# ISYE 6644 Mini-Project 2: Tausworthe Pseudo-Random Number Generator

May 3, 2021

Kaitlyn Boyle

## Abstract:

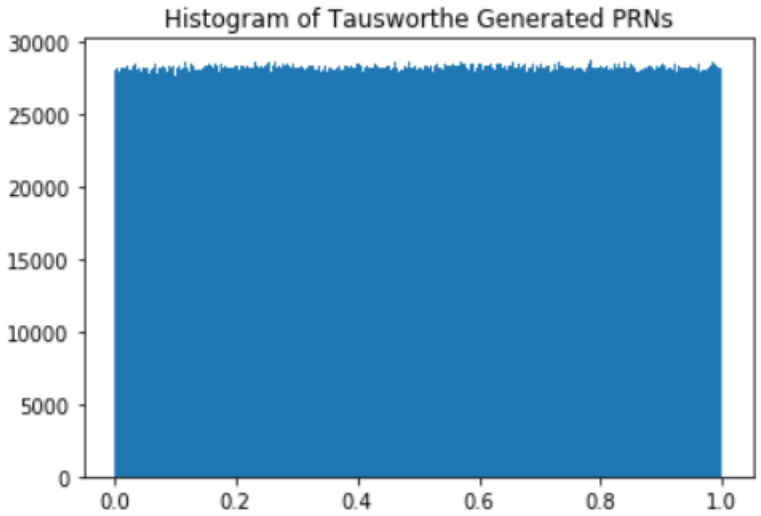
In this project, I implemented a Tausworthe pseudo-random number generator using Python and performed several statistical tests to determine if the generator produces independent and uniformly distributed pseudo-random numbers. Independence and uniform distribution are necessary requirements for good pseudo-random number generators. I performed a Chi-square goodness of fit test to determine if the random numbers generated are approximated uniformly distributed and a Runs Above and Below the Mean test and correlation test to determine if the random numbers are independent and uncorrelated. Based on these results, we can conclude that the implementation of the Tausworthe generator produced pseudo-random numbers that are independent and uniformly distributed.

## Background:

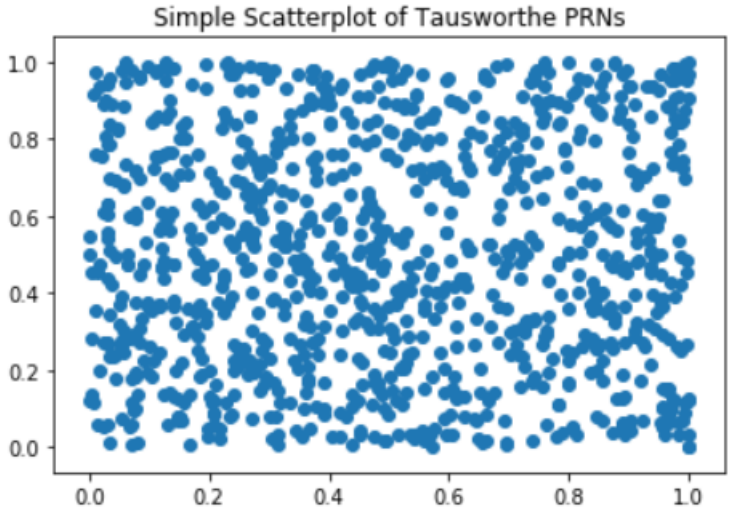
Random number generation is the basis to many simulation applications. Uniform random numbers are used in simulation to create other types of random variables. Because it is nearly impossible to generate truly random numbers, we want to use an algorithm to produce pseudo-random numbers that are deterministic but appear random. The most important properties of a pseudo-random number generator are that the output appears to be independent and identically uniformly distributed, the algorithm is fast, and the generator can replicate any output that it produces. The most widely used generator is called a linear congruential generator. They are of the form $X_i = (aX_{i-1} + c) \bmod m$, where $X_0$ is the seed, and $R_i = X_i / m$. $R_i$ is the generated pseudo random number from the algorithm. The constants a, c, and m can be chosen to get good statistical qualities, such as a long cycle length. One popular linear congruential generator is called the Tausworthe generator. The Tausworthe generator was introduced by Robert Tausworthe in 1965 in his paper “Random Numbers Generated by Linear Recurrence Module Two” (Tausworthe, 1965). Tausworthe introduced the pseudo-random number generator by performing several statistical tests on the generator to show that it produces uniformly distributed independent pseudo-random numbers. The Tausworthe generator is known to be fast, easily implemented and produces pseudo-random numbers that are independent and uniformly distributed. It is also capable of reproducing any sequences it outputs. The Tausworthe Generator algorithm is defined as a sequence of binary digits $B_1, B_2, \ldots$ by $B_i = (\sum_{j=1}^{q} c_j B_{i-j}) \bmod 2$ where $c_j$ is 0 or 1. I implemented this in Python using $B_i = (B_{i-r} + B_{i-q}) \bmod 2$, where r and q are relatively large prime numbers. I used this to produce a sequence of binary digits of length $2^q$. I initiated the binary sequence by setting Bi through Bq to be 1. I then produced the sequence of binary digits and converted the $B_i$’s to Unif(0,1)’s using the formula $u = (\text{l-bits in base 2})/2^l$, a formula from the module 6 lecture notes on the Tausworthe generator (Goldsman, 2018). L is also a prime number. I will show that this generator is easily implemented, can reproduce its results, and generates independent and uniformly distributed pseudo-random numbers.

# Main Findings:

I wrote the code to implement the generator in Python, initializing the sequence with r = 23, q = 29, and l = 19. I stored the binary sequence and the pseudo-random numbers to perform statistical tests on. These results are easily implemented in Python by simply defining a function to produce the binary sequence and convert the sequence to Unif(0,1)s using formulas. Since the binary sequence is based on the initialization of a binary sequence of $B_1$ to $B_q$ of all 1's, the results are easily reproducible.
After writing the algorithm in Python, I used the generator to produce $2^q$ (28,256,365) pseudo-random numbers, a large number that will be useful in testing the statistical properties of the generator. After producing the pseudo-random numbers, I then needed to check that the pseudo-random numbers are uniformly distributed and independent of each other. I started with plotting the histogram of all the pseudo-random numbers in Python.



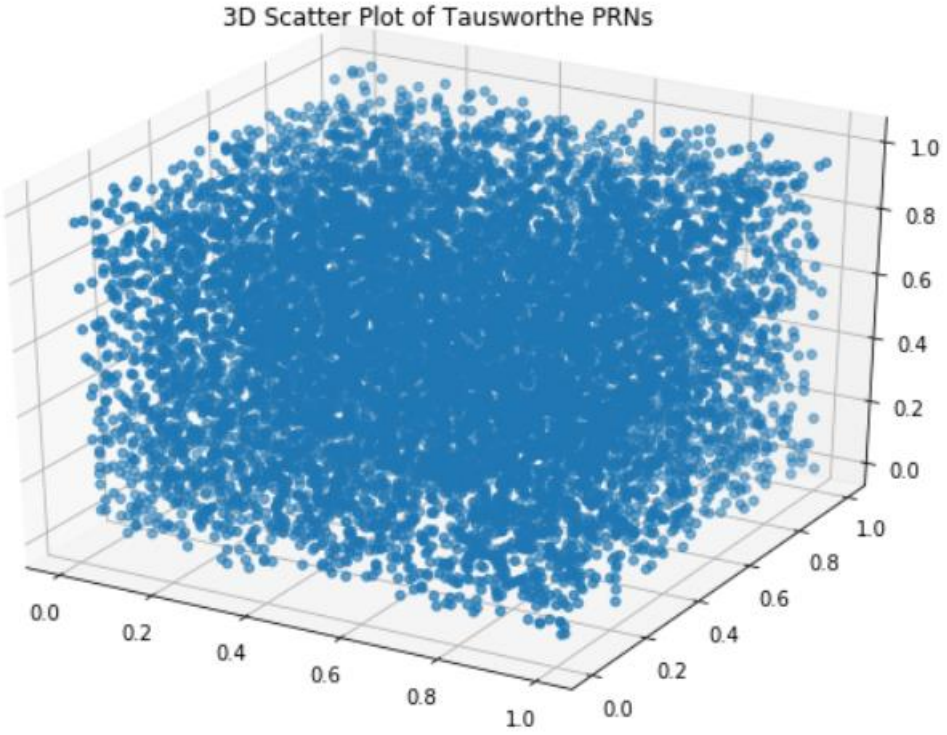

From the histogram, it is easy to see that the pseudo-random numbers follow a uniform distribution. I then plotted a scatterplot of pairs of ($U_{i-1}$, $U_i$) pseudo-random numbers to check for randomness in the




distribution and to check for any patterns in the generated data.

I used a sample of 1,000 pseudo-random numbers from the data so that we can see any patterns and not overwhelm the plot with too many data points. We can see from this plot that there is randomness

throughout the data, and there does not appear to be any patterns in the data. I also plotted a 3D scatterplot using ($U_{i-2}$, $U_{i-1}$, $U_i$) pairs to get another look at the data.

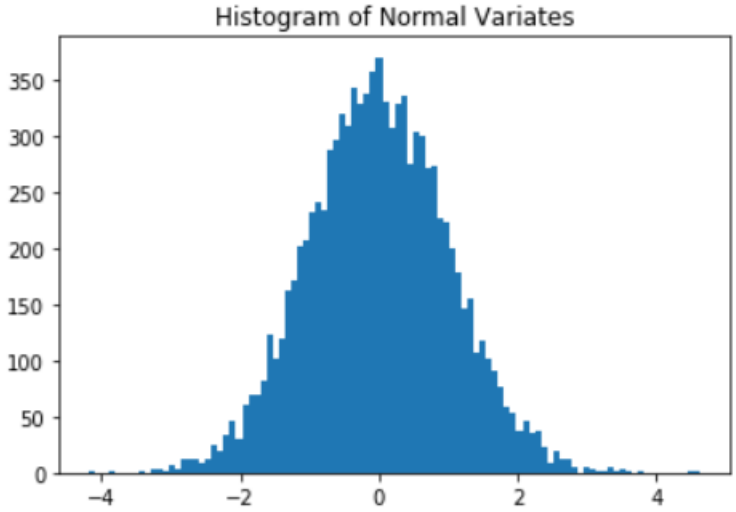



Using a sample of 10,000 pseudo-random generated numbers, this 3D scatterplot provides a better view of the data and makes it clear that there are no patterns in the data, and the pseudo-random numbers are distributed randomly throughout the plot.

Statistical tests for determining the uniform distribution and independence are a vital step to producing pseudo-random numbers. All test procedures are taken from the module 6 lecture slides, "Generating Uniform Random Numbers" from week 8 of the course (Goldsman, 2018). I started with a goodness of fit test. Goodness of fit tests are used to check for the uniform distribution. I used an approximation of the chi-squared goodness of fit test. The chi-squared goodness of fit test has a null hypothesis that the observations being tested are uniform. It uses a test statistic $X_0^2 = \sum_{i=1}^{k} (O_i - E_i)^2/E_i$, where $O_i$ are the observed values and $E_i$ are the expected values. The data is split into k bins, and the null hypothesis is rejected for large values of $X_0$. Since we have an exceptionally large number of observations and would need a large value of k to split the data, we can use an approximation of the test instead. I used the formula from lecture notes, $X^2_{a,\,k-1} = (k-1)\,[1 - (2/9(k-1)) + z_a(\text{Sqrt}(2/(9(k-1))))]^3$. We can then compare this approximation to the chi-squared test statistic from the chi-square table. I used an alpha level of 0.05, and k-1 = 999, so the critical value was 1074.679. The approximated test statistic was calculated as 913.298, which is less than the critical value, so we fail to reject the null hypothesis and conclude that the pseudo-random numbers from the generator are uniformly distributed.

Another important aspect to test for is independence amongst the pseudo-random numbers. I started with implemented a Runs Above and Below the Mean test. This tests for independence of the Pseudo-random numbers. I implemented this is Python, defining a function to count the above and below runs, where n1 = number of above runs, n2 = the number of below runs, and the mean of the pseudo-random numbers used in the test. A portion of the code is adopted from an article "Runs Test of Randomness in Python" (GeeksForGeeks, 2021), and the formula used to produce the test statistic is from the lecture from Bucknell University, "Runs Test*"* (Xiannong, 2002). The test statistic is $Z_0 = (B - E[B])/\text{sqrt}(\text{Var}(B))$. The null hypothesis is that the pseudo-random numbers are independent. We reject the null hypothesis if $Z > Z_{a/2}$. I used an alpha value of 0.05, so the critical value was 1.96, and the calculated $Z_0$ from the

Python implementation is 0.619. Since this value is lower than 1.96, we fail to reject the null hypothesis, and conclude that the pseudo-random numbers are independent.
Another test for independence is the correlation test to check for correlation between the pseudo-random numbers. I defined a function in Python to check for correlation, which is defined as $p = (12/(n-1) * \sum_i^{n-1} R_k R_{1+k}) - 3$. $Z_0 = p / \text{sqrt}(\text{Var}(p))$, and we reject the null hypothesis of independent pseudo-random numbers if $|Z_0| < z_{a/2}$. I again used an alpha of 0.05, so the critical value was 1.96. I calculated $Z_0 = 0.814$ in Python, and since $Z_0 < 1.96$, we fail to reject the null hypothesis and conclude that the Pseudo-random numbers are uncorrelated and independent.
The final step in the project is to use the Uniform pseudo-random numbers to generate Normal deviates. Normally, we could use the Inverse Transform method to do this, but the Normal distribution's c.d.f. does not have an analytical form. I instead used Scipy's statistics package and the norm.ppf() function to generate Normal deviates. The function uses the inverse of the Normal c.d.f. and takes an input of a Uniform random number. I used 10,000 random numbers generated from my Tausworthe generator to produce 10,000 normal random deviates. I then plotted their histogram to visualize the deviates.




This histogram shows that the 10,000 produced deviates approximately follow a normal distribution, centered about 0.

# Conclusions:

Generating pseudo-random numbers is an essential step of many simulation applications, and testing for the uniform distribution and independence amongst pseudo-random numbers is an important part of the process. I used several tests to check for these requirements. I used a histogram and scatterplots to visualize the pseudo-random numbers from the Tausworthe generator implementation, which showed that the pseudo-random numbers were uniformly distributed. I used a chi-squared goodness of fit test to test for the Uniform distribution, and from the test statistic I concluded that the pseudo-random numbers are uniformly distributed. To test for independence, I used a Runs Above and Below the Mean test. I found a Z test statistic lower than the Z critical value and concluded that the pseudo-random numbers are independent. I also used a correlation test to check for independence and correlation between the pseudo-random numbers. I found a Z test statistic lower than the Z critical value, and again concluded that the pseudo-random numbers are independent and uncorrelated. Since the pseudo-random numbers passed the tests for independence and the Uniform distribution, I used a sample of 10,000 of generated numbers to generate 10,000 Normal deviates. I plotted a histogram to visualize the Normal distribution of the Normal deviates produced from the Pseudo-random numbers.
Overall, we found that the Tausworth generator is a good pseudo-random number generator. The generator creates independent and uniformly distributed pseudo-random numbers, is efficient and fast to

implement, and can be replicated. These are all the hallmarks of a good pseudo-random number generator.
In the future, I would rewrite my code for the generator to implement faster. It can take a long time to produce enough pseudo-random numbers from the generator I created because of the way I am storing it in Python. It is computationally expensive when it could be simpler. For q and r close to but less than 20, such as 19 and 17, the generator is very quick. But when using larger q and r, such as 29 and 23, the generator can take an exceptionally long time to implement, which is a flaw in my code. This can be improved with better coding, and not storing all binary digits that are produced.

# Appendix:

All code can be found in the Jupyter notebook. I have included the code for the functions defined for the Tausworthe generator here, but all implementation of graphs and statistical tests can be found in the Jupyter notebook.

```
def taus_period(r, q):
    initial = '1'*q
    full = initial
    k = (2**q)-1
    y = q + 1
    for x in range(k):
        bi = int(full[y-r-1]) + int(full[y-q-1])
        new = bi%2
        full = full + str(new)
        y += 1
    return full
g = taus_period(23, 29)
len(g)
```

Code to generate binary sequence

```
def convert(l):
    n = len(g)
    y = l
    x = 0
    randoms = []
    while y < n+1:
        if y == n:
            l_str = g[x:]
        else:
            l_str = g[x:y]
        x += l
        y += l
       # if l_str == l:
        convert_b = int(l_str, base=2)
        prn = convert_b/(2**l)
        randoms.append(prn)
    return randoms
prns = convert(19)
print(len(prns))
prns[0:20]
```

Code to convert binary string to Unif(0,1) pseudo-random numbers

# References


Christos Alexopoulus, D. G. (2018, May 26). *Generating Uniform Random Numbers.* Retrieved from Georgia Institue of Technology: Module06-RandomNumberGenerationSlides_180526.pdf

*Linear-Feedback Shift Register*. (2021, May 2). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Linear-feedback_shift_register

*Random Number Generation*. (2021, May 2). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Random_number_generation

*Runs Test of Randomness in Python*. (2020, June 08). Retrieved from GeeksForGeeks: https://www.geeksforgeeks.org/runs-test-of-randomness-in-python/

*Scipy.stats.norm Documentation*. (2021, April 26). Retrieved from SciPy.Org: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html

Tausworthe, R. C. (1965, May). *Random Numbers Generated by Linear Recurrence Module Two.* Retrieved from American Mathematical Society: https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0184406-1/S0025-5718-1965-0184406-1.pdf

Xiannong, M. (2002, October 18). *Runs Test*. Retrieved from Bucknell University: https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node44.html