 665033$,	$2^{64} - 959417$
7	$2^{31} - 6489$,	$2^{31} - 50949$,	$2^{31} - 55341$
7	$2^{32} - 5453$,	$2^{32} - 36233$,	$2^{32} - 37277$
7	$2^{32} - 40313$,	$2^{32} - 45737$	
7	$2^{63} - 52425$,	$2^{63} - 92181$	
7	$2^{63} - 152541$,	$2^{63} - 379521$	
7	$2^{64} - 51149$,	$2^{64} - 225257$	
11	$2^{32} - 30833$,	$2^{32} - 86357$	
13	$2^{32} - 9653$,	$2^{32} - 65129$	

Figure 2: Values of m and k

Figure 2 lists some values of m and k. All the m_j are selected so that $(m_j^k - 1)/(m_j - 1)$ is prime, and so that the least common multiple of $(m_j^k - 1)$ is $(m_j^k - 1)...(m_j^k - 1)/2^{J-1}$ (which is the largest possible length for the combination. In most case, $(m_j - 1)/2$ is also prime. L'Ecuyer's implementation of MRG are easier and more efficient when certain constraints are imposed on the coefficients $a_{j,i}$:

B. The product $a_{j,i}(m_j - 1)$ is less than than 2^{53}.

C. The coefficient $a_{j,i}$ satisfies $a_{j,i}(m_j \bmod a_{j,i}) < m_j$.

When either condition (B) or (C) is applied, requiring certain coefficients to be set to zero, it often becomes necessary to use combinations in order to achieve favorable figures of merit (M_T). This is because the capabilities of an MRG are inherently limited when such constraints are placed on its coefficients. Usually, figures of merit whose definition is described in L'Ecuyer, P. (1999) should be as close to 1 [5]. The choice of T in M_T is arbitrary, T = 8, 16, 32. It gives generators with good lattice structures in small, medium and large dimensions.

Figure 3 shows a table obtained from L'Ecuyer, P. (1999) of CMRGs with best values of M_{32} [5]. For this project, the CMRG with $J = 2$, $k = 3$, $m_1 = 2^{32} - 209$, $m_2 = 2^{32} - 22853$, $a_{11} = a_{22} = 0$ and with Condition B in effect was selected. This CMRG has $M_{32} = 0.63359$, which is the largest value of M_{32}. The CMRG is named as MRG32k3a and has a period length of 2^{191}. Note that the initial seed is 6-vector $(Z_{1,0}, Z_{1,1}, Z_{1,2}, Z_{2,0}, Z_{2,1}, Z_{2,2})$ and the first returned random number would be indexed as U_3 [3].

m_1 / m_2	Cd.	a_{12} / a_{21}	a_{13} / a_{23}		M_8	M_{16}	M_{32}
			$J = 2, k = 3$				
$2^{31} - 1$	B	1670453	−3445492				
$2^{31} - 21069$	B	2197254	−1967928		0.64954	0.63638	0.63442
$2^{31} - 21069$	B, C	26697	−94635				
$2^{31} - 43725$	B, C	17207	−32449		0.64585	0.63562	0.63257
$2^{32} - 209$	B	1403580	−810728				
$2^{32} - 22853$	B	527612	−1370589		0.68561	0.63940	0.63359
$2^{63} - 6645$	C	1754669720	−3182104042				
$2^{63} - 21129$	C	31387477935	−6199136374		0.66021	0.62700	0.62700
$2^{63} - 21129$	C	18010381385	−5837607579				
$2^{63} - 275025$	C	3444163371	−3141078384		0.63477	0.63393	0.61218
			$J = 2, k = 5$				
m_1 / m_2	Cd.	a_{12} / a_{21}	a_{14} / a_{23}	a_{15} / a_{25}	M_8	M_{16}	M_{32}
$2^{31} - 22641$	B+	343567	1162681	−1838005			
$2^{31} - 46365$	B+	1358258	449185	−619098	0.65922	0.63317	0.62644
$2^{32} - 18269$	B	1154721	1739991	−1108499			
$2^{32} - 32969$	B	1776413	865203	−1641052	0.66340	0.61130	0.61130
			$J = 3, k = 7$				
m_1 / m_2 / m_3	Cd.	a_{11} / a_{22} / a_{33}	a_{14} / a_{25} / a_{36}	a_{17} / a_{27} / a_{37}	M_8	M_{16}	M_{32}
$2^{31} - 6489$	B	1004479	719020	−3542530			
$2^{31} - 50949$	B	3259273	533655	−3434331			
$2^{31} - 55341$	B	1193874	2375699	−589692	0.70833	0.61275	0.61275
$2^{32} - 5453$	B	1025652	1495670	−1555702			
$2^{32} - 36233$	B	1790017	1978132	−1015534			
$2^{32} - 37277$	B	1227190	1019889	−847163	0.68699	0.64588	0.64251

Figure 3: CMRGs with good Figure of Merit

2.3 Mersenne Twister Pseudo-Random Number Generator

Makoto Matsumoto and Takuji Nishimura in 1997 proposed a new random number generator called the Mersenne Twister (MT), and this project deals with the most commonly used version of the Mersenne Twister algorithm, MT19937 based on the Mersenne prime $2^{19937} - 1$ which is also the period of the generator [7]. It is a variant of the Twisted Generalized Feedback Shift Register (TGFSR) algorithm which was introduced by Matsumoto and Kurita in 1992, and then further improved upon in 1994. The generator exhibits excellent statistical properties and performs well in maintaining multidimensional uniformity. It is relatively fast, especially when compared to other algorithms of similar quality. This makes it a popular choice for simulations that demand large volumes of high-quality random numbers [8].

The MT algorithm generates a sequence of word vectors (w-dimensional row vectors over the two-element field $\mathbb{F}_2 = \{0, 1\}$), which are considered to be uniform pseudorandom integers between 0 and $2^w - 1$. Dividing by $2^w - 1$ gives realization in [0,1]. The algorithm is based on the following linear recurrence

$$\mathbf{x}_{k+n} := \mathbf{x}_{k+m} \oplus (\mathbf{x}_k^u | \mathbf{x}_{k+1}^l) A, \text{ for } k \geq 0, m < n \in \mathbb{N} \tag{13}$$

Each value x_i has a word length of w that is represented by w 0–1 bits (with the least significant bit at the right). The expression $(\mathbf{x}_k^u | \mathbf{x}_{k+1}^l)$ denotes the concatenation of $w - r$ most significant bits of $\mathbf{x}_k$ and the r least significant bits of $\mathbf{x}_{k+1}$ for some $0 \leq r \leq w$. It can be calculated with a bit-wise AND operator. $\mathbf{A}$ is a $w \times w$ bit-matrix given by

$$A = \begin{bmatrix} 0 & 1 & 0 & ... & 0 & 0 \\ 0 & 0 & 1 & ... & 0 & 0 \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ . & . & . & ... & . & . \\ 0 & 0 & 0 & ... & 0 & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & ... & a_1 & a_0 \end{bmatrix} \tag{14}$$

where all the entries of A_i are either 0 or 1. Then the calculation of xA can be done using only bitwise operations:

$$\mathbf{x}A = \begin{cases} \text{shiftright}(\mathbf{x}) & \text{if } x_0 = 0 \\ \text{shiftright}(\mathbf{x}) \oplus a & \text{if } x_0 = 1 \end{cases} \tag{15}$$

where $a = (a_{w-1}, a_{w-2}, ..., a_0)$ and $\mathbf{x} = (x_{w-1}, x_{w-2}, ..., x_0)$. Thus, the entire operation on the recurrence is a calculation involving bitshift, bitwise XOR, bitwise OR, and bitwise AND operations. The MT is cascaded with a tempering transform to compensate for the reduced dimensionality of equidistribution (because of the choice of A being in the rational normal form) [8]. This *tempering* is defined as

$$\begin{aligned} y &:= \mathbf{x} \oplus (\mathbf{x} >> u) \\ y &:= y \oplus ((y << s) \text{ AND } \mathbf{b}) \\ y &:= y \oplus ((y << t) \text{ AND } \mathbf{c}) \\ z &:= y \oplus (y >> l) \end{aligned} \tag{16}$$

where *l, s, t* and u are integers, $\mathbf{b}$ and $\mathbf{c}$ are selected bitmasks of word size. The operation $(\mathbf{x} >> u)$ denotes bitwise shift right by u-bit and $(\mathbf{x} << u)$ denotes bitwise shift left by u-bit.

The 32-bit implementation of the MT has the following coefficients:

$$\begin{aligned} w, n, m, r &= 32, 624, 397, 31 \\ a &= \text{0x9908b0df} \\ u, s, t, l &= 11, 7, 15, 18 \\ \text{b} &= \text{0x9d2c5680} \\ \text{c} &= \text{0xefc60000} \end{aligned}$$

The state needed for a MT implementation is an array of n values of w bits each. To initialize the array, a w-bit seed value is used to supply x_0 through x_{n-1} by setting x_0 to the seed value and thereafter setting

$$x_i = f \times (x_i \oplus (x_{i-1} >> (w-2))) + i \tag{17}$$

for i from 1 to $n-1$. The multiplier f taken as 1812433253 [9].

The long periodicity of MT makes it suitable for simulations, statistics, and cryptography. In terms of statistical qualities, it demonstrates uniformity and independence. As a result, it creates random numbers faster than hardware-implemented methods [7]. MT is used as default PRNG by various software and libraries, including in Python.

3. ALGORITHM DESIGN AND CODE IMPLEMENTATION

This section examines the algorithms utilized for the implementation of the three pseudo-random number generators: Tausworthe, L'Ecuyer, and Mersenne Twister. With the mathematical foundation discussed in the previous section, this section focuses on explaining the logical constructs behind each algorithm, highlighting their unique characteristics and operations. Corresponding code blocks demonstrating the Pythonic implementation can be found in the **Appendix 8.2** section. These code blocks are intended to provide a clear and concise representation of how the algorithms are translated into programming logic. Further, the code files are also included at the time of project submission.

3.1 Tausworthe Pseudo-Random Number Generator

The Tausworthe pseudo-random number generator (PRNG) requires several input parameters: a binary seed provided in a string format, integer lag parameters (r and q), and an integer bit-grouping parameter l. Values of r and q must be carefully chosen such that r and q are relatively prime which ensures longer period. Once the Tausworthe class is initialized, the algorithm generates a specified number of uniform random numbers by accepting an integer input n.

Algorithm 1 outlines the detailed procedure of the process. Initially, the algorithm calculates the number of additional bits required to extend the *seed*. It then initializes a *bins_array*, which undergoes a bitwise XOR operation to iteratively append new bits. The specifics of this XOR operation have been previously discussed (*See Section* **2.1**). Subsequently, the binary values in the array are converted to decimal format [10], and finally, these decimal values are normalized to generate uniform random numbers Unif(0,1). The Python code snippet is shown in **Appendix 8.2.1**.

Algorithm 1 Tausworthe PRNG

Require:
Binary *seed* (in string format)
Secondary lag parameter r (integer)
Lag parameter q (integer)
Bit grouping parameter l (integer)

Ensure: Uniform random numbers Unif$(0, 1)$

1: Validate inputs:
Ensure seed is binary and in string format.
Ensure r and q are integers such that $\gcd(r, q) = 1$ and $0 < r < q$.
Ensure length(seed) $> r$ and length(seed) $\geq q$.
Ensure l is an integer.

2: Calculate number of bits to add:

$$\text{bits\_to\_add} = (n \cdot l) - \text{length(seed)}$$

3: Initialize a bit array:
bins_array $\leftarrow$ *seed* that is concatenated with zeros of length *bits_to_add*.

4: **for** $i =$ length(seed) to length(bins_array) **do**
Perform XOR operation:

$$\text{bins\_array}[i] = \text{bins\_array}[i - r] \oplus \text{bins\_array}[i - q]$$

5: **end for**

6: Reshape *bins_array* to a 2D array with shape (n, l).

7: Convert binary values in each row to decimal.

8: Normalize decimal values to obtain Uniform(0, 1):

$$\text{unif\_array} = \frac{\text{decimal\_value}}{2^l}$$

return unif

3.2 L'Ecuyer Pseudo-Random Number Generator

L'Ecuyer's CMRG requires two seed vectors for initialization. The report concentrates on the specific implementation of MRG32k3a, with the corresponding seed values detailed in Section **2.2**. The algorithm for implementation is defined in **Algorithm 2**. Note that the algorithm adopts the Pythonic indexing values, that is the first element in an array has the index of 0, and so forth. Upon initializing the class with these seed values, the process begins by constructing two arrays, X_1 and X_2. These arrays are created by concatenating the respective seeds, S_1 and S_2, with n zeros, where n is the user-defined number of uniform random variables Unif(0,1) to be generated. The next step involves computing the subsequent values in the arrays using a linear congruential operation. Finally, the uniform random numbers are generated following Eq (12). The Python code snippet is shown in **Appendix 8.2.2**.

Algorithm 2 MRG32k3a PRNG

Require:
Constants:
$a_1 = [0, 1403580, -810728]$
$a_2 = [527612, 0, -1370589]$
Modulus parameters:
$m_1 = (2^{32}) - 209$
$m_2 = (2^{32}) - 22853$

Ensure: Uniform random numbers Unif$(0, 1)$

1: Initialize seeds $S1$ and $S2$:
Ensure all elements of $S1$ are less than m_1.
Ensure all elements of $S2$ are less than m_2.
2: Define arrays $X1$ and $X2$:
$X1 = S1$ concatenated with n zeros.
$X2 = S2$ concatenated with n zeros.
3: Initialize unif:
Create an array unif of n zeros to store the generated numbers.
4: **for** $i = 3$ to $n + 3$ **do**
5: Compute $X1[i]$ using linear congruential operation:

$$X1[i] = \left(\sum_{j=0}^{2} a1[j] \cdot X1[i - j - 1] \right) \mod m1$$

6: Compute $X2[i]$ using linear congruential operation:

$$X2[i] = \left(\sum_{j=0}^{2} a2[j] \cdot X2[i - j - 1] \right) \mod m2$$

7: Compute unif[3]:

$$\text{unif}[i] = \frac{(X1[i] - X2[i] \mod m1)}{m1 + 1}$$

8: **end for**
return unif

3.3 Mersenne Twister Pseudo-Random Number Generator

As outlined in Section 2.3, **Algorithm 3** demonstrates the implementation of the Mersenne Twister PRNG. The process begins with the initialization of the class using pre-defined constants required in the procedure. An initial state vector, *mt*, is created, with its first element set to the user-provided seed value. The algorithm then proceeds to populate the state vector through a straightforward Linear Congruential step. This step includes the multiplier f = 81243325 [9]. Following this initialization phase, the subsequent operations of Twisting and Tempering are applied to refine and generate the desired pseudo-random numbers. Note the use of the function roll in the Twisting procedure. The function

enables the array *mt* to loop around itself with the second argument specifying which position to loop around from. The Python code snippet is shown in **Appendix 8.2.3**.

Algorithm 3 Mersenne Twister (MT19937) Algorithm

Require:
Parameters:
$w = 32$, $N = 624$, $M = 397$, $r = 31$
MATRIX_A = 0x9908b0df (constant)
$u = 11$, $s = 7$, $t = 15$, $l = 18$ (tempering parameters)
d = 0xffffffff (mask for 32-bit integers)
f = 1812433253 (multiplier)
B = 0x9d2c5680 (Tempering mask)
C = 0xefc60000 (Tempering mask)
UPPER_MASK = 0x80000000 (To extract the higher (w - r) significant bits)
LOWER_MASK = 0x7fffffff (To extract the lower r significant bits for the next number)
mag01 = [0x0, MATRIX_A] (matrix **A**)
Ensure: Uniform random numbers Unif(0, 1)
Initialize a counter $mti = N$
Initialize state vector mt with N zeros:
Set mt[0] = *seed*
for $i = 1$ to $N - 1$ **do**
(Update the state vector using a simple Linear Congruential method)

$$\text{mt}[i] = f \cdot (\text{mt}[i-1] \oplus (\text{mt}[i-1] \gg (w-2))) + i$$

end for
procedure TWIST
Compute intermediate y:

$$y = (\text{mt \& UPPER\_MASK}) + (\text{roll(mt, -1) \& LOWER\_MASK})$$

Update mt:

$$\text{mt} = \text{roll(mt, -M)} \ \oplus (y >> 1) \oplus \text{mag01[y AND 0x1]}$$

Reset $mti = 0$.
end procedure
procedure TEMPER
if mti $\geq N$ **then**
Perform Twist().
end if
Initialize y as mt[mti]; increment mti.
Perform tempering operations:

$$y \leftarrow y \oplus (y >> u)$$
$$y \leftarrow y \oplus ((y << s) \text{ AND B})$$
$$y \leftarrow y \oplus ((y << t) \text{ AND C})$$
$$y \leftarrow y \oplus (y >> l)$$

return $y/(2^w - 1)$.
end procedure
procedure GENERATE UNIFORM NUMBERS
Initialize unif array with n zeros.
for $i = 0$ to $n - 1$ **do**
unif[i] $\leftarrow$ Temper().
end for
return unif.
end procedure

4. STATISTICAL ANALYSIS

4.1 Performance Metrics

Performance metrics of the three PRNGs are analyzed, specifically on the time required to generate a set of random numbers. **Table 1** provides a summary of the time taken by each generator to produce 1 million random numbers. While all the generators exhibit impressive speed, it is observed that the Tausworthe generator performs relatively slower compared to the others. This may be attributed to the conversion of strings to integers during the bitwise operations. Despite this limitation, the Python implementation remains highly efficient and reliable for practical use.

PRN Generator	Time
Tausworthe	10.587202 s
L'Ecuyer	1.922203 s
Mersenne Twister	4.015129 s

Table 1: Performance metric

4.2 Tests for Distribution Uniformity

The Distribution Uniformity tests examine whether the generated numbers adhere to the expected Uniform distribution.

4.2.1 Histogram

A histogram of a Unif(0, 1) distribution typically exhibits a flat, rectangular shape across the interval, indicating that all values within the range are equally likely and occur with consistent frequency. **Figure 4** demonstrates this characteristic. (See **Appendix 8.2.4** for code implementation)

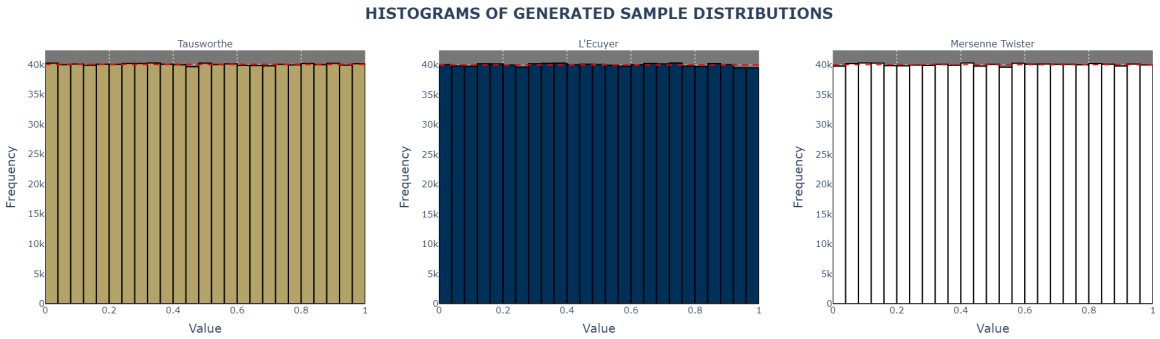

Figure 4: Histograms of generated sample distribution

4.2.2 Chi-Square Goodness-of-Fit Test

A goodness-of-fit test is a statistical hypothesis test that is used to assess formally whether the observations are an independent sample from a particular distribution [3]. One such goodness-of-fit test is the chi-square test developed by K. Pearson in 1900. A chi-square test may be thought of as a more formal comparison of a histogram with the fitted density or mass function. The histogram in Section 4.2.1 serves be basis for the chi-square test. The chi-square test is defined for the hypothesis:

H_0: The data follows a Uniform distribution
H_1: The data do not follow a Uniform distribution

To compute the test statistic used in the chi-square goodness-of-fit test, the range of possible observed values of the quantity of interest were partitioned into k bins. For the sake of analysis, k is chosen as 5. Thus, the bins are [0.0, 0.2), [0.2, 0.4), [0.4, 0.6), [0.6, 0.8), [0.8, 1.0]. The number of observations that fell in the bins were noted. The test statistic is thus defined as

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} \tag{18}$$

where O_i denotes the number of observations that fall in bin i, and E_i denotes the number of observations expected to fall in bin i. For the case when the distribution is Uniform, $E_i = 20,000$ for 1 million sample size and $bin = 5$. The test statistic follows a chi-square distribution with k-s-1 degrees of freedom, where k is the number of non-empty cells, and c is the number of estimated parameters. In this case, degrees of freedom is 4. Therefore, the hypothesis that the data are from a Uniform distribution is rejected if

$$\chi^2 > \chi^2_{1-\alpha,k-s-1} \tag{19}$$

where $\chi^2_{1-\alpha,k-s-1}$ is the chi-square critical value with $k - s - 1$ degrees of freedom at confidence level $100(1-\alpha)\%$. The results of the Chi-Square Goodness-of-Fit test on the generator are presented in **Figures 5 - 7**. (See **Appendix 8.2.5** for code implementation)

```
Testing Goodness-of-Fit for the Tausworthe PRN Generator
Hypothesis:
H₀: The random numbers are identically distributed
H₁: The random numbers are not identically distributed
----------------------
The intervals for the range (0, 1) divided into 5 sections:
[(0.0, 0.2), (0.2, 0.4), (0.4, 0.6), (0.6, 0.8), (0.8, 1.0)]
The expected number of random numbers in each bin is 200000
The observed number of random numbers in each intervals: [200099 200580 199768 199289 200264]
The chi_squared statistic = 4.87621
The critical chi-squared value at confidence level 95.0% = 9.487729036781154
We see that 4.87621 < 9.487729036781154. Thus,
----> We fail to reject the null hypothesis H₀ <----
The sequence of numbers are in fact identically distributed
```

Figure 5: Testing the Goodness-of-Fit for the Tausworthe PRNG

```
Testing Goodness-of-Fit for the L'Ecuyer PRN Generator
Hypothesis:
H₀: The random numbers are identically distributed
H₁: The random numbers are not identically distributed
----------------------
The intervals for the range (0, 1) divided into 5 sections:
[(0.0, 0.2), (0.2, 0.4), (0.4, 0.6), (0.6, 0.8), (0.8, 1.0)]
The expected number of random numbers in each bin is 200000
The observed number of random numbers in each intervals: [200065 200392 199969 200544 199030]
The chi_squared statistic = 6.97843
The critical chi-squared value at confidence level 95.0% = 9.487729036781154
We see that 6.97843 < 9.487729036781154. Thus,
----> We fail to reject the null hypothesis H₀ <----
The sequence of numbers are in fact identically distributed
```

Figure 6: Testing the Goodness-of-Fit for the L'Ecuyer PRNG

```
Testing Goodness-of-Fit for the Mersenne Twister PRN Generator
Hypothesis:
H₀: The random numbers are identically distributed
H₁: The random numbers are not identically distributed
----------------------
The intervals for the range (0, 1) divided into 5 sections:
[(0.0, 0.2), (0.2, 0.4), (0.4, 0.6), (0.6, 0.8), (0.8, 1.0)]
The expected number of random numbers in each bin is 200000
The observed number of random numbers in each intervals: [200263 199539 199864 200244 200090]
The chi_squared statistic = 1.8391099999999998
The critical chi-squared value at confidence level 95.0% = 9.487729036781154
We see that 1.8391099999999998 < 9.487729036781154. Thus,
----> We fail to reject the null hypothesis H₀ <----
The sequence of numbers are in fact identically distributed
```

Figure 7: Testing the Goodness-of-Fit for the Mersenne Twister PRNG

4.2.3 Kolmogórov-Smirnov Test

The Kolmogórov-Smirnov (K-S) test for goodness-of-fit is a non-parametric statistical method used to compare an empirical distribution function with the cumulative distribution function of a hypothesized distribution [3]. One of its key advantages is that the distribution of the K-S test statistic does not rely on the underlying cumulative distribution function being evaluated. Additionally, the K-S test is exact, unlike the chi-square goodness-of-fit test, which requires a sufficiently large sample size for its approximations to be valid [11]. Despite these strengths, the K-S test has notable limitations:

1. It is applicable only to continuous distributions.
2. The test tends to be more sensitive near the center of the distribution and less so at the tails.
3. A significant drawback is the requirement for the distribution to be fully specified. If parameters such as location, scale, or shape are estimated from the data, the critical region of the K-S test becomes invalid and typically needs to be determined through simulation.

The K-S test is defined for the hypothesis:

H_0: The data follows a Uniform distribution
H_1: The data do not follow a Uniform distribution

To define the K-S test statistic, the cumulative distribution function of the hypothesized Unif(0, 1) distribution, $\hat{F}(X_i)$ was utilized, and compute the following

$$\begin{aligned} D^+ &= max_{1\leq i\leq n}\{\frac{i}{n} - \hat{F}(X_i)\} \\ D^- &= max_{1\leq i\leq n}\{\hat{F}(X_i) - \frac{i-1}{n}\} \end{aligned} \tag{20}$$

and finally letting

$$D = max\{D^+, D^-\} \tag{21}$$

The hypothesis regarding the distributional form is rejected if the test statistic, D, is greater than the critical value obtained from a K-S table at the confidence level $100(1-\alpha)\%$. After performing the test on the 1 million random numbers generated, the **Figures 8 - 10** shows the result. (See **Appendix 8.2.6** for code implementation)

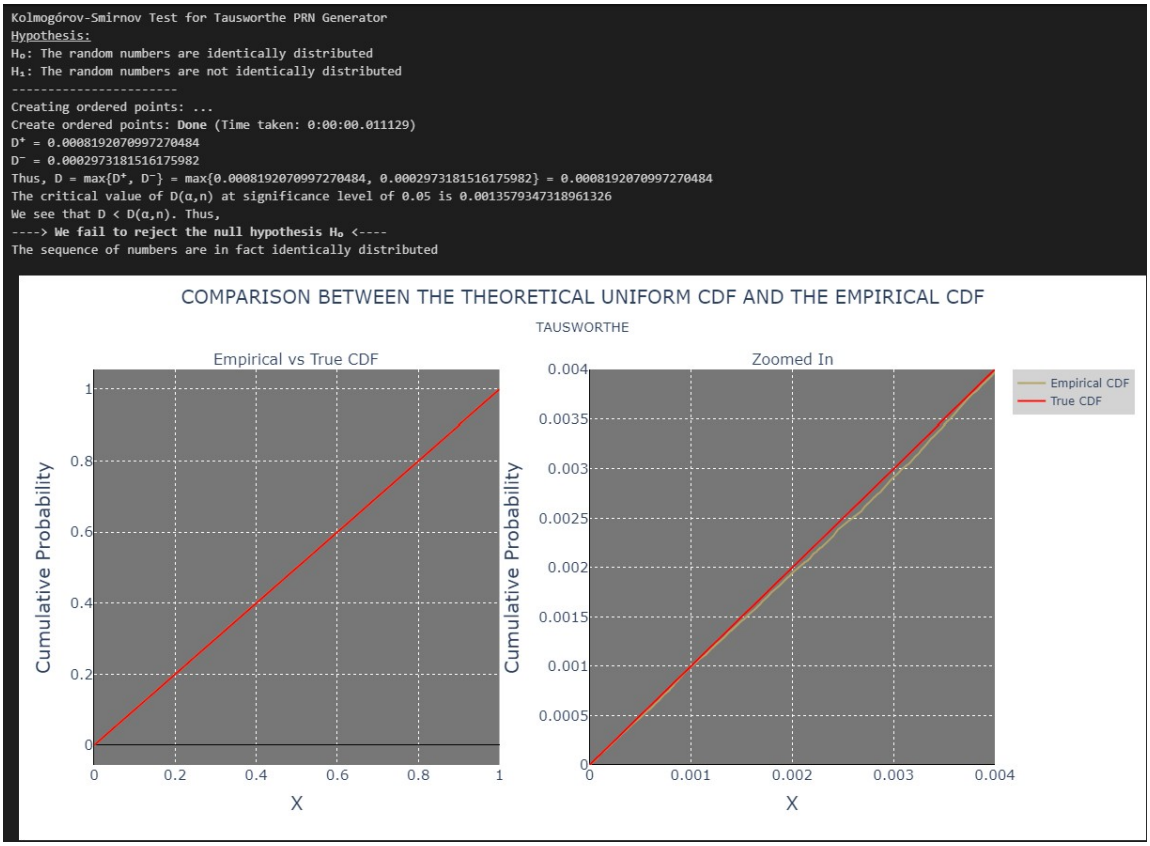

Figure 8: Kolmogórov-Smirnov (K-S) test for the Tausworthe PRNG

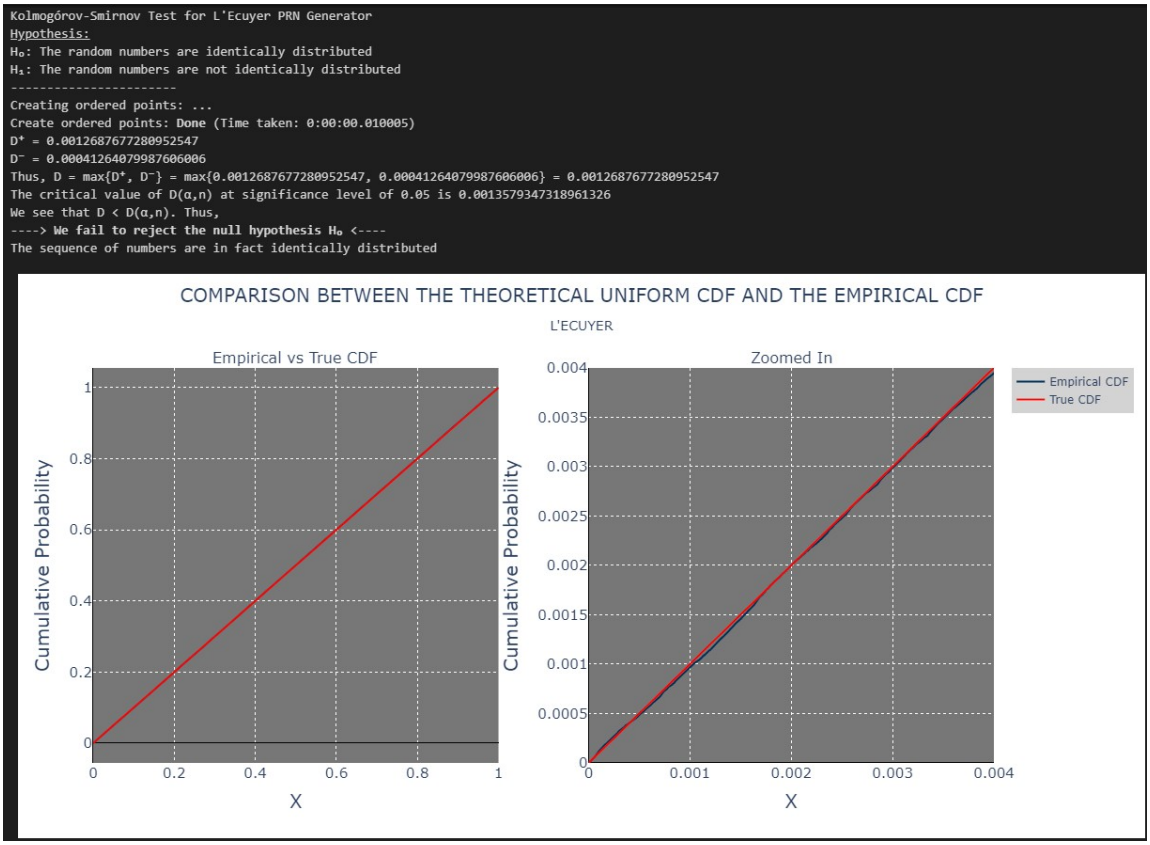

Figure 9: Kolmogórov-Smirnov (K-S) test for the L'Ecuyer PRNG

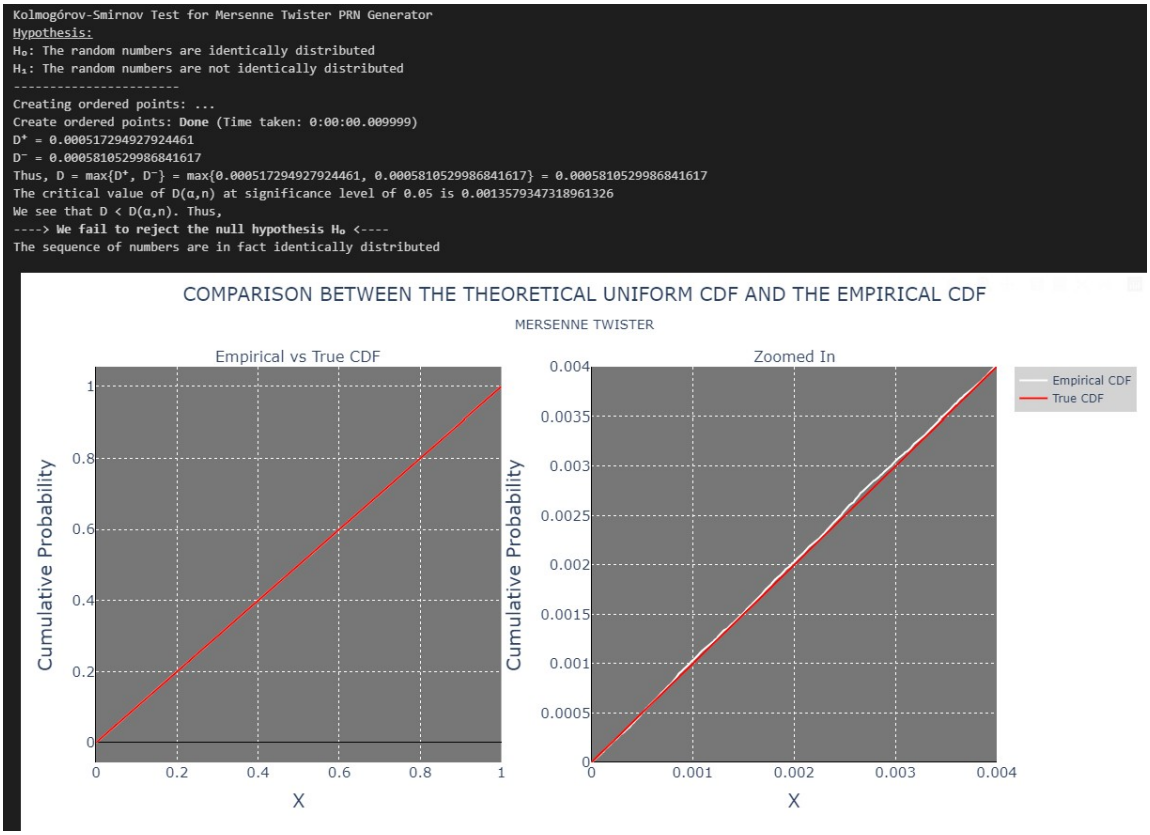

Figure 10: Kolmogórov-Smirnov (K-S) test for the Mersenne Twister PRNG

4.3 Tests for Independence

This section focuses on tests for Independence. These tests will evaluate whether the generated numbers are free from correlation.

4.3.1 Serial Correlation Test

Serial correlation, or autocorrelation occurs when a variable and a lagged version of itself are observed to be correlated over periods of time. It quantifies the similarity (or dissimilarity) between observations of a random variable at different points in time. It is an essential mathematical tool for identifying repeating patterns or hidden periodicities. Serial Correlation is widely used in signal processing, time domain and time series analysis [12].

The most common way of measuring a linear correlation is by computing the Pearson correlation coefficient (ρ).

$$\rho = \frac{Cov(R_i, R_{i+k})}{\sqrt{Var(R_i)}\sqrt{Var(R_{i+k})}} \tag{22}$$

The Pearson coefficient, ρ lies between -1 and +1 that measures the strength and direction of the relationship between two variables. In the equation above, the coefficient is measuring the correlation between the i-th number and the $(i+k)$-th. k = 1 is referred as the lag-1 correlation of R_i's.

Note that uncorrelated does not necessarily mean random. Data that has significant correlation is not random. However, data that does not show significant correlation can still exhibit non-randomness in other ways. Correlation is just one measure of randomness.

Eq. 22 can be slightly modified since correlation of a Unif(0,1) distribution is being tested here. Suppose the correlation coefficient between consecutive random numbers from a sample distribution with lag k is of interest. Then the samples are represented as:

$$\begin{aligned} X &= \{R_1, R_2, ..., R_{n-k}\} \\ Y &= \{R_{k+1}, R_{k+2}, ..., R_{n-k}, R_{n-k+1}, ..., R_n\} \end{aligned} \tag{23}$$

The coefficient can be re-written as:

$$\rho = \frac{(E[X - E[X]))(E[Y - E[Y]])}{\sqrt{Var(X)}\sqrt{Var(Y)}} \tag{24}$$

where the definition of Cov(X, Y) has been substituted. And for a Unif(0,1) distribution, the expected value and the variance for all i's are:

$$\begin{aligned} E[R_i] &= \frac{1}{2} \\ Var(R_i) &= \frac{1}{12} \end{aligned} \tag{25}$$

Substituting these equations

$$\begin{aligned} \rho &= \frac{(E[X - E[X]))(E[Y - E[Y]])}{\sqrt{Var(X)}\sqrt{Var(Y)}} \\ \text{or, } \rho &= \frac{E[XY] - E[X]E[Y]}{\sqrt{Var(X)}\sqrt{Var(Y)}} \\ \text{or, } \rho &= \frac{E[XY] - \frac{1}{2}.\frac{1}{2}}{\sqrt{\frac{1}{12}}\sqrt{\frac{1}{12}}} \\ \text{or, } \rho &= \frac{E[XY]}{\sqrt{\frac{1}{12}}\sqrt{\frac{1}{12}}} - \frac{\frac{1}{2}.\frac{1}{2}}{\sqrt{\frac{1}{12}}\sqrt{\frac{1}{12}}} \end{aligned}$$

$$\text{or, } \rho = 12E[XY] - 3 \tag{26}$$

Now lets evaluate $E[XY]$,

$$E[XY] = \frac{1}{n-k}\sum_{i=1}^{n-k} R_i R_{i+k} \tag{27}$$

Thus, the final equation can be written as

$$\rho = \frac{12}{n-k}\sum_{i=1}^{n-k} R_i R_{i+k} - 3 \tag{28}$$

Figure 11 illustrates the relationship between the correlation coefficient and the lag for the three cases under consideration. Across all scenarios, the correlation coefficient remains consistently close to zero over 500 lags. This observation indicates that the random numbers generated by the algorithm exhibit negligible to no correlation, affirming their independence. (See **Appendix 8.2.7** for code implementation)

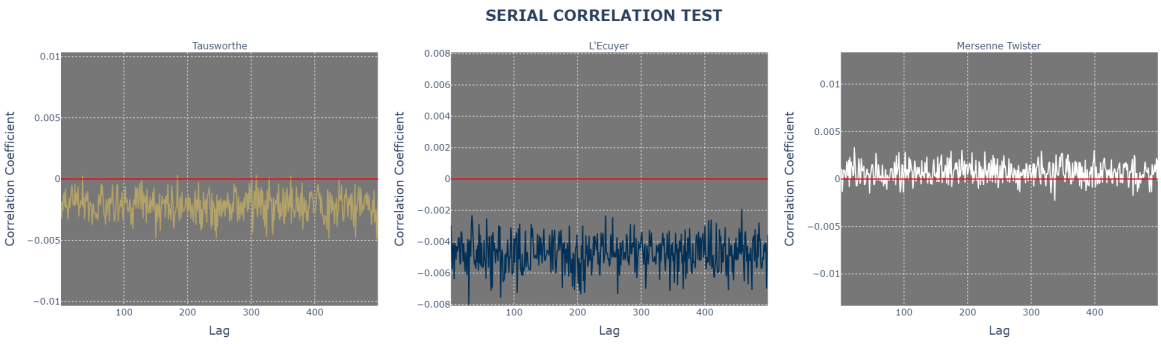

Figure 11: Serial Correlation Test

4.3.2 Von Neumann Ratio Test

The Von Neumann ratio test is a statistical test used to determine if a series of observations are independent. Von Neumann (1941) considered n successive observations from a normal population with mean μ and variance σ^2 [13]. Using the (slightly) biased estimator of σ^2,

$$S^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2 \tag{29}$$

and the mean squared difference between successive observations,

$$d^2 = \frac{1}{n-1}\sum_{i=1}^{n-1}(X_{i+1} - X_i)^2 \tag{30}$$

Von Neumann examined the ratio $V = d^2/S^2$. This ratio is intuitively appealing as a measure of independence between successive observations, because S^2 represents a measure of variability that does not consider the order of observations, whereas the importance of order is explicit in d^2 [14].

Thus the von Neumann statistic,

$$VN = \frac{d^2}{S^2} \tag{31}$$

It is known that $(VN - \mu)/\sigma$ is asymptotically standard normal, where $\mu = \frac{2n}{n-1}$ and $\sigma^2 = \frac{4(n-2)}{n^2-1}$. Thus, the hypothesis for such a test for independence is

H_0: The random numbers are independent
H_1: The random numbers are not independent

If the normalized Von Neumann statistic exceeds the critical value corresponding to a confidence level of $100(1-\alpha)\%$, as determined from the z-table, the null hypothesis is rejected. This indicates that the observed data does not conform to the assumed pattern under the null hypothesis at the specified confidence level. **Figures 12 - 14** displays the results of such a test. (See **Appendix 8.2.8** for code implementation)

```
Von Neumann Ratio Test for Tausworthe PRN Generator
Hypothesis:
H₀: The random numbers are independent
H₁: The random numbers are not independent
----------------------
von Neumann statistic after normalization with μ and σ, Z_stat = 0.3960067585176104
The critical z_score at significance level α = 0.05, Z(α, n) is 1.959963984540054
We see that |Z_stat| < Z(α, n). Thus,
----> We fail to reject the null hypothesis H₀ <----
The sequence of numbers are in fact independent
```

Figure 12: Von Neumann Ratio Test for the Tausworthe PRNG

```
Von Neumann Ratio Test for L'Ecuyer PRN Generator
Hypothesis:
H₀: The random numbers are independent
H₁: The random numbers are not independent
----------------------
von Neumann statistic after normalization with μ and σ, Z_stat = 1.8144096119087991
The critical z_score at significance level α = 0.05, Z(α, n) is 1.959963984540054
We see that |Z_stat| < Z(α, n). Thus,
----> We fail to reject the null hypothesis H₀ <----
The sequence of numbers are in fact independent
```

Figure 13: Von Neumann Ratio Test for the L'Ecuyer PRNG

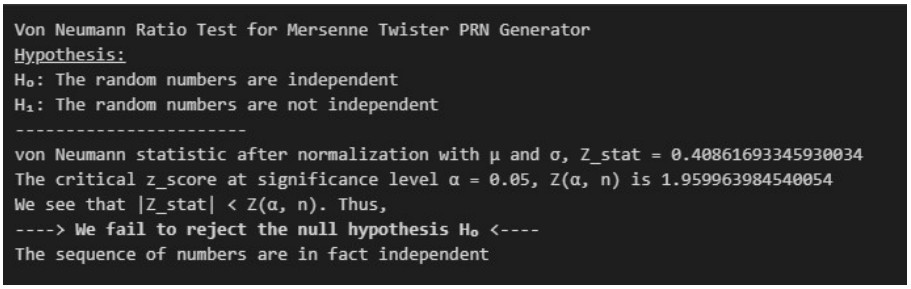

Figure 14: Von Neumann Ratio Test for the Mersenne Twister PRNG

4.3.3 Runs Test

Runs test is a statistical procedure which determines whether a sequence of data within a given distribution have been derived with a random process or not. It is sometimes called the Geary test, and it is a nonparametric test. It is an alternative test to test autocorrelation in the data. The runs test checks a randomness hypothesis for a data sequence. More specifically, it tests the null hypothesis that the data are random, i.e., that each data point is independent of others. Two different runs test will be looked into: (1) Runs Up and Down, (2) Runs Above and Below the Mean.

A "run" in this context is a series of increasing or decreasing values. For instance, in a sequence of coin tosses, a run might be a series of heads or tails. If the coin is fair and tosses are independent, both heads and tails occurring in a random pattern should be seen, with roughly equal numbers of runs of each type and of varying lengths [15]. The hypothesis for such a test in the example of 1 million numbers generated by the PRNGs is defined as

H_0: The random numbers are independent
H_1: The random numbers are not independent

The first step in the runs test is to count the number of runs in the data sequence. For example, consider the following series of numbers, **R**

$$0.41\ 0.68\ 0.89\ 0.84\ 0.74\ 0.91\ 0.55\ 0.71\ 0.36\ 0.30\ 0.09$$

Here the associated runs are

$$+\ +\ -\ -\ +\ -\ +\ -\ -\ -$$

The number of runs for this series is 6. This represents the Runs Up and Down. If **A** is the total number of runs in a truly random sequence, the mean μ_A and variance σ_A^2 of **A** is given by

$$\begin{aligned} \mu_A &= \frac{2N-1}{3} \\ \sigma_A^2 &= \frac{16N-29}{90} \end{aligned} \tag{32}$$

where N is the number of observations. For $N > 20$, the distribution of **A** is reasonably approximated by a normal distribution [16]

$$\mathbf{A} \approx Nor\left(\frac{2N-1}{3}, \frac{16N-29}{90}\right) \tag{33}$$

Converting it to a standardized normal distribution by

$$Z_0(A) = \frac{A-\mu_A}{\sigma_A^2} \tag{34}$$

Failure to reject the hypothesis of independence occurs when $-z_{\alpha/2} \leq Z_0(A) \leq z_{\alpha/2}$, where the α is the level of significance. **Figures 15 - 17** displays the results for the Runs Up and Down test performed on the three generators.

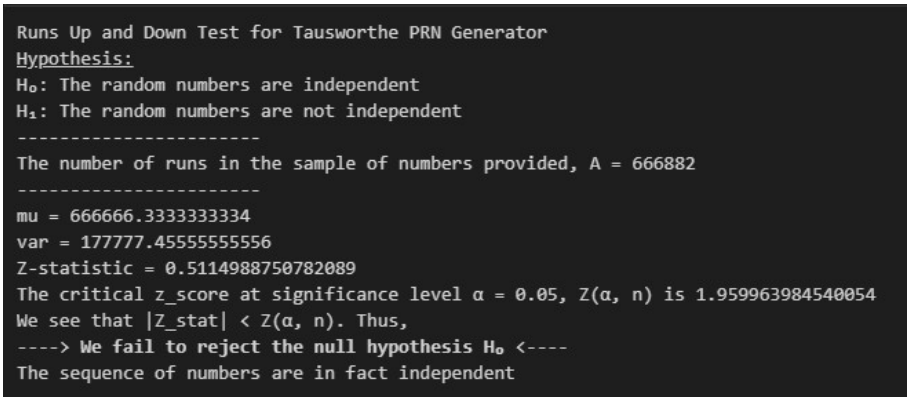

Figure 15: Runs Up and Down Test for the Tausworthe PRNG

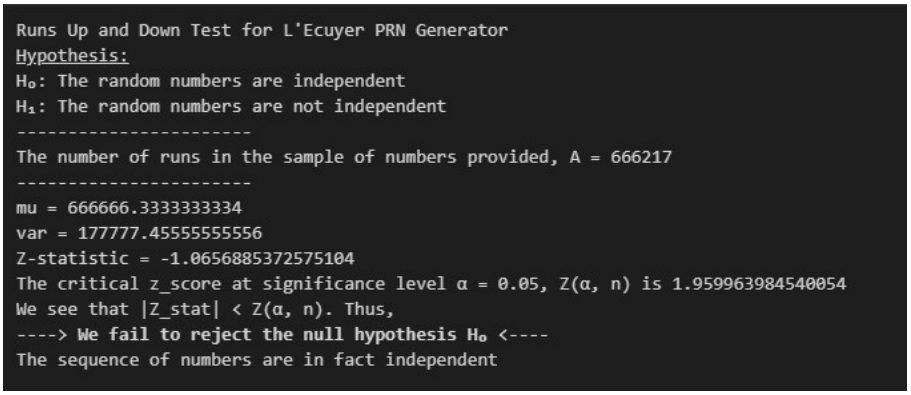

Figure 16: Runs Up and Down Test for the L'Ecuyer PRNG

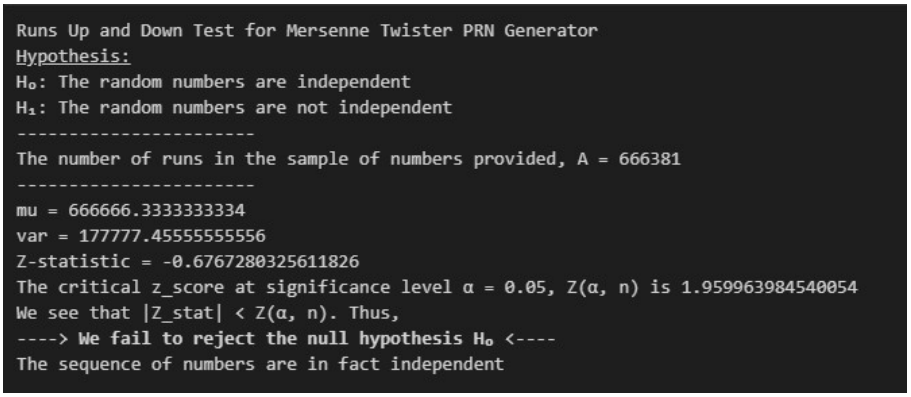

Figure 17: Runs Up and Down Test for the Mersenne Twister PRNG

Runs Above and Below the Mean is another type of the runs test, but it considers runs of values above and below a fixed level - in this case, the mean of a uniform distribution (0.5). For the series **R** described above, if $R_i \geq 0.5$, put a (+), while if $R_i < 0.5$, put a (-). If **B** is the associated run, then associated run looks like

$$- + + + + + + + + - - -$$

And $\mathbf{B} = 3$. Let N_1 be the number of (+) and N_2 be the number of (-), if N is large and R_i are actually independent, then the number of runs **B** is asymptotically Normally distributed as follows

$$B \approx Nor(\mu_B, \sigma_B^2) \tag{35}$$

where,

$$\begin{aligned} \mu_B &= \frac{2N_1N_2}{N} + \frac{1}{2} \\ \sigma_B^2 &= \frac{2N_1N_2(2N_1N_2 - N)}{N^2(N-1)} \end{aligned} \tag{36}$$

And the test statistic is

$$Z_0(B) = \frac{B - \mu_B}{\sigma_B^2} \tag{37}$$

Failure to reject the hypothesis of independence occurs when $-z_{\alpha/2} \leq Z_0(B) \leq z_{\alpha/2}$, where the α is the level of significance. **Figures 18, 19, 20** displays the results for the Runs Above and Below the Mean test performed on the three generators. (See **Appendix 8.2.9** for code implementation)

```
Runs Above and Below the Mean Test for Tausworthe PRN Generator
Hypothesis:
H₀: The random numbers are independent
H₁: The random numbers are not independent
-----------------------
The number of runs in the sample of numbers provided, B = 500026
-----------------------
Number of samples greater than or equal to 0.5 = 499751
Number of samples less than 0.5 = 500249
-----------------------
mu = 500000.375998
var = 249999.62599776537
Z-statistic = 0.05124804233383092
The critical z_score at significance level α = 0.05, Z(α, n) is 1.959963984540054
We see that |Z_stat| < Z(α, n). Thus,
----> We fail to reject the null hypothesis H₀ <----
The sequence of numbers are in fact independent
```

Figure 18: Runs Above and Below the Mean Test for the Tausworthe PRNG

```
Runs Above and Below the Mean Test for L'Ecuyer PRN Generator
Hypothesis:
H₀: The random numbers are independent
H₁: The random numbers are not independent
-----------------------
The number of runs in the sample of numbers provided, B = 500177
-----------------------
Number of samples greater than or equal to 0.5 = 499284
Number of samples less than 0.5 = 500716
-----------------------
mu = 499999.474688
var = 249998.72468880127
Z-statistic = 0.35505152960356356
The critical z_score at significance level α = 0.05, Z(α, n) is 1.959963984540054
We see that |Z_stat| < Z(α, n). Thus,
----> We fail to reject the null hypothesis H₀ <----
The sequence of numbers are in fact independent
```

Figure 19: Runs Above and Below the Mean Test for the L'Ecuyer PRNG

```
Runs Above and Below the Mean Test for Mersenne Twister PRN Generator
Hypothesis:
H₀: The random numbers are independent
H₁: The random numbers are not independent
-----------------------
The number of runs in the sample of numbers provided, B = 499955
-----------------------
Number of samples greater than or equal to 0.5 = 500154
Number of samples less than 0.5 = 499846
-----------------------
mu = 500000.452568
var = 249999.70256775225
Z-statistic = -0.09090519007630189
The critical z_score at significance level α = 0.05, Z(α, n) is 1.959963984540054
We see that |Z_stat| < Z(α, n). Thus,
----> We fail to reject the null hypothesis H₀ <----
The sequence of numbers are in fact independent
```

Figure 20: Runs Above and Below the Mean Test for the Mersenne Twister PRNG

4.3.4 Spatial Distribution

Another approach to evaluating the generated random numbers is through 2D and 3D scatter plots of adjacent values. These visualizations help identify any potential patterns within the sample distribution. **Figures 21 and 22** present the scatter plots for 1,000 data points generated by the algorithms. Upon closer examination, no discernible patterns are observed, suggesting randomness in the generated sample. (See **Appendix 8.2.10** for code implementation)

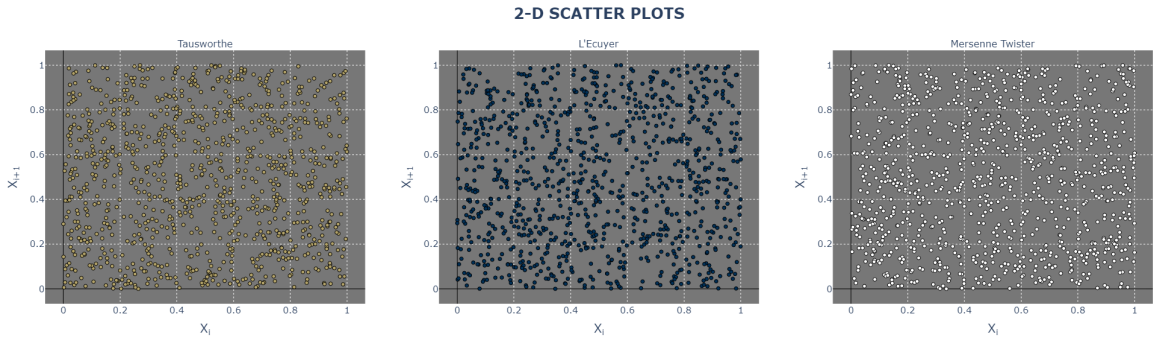

Figure 21: 2D Scatter plots

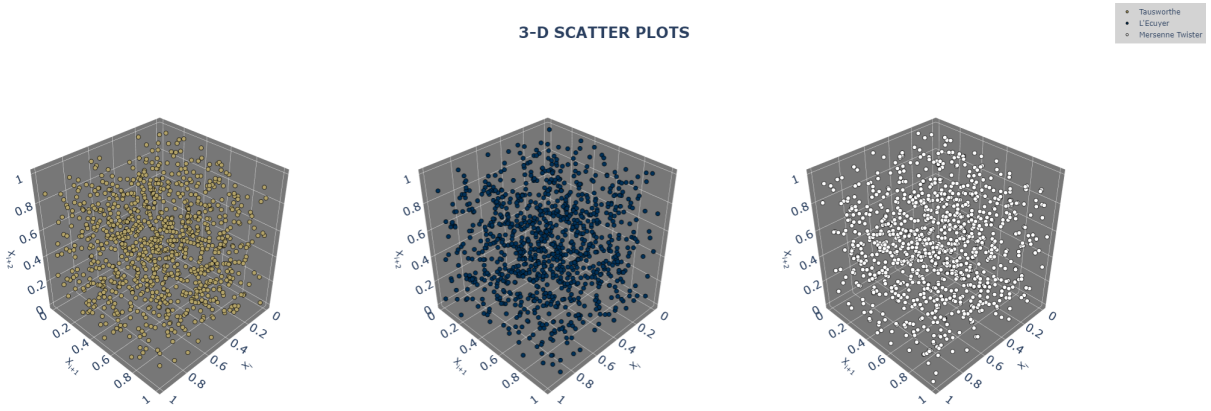

Figure 22: 3D Scatter plots

It is important to recognize that no single test can definitively establish whether a sequence of numbers is truly random and independent. Each test serves only to disprove randomness or provide evidence supporting it. A comprehensive assessment of a PRNG requires the application of multiple statistical tests, each targeting specific aspects of randomness and independence.

A range of tests to evaluate both the independence and uniformity of the generated sequences were conducted. Tests for independence, such as scatter plots, serial correlation, runs tests, and the Von Neumann ratio test, indicated negligible correlations and confirmed the lack of discernible patterns. Additionally, tests for uniformity, including histograms, the Kolmogórov-Smirnov test, and the chi-squared goodness-of-fit test, demonstrated that the distributions closely resembles the theoretical uniform distribution.

In conclusion, the combined results of these tests provide strong evidence that the sequences generated by the PRNGs meet the criteria for independence and uniformity, two fundamental properties of high-quality random number generators.

5. PRACTICAL APPLICATIONS

This section focuses on the practical applications of the generated random sequence of numbers. It examines methods for estimating the value of π, performing a random walk, and generating other random variates such as Normal, Exponential, and Triangular distributions.

5.1 Monte Carlo Simulation for estimating the value of π

Monte Carlo methods represent a versatile class of computational algorithms that use repeated random sampling to achieve numerical results. An example of applying the Monte Carlo method is the estimation of π. This involves simulating random (x, y) points within a two-dimensional plane, where the domain is a square with side length $2r$, centered at (0, 0). Within the same domain, a circle of radius r is inscribed inside the square. By calculating the ratio of the number of points that fall within the circle to the total number of generated points, an estimate of π can be obtained [17].

The area of the square is $4r^2$ unit sq, and the area of the circle inscribed is πr^2. And the ratio of these two areas is

$$\frac{\text{Area of the circle}}{\text{Area of the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \tag{38}$$

Now for a very large number of points generated,

$$\frac{\pi}{4} = \frac{\text{No. of points generated in the circle}}{\text{Total number of points}}$$

that is,

$$\pi = 4 \times \frac{\text{No. of points generated in the circle}}{\text{Total number of points}} \tag{39}$$

For the sake of visual clarity, 10,000 points were generated, with each point representing a coordinate within the unit square area. The **Figures 23, 24, 25** display scatter plots showing points that fall inside the circle and those that fall outside of it. It is also evident that increasing the number of data points brings the estimated value closer to the true value. Using **Eq 39**, an estimate of π is calculated. It is important to note that this estimation depends on the initial *seed* value used in the generator. Nevertheless, the algorithm demonstrates a high level of accuracy in its estimations. (See **Appendix 8.2.11** for code implementation)

$$\pi_{\text{Tausworthe}} = 3.1432, \qquad \pi_{\text{L'Ecuyer}} = 3.1464, \qquad \pi_{\text{Mersenne Twister}} = 3.14$$

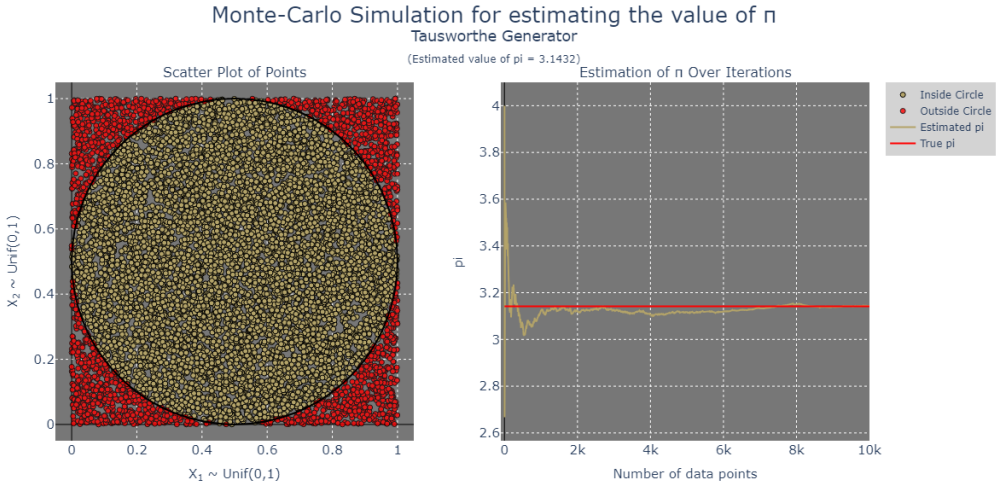

Figure 23: Estimation of π: Tausworthe PRNG

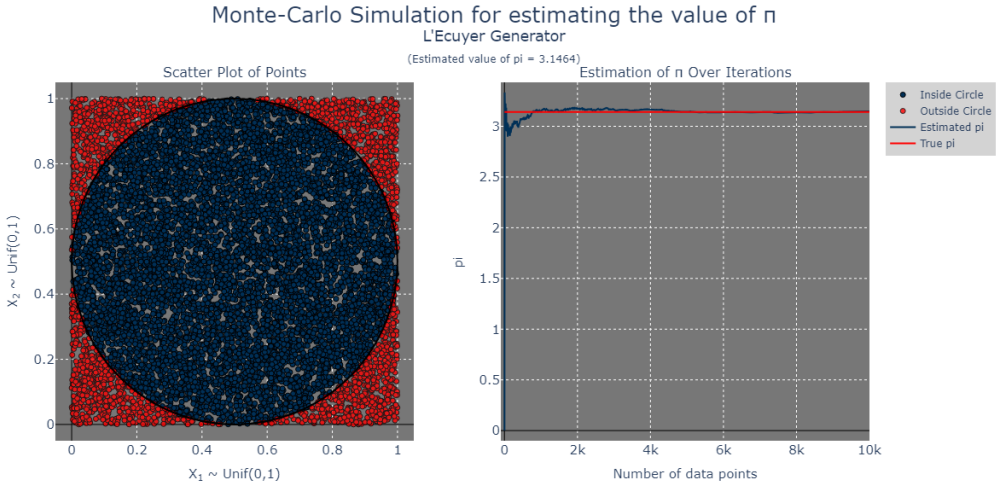

Figure 24: Estimation of π: L'Ecuyer PRNG

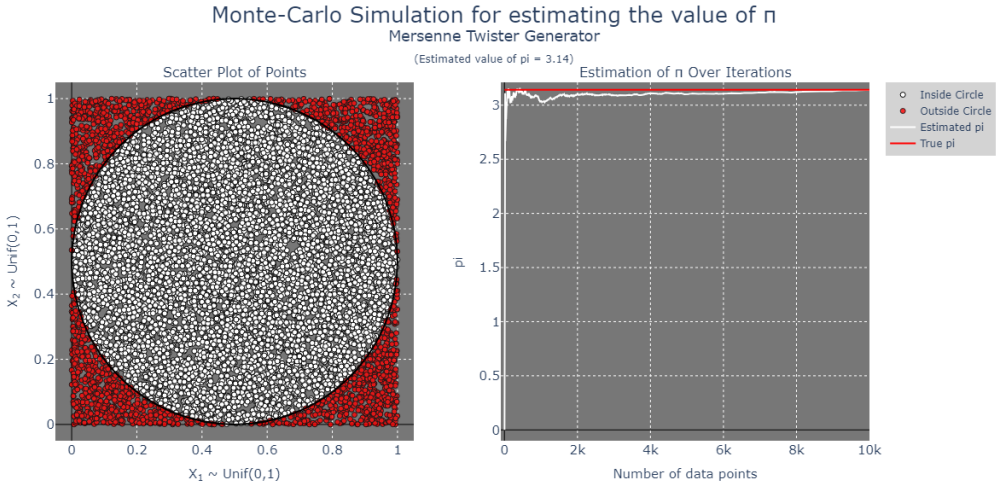

Figure 25: Estimation of π: Mersenne Twister PRNG

5.2 Random Walk

A random walk, often referred to as a drunkard's walk, represents a stochastic process involving a sequence of random steps within a given mathematical space. A simple example is a random walk along the integer number line $\mathbb{Z}$, which begins at 0 and moves either +1 or -1 at each step, with equal probability [18]. For this section, the random sequence generated from a Unif(0, 1) distribution was utilized, assigning a value of +1' to numbers greater than or equal to 0.5 and -1' otherwise. The figure illustrates the displacement of the sequence as a function of the step number. (See **Appendix 8.2.12** for code implementation)

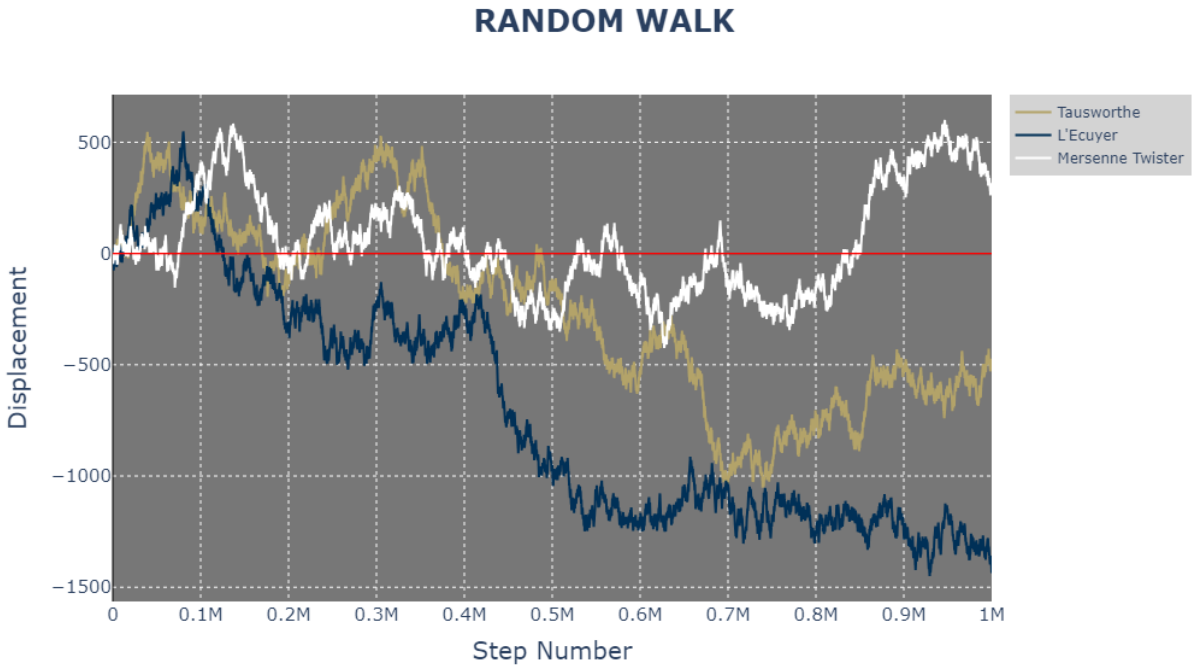

Figure 26: Random Walk

Random walks have numerous practical applications across various fields [18]:

1. **Financial Economics:** The random walk hypothesis is often employed to model stock prices and other financial factors.
2. **Population Genetics:** In genetics, random walks are used to describe the statistical properties of processes like genetic drift and the random fluctuations in selection over extended periods.
3. **Physics:** Random walks serve as simplified models for phenomena such as Brownian motion and diffusion, which describe the random movement of particles in liquids and gases.
4. **Semiconductor Manufacturing:** In this field, random walks are applied to study the effects of thermal treatments at smaller technology nodes. They help analyze the diffusion of dopants, impurities, and defects during critical fabrication processes.

5.3 Random Variate Generation

Using random samples from a Unif(0, 1) distribution, it is possible to generate random variates from other distributions by employing techniques such as the Inverse Transform Theorem, the Box-Muller Method, and Convolution. These methods enable the transformation of uniformly distributed random numbers into values that follow specific probability distributions required for various applications.

This section outlines the procedures used to generate random variates for the Exponential distribution Exp(1), the Triangular distribution Tria(0, 1, 2), and the standard Normal distribution Nor(0, 1).

The Exponential distribution is commonly used to model processes with constant hazard rates, such as interarrival times in Poisson processes. The Triangular distribution, characterized by its three points (minimum, mode, maximum), is often employed in simulations and decision analysis due to its simplicity and ability to represent bounded data. The Normal distribution, being essential in statistical modeling and widely observed in natural phenomena, serves as a cornerstone in probabilistic analysis. (See **Appendix 8.2.13** for code implementation)

5.3.1 Exponential Distribution via Inverse Transform Theorem

Let Y $\sim$ Exp(1), with CDF $F(y) = 1 - exp\{-y\}$ then via the Inverse Transform method, if X $\sim$ Unif(0, 1) is given, Y can be generated as follows:

$$Y = -ln(1 - X) \tag{40}$$

Figures 27-29 describes the output for such a process.

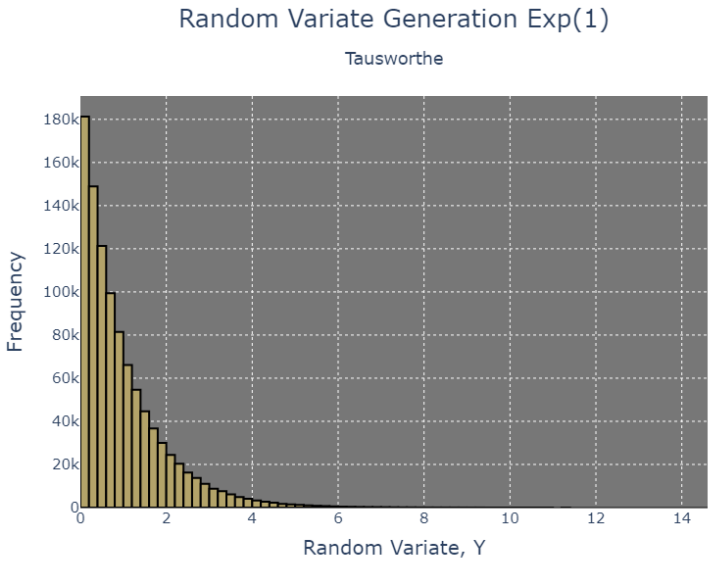

Figure 27: Exponential Distribution via Inverse Transform: Tausworthe PRNG

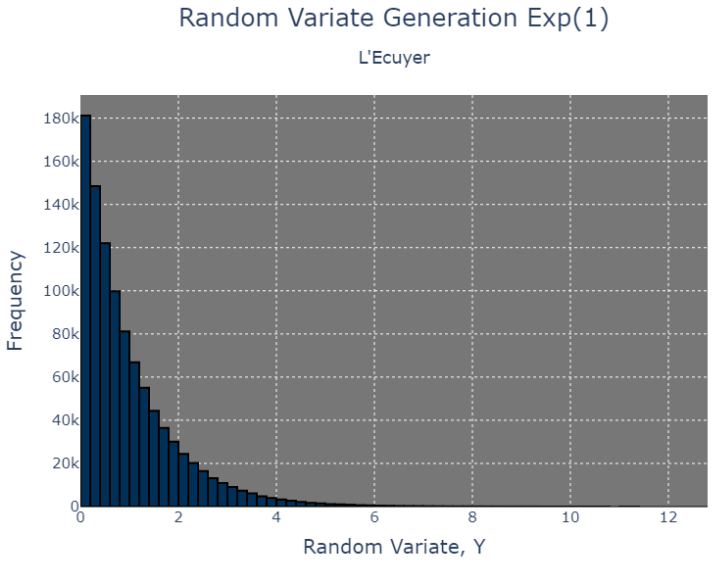

Figure 28: Exponential Distribution via Inverse Transform: L'Ecuyer PRNG

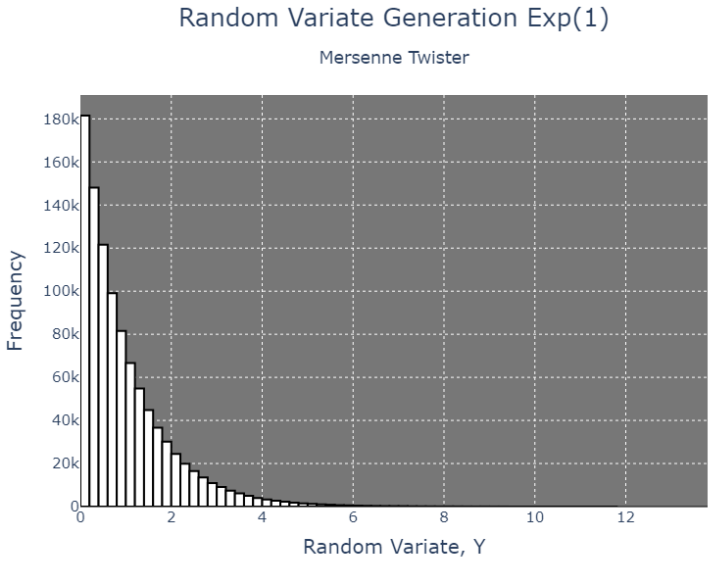

Figure 29: Exponential Distribution via Inverse Transform: Mersenne Twister PRNG

5.3.2 Triangular Distribution via Convolution

Given two distributions, X $\sim$ Unif(0, 1) and Y $\sim$ Unif(0, 1)), the desired Tria(0, 1, 2) distribution can be obtained by defining Z = X + Y . **Figures 30 - 32** illustrate the results of this process.

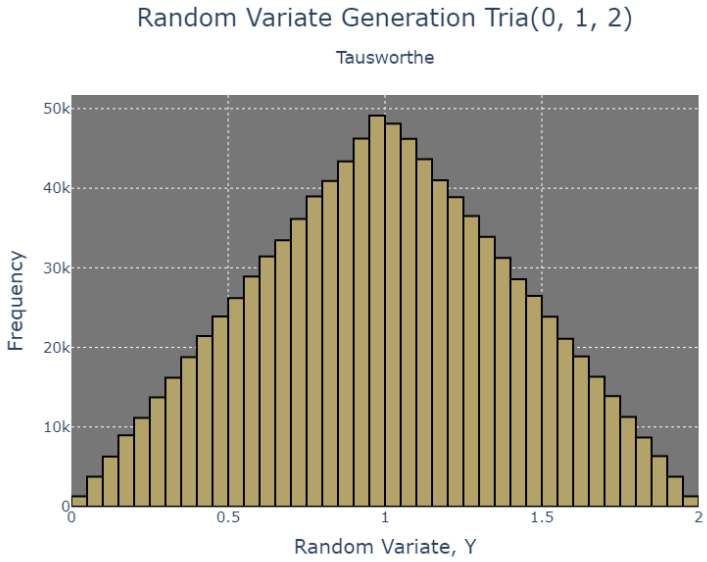

Figure 30: Triangular Distribution via Convolution: Tausworthe PRNG

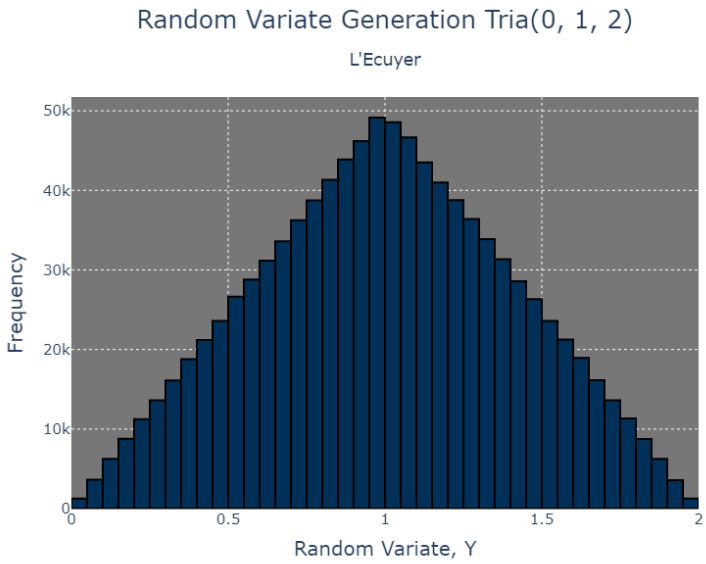

Figure 31: Triangular Distribution via Convolution: L'Ecuyer PRNG

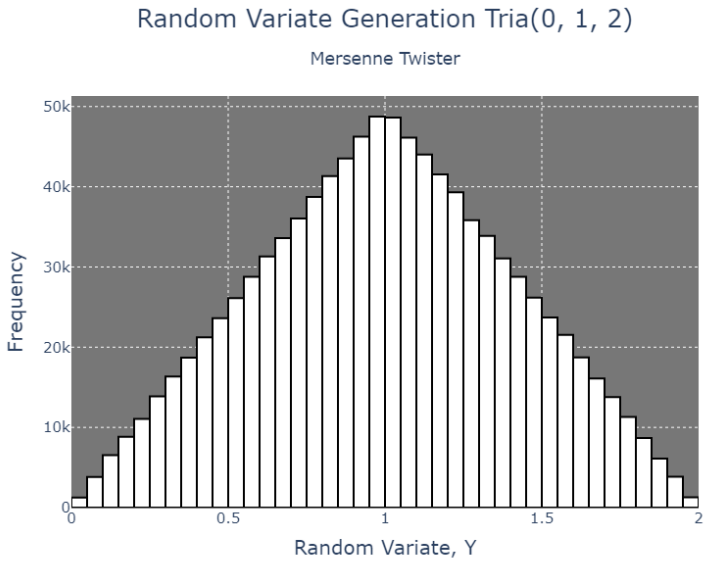

Figure 32: Triangular Distribution via Convolution: Mersenne Twister PRNG

5.3.3 Standard Normal Distribution via Box-Muller Transformation

Given two distributions, X $\sim$ Unif(0, 1) and Y $\sim$ Unif(0, 1)), let

$$\begin{aligned} Z_1 &= \sqrt{-2lnX}cos(2\pi Y) \\ Z_2 &= \sqrt{-2lnX}sin(2\pi Y) \end{aligned} \tag{41}$$

Then Z_1 and Z_2 are independent random variables with a standard normal distribution. **Figures 33 - 35** describes the result of such a process.

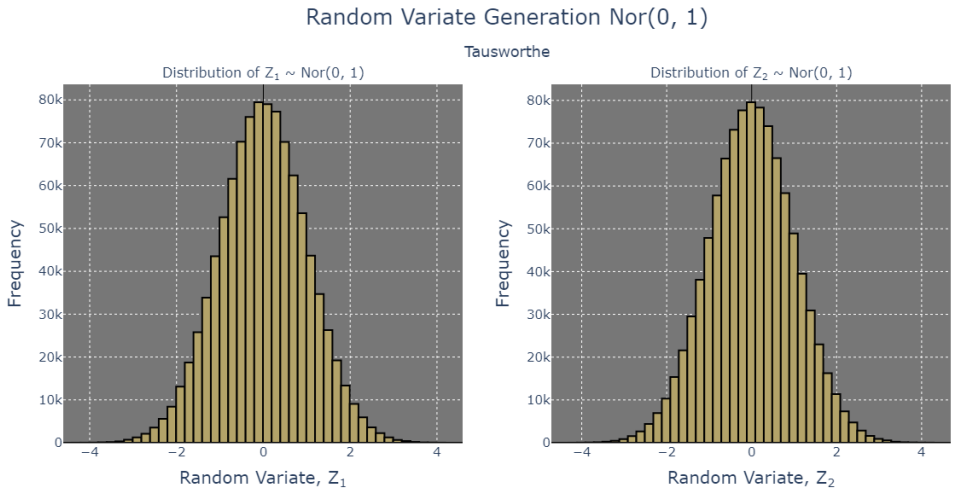

Figure 33: Standard Normal Distribution via Box-Muller: Tausworthe PRNG

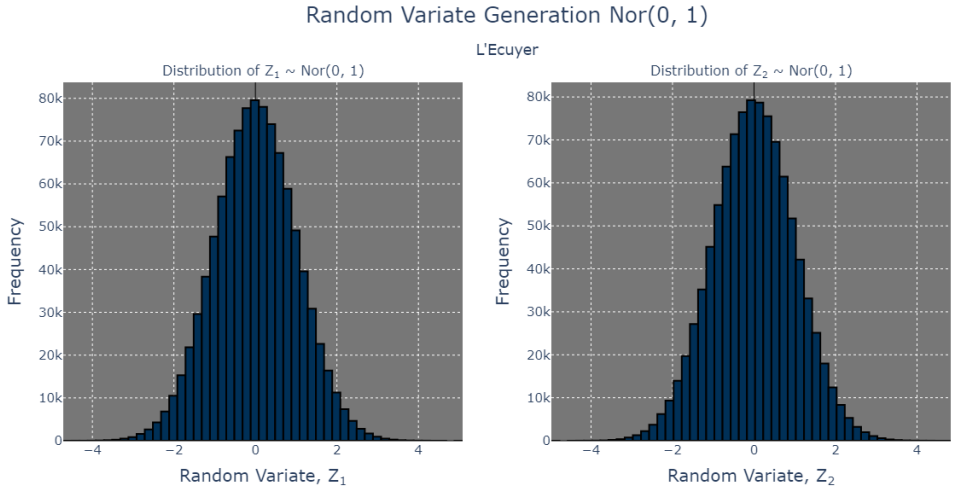

Figure 34: Standard Normal Distribution via Box-Muller: L'Ecuyer PRNG

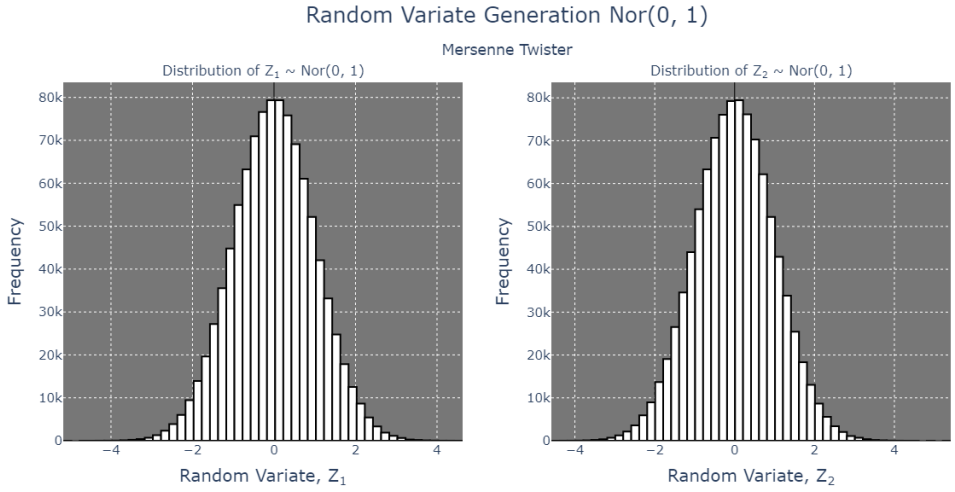

Figure 35: Standard Normal Distribution via Box-Muller: Mersenne Twister PRNG

6. CONCLUSION

The evaluation of pseudo-random number generators presented in this report highlights the importance of comprehensive testing to ensure their reliability and effectiveness in practical applications.

Tests for independence—such as scatter plots, correlation analysis, runs tests, and the Von Neumann ratio test—showed that the generated sequences exhibit negligible to no correlation, confirming their ability to produce random, independent values. Additionally, tests for uniformity, including histograms, chi-squared goodness-of-fit tests, and the Kolmogórov-Smirnov test, demonstrated that the distributions closely align with the theoretical uniform distribution. These findings validate the correct implementation of the three types of PRNGs.

The practical applications explored, such as the estimation of π, random walks, and the generation of random variates including Exponential, Normal, and Triangular variates. While multiple tests were employed to assess various aspects of randomness and independence, it remains impossible to definitively prove true randomness. Each test contributes valuable evidence but can only suggest or disprove randomness within the bounds of statistical theory.

In conclusion, the combination of statistical tests, theoretical analysis, and practical applications provides strong evidence supporting the quality of the PRNGs implemented.

7. REFERENCES

[1] Jones, D., & UCL Bioinformatics Unit. (2010, January 1). Good practice in (pseudo) random number generation for bioinformatics applications. http://www.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf

[2] Zhurbenko, I. G., & Smirnova, O. S. (1989). Some properties of random number generators. Journal of Soviet Mathematics, 47(5), 2703–2707. https://doi.org/10.1007/bf01095595

[3] Law, A. M. (2015). Simulation modeling and analysis. Mcgraw-Hill Education.

[4] Tausworthe, R. C. (1965). Random Numbers Generated by Linear Recurrence Modulo Two. Mathematics of Computation, 19(90), 201. https://doi.org/10.2307/2003345

[5] L'Ecuyer, P. (1999). Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. Operations Research, 47(1), 159–164. https://doi.org/10.1287/opre.47.1.159

[6] Grube, A. (1973). Mehrfach rekursiv-erzeugte Pseudo-Zufallszahlen. ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift Für Angewandte Mathematik Und Mechanik, 53(12). https://doi.org/10.1002/zamm.197305312116

[7] Matsumoto, M., & Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation, 8(1), 3–30. https://doi.org/10.1145/272991.272995

[8] Bradley, T., Toit, J. du, Tong, R., Giles, M., & Woodhams, P. (2011). Parallelization Techniques for Random Number Generators. Elsevier EBooks, 231–246. https://doi.org/10.1016/b978-0-12-384988-5.00016-4

[9] Donald Ervin Knuth. (2014). The art of computer programming. 3 Sorting and searching (3rd ed., Vol. 2, p. 107). Addison-Wesley.

[10] Sander, E. (2013, March 19). Binary numpy array to list of integers? Stack Overflow. https://stackoverflow.com/questions/15505514/binary-numpy-array-to-list-of-integers

[11] National Institute of Standards and Technology. (2019). 1.3.5.16. Kolmogorov-Smirnov Goodness-of-Fit Test. Nist.gov. https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm

[12] Autocorrelation. (2020, February 3). Wikipedia. https://en.wikipedia.org/wiki/Autocorrelation

[13] von Neumann, J. (1941). Distribution of the Ratio of the Mean Square Successive Difference to the Variance. The Annals of Mathematical Statistics, 12(4), 367–395. https://doi.org/10.1214/aoms/1177731677

[14] D, W. J. (1941). Moments of the Ratio of the Mean Square Successive Difference to the Mean Square Difference in Samples From a Normal Universe. The Annals of Mathematical Statistics, 12(2), 239–241. JSTOR. https://doi.org/10.2307/2235775

[15] 1.3.5.13. Runs Test for Detecting Non-randomness. (n.d.).www.itl.nist.gov. https://www.itl.nist.gov/div898/handbook/eda/section3/eda35d.htm

[16] Runs Tests. (n.d.). www.eg.bucknell.edu.https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node44.html

[17] Estimating the value of Pi using Monte Carlo. (2017, August 15). GeeksforGeeks. https://www.geeksforgeeks.org/estimating-value-pi-using-monte-carlo/

[18] Wikipedia Contributors. (2019, April 20). Random walk. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Random$_$walk

8. APPENDIX

8.1 Development Tools and Versions

	Version
Visual Studio Code	1.99.2
Jupyter core packages	
IPython	8.27.0
ipykernel	6.29.5
ipywidgets	8.1.5
jupyter_client	8.6.2
jupyter_core	5.7.2
jupyter_server	2.14.2
jupyterlab	4.2.5
nbclient	0.10.0
nbconvert	7.16.6
nbformat	5.10.4
notebook	7.2.2
qtconsole	not installed
traitlets	5.14.3
Python	3.12.5
Libraries	
Numpy	2.0.2
Plotly	6.0.1
SciPy	1.14.1
tqdm	4.67.0

8.2 Code Snippet

8.2.1 Tausworthe Pseudo-Random Generator

```
class Tausworthe:
    def __init__(self, class_parameters):
        assert isinstance(class_parameters['seed'], str),   f'Provide a seed that is in string format'
        assert self.is_binary(class_parameters['seed']),    f'Provide a seed that is binary\
                                                            and in string format'
        assert isinstance(class_parameters['r'], int),      f'The Secondary Lag parameter r must be an\
                                                            integer greater than 0. A good choice would\
                                                            be to select r such that gcd(r, q) = 1,\
                                                            that is r and q are relatively prime.\
                                                            For example 53, 61, 67, 71, 73'
        assert isinstance(class_parameters['q'], int),      f'The Lag parameter q must be an integer\
                                                            greater than 0. A good choice is to select\
                                                            prime numbers such that gcd(r, q) = 1 to\
                                                            ensure a long period.\
                                                            Examples: 97, 103, 127, 131, 149'
        assert len(str(class_parameters['seed'])) > class_parameters['r'] and \
                    len(str(class_parameters['seed'])) >= class_parameters['q'],   f'The Lag parameters,\
                                                                                   r and q should be\
                                                                                   shorter than length\
                                                                                   of seed'
        assert math.gcd(class_parameters['r'], class_parameters['q']) == 1, f'Ensure r and q are\
                                                                            relatively prime to\
                                                                            ensure full period'
        assert 0 < class_parameters['r'] < class_parameters['q'],  f'The length of the Lag\
                                                                   parameters should be\
                                                                   0 < r < q'
        assert isinstance(class_parameters['l'], int),  f'The Bit Grouping parameter l\
                                                        should be an integer'
        self.seed = np.array(list(class_parameters['seed']), dtype = np.int32)
        self.r = class_parameters['r']
        self.q = class_parameters['q']
        self.l = class_parameters['l']

    # staticmethod to check if seed is binary
    @staticmethod
    def is_binary(seed):
        return all(char in '01' for char in str(seed))

    def get_unif(self, n, disable_tqdm = False, enable_print = True):
        st = time.time()
        # Calculate the number of bits to add to the seed:
        bits_to_add = (n * self.l) - len(self.seed)

        # Create a new numpy array to store the generated bits after the XOR operation.
        # The new numpy array is intialized as the seed and an array of zeros of
        # length (bits_to_add), concatenated.
        # The dtype for the zeros array was set as 'object' to avoid the issue of overflow
        bins_array = np.concatenate([self.seed, np.zeros(bits_to_add, dtype = object)])

        # Loop through the bins_array and perform the XOR operation.
        # The range for the loop starts at the end of the seed
        for i in tqdm(range(len(self.seed),
                            len(bins_array)),
                            desc = f'Generating bits',
                            disable = disable_tqdm):
            bins_array[i] = bins_array[i - self.r] ^ bins_array[i - self.q]
```

```
        # Reshape the bins_array with the number of PRNs to be generated and the bit grouping parameter.
        # That is from 1D of shape (n * l + len(seed),) to 2D of shape (n, l)
        bins_array = bins_array.reshape(n, self.l)

        # Convert the bins_array (which now contains n binary values) into decimals
        # ref: https://stackoverflow.com/questions/15505514/binary-numpy-array-to-list-of-integers
        # dtype of object was used to avoid the issue of overflow as I have faced
        # with large n and l values
        num_array = bins_array.dot(1 << np.arange(bins_array.shape[-1] - 1, -1, -1, dtype = object))

        # Convert the num_array (which contains n decimal values) into Unif(0, 1)
        # Divide by np.float32(2 ** l) to avoid issue of floating-point precision issues
        # as I have faced before
        unif_array = num_array / np.float32(2 ** self.l)

        et = time.time()
        t = timedelta(seconds = et - st)

        if enable_print:
            print(f'-------------------------')
            print(f'Time taken to generate {n} Unif(0, 1) random numbers: {t}')
            print(f'Random Number generation rate: {t/n} /s')
            print(f'-------------------------')

        return unif_array.astype(float)
```

8.2.2 L'Ecuyer Pseudo-Random Generator

```
class MRG32k3a:

    #Constants to be used in the generator
    a1 = [0, 1_403_580, -810_728]
    a2 = [527_612, 0, -1_370_589]
    m1, m2 = (2 ** 32) - 209, (2 ** 32)  - 22_853

    def __init__(self, class_parameters):
        assert all(s < MRG32k3a.m1 for s in class_parameters['s1']),    f'Seed S1 must be less than\
                                                                        {MRG32k3a.m1} and not all zero'
        assert all(s < MRG32k3a.m2 for s in class_parameters['s2']),    f'Seed S2 must be less than\
                                                                        {MRG32k3a.m2} and not all zero'
        self.s1 = np.array(class_parameters['s1'])
        self.s2 = np.array(class_parameters['s2'])


    # function to generate the period of the L'Ecuyer generator
    def period(self):
        p = ((MRG32k3a.m1 ** 3 - 1) * (MRG32k3a.m2 ** 3 - 1)) / 2
        print(f'The period for the MRG32k3a generator is {p}')

    def get_unif(self, n, disable_tqdm = False, enable_print = True):
        # Define X_1i and X_2i as arrays which will store the numbers generated from the LCG
        # The length of the arrays was set to length of initial seed plus number of random numbers to
        # be generated as defined by user. This was done using np.concatenate
        X_1i = np.concatenate([self.s1, np.zeros(n, dtype = object)])
        X_2i = np.concatenate([self.s2, np.zeros(n, dtype = object)])

        # Create an np.zeros array to store the unif numbers generated
```

```
        unif = np.zeros(n, dtype = object)

        st = time.time()
        # Define a loop to run from range(3, n + 3). Starting range 3 was chosen to
        # access the third position from the X_1i and X_2i arrays.
        for i in tqdm(range(3, n + 3), desc = f'Progress', disable = disable_tqdm):
            X_1i[i] = sum(a1i * x1 for a1i, x1 in zip(MRG32k3a.a1[::-1], X_1i[i - 3:i:1])) % MRG32k3a.m1
            X_2i[i] = sum(a2i * x2 for a2i, x2 in zip(MRG32k3a.a2[::-1], X_2i[i - 3:i:1])) % MRG32k3a.m2

            unif[i-3] = ((X_1i[i] - X_2i[i]) % MRG32k3a.m1) / (MRG32k3a.m1)
        et = time.time()
        t = timedelta(seconds = et - st)
        if enable_print:
            print(f'-------------------------')
            print(f'Time taken to generate {n} Unif(0, 1) random numbers: {t}')
            print(f'Random Number generation rate: {t/n} /s')
            print(f'-------------------------')
        return unif.astype(float)
```

8.2.3 Mersenne Twister Pseudo-Random Generator

```
class MT19937:
    # Parameters
    w, N, M, r = 32, 624, 397, 31
    MATRIX_A = 0x9908b0df
    u, s, t, l = 11, 7, 15, 18 # Tempering parameters
    d = 0xffffffff #
    f = 1812433253 # See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier.
    TEMPERING_MASK_B = 0x9d2c5680 # Tempering mask
    TEMPERING_MASK_C = 0xefc60000 # Tempering mask
    UPPER_MASK = 0x80000000 # To extract the higher (w - r) significant bits
    LOWER_MASK = 0x7fffffff # To extract the lower r significant bits for the next number
    mag01 = np.array([0x0, MATRIX_A]) # mag01[x] = x * MATRIX_A for x = {0, 1}


    def __init__(self, class_parameters):
        self.mti = MT19937.N # A counter to keep track everytime a number is generated

        # set up array for state vectors
        self.mt = np.zeros(MT19937.N, dtype = np.uint32)

        # use the seed as the first state. AND operation to ensure the seed is 32-bit
        self.mt[0] = np.uint32(class_parameters['seed'] & MT19937.d)

        # update the state vector using a simple Linear Congruential method.
        # Note the use of various np.uintxx. NumPy enforces strict 32-bit integerlimits.
        # In order to avoid Runtime Overflow error caused by large numbers, I converted values
        # to uint64 and back to uint32
        for i in range(1, MT19937.N):
            self.mt[i] = np.uint32(np.uint64(MT19937.f) *\
            np.uint64(self.mt[i - 1] ^ (self.mt[i - 1] >> (MT19937.w - 2))) + np.uint64(i))



    def twist(self):
        # This is equivalent to getting the most significant bit from mt[i] and concatenating
        # to the least significant bit of the mt[i+1]. The np.roll allows to loop around to the
```

```
        # beginning of the array. Next is the XOR operation of three figures.
        # The first is the state vector starting at position 397 (achieved using np.roll),
        # next is a bitwise right shift of y, and then its the mag01 which is either 0 or
        # MATRIX_A. The mti counter is reset to zero.
        y = np.add((self.mt & MT19937.UPPER_MASK).astype(np.uint32),
                   (np.roll(self.mt, -1) & MT19937.LOWER_MASK).astype(np.uint32))
        self.mt = np.roll(self.mt, -self.M) ^ np.bitwise_right_shift(y, 1)\
                   ^ MT19937.mag01[y & 0x1]
        self.mti = 0

    def temper(self):
        # Tempering checks and performs twisting if it has not been done.
        if self.mti >= MT19937.N:
            self.twist()

        # Set up an intermediate variable that performs tempering as described below
        # for each of the state vectors
        y = self.mt[self.mti]
        self.mti += 1

        # Tempering steps which includes various bitwise shifts
        # TEMPERING_SHIFT_U(y)
        y ^= np.bitwise_right_shift(y, MT19937.u)
        # TEMPERING_SHIFT_S(y)
        y ^= np.bitwise_left_shift(y, MT19937.s) & MT19937.TEMPERING_MASK_B
        # TEMPERING_SHIFT_T(y)
        y ^= np.bitwise_left_shift(y, MT19937.t) & MT19937.TEMPERING_MASK_C
        # TEMPERING_SHIFT_L(y)
        y ^= np.bitwise_right_shift(y, MT19937.l)

        return y / (2 ** MT19937.w - 1)


    def get_unif(self, n: int, enable_print = True, disable_tqdm = False):
        # Set up an initial array of zeros which will be filled with the random numbers
        unif = np.zeros(n)
        st = time.time()
        for i in tqdm(range(n), desc = f'Generating random numbers:', disable = disable_tqdm):
            unif[i] = self.temper()
        et = time.time()
        t = timedelta(seconds = et - st)
        if enable_print:
            print(f'-------------------------')
            print(f'Time taken to generate {n} Unif(0, 1) random numbers: {t}')
            print(f'Random Number generation rate: {t/n} /s')
            print(f'-------------------------')

        return unif
```

8.2.4 Histogram generation

```
def hist(unifs):
    # Add an assertion to check if all the distributions have the same length.
    assert len({len(unif['unif']) for unif in unifs}) == 1, 'For better visualization, \
                                                            enter uniforms of same length'

    # n_bins will determine the bin size of the histograms
    n_bins = 25
```

```
    # Create subplots grid of 1 row and 3 columns
    fig = make_subplots(rows = 1, cols = len(unifs), subplot_titles = [unif['name'] for unif in unifs])

    # Add histogram traces
    for i, gen in enumerate(unifs):
      fig.add_trace(go.Histogram(x = gen['unif'],
                                 xbins = dict(start = 0, end = 1, size = 1/n_bins),
                                 marker = dict(line = dict(color = 'black', width = 2),
                                               color = gen['color']),
                                 hovertemplate = "<br>".join([f'{gen['name']}', "Range: %{x}",
                                                              "Frequency of Random Numbers: %{y}"]),
                                 name = ''), row = 1, col = i + 1)
      fig.add_hline(y = len(gen['unif']) / n_bins,
                    line_width = 2,
                    line_dash = "dash",
                    line_color = "red")
      fig.update_xaxes(title_text="Value", title_font_size = 20, tickfont_size = 15, gridwidth = 1,
                       griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1, row = 1, col = i + 1)
      fig.update_yaxes(title_text="Frequency", title_font_size = 20, tickfont_size = 15, gridwidth = 1,
                       griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1, row = 1, col = i + 1)

    # Update layout for better visualization
    fig.update_layout(title = {'text' : '<b>Histograms of Generated Sample Distributions</b>',
                               'x' : 0.5,
                               'y' : 0.95,
                               'xanchor' : 'center',
                               'yanchor' : 'top',
                               'font' : {'textcase' : 'upper',
                                         'size' : 25}
                                         },
                      width = 2000,
                      height = 600,
                      showlegend = False,
                      paper_bgcolor = 'white',
                      plot_bgcolor = '#777777'
                     )
    fig.show()
```

8.2.5 Chi-square Goodness-of-Fit Test

```
def gof(gen_list, gen, bins, alpha):
    # Get the required distribution
    for item in gen_list:
        if item['name'] == gen:
            unif = item['unif']
            nameU = item['name']
            colorU = item['color']

    # Print the hypothesis
    print(f'Testing Goodness-of-Fit for the {nameU} PRN Generator')
    print(f'\033[4mHypothesis:\033[0m')
    print(f'H\u2080: The random numbers are identically distributed')
    print(f'H\u2081: The random numbers are not identically distributed')
    print(f'------------------------')

    # Count the number of random number in 'k' bins
    unif_counts, unif_bin_edges = np.histogram(unif, bins = bins, range = (0.0, 1.0))
```

```
    # Get the intervals
    intervals = [(i/bins, (i+1)/bins) for i in range(bins)]
    print(f'The intervals for the range (0, 1) divided into {bins} sections:')
    print(intervals)

    # Calculate the expected value for each bin, E
    E = int(len(unif) / bins)
    print(f'The expected number of random numbers in each bin is {E}')

    print(f'The observed number of random numbers in each intervals: {unif_counts}')

    # Claculate the chi-squared statistic
    chi_squared_stat = sum((unif_counts - E) ** 2 / E)
    print(f'The chi_squared statistic = {chi_squared_stat}')

    # Calculate the chi_squared value at the desired significance level
    df = bins - 1 # df = degree of freedom
    chi_squared = chi2.ppf(1 - alpha, df)
    print(f'The critical chi-squared value at confidence level {(1 - alpha) * 100}% = {chi_squared}')

    if chi_squared_stat <= chi_squared:
        print(f'We see that {chi_squared_stat} < {chi_squared}. Thus,')
        print(f'----> \033[1mWe fail to reject the null hypothesis H\u2080\033[0m <----')
        print(f'The sequence of numbers are in fact identically distributed')
    else:
        print(f'We see that {chi_squared_stat} > {chi_squared}. Thus,')
        print(f'----> \033[1mReject the null hypothesis H\u2080\033[0m <----')
        print(f'The sequence of numbers are in fact not identically distributed')
```

8.2.6 Kolmogórov-Smirnov Test

```
def ks(gen_list, gen, alpha):
    # Get the distribution
    for item in gen_list:
        if item['name'] == gen:
            unif = item['unif']
            nameU = item['name']
            colorU = item['color']

    # Print the hypothesis
    print(f'Kolmogórov-Smirnov Test for {nameU} PRN Generator')
    print(f'\033[4mHypothesis:\033[0m')
    print(f'H\u2080: The random numbers are identically distributed')
    print(f'H\u2081: The random numbers are not identically distributed')
    print(f'-----------------------')

    # Sample (or distribution) size, n
    n = len(unif)

    # Compute indices
    indices = np.arange(1, (n + 1))

    # Ordered points
    print(f'Creating ordered points: ...')
    st = time.time()
    unif_ordered = np.sort(unif)
    t = timedelta(seconds = time.time() - st)
```

```
print(f'Create ordered points: \033[1mDone\033[0m (Time taken: {t})')

# Calculate D_plus
D_plus = max((indices/n) - unif_ordered)
print(f'D\u207A = {D_plus}')


# Calculate D_minus
D_minus = max((unif_ordered - (indices - 1) / n))
print(f'D\u207B = {D_minus}')


# Calcuate D = max{D_plus, D_minus}
D = max(D_plus, D_minus)
print(f'Thus, D = max\u007bD\u207A, D\u207B\u007d = max\u007b{D_plus}, {D_minus}\u007d = {D}')

# Get the K-S critical value at the significance level
D_ks = ksone.ppf(1 - alpha/2, n)
print(f'The critical value of D(\u03B1,n) at significance level of {alpha} is {D_ks}')

# Check condition for the hypothesis testing
if D < D_ks:
  print(f'We see that D < D(\u03B1,n). Thus,')
  print(f'----> \033[1mWe fail to reject the null hypothesis H\u2080\033[0m <----')
  print(f'The sequence of numbers are in fact identically distributed')
else:
   print(f'We see that D > D(\u03B1,n). Thus,')
   print(f'----> \033[1mReject the null hypothesis H\u2080\033[0m <----')
   print(f'The sequence of numbers are in fact not identically distributed')

trace1 = go.Scatter(x = np.concatenate([np.array([0]), unif_ordered, np.array([1])],),
                    y = np.arange(len(unif_ordered) +2) / (n + 1),
                    mode = 'lines',
                    marker = dict(color = colorU),
                    hovertemplate = '<br>'.join([f'{nameU}', 'Cumulative prob: %{y}', 'X: %{x}']),
                    showlegend = False,
                    name = '')
trace2 = go.Scatter(x = np.linspace(start = 0, stop = 1, num = n+2),
                    y = np.linspace(start = 0, stop = 1, num = n+2),
                    mode = 'lines', marker = dict(color = 'red'),
                    hovertemplate = '<br>'.join([f'True CDF', 'Cumulative prob: %{y}', 'X: %{x}']),
                    showlegend = False,
                    name = '')

fig = make_subplots(rows = 1, cols = 2, subplot_titles = ['Empirical vs True CDF', 'Zoomed In'])
fig.add_trace(trace1, row = 1, col = 1)
fig.add_trace(trace2, row = 1, col = 1)
fig.add_trace(trace1.update(name = 'Empirical CDF', showlegend = True), row = 1, col = 2)
fig.add_trace(trace2.update(name = 'True CDF', showlegend = True), row = 1, col = 2)
fig.update_xaxes(title_text = 'X', title_font_size = 20, tickfont_size = 15, gridwidth = 1,
                 griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1)
fig.update_yaxes(title_text = 'Cumulative Probability', title_font_size = 20, tickfont_size = 15,
                 gridwidth = 1, griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1)
fig.update_layout(title = {'text' : f'Comparison between the theoretical uniform CDF and \
                                    the empirical CDF<br><sub>{nameU}</sub>',
                           'x' : 0.5,
                           'y' : 0.95,
                           'xanchor' : 'center',
                           'yanchor' : 'top',
                           'font' : {'textcase' : 'upper',
                                     'size' : 20}
```

```
                                        },
                    width = 1200,
                    height = 600,
                    paper_bgcolor = 'white',
                    plot_bgcolor = '#777777',
                    legend=dict(bgcolor='lightgray'))
    fig.update_xaxes(range = [0, 0.004], row = 1, col = 2)
    fig.update_yaxes(range = [0, 0.004], row = 1, col = 2)
    fig.show()
```

8.2.7 Serial Correlation Test

```
def correlation(unifs, lag = 500):
    # Add an assertion to check if all the distributions have the same length.
    assert len({len(unif['unif']) for unif in unifs}) == 1, 'For better visualization, \
                                                            enter uniforms of same length'

    # For better visualization, let's set the max lag to be 500
    assert lag <= 500, f'Max lag should be less than 500'

    # length of the distribution
    n = len(unifs[0]['unif'])

    # Make sure that the lag is lesser than the length of the distribution
    assert lag < n, f'Value of Lag should be less than length of the uniform distribution array'

    # Create subplots grid of 1 row and 3 columns
    fig = make_subplots(rows = 1, cols = len(unifs), subplot_titles = [unif['name'] for unif in unifs])

    # Variable 'corr' will store the correlation coefficients at
    # different lags for the three distributions generated from the PRNGs
    for i, unif in enumerate(unifs):
        corr = np.zeros(shape = lag)
        for j in range(1, lag + 1):
            X = unif['unif'][:n-j]
            Y = unif['unif'][j:]
            XY = np.sum(np.multiply(X, Y))
            corr[j-1] = (12 * XY) / (n - j) - 3


        fig.add_trace(go.Scatter(x = np.arange(start = 1, stop = lag),
                                 y = corr,
                                 mode = 'lines',
                                 marker = dict(color = unif['color']),
                                 hovertemplate = "<br>".join([f'{unif['name']}', 'Corr : %{y}',
                                                              'Lag : %{x}']),
                                 name = ''), row = 1, col = i + 1)
        fig.add_hline(y = 0, line_width = 2, line_color = 'red')
        fig.update_xaxes(title_text='Lag', title_font_size = 20, tickfont_size = 15,
                         gridwidth = 1, griddash = 'dot', zerolinecolor = 'black',
                         zerolinewidth = 1, row = 1, col = i + 1)
        fig.update_yaxes(title_text='Correlation Coefficient', title_font_size = 20,
                         tickfont_size = 15, range = [-max(corr) - 0.01, max(corr) + 0.01],
                         gridwidth = 1, griddash = 'dot', zerolinecolor = 'black',
                         zerolinewidth = 1, row = 1, col = i + 1)
```

```
    fig.update_layout(title = {'text' : '<b>Serial Correlation Test</b>',
                               'x' : 0.5,
                               'y' : 0.95,
                               'xanchor' : 'center',
                               'yanchor' : 'top',
                               'font' : {'textcase' : 'upper',
                                         'size' : 25}
                                        },
                      width = 2000,
                      height = 600,
                      showlegend = False,
                      paper_bgcolor = 'white',
                      plot_bgcolor = '#777777'
                     )
    fig.show()
```

8.2.8 Von Neumann Ratio Test

```
def vn(gen_list, gen, alpha):
    # Get the distribution
    for item in gen_list:
        if item['name'] == gen:
            unif = item['unif']
            nameU = item['name']
            colorU = item['color']

    # Print the hypothesis
    print(f'Von Neumann Ratio Test for {nameU} PRN Generator')
    print(f'\033[4mHypothesis:\033[0m')
    print(f'H\u2080: The random numbers are independent')
    print(f'H\u2081: The random numbers are not independent')
    print(f'-----------------------')

    # number of samples
    n = np.float64(len(unif))

    # mean-squared difference between successive observations
    d2 = np.sum(np.square(np.diff(unif))) / (n-1)

    # von Neumann statistic
    vn = d2 / np.var(unif)

    # Assuming we have generated a large number of samples
    mu = np.float64(2*n/(n-1))
    sigma2 = np.float64(4*n/(n**2 - 1))
    vn_norm = abs((vn - mu) / (sigma2 ** (0.5)))
    print(f'von Neumann statistic after normalization \
            with \u03BC and \u03C3, Z_stat = {vn_norm}')

    # Get the z-score value from the significance level
    z_score = st.norm.ppf(1-alpha/2)
    print(f'The critical z_score at significance level\
            {chr(945)} = {alpha}, Z({chr(945)}, n) is {z_score}')

    if abs(vn_norm) < z_score:
        print(f'We see that |Z_stat| < Z({chr(945)}, n). Thus,')
        print(f'----> \033[1mWe fail to reject the null hypothesis H\u2080\033[0m <----')
```

```
        print(f'The sequence of numbers are in fact independent')
    else:
        print(f'We see that |Z_stat| > Z({chr(945)}, n). Thus,')
        print(f'\033[1mReject the null hypothesis H\u2080\033[0m')
        print(f'The sequence of numbers are in fact not independent')
```

8.2.9 Runs Test

```
def runsUD(gen_list, gen, alpha):
    # Get the distribution
    for item in gen_list:
        if item['name'] == gen:
            unif = item['unif']
            nameU = item['name']
            colorU = item['color']

    # Print the hypothesis
    print(f'Runs Up and Down Test for {nameU} PRN Generator')
    print(f'\033[4mHypothesis:\033[0m')
    print(f'H\u2080: The random numbers are independent')
    print(f'H\u2081: The random numbers are not independent')
    print(f'-----------------------')

    # number of samples
    n = len(unif)

    # algorith to get the number of runs, A
    # Logic: np.diff(unif) gives a numpy array with discrete differences along the given axis,
    # np.sign() converts the array's positive values to 1 and negative values to -1. Now compare
    # adjacent values for inflection point, that is when 1 becomes -1 and vice-versa.
    A = np.sum((np.sign(np.diff(unif))[:-1] != np.sign(np.diff(unif))[1:]))
    print(f'The number of runs in the sample of numbers provided, A = {A}')
    print(f'-----------------------')

    # Get the mu and var
    mu = np.float64((2*n - 1)/3)
    var = np.float64((16*n - 29)/90)
    print(f'mu = {mu}')
    print(f'var = {var}')


    # Calculate the test statistic
    Z_stat = (A - mu)/(var ** 0.5)
    print(f'Z-statistic = {Z_stat}')

    # Get the critical z-score value from the significance level
    z_score = st.norm.ppf(1 - alpha / 2)
    print(f'The critical z_score at significance level \
            {chr(945)} = {alpha}, Z({chr(945)}, n) is {z_score}')

    if abs(Z_stat) < z_score:
        print(f'We see that |Z_stat| < Z({chr(945)}, n). Thus,')
        print(f'----> \033[1mWe fail to reject the null hypothesis H\u2080\033[0m <----')
        print(f'The sequence of numbers are in fact independent')
    else:
        print(f'We see that |Z_stat| < Z({chr(945)}, n). Thus,')
        print(f'\033[1mReject the null hypothesis H\u2080\033[0m')
```

```
        print(f'The sequence of numbers are in fact not independent')

def runsABM(gen_list, gen, alpha):
    # Get the distribution
    for item in gen_list:
        if item['name'] == gen:
            unif = item['unif']
            nameU = item['name']
            colorU = item['color']

    # Print the hypothesis
    print(f'Runs Above and Below the Mean Test for {nameU} PRN Generator')
    print(f'\033[4mHypothesis:\033[0m')
    print(f'H\u2080: The random numbers are independent')
    print(f'H\u2081: The random numbers are not independent')
    print(f'-----------------------')

    # number of samples
    n = np.float64(len(unif))

    # Get the number of runs, B
    # Logic: We subtract 0.5 from each element of array, and use
    # np.sign() to denote the value as 1 if it is positive and -1 if neagtive.
    # Compare adjacent values for sign change, and add 1 to account for the extra
    # comparison.
    B = np.sum(np.sign(unif - 0.5)[:-1] != np.sign(unif - 0.5)[1:]) + 1
    print(f'The number of runs in the sample of numbers provided, B = {B}')
    print(f'-----------------------')

    # Get n1, number of observations greater than equal to 0.5, and
    # n2 = number of observations less than 0.5
    n1 = np.float64(np.sum(unif >= 0.5))
    n2 = np.float64(n - n1)
    print(f'Number of samples greater than or equal to 0.5 = {int(n1)}')
    print(f'Number of samples less than 0.5 = {int(n2)}')
    print(f'-----------------------')

    # Calculate mu and variance using n1 and n2
    mu = np.float64(((2 * n1 * n2)/ n) + 0.5)
    var = np.float64((2 * n1 * n2 * (2 * n1 * n2 - n)) / (n * n * (n - 1)))
    print(f'mu = {mu}')
    print(f'var = {var}')

    # Calculate the test statistic, Z_stat
    Z_stat = (B - mu) / (var ** 0.5)
    print(f'Z-statistic = {Z_stat}')

    # Get the critical z-score value from the significance level
    z_score = st.norm.ppf(1-alpha/2)
    print(f'The critical z_score at significance level\
            {chr(945)} = {alpha}, Z({chr(945)}, n) is {z_score}')

    # Check the condition for the hypothesis
    if abs(Z_stat) < z_score:
        print(f'We see that |Z_stat| < Z({chr(945)}, n). Thus,')
        print(f'----> \033[1mWe fail to reject the null hypothesis H\u2080\033[0m <----')
        print(f'The sequence of numbers are in fact independent')
    else:
        print(f'We see that |Z_stat| < Z({chr(945)}, n). Thus,')
        print(f'\033[1mReject the null hypothesis H\u2080\033[0m')
```

```
        print(f'The sequence of numbers are in fact not independent')
```

8.2.10 Spatial Distribution

```
def scatter(unifs):
    # Add an assertion to check if all the distributions have the same length.
    assert len({len(unif['unif']) for unif in unifs}) == 1, 'For better visualization, \
                                                            enter uniforms of same length'

    # Create subplots grid of 1 row and 3 columns
    fig = make_subplots(rows = 1, cols = len(unifs), subplot_titles = [unif['name'] for unif in unifs])

    # Add scatter traces
    for i, gen in enumerate(unifs):
      fig.add_trace(go.Scatter(x = gen['unif'][0:999],
                               y = gen['unif'][1:1000],
                               mode = 'markers',
                               marker = dict(line = dict(color ='black', width=1), color = gen['color']),
                               hovertemplate = "<br>".join([f'{gen['name']}', 'X<sub>i</sub> : %{x}',
                               'X<sub>i+1</sub> : %{y}']), name = ''), row = 1, col = i + 1)

      fig.update_xaxes(title_text='X<sub>i</sub>', title_font_size = 20, tickfont_size = 15,
                       gridwidth = 1, griddash = 'dot', zerolinecolor = 'black',
                       zerolinewidth = 1, row = 1, col = i + 1)
      fig.update_yaxes(title_text='X<sub>i+1</sub>', title_font_size = 20, tickfont_size = 15,
                       gridwidth = 1, griddash = 'dot', zerolinecolor = 'black',
                       zerolinewidth = 1, row = 1, col = i + 1)

    # Update layout for better visualization
    fig.update_layout(title = {'text' : '<b>2-D Scatter Plots</b>',
                               'x' : 0.5,
                               'y' : 0.95,
                               'xanchor' : 'center',
                               'yanchor' : 'top',
                               'font' : {'textcase' : 'upper',
                                         'size' : 25}
                                         },
                      width = 2000,
                      height = 600,
                      showlegend = False,
                      paper_bgcolor = 'white',
                      plot_bgcolor = '#777777'
                    )
    fig.show()

def scatter3d(unifs):
    # Add an assertion to check if all the distributions have the same length.
    assert len({len(unif['unif']) for unif in unifs}) == 1, 'For better visualization,\
                                                            enter uniforms of same length'

    # Create subplots grid of 1 row and 3 columns
    fig = make_subplots(rows = 1, cols = len(unifs),
        specs=[[{'type': 'scene'}, {'type': 'scene'}, {'type': 'scene'}]],
        subplot_titles = [unif['name'] for unif in unifs])

    # Define scene settings
    scene_settings = {'xaxis' : {'title' : 'X<sub>i</sub>',
```

```
                        'tickfont' : {'size' : 15},'title_font' : {'size' : 15},
                                'backgroundcolor' : '#777777', 'gridwidth' : 1},
                    'yaxis' : {'title' : 'X<sub>i+1</sub>',
                        'tickfont' : {'size' : 15}, 'title_font' : {'size' : 15},
                                'backgroundcolor' : '#777777', 'gridwidth' : 1},
                    'zaxis' : {'title' : 'X<sub>i+2</sub>',
                        'tickfont' : {'size' : 15},'title_font' : {'size' : 15},
                                'backgroundcolor' : '#777777', 'gridwidth' : 1},
                    'camera' : {'eye' : {'x' : 2.0, 'y' : 2.0, 'z' : 2.0}}
                    }

    # Add scatter traces
    for i, gen in enumerate(unifs):
      fig.add_trace(go.Scatter3d(x = gen['unif'][0:999],
                                 y = gen['unif'][1:1000],
                                 z = gen['unif'][2:1001],
                                 mode = 'markers',
                                 marker = {'line' : {'color' : 'black', 'width' : 1},
                                                    'size' : 4, 'color' : gen['color']},
                                 hovertemplate = "<br>".join([f'{gen['name']}', "X<sub>i</sub> : %{x}",
                                                "X<sub>i+1</sub> : %{y}", "X<sub>i+2</sub> : %{z}"]),
                                 name = gen['name'], hoverlabel={'bgcolorsrc' : 'blue'}),
                                 row = 1, col = i + 1)

    # Update layout for better visualization
    fig.update_layout(title = {'text' : '<b>3-D Scatter Plots</b>',
                                 'x' : 0.5,
                                 'y' : 0.95,
                                 'xanchor' : 'center',
                                 'yanchor' : 'top',
                                 'font' : {'textcase' : 'upper',
                                           'size' : 25}
                                          },
                      scene = scene_settings,
                      scene2 = scene_settings,
                      scene3 = scene_settings,
                      width = 2000,
                      height = 800,
                      showlegend = True,
                      legend=dict(bgcolor='lightgray'),
                      margin = {'l' : 0, 'r' : 0, 't' : 0, 'b' : 0}
                     )
    fig.show()
```

8.2.11 Monte Carlo Simulation for estimating the value of π

```
def MC(gen_list, gen):
    # Get the distributions
    for item in gen_list:
        if item['name'] == gen:
            unif1 = item['unif'][0:10000]
            unif2 = item['unif2'][0:10000]
            nameU = item['name']
            colorU = item['color']
    # Check whether the distributions are of the same length
    assert len(unif1) == len(unif2), 'The two samples should be of the same length'
```

```
# Get the length of the distribution
n = len(unif1)
# Create an array of integers. This will be used in subplot number 2
indices = np.arange(1, n + 1)

# Pairing corresponding elements from the two distributions as cordinates in a unit square,
# and calculating the distance of the points from the center (0.5, 0.5)
dist = np.sqrt(np.square(unif1 - 0.5) + np.square(unif2 - 0.5)) # Distance of each point \
                                                                from the center (0.5, 0.5)
points_in_circle = np.sum(dist <= 0.5) # total number of points that\
                                       are inside the circle of radius 0.5
pi = 4 * np.divide(np.cumsum(dist <= 0.5), indices) # pi is obtained as \
                                            pi = 4 * (points in circle/total number of points)

# Obtain cordinates of points that are in and outside the circle
x_in = unif1[np.where(dist <= 0.5)]
y_in = unif2[np.where(dist <= 0.5)]
x_out = unif1[np.where(dist > 0.5)]
y_out = unif2[np.where(dist > 0.5)]

print(f'Points that fall within the circle = {points_in_circle}')
print(f'Total number of points = {len(unif1)}')
print(f'------------------------')
pi_estimated = 4 * (points_in_circle/len(unif1))
print(f'Estimation of value of \u03C0 = {pi_estimated}')

trace1 = go.Scatter(x = x_in, y = y_in, mode = 'markers',
        marker = dict(line = dict(color = 'black', width = 1),
        color = colorU), name = 'Inside Circle')

trace2 = go.Scatter(x = x_out, y = y_out, mode = 'markers',
        marker = dict(line = dict(color = 'black', width = 1),
        color = 'red'), name = 'Outside Circle', opacity = 0.8)

trace3 = go.Scatter(x = indices, y = pi, mode = 'lines',
        marker = dict(color = colorU), name = 'Estimated pi')

trace4 = go.Scatter(x = indices, y = np.repeat(math.pi, len(indices)),
        mode = 'lines', marker = dict(color = 'red'), name = 'True pi')

# Create subplots grid of 1 row and 3 columns
fig = make_subplots(rows = 1, cols = 2,
    subplot_titles = ["Scatter Plot of Points", "Estimation of  Over Iterations"])

fig.add_trace(trace1, row = 1, col = 1)
fig.add_trace(trace2, row = 1, col = 1)
fig.add_trace(trace3, row = 1, col = 2)
fig.add_trace(trace4, row = 1, col = 2)
fig.add_shape(type = "circle", xref = "x", yref = "y", x0 = 0, y0 = 0, x1 = 1, y1 = 1,
            line_color = "black", line_width = 2, row = 1, col = 1)
fig.update_layout(title = {'text' : f'Monte-Carlo Simulation for estimating the value\
                           of <br><sup>{nameU} Generator</sup><br><sup><sup>(Estimated value\
                           of pi = {pi_estimated})</sup>',
                              'x' : 0.5,
                              'y' : 0.95,
                              'xanchor' : 'center',
                              'yanchor' : 'top',
                              'font' : {'size' : 25}
                              },
                    width = 1200,
                    height = 600,
```

```
                        paper_bgcolor = 'white',
                        plot_bgcolor = '#777777',
                        legend=dict(bgcolor='lightgray')
                    )
    fig.add_annotation(text = 'pi', xref = 'paper', yref = 'paper', y = math.pi,
                        font=dict(size=15, color="red"), showarrow = False)
    fig.update_xaxes(range = [-0.05, 1.05], title = 'X<sub>1</sub> ~ Unif(0,1)',
                        tickfont_size = 15, title_font_size = 15, scaleanchor = 'y',
                        gridwidth = 1, griddash = 'dot', zerolinecolor = 'black',
                        zerolinewidth = 1, row = 1, col = 1, constrain = 'domain')
    fig.update_yaxes(range = [-0.05, 1.05], title = 'X<sub>2</sub> ~ Unif(0,1)',
                    tickfont_size = 15, title_font_size = 15, gridwidth = 1,
                    griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1,
                    row = 1, col = 1, constrain = 'domain')
    fig.update_yaxes(range = [min(pi[0:8] - 0.1), max(pi[0:20] + 0.1)],
                    title = 'pi', tickfont_size = 15, title_font_size = 15,
                    gridwidth = 1, griddash = 'dot', zerolinecolor = 'black',
                    zerolinewidth = 1, row = 1, col = 2)
    fig.update_xaxes(range = [-100, n], title = 'Number of data points',
                    tickfont_size = 15, title_font_size = 15, gridwidth = 1,
                    griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1,
                    row = 1, col = 2)
    fig.show()
```

8.2.12 Random Walk

```
def random_walk(unifs):

    fig = go.Figure()
    # Loop through the three generators
    for unif in unifs:
        # Inside the np.cumsum function, we take each value and assign it -1 if it is less than
        # zero, and +1 if it is greater than or equal to zero. And the cumsum function adds is
        # is just performing a cumulative sum
        walk = np.cumsum(2*(unif['unif'] >= 0.5) - 1)
        trace = go.Scatter(x = np.arange(start = 0, stop = len(unif['unif']), step = 1),
                           y = walk,
                           mode = 'lines',
                           marker = dict(color = unif['color']),
                           name = unif['name'])
        fig.add_trace(trace)
        fig.add_hline(y = 0, line_width = 1, line_color = 'red')

    fig.update_xaxes(title_text = 'Step Number', title_font_size = 20,\
                    tickfont_size = 15, gridwidth = 1, griddash = 'dot',\
                    zerolinecolor = 'black', zerolinewidth = 1)
    fig.update_yaxes(title_text = 'Displacement', title_font_size = 20,\
                    tickfont_size = 15, gridwidth = 1, griddash = 'dot',\
                    zerolinecolor = 'black', zerolinewidth = 1)
    fig.update_layout(title = {'text' : '<b>Random Walk</b>',
                               'x' : 0.5,
                               'y' : 0.95,
                               'xanchor' : 'center',
                               'yanchor' : 'top',
                               'font' : {'textcase' : 'upper',
                                         'size' : 25}
                                        },
                    legend=dict(bgcolor='lightgray'),
```

```
                    width = 1000,
                    height = 600,
                    paper_bgcolor = 'white',
                    plot_bgcolor = '#777777')
    fig.show()
```

8.2.13 Random Variate Generation

```
def exp(gen_list, gen):
    for item in gen_list:
        if item['name'] == gen:
            unif = item['unif']
            nameU = item['name']
            colorU = item['color']

    exp_1 = - np.log(1 - unif)
    print(f'----------------------------')
    print(f'Y ~ Exp(1): {exp_1}')
    print(f'----------------------------')
    fig = go.Figure(data = [go.Histogram(x = exp_1,
                                         xbins = dict(start = min(exp_1), end = max(exp_1), size = 1/5),
                                         marker=dict(line=dict(color='black', width=2), color = colorU),
                                         hovertemplate="<br>".join([f'{nameU}', 'Range: %{x}',
                                                                    'Count of Random Numbers: %{y}']),
                                         name = '')])
    fig.update_layout(title = {'text' : f'Random Variate Generation Exp(1)<br><sub>{nameU}</sub>',
                               'x' : 0.5,
                               'y' : 0.95,
                               'xanchor' : 'center',
                               'yanchor' : 'top',
                               'font' : {'size' : 25}},
                      width = 800,
                      height = 600,
                      paper_bgcolor = 'white',
                      plot_bgcolor = '#777777')
    fig.update_xaxes(title_text = 'Random Variate, Y', title_font_size = 20, tickfont_size = 15,
                     gridwidth = 1, griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1)
    fig.update_yaxes(title_text = 'Frequency', title_font_size = 20, tickfont_size = 15,
                     gridwidth = 1, griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1)
    fig.show()

    return exp_1

def tria(gen_list, gen):
    # Get the distribution
    for item in gen_list:
        if item['name'] == gen:
            unif1 = item['unif']
            unif2 = item['unif2']
            nameU = item['name']
            colorU = item['color']

    # Add the two distributions to generate the triangular distribution
    tria_0_1_2 = np.add(unif1,unif2)
    print(f'----------------------------')
    print(f'Y ~ Tria(0, 1, 2): {tria_0_1_2}')
    print(f'----------------------------')
    fig = go.Figure(data = [go.Histogram(x = tria_0_1_2,
```

```
                                        xbins = dict(start = 0, end = 2, size = 1/20),
                                        marker=dict(line=dict(color='black', width=2), color = colorU),
                                        hovertemplate="<br>".join([f'{nameU}', 'Range: %{x}',
                                        'Count of Random Numbers: %{y}']),
                                        name = '')])
    fig.update_layout(title = {'text' : f'Random Variate Generation Tria(0, 1, 2)<br><sub>{nameU}</sub>',
                               'x' : 0.5,
                               'y' : 0.95,
                               'xanchor' : 'center',
                               'yanchor' : 'top',
                               'font' : {'size' : 25}},
                      width = 800,
                      height = 600,
                      paper_bgcolor = 'white',
                      plot_bgcolor = '#777777')
    fig.update_xaxes(title_text = 'Random Variate, Y', title_font_size = 20, tickfont_size = 15,
                         gridwidth = 1, griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1)
    fig.update_yaxes(title_text = 'Frequency', title_font_size = 20, tickfont_size = 15,
                         gridwidth = 1, griddash = 'dot', zerolinecolor = 'black', zerolinewidth = 1)
    fig.show()

    return tria_0_1_2

def nor(gen_list, gen):
    # Get the unif distribution
    for item in gen_list:
        if item['name'] == gen:
            unif1 = item['unif']
            unif2 = item['unif2']
            nameU = item['name']
            colorU = item['color']

    # Use the Box-Muller equations to generate Nor
    z1 = np.sqrt(-2 * np.log(unif1)) * np.cos(2 * math.pi * unif2)
    z2 = np.sqrt(-2 * np.log(unif1)) * np.sin(2 * math.pi * unif2)
    print(f'----------------------------')
    print(f'Z\u2081 ~ Nor(0, 1): {z1}')
    print(f'----------------------------')
    print(f'Z\u2082 ~ Nor(0, 1): {z2}')
    print(f'----------------------------')

    trace1 = go.Histogram(x = z1,
                         xbins = dict(start = min(z1), end = max(z1), size = 1/5),
                         marker=dict(line=dict(color = 'black', width = 2), color = colorU),
                         hovertemplate="<br>".join([f'{nameU}', 'Range: %{x}',
                         'Count of Random Numbers: %{y}']),
                         name = '')
    trace2 = go.Histogram(x = z2,
                         xbins = dict(start = min(z2), end = max(z2), size = 1/5),
                         marker=dict(line=dict(color = 'black', width = 2), color = colorU),
                         hovertemplate="<br>".join([f'{nameU}', 'Range: %{x}',
                         'Count of Random Numbers: %{y}']),
                         name = '')
    fig = make_subplots(rows = 1, cols = 2, subplot_titles = ["Distribution of Z<sub>1</sub> ~ Nor(0, 1)",
                                                              "Distribution of Z<sub>2</sub> ~ Nor(0, 1)"])
    fig.add_trace(trace1, row = 1, col = 1)
    fig.add_trace(trace2, row = 1, col = 2)
    fig.update_layout(title = {'text' : f'Random Variate Generation Nor(0, 1)<br><sub>{nameU}</sub>',
                               'x' : 0.5,
                               'y' : 0.95,
                               'xanchor' : 'center',
```

```
                            'yanchor' : 'top',
                            'font' : {'size' : 25}},
                width = 1200,
                height = 600,
                paper_bgcolor = 'white',
                plot_bgcolor = '#777777')
    fig.update_xaxes(title_text = 'Random Variate, Z<sub>1</sub>', title_font_size = 20,
                tickfont_size = 15, gridwidth = 1, griddash = 'dot',
                zerolinecolor = 'black', zerolinewidth = 1, row = 1, col = 1)
    fig.update_xaxes(title_text = 'Random Variate, Z<sub>2</sub>', title_font_size = 20,
                tickfont_size = 15, gridwidth = 1, griddash = 'dot',
                zerolinecolor = 'black', zerolinewidth = 1, row = 1, col = 2)
    fig.update_yaxes(title_text = 'Frequency', title_font_size = 20, gridwidth = 1,
                griddash = 'dot', tickfont_size = 15, zerolinecolor = 'black',
                zerolinewidth = 1)
    fig.update_layout(showlegend = False)
    fig.show()

    return z1, z2
```