ISYE 6644 SIMULATION: EXPLORATION OF UNIFORM RANDOM NUMBER GENERATION THROUGH IMPLMENENTATION, VISUALIZATION, AND INDEPEDENCE AND EFFICIENCY TESTING OF SEVERAL GENERATORS

Matthew Kesselman (Team 76, solo group)
Matthewkesselman3@gatech.edu

ABSTRACT

The effective generation of random numbers is critical in a broad array of fields and disciplines—everything from selecting subjects in randomized trials to ensuring "fairness" in gambling to complex simulations—and the need for *quality* random number generation is likely to grow ever more crucial as novel applications demand random numbers as inputs and emerging technologies threaten to predict pseudorandom numbers[1] (which could lead to exploitive behavior). Software uses pseudorandom numbers, numbers that *should* emulate the distribution of truly random numbers (in other words, be statistically independent), but have a mathematical origin and yield a regular pattern kicked off by a "seed" value. This project explores a class of pseudorandom number generators—uniform random number generators (from 0 to 1)—through the implementation and examination of seven different generators using Python: five linear congruential generators of various forms, one generator using the "mid-squared method", and one generator from the Python package Random, an implementation of the Mersenne Twister[2] algorithm. For each generator, visualizations were created to assess the uniformity and independence of the generators' output both in two-dimensional and three-dimensional spaces. Following these visual inspections, multiple experiments—goodness of fit tests, up down tests, correlation tests—were performed to assess the uniformity of the generators' output. Lastly, a final experiment examines the generation time of different generators (as implemented) to describe and account for the efficiency of different generators.

GENERATORS: IMPLEMENTATION

This project focuses primarily on one class of random number generators, linear congruential generators (LCG), supplemented with a study of the "mid-square" uniform generator and Python's Random package's Mersenne Twister implementation. Linear congruential generators are defined by a sequence starting with an arbitrary "starting seed" X_0 and takes the form:

$$X_{k+1} = (a \cdot X_k + c) \; mod \; m$$

Uniforms are generated out from this sequence by dividing by the modulus m:

$$U_k = X_k \; / m$$

[1] G. Amigo, L. Dong and R. J. Marks Ii, "Forecasting Pseudo Random Numbers Using Deep Learning," *2021 15th International Conference on Signal Processing and Communication Systems (ICSPCS)*, Sydney, Australia, 2021, pp. 1-7, doi: 10.1109/ICSPCS53099.2021.9660301.
[2] https://docs.python.org/3/library/random.html

An LCG where $c = 0$ is a special form of an LCG, a **multiplicative** linear congruential generator[3]. A further specialized form of a multiplicative LCG is one where the modulus m is a prime number (ex a "Mersenne prime"[4], a prime number that is one less than the power of two: 2^n-1) and a is a primitive root of m (ex: $a = 7^5$ for $m = 2^{31}$-1), known as an Lehmer LCG[5].

The LCGs implemented for this project are as follows:

- "parkMillerOne", defined by: $X_i = 16807 \cdot X_{i-1} \; mod \; 2^{31} - 1$
- "parkMillerTwo", defined by: $X_i = 48271 \cdot X_{i-1} \; mod \; 2^{31} - 1$
- "advancedLehmerLCG", defined by: $X_i = 50653 \cdot X_{i-1} \; mod \; 2^{61} - 1$
- "randu", defined by: $X_i = 65539 \cdot X_{i-1} \; mod \; 2^{31}$
- "simpleLCG", defined by: $X_i = 17 \cdot (X_{i-1} + 2) \; mod \; 94$

These various LCGs of study were procured from both positive and negative historical connotations (one example on RANDU, "When this chapter was first written in the late 1960's, a truly horrible random number generator called RANDU was commonly used on most of the world's computers" (Donald Knuth, The Art of Computer Programming), and a generic, simple generator with a small, incomplete period (meaning it does not loop fully through its modulus), "simpleLCG."

The Linear Congruential Generators: Implementation

The following are the LCGs' implementation, along with brief descriptions:

parkMillerOne (a multiplicative linear congruential generator using the Mersenne prime, m = 2^{31} – 1 and a= 7^5) developed by Stephen Park and Keith Miller[6], taking the form:

$parkMillerOne: X_i = 16807 \cdot X_{i-1} \; mod \; 2^{31} - 1$

```
class parkMillerOne():

    def __init__(self, seed=33):
        self.seed = seed
        self.current_state = 16807 * seed % (2**31-1)
        self.next_state = 16807 * self.current_state % (2**31-1)

    def getNextState(self):
        self.current_state = self.next_state
        self.next_state = 16807 * self.current_state % (2**31-1)

        return self.current_state

    def getDenom(self):
        return (2**31-1)
```

[3] J. Harris. 2003. "SOME NOTES ON MULTIPLICATIVE CONGRUENTIAL RANDOM NUMBER GENERATORS WITH MERSENNE PRIME MODULUS." Journal of the South Carolina Academy of Science l(l):28-32 Fall 2003.
[4] Mersenne Research, Inc. "List of Known Mersenne Prime Numbers" https://www.mersenne.org/primes/
[5] W. H. Payne, J. Rabung, and T. Bogyo "Coding the Lehmer Pseudorandom Number Generator" *COMMUNICATIONS OF THE ACM* Volume 12 / Number 2 / February, 1969 https://www.firstpr.com.au/dsp/rand31/p85-payne.pdf
[6] S. Park and K. Miller " RANDOM NUMBER GEUERATORS: GOOD ONES ARE HARD TO FIND" *COMMUNICATIONS OF THE ACM July 1993/Vol,36, No.7*. https://www.firstpr.com.au/dsp/rand31/p105-crawford.pdf#page=4

Park and Miller referred to the above generator as the "**minimal standard**."[6] This project explored Park, Miller, and Stockmeyer's alternative suggestion, which they offered following some criticism[7]: parkMillerTwo, changing the multiplier $a = 48271$.

$parkMillerTwo$: $X_i = 48271 \cdot X_{i-1} \; mod \; 2^{31} - 1$

```
class parkMillerTwo():

    def __init__(self, seed=33):
        self.seed = seed
        self.current_state = 48271 * seed % (2**31-1)
        self.next_state = 48271 * self.current_state % (2**31-1)

    def getNextState(self):
        self.current_state = self.next_state
        self.next_state = 48271 * self.current_state % (2**31-1)

        return self.current_state

    def getDenom(self):
        return (2**31-1)
```

Another, more potentially computationally complex with a longer period Lehmer LCG is "advancedLehmerLCG".

$advancedLehmerLCG$: $X_i = 50653 \cdot X_{i-1} \; mod \; 2^{61} - 1$ (Note: 37 is a primitive root of the Mersenne number $2^{61} - 1$, $37^3 = 50{,}653$)

```
class advancedLehmerLCG():

    def __init__(self, seed=33):
        self.seed = seed
        self.current_state = 50653 * seed % (2**61-1)
        self.next_state = 50653 * self.current_state % (2**61-1)

    def getNextState(self):
        self.current_state = self.next_state
        self.next_state = 50653 * self.current_state % (2**61-1)

        return self.current_state

    def getDenom(self):
        return (2**61-1)
```

RANDU as "randu", an LCG used for an extensive period of computing history but known to be controversial.

[7] S. Park, K. Miller, and P. Stockmeyer "Technical Correspondence" *Communications of the ACM October 1988 Volume 31 Number 10*. https://www.firstpr.com.au/dsp/rand31/p1192-park.pdf

$randu: X_i = 65539 \cdot X_{i-1} \, mod \, 2^{31}$

```
class randu():

    def __init__(self, seed=33):
        self.seed = seed
        self.current_state = 65539 * seed % 2**31
        self.next_state = 65539 * self.current_state % 2**31

    def getNextState(self):
        self.current_state = self.next_state
        self.next_state = 65539 * self.current_state % 2**31

        return self.current_state

    def getDenom(self):
        return 2**31
```

simpleLCG, a simpler, arbitrary LCG offering a unique example to examine the tradeoff of simplicity and performance.

$simpleLCG: X_i = 17 \cdot (X_{i-1} + 2) \, mod \, 94$

```
class simpleLCG():
    def __init__(self, seed=33):
        self.seed = seed
        self.current_state = 17 * (seed+2) % (94)
        self.next_state = 17 * (self.current_state+2) % (94)

    def getNextState(self):
        self.current_state = self.next_state
        self.next_state = 17 * (self.current_state+2) % (94)

        return self.current_state

    def getDenom(self):
        return (94)
```

Other Generators (Mid-Square Method, Python's Mersenne Twister): **Implementation**

Accompanying the LCGs implemented in this project, two other alternative pseudorandom number generators were tested: the Mid-square method and the Python Random package's implementation of Mersenne Twister. Mersenne Twister works by initializing an initial "state" value (one of 624), then applies a "twist" function which, by using an "XOR" function, generates both a new pseudorandom number and a new state[8]. It works over a period of 2^{19937} - 1. The mid-

[8] https://research.nccgroup.com/2021/10/18/cracking-random-number-generators-using-machine-learning-part-2-mersenne-twister/#1.

square method is a technique developed by John von Neumann[9] where the middle digits of a number are taken as the number of the sequence, then squared, whereupon the middle digits are again taken as part of the sequence. For this implementation, the third to sixth digit (inclusive) are taken as the next part of the sequence and the seed number must be a four-digit number.

getMersenne is a simple procedure to use Python's Random package to generate uniform random numbers (the number of uniforms = *size*) via the Mersenne Twister algorithm.

```
import random
def getMersenne(size=100000):
    mersenneUniforms = []

    for i in range(0, size):
        mersenneUniforms.append(random.random())

    return mersenneUniforms
```

midSquareMethod is an implementation of the mid-square method as described (using the third to sixth digit, inclusive, of the squared seed/next number in the sequence).

```
class midSquareMethod():
    def __init__(self, seed=5473): # seed must be 4 digit number
        self.seed = seed
        self.current_state = int(str(seed**2).zfill(8)[2:6])
        self.next_state = int(str(self.current_state**2).zfill(8)[2:6])

    def getNextState(self):
        self.current_state = self.next_state
        self.next_state = int(str(self.current_state**2).zfill(8)[2:6])

        return self.current_state

    def getDenom(self):
        return 10000
```

Lastly, a function getResults was written to extract sequences of numbers, and corresponding uniforms, of the various LCGs and the midSquareMethod.

```
def getResults(generator, size=100000):
    nums = []
    uniforms = []

    for i in range(0, size):
        nums.append(generator.getNextState())
        uniforms.append(nums[i]/generator.getDenom())

    return nums, uniforms
```

[9] J. von Neumann, "Various Techniques Used in Connection with Random Digits," *Monte Carlo Method (*A. S. Householder, G. E. Forsythe, and H. H. Germond, eds.), National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office, 1951, pp. 36–38.

METHODS

This project used two classes of **visualizations** to assess uniformity:

- 2D histograms depicting the distribution of generated uniforms for each generator across 50 equally spaced bins (0.0 to .02, .02 to .04… .96 to .98, .98 to 1.0). If the generated numbers were truly uniform (and likely independent), one would expect a roughly flat rectangle of evenly distributed numbers. This visualization used the Python package matplotlib's hist() function to plot the histograms.
- 3D visualizations depicting the distribution of triple subsequential tuple pairs of uniforms for each generator (R_k, R_{k+1}, R_{k+2}). If the generated numbers were truly independent, one would expect a fully (randomly) filled 3D cube, rather than any specific geometries (planes, points, patterns, etc) being formed. This visualization used the Python package matplotlib's scatter3D() function to plot the histograms.

The implementation for the visualizations can be found below:

```
import matplotlib.pyplot as plt
def graphingHistogram (uniforms):
    plt.hist(uniforms,50)
    plt.show()

def graphing3D (uniforms):
    tuplesForGraphing = []
    for i in range(2, len(uniforms)):
        tuplesForGraphing.append((uniforms[i-2],uniforms[i-1],uniforms[i]))

    ax = plt.axes(projection='3d')
    x, y, z = zip(*tuplesForGraphing)
    ax.scatter3D(x, y, z, c=z, cmap='winter');

    ax.view_init(-140, 60)

    return plt
```

Accompanying these visualizations, this project used three **statistical tests for assessing the uniformity and independence** of the selected generators:

- A Chi-Square Goodness of Fit test with five groups. Chi-Square tests examine how the expected distribution of an independent, uniform group of random numbers would distribute (in this case one fifth of the population in each group) versus how they truly distribute. A generator yielding random numbers statistically different from the expected split distribution suggests a generator does not produce uniform, independent numbers. The formula for a Chi-Squared test is found below (where O is actual values for i groups and E expected values)

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

- An updown test. An updown test examines the number of run-ups and run-downs in a sample compared to an expected number of runs for a truly independent, uniform random sample. A "run-up" or "run-down" is defined as a sequence of numbers

proceeding in a that specified order, for instance .2, .3, .2, .1, .9, would represent three runs, an "up" followed by a "down" followed by a final "up" (+--+), two ups, one down, three runs total. The expected number of runs and the expected variance of runs can be compared to the actual number of runs to generate a Z-score (which can be used to determine statistical significance from a null hypothesis for a given *alpha*). These expected runs and deviation and associated Z-score is found below (where *n* is the sample size and *A* is the expected number of runs and *a* is the desired alpha of significance):

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

$$Z = \frac{A - E[A]}{\sqrt{Var(A)}}, reject\ null\ hypothesis\ if\ |Z| > z_{a/2}$$

- A correlations test. A correlation test examines the correlation between numbers in a sequence. There can be different "lags" of testing; this experiment uses a lag-1 correlation, where the previous number in the sequence is compared to the next. If all numbers are truly independent, then the correlation should be zero or very close to zero. The expected correlation (0) and the expected variance can be compared against the actual correlation to generate a Z-score (which can be used to determine statistical significance from a null hypothesis for a given *alpha*). An estimator to calculate the sample's sequential correlation ($\hat{p}$) and the expected correlation and deviance and associated Z-Score is found below (where R_k represents a uniform number in sequence and *n* is the sample size and *a* is the desired alpha of significance):

$$\hat{p} \equiv \left(\frac{12}{n-1}\sum_{k=1}^{n-1} R_k R_{k-1}\right) - 3$$

$$\hat{p} \approx Nor\left(0, \frac{13n-19}{(n-1)^2}\right)$$

$$Z = \frac{\hat{p}}{\sqrt{Var(\hat{p})}}, reject\ null\ hypothesis\ if\ |Z| > z_{a/2}$$

The implementation for the tests can be found on the following page:

```
def goodnessOfFit(uniforms, k=5): # Chi-squared test (5 groups)

    grps = []

    for z in range(0, k):
        sums = sum(1 for i in uniforms if i>(1/k)*z and i<=(1/k)*(z+1))
        grps.append(sums)

    expectedDistribution = len(uniforms)/k
    chiSquaredScore = 0

    for group in grps:
        chiSquaredScore += ((group - expectedDistribution)**2/expectedDistribution)


    return chiSquaredScore, grps

def updownTest(uniforms):
    upFlag = 0
    runs = 0

    for i in range(1, len(uniforms)):
        if(uniforms[i]>uniforms[i-1] and upFlag==0):
            runs+=1
            upFlag = 1
        if(uniforms[i]<uniforms[i-1] and upFlag==1):
            runs+=1
            upFlag=0

    expectedRuns = (2*len(uniforms)-1) / 3
    expectedDeviation = (16*len(uniforms)-29)/90

    z_score = (runs-expectedRuns)/(expectedDeviation**.5)

    return z_score, runs, expectedRuns

#  lag-1 correlation
def correlationTests(uniforms):
    p_hat = 0
    n = len(uniforms)

    for i in range(0, n-1):
        p_hat += uniforms[i]*uniforms[i+1]

    p_hat = (p_hat*(12/(n-1))) - 3
    var_p_hat = (13*n - 19)/((n-1)**2)

    z_score = p_hat/(var_p_hat**.5)

    return z_score, p_hat
```

This project also explored the **time differences in generation** for the various generators. Time to generate can be seen as a proxy for processing productivity. Computing has made great strides since the initial creation of these generators (particularly LCGs, mid-square method), but small differences in trials of even millions of numbers generated could imply significant efficiency differences over continuous, accumulating use of a generator. An implementation of the timed

run experiment is found below (note the MersenneTwister uses Python's random's implementation):

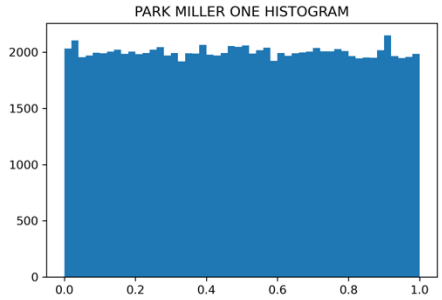
```
import time
def timedRun(uniform, size=100000, mersenne=False):
    if mersenne:
        start = time.time()
        getMersenne(size)
        end = time.time()
    else:
        start = time.time()
        getResults(uniform,size=size)
        end = time.time()

    return end-start
```

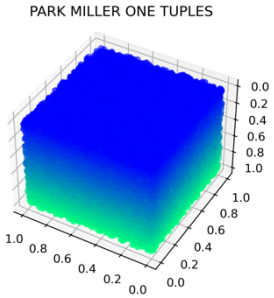
For all generators, all tests and visualization used 100,000 random uniforms (the same 100,000 100,000 population for each respective generator), except for the timedRun experiment which generated 1,000,000 uniforms per generator. All tests (including the timed test) were run on a MacBook Air (M2, 2022) configured with 16 GB of RAM.

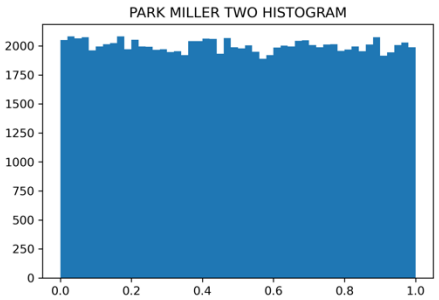
RESULTS

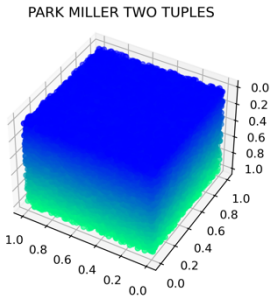
Visualizations

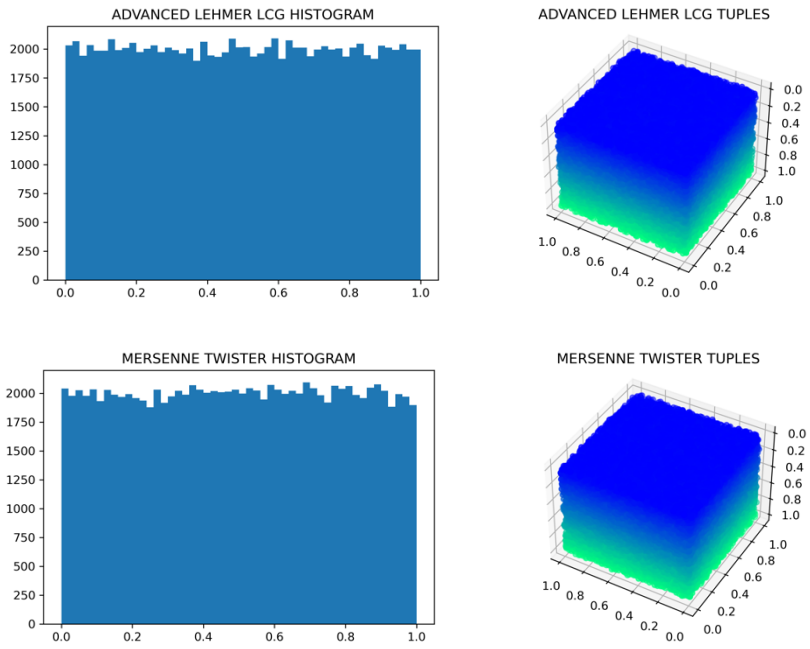
parkMillerOne & parkMillerTwo & advancedLehmerLCG & MersenneTwister





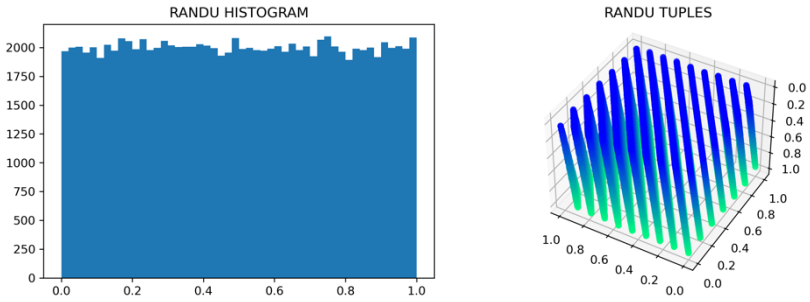






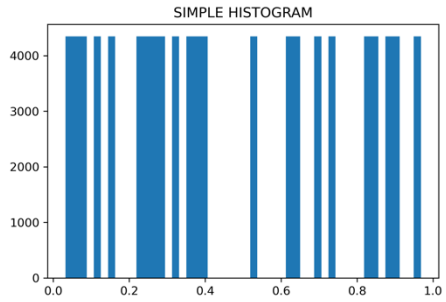
All 2D histograms for parkMillerOne, parkMillerTwo, advancedLehmerLCG, and Mersenne Twister appear roughly uniform (i.e. evenly distributed) and each of their respective 3D tuple graphs are fully filled cubes with no particular geometry, implying that each uniform is independent.

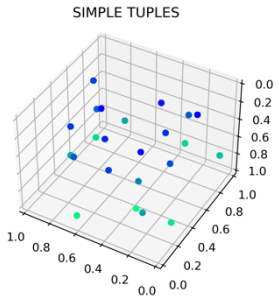
randu



The 2D histogram for randu appears roughly uniform, however 3D tuple graph clearly shows 15 distinct 2D planes, implying that the distribution of uniforms is not independent.

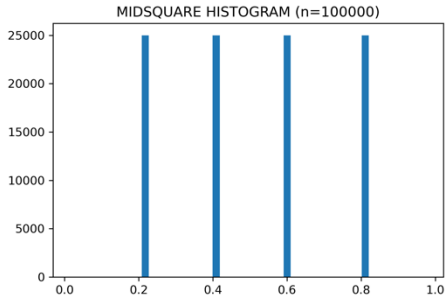
simpleLCG

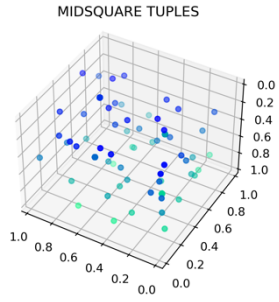




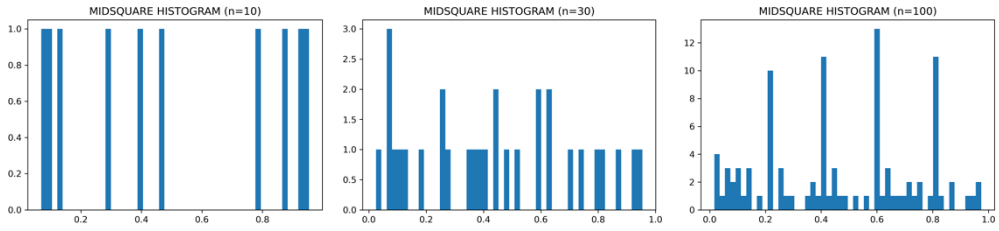
The 2D histogram for simpleLCG is clearly not uniform (there are ranges for values that are never selected) but is instead very consistent (almost akin to a specialized, repeating die). This is sensible given its low period. Likewise, its 3D tuple graph is neither fully filled nor contains planes but are distinct points.

midSquare





The 2D histogram for midSquare is clearly not uniform, in fact there appears to be only 4 distinct bins captured (any other results are too small to see). The 3D tuple graph however has more distinct points (still not a completed cube. The 2D histograms plotted below for midSquare for n=10, n=30, n=100, demonstrate how the method begins to degenerate as more uniforms are measured for seed = 5473. Like simpleLCG, midSquare produces a distribution more akin to a specialized, wonky die than an independent, uniform result, with the added twist of degeneration.



The midSquare method degenerating to four distinct results as n grows.

Tests of independence and efficiency

Chi-Square Goodness of Fit Test Results

Generators	χ^2 scores (first seed)	χ^2 scores (second seed)	Pass 95% confidence interval (k=5, degrees of freedom=4)
parkMillerOne	**2.00**	**1.29**	**Fail to reject null (both tests)**
parkMillerTwo	**7.65**	**2.02**	**Fail to reject null (both tests)**
advancedLehmerLCG	**2.57**	**2.03**	**Fail to reject null (both tests)**
randu	**2.33**	**1.81**	**Fail to reject null (both tests)**
simpleLCG	**23817.6**	**23817.1**	**Reject null (both tests)**
MersenneTwister (Python Random package's random)	**5.62**	**1.48**	**Fail to reject null (both tests)**
Mid-Square Method	**24955.02**	**24972.51**	**Reject null (both tests)**

Given the visualizations, there are no surprises in the independence test outcomes, however it is notable that the χ^2 scores for *the first seed for both the parkMillerTwo generator and the MersenneTwister generator yielded higher χ^2 scores than the rest of the generators and sample seeds that passed the test (those values were 7.65, 5.62 respectively). This encourages performing additional tests with additional sets of seeds. Also interestingly, the Mid-Square method yielded a higher (worse) χ^2 score than the "simpleLCG", the LCG with the known small and non-full period. randu also manages to appear independent in this test, highlighting the importance of the 3D visualization.*

Up, Down Run Test Results

Generators	*Z* score from runs test (first seed)	*Z* score from runs test (second seed)	Pass 95% confidence interval
parkMillerOne	**0.37**	**1.05**	**Fail to reject null (both tests)**
parkMillerTwo	**-0.80**	**-1.14**	**Fail to reject null (both tests)**
advancedLehmerLCG	**-.02**	**-1.17**	**Fail to reject null (both tests)**
randu	**0.8**	**-1.81**	**Fail to reject null (both tests)**
simpleLCG	**21.7**	**21.7**	**Reject null (both tests)**
MersenneTwister (Python Random package's random)	**0.13**	**1.27**	**Fail to reject null (both tests)**
Mid-Square Method	**-124.94**	**-124.95**	**Reject null (both tests)**

Again, there are no surprises in the independence test outcomes. randu again suggests to be independent (though at a Z-score of -1.81 in the second seed, it nearly fails, and with a wider confidence interval, it would have failed), stressing the importance of the 3D visualization.

Correlations Test Results

Generators	*Z* score from correlations test (first seed)	*Z* score from runs test (second seed)	Pass 95% confidence interval
parkMillerOne	**-0.23**	**-1.16**	**Fail to reject null (both tests)**
parkMillerTwo	**-0.71**	**0.91**	**Fail to reject null (both tests)**
advancedLehmerLCG	**-0.62**	**-0.82**	**Fail to reject null (both tests)**
randu	**-0.06**	**-0.16**	**Fail to reject null (both tests)**
simpleLCG	**-35.2**	**-35.2**	**Reject null (both tests)**
MersenneTwister (Python Random package's random)	**0.62**	**0.17**	**Fail to reject null (both tests)**
Mid-Square Method	**-10.60**	**10.61**	**Reject null (both tests)**

Same results as the runs test.

Efficiency Test Results

Generators	Time to generate 1 million uniforms (first seed)	Time to generate 1 million uniforms (second seed)	Avg time to generate 1 million uniforms	% Slower than simpleLCG
parkMillerOne	**0.434**	**0.435**	**0.435**	**22.9%**
parkMillerTwo	**0.427**	**0.445**	**0.436**	**23.2%**
advancedLehmerLCG	**0.563**	**0.580**	**0.572**	**61.6%**
randu	**0.429**	**0.435**	**0.432**	**22.0%**
simpleLCG	**0.354**	**0.353**	**0.354**	**0%**
MersenneTwister (Python Random package's random)* *Not included in comparison (different implementation)	**0.0995**	**0.0992**	**0.0994**	**N/A**
Mid-Square Method	**0.772**	**0.779**	**0.778**	**119.8%**

MersenneTwister is the most efficient generator, however this may be primarily due to implementation differences. Of the generators implemented through the same methodology (all others), simpleLCG is the fastest, followed by randu, parkMillerOne, parkMillerTwo which all are grouped closely, then advancedLechmerLCG at 61% slower, ending with Mid-Square method which is the slowest by less than half the speed.

Summarizing these results:

- parkMillerOne, parkMillerTwo, advancedLehmerLCG are all relatively indistinguishable in both visualization, tests of independence and of efficiency. All balance quality of uniform generation and efficiency.

- randu clearly fails independence when looking at its geometry but passes independence tests (barely in some cases). This may explain how it "got away" with being used in computing for so long, despite having obvious problems.

- MersenneTwister is the best performing generator: It is the most efficient (fastest), passes all tests for uniformity and independence, and has an immensely along period, 2^{19937}-1. All of these are great arguments for it being the default pseudorandom number generator for a wide number of applications.

- The most efficient (fastest) self-implemented generator is the "simpleLCG." SimpleLCG fails to generate i.i.d (independent, individually distributed) uniform variables, but instead yields a very specific, repeatable distribution. Though it is unhelpful as a generic uniform pseudorandom number generator, its efficiency does demonstrate the opportunity to fit a generator to purpose (i.e., generate exactly for the function one needs, and nothing beyond, if computing power is a constraint).

    - Note: If MersenneTwister was implemented in similar fashion in Python and not in C[10], a faster language, it may have been slower than the other generators.

- Mid-Square Method is the worst performing generator, yielding degenerating results at inefficient performance.

CONCLUSION

This project explored an array of pseudorandom number generators through the lens of several tools—implementations, visualizations, tests of independence and efficiency). Through those efforts, a broad arsenal of approaches was created, providing a toolkit to examine other future and past pseudorandom number generators. Future extensions of this work could explore other classes of generators beyond pseudorandom number generator, like "true" random number generators[11], generating numbers using hardware devices informed by environmental randomness. Testing the quality of the generation of these devices is critical as the origin of their outputs are more obfuscated than pseudorandom number generators which have defined mathematical underpinnings.

---

[10] https://docs.python.org/3/library/random.html#:~:text=Python%20uses%20the%20Mersenne%20Twister,random%20number%20generators%20in%20existence.

[11] S. Park, B. G. Choi, T. Kang, K. Park, Y. Kwon, J. Kim, "Efficient hardware implementation and analysis of true random-number generator based on beta source" *ETRI Journal* Volume 42, Issue4 Special Issue on SoC and AI processors, August 2020 pp. 518-526.

This project also confirmed the sensibility behind the status quo of generators today. What stands on top of the heap are quality generators (the various multiplicative LCGs with long periods, MersenneTwister), and what has fallen by the wayside (randu, Mid-Square method) are riddled with issues. If there are any fringe applications for degenerating generators, closer examination of the results of the Mid-Square Method could be warranted to observe how results degenerate for various seeds. With computing power and the complexity of tools available continuing to increase, there will surely be more advances in the science (and art) of pseudorandom number generators and a new standard will emerge, just as the next Mersenne number waits to be discovered (the last being $2^{82589933}$-1 in 2018)[4].