Tausworthe Pseudo-Random Number Generator Implementation

Jay Nonaka

Abstract

The focus of this project is the code implementation of the Tausworthe pseudo-random number (PRN) generator. Developed by Peter Tausworthe in 1965, this PRN generator is an application of linear-feedback shift register (LFSR) where the generated PRN is a deterministic and linear function of its previous state. This generator possesses many excellent properties including long periods, relatively easy implementation, and computational efficiency. A walk-through of the Tausworthe PRN generator code implementation and steps to initialize and generate pseudo-random numbers are documented.

After code implementation documentation, the Tausworthe PRN generator is initialized with default parameters. The quality of the generator with the default parameters is assessed via statistical tests and visual inspection of adjacent plots, and the implemented Tausworthe PRN generator is used to generate a pseudo-random unit normal via a Box-Muller transformation. As a result of this project, it is concluded that the Tausworthe PRN generator is an effective pseudo-random number generator which can be implemented easily.

.

Background & Problem Description

Pseudo-random number (PRN) generators are algorithms designed to produce number sequences that are approximately random. The numbers generated from these algorithms are not truly random as the number sequence generated by these algorithms are deterministic based on the initial state (i.e., seed) used when initializing the generator. However because of this property, PRN generators have the advantage of being able to produce the same PRNs if the seed remains the same. PRNs are particularly important in the realm of simulations as it allows for the generation of random variables which can be easily reproduced.

The Tausworthe PRN generator was first developed in 1965 by Peter Tausworthe. Prior to the Tausworthe PRN generator, there were other algorithms/methodologies that were utilized to produce PRNs. Some early examples include random devices (e.g., coin flips, Geiger counters, etc.), random number tables, and the mid-square method developed by John von Neuman. However, these early algorithms and methodologies various shortcomings which included:

- Relatively short periods (i.e., the length of the repeating cycle) depending on the seed
- Poor fit against a true uniform distribution
- Correlation among adjacent PRNs (i.e., PRNs are not independent)
- Unable or difficult to replicate the same sequence of PRNs (i.e., not reproducible)

When the right parameters are selected, the PRNs from a Tausworthe PRN generator do not have the above shortcomings. Additionally, the Tausworthe PRN generator is easy to implement and computationally efficient. While there are more advanced and secure PRN generators today, the Tausworthe PRN generator was influential for its good properties.

In the following Main Findings section, the below topics and results will be discussed and analyzed:

- Tausworthe PRN Algorithm Explanation
- Code Implementation Documentation

- Initialization and PRN Generation
- Statistical Tests and Adjacent Plots
- Unit Normal Generation (Box Muller Method)

Main Findings

Tausworthe PRN Generator Algorithm

Given an initial sequence of binary digits $B_1, B_2, B_3, \ldots$ (the seed), the i[th] binary digit B_i can be defined as

$$B_i = \left(\sum_{j=1}^{q} c_j B_{i-j} \right) mod\ 2$$

Put simply, B_i is calculated by taking the summation of the q previous binary bits multiplied by a constant c_j (0 or 1) and applying a $mod\ 2$ to generate a binary value for B_i. A more computationally efficient version of this algorithm can be written as

$$B_i = \left(B_{i-r} + B_{i-q}\right) mod\ 2 = B_{i-r}\ XOR\ B_{i-q} \qquad 0 < r < q$$

In the above definition, the algorithm simply applies the XOR operator (i.e., either this or that, but not both) between the r^{th} previous binary bit (B_{i-r}) and the q^{th} previous binary bit (B_{i-q}). In other words, B_i is equal 0 if $B_{i-r} = B_{i-q}$ and 1 if if $B_{i-r} \neq B_{i-q}$. Finally to generate a PRN, take a sequence of l bits in base 2 and divide by 2^l to convert to a base 10 PRN between 0 and 1.

Code Implementation Documentation

The Tausworthe PRN generator is implemented in a Jupyter Notebook via the TausworthePRNG class. This class requires the user to input the following four parameters (in order) to initialize:

- $seed$ – A binary value that sets the initial state of the Tausworthe PRN generator
- r – The r^{th} previous binary bit (first binary bit input for the XOR operator)
- q – The q^{th} previous binary bit (second binary bit input for the XOR operator)
- l – The length of bits in base 2 to be used to generate a base 10 PRN between 0 and 1

The user can select any value for each of the parameters above to customize the resulting Tausworthe PRN generator.

The TausworthePRNG class contains the following functions:

- generate_num – Main function. Generates one PRN (base 10)
- convert_bin_to_num – Helper function utilized within generate_num. Converts binary sequence of length l to a base 10 PRN
- update_bin_seq – Helper function utilized within generate_num. Updates the binary sequence by applying the XOR operator to generate new binary bits in the entire sequence
- checks – Basic checks performed on the inputted parameters ($seed$, r, q, and l). The class will fail to initialize if an input error is detected with one or more of the parameters.

Initialization and PRN Generation

TausworthePRNG class can be initialized and used to generate PRNs in two steps:

1. Set the parameter values and initialize the class. The default values set in the Jupyter Notebook are $r = 17$, $q = 27$, $l = 24$, and $seed = 10100010101100$ (10,412 in base 10)

```
r, q, l = 17, 27, 24
seed = 0b10100010101100

TPRN_generator = TausworthePRNG(seed, r, q, l)
```

2. Call the generate_num() function to generate a single PRN. Alternatively, list comprehension can be used to generate multiple PRNs at once (2,000 PRNs are generated in the code block below)

```
# Change the below number to the number of random numbers desired
rand_nums_to_generate = 2000

# Generated random numbers in a list
rand_nums = [TPRN_generator.generate_num() for i in range(rand_nums_to_generate)]
```

Statistical Tests and Adjacent Plots

To assess the quality of the Tausworthe PRN generator with the default parameters ($r = 17$, $q = 27$, $l = 24$, and $seed = 10100010101100$), several statistical tests are performed, and adjacent plots are generated. To begin, two thousand PRNs are generated using this Tausworthe PRN generator.

First to test for goodness-of-fit against a Unif(0,1) distribution, a Chi-Squared (χ^2) Test is performed. The null hypothesis for this test is that the randomly generated PRNs are uniform. When using $k = 10$ bins, the resulting p-value from the Chi-Squared test is 0.843. As a result, the null hypothesis is not rejected, and the PRNs are a good fit against a Unif(0,1).

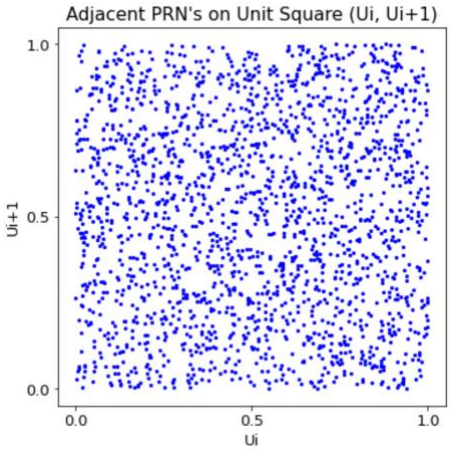
	(0.0, 0.1]	(0.1, 0.2]	(0.2, 0.3]	(0.3, 0.4]	(0.4, 0.5]	(0.5, 0.6]	(0.6, 0.7]	(0.7, 0.8]	(0.8, 0.9]	(0.9, 1.0]
PRN Count	185	200	194	196	196	202	224	209	199	195

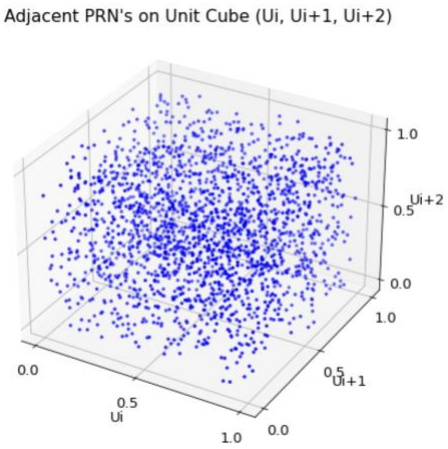
```
Null Hypothesis: The randomly generated data points are uniform
alpha = 0.05
**************************
Chi-Square = 4.9
p value = 0.843
**************************
Failed to reject the null hypothesis
```

Next, an independence test is performed to ensure that each PRN is independent (i.e., not correlated with other PRNs). To test for independence, a runs test is employed to count the number of positive or negative runs observed in the generated sequence fo PRNs. The resulting p-value of this test is 0.958, so the null hypothesis is not rejected. As a result, the generated PRNs are independent.

```
Null Hypothesis: The randomly generated data points are independent
alpha = 0.05
**************************
Z-statistic = 0.052
p value = 0.958
**************************
Failed to reject the null hypothesis
```

Finally, adjacent PRNs are plotted on a unit square and a unit cube. Plotting adjacent PRNs serves as a visual check to ensure that there are not any discernable patterns apparent in the plot. Observing any type of pattern in the plots would indicate that PRNs are not random. As can be seen below, both plots appear to be random as no obvious pattern is visible.





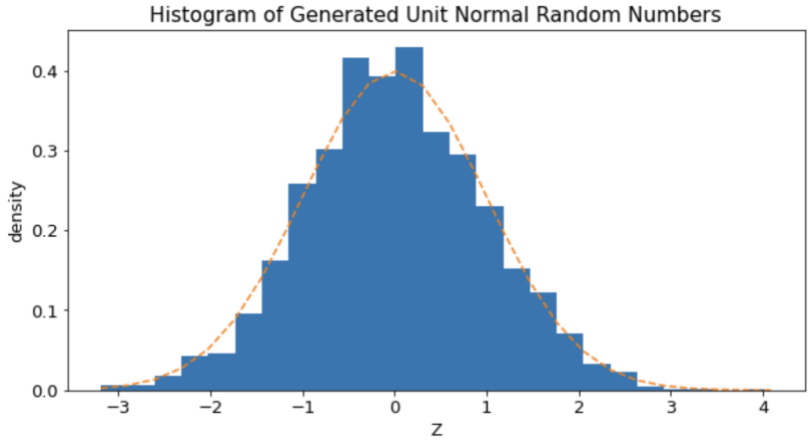
As a result of the statistical tests and the visual inspection of the adjacent PRN plots, the implemented Tausworthe PRN generator with this set of parameters can produce good PRNs.

Unit Normal Generation (Box Muller Method)

For further application, the Tausworthe PRN generator can be used to generate a unit normal random variable. Two sets of uniform PRNs can be transformed into unit normal random variables using the Box-Muller transformation. The equation for the Box-Muller transformation is

$$Z = \sqrt{-2\ln(U_1)}\cos(2\pi U_2)$$

A second set of two thousand PRNs are generated using the Tausworthe PRN generator with the same parameters. With both sets of PRNs, unit normal random variables are generated. The below density histogram of the unit normal random variables shows that generated variables closely follow the probability density function of a unit normal distribution. This histogram serves as further proof that the the implemented Tausworthe PRN generator with this set of parameters can produce good PRNs



Conclusion

In conclusion, the Tausworthe PRN generator is a highly effective PRN generator. The statistical tests performed indicate that the generator with the default parameters produces PRNs that are both independent and fits well against a true uniform distribution, and the visual inspection of adjacent PRN

plots exhibit good randomness. Finally, the unit normal random variables generated via the Box-Muller transformation of the generated PRNs result in an approximate unit normal distribution density which closely follows the probability density function of a true unit normal distribution. By all measures discussed in this project, the Tausworthe PRN generator with the default parameters is a good PRN generator especially when considering the relatively ease to implement and the computation efficiency.