Nathan Glotzbach

PHYS 580

Project: PRNG Evaluation

I. Introduction

The purpose of this project is to write a program that performs selected tests which evaluate the quality of pseudo-random number generators. Three tests were implemented, and six generation algorithms were tested. In addition to these tests, the speeds of the generation algorithms were measured. This report will discuss the tests and generation algorithms, the results obtained, and suggestions for future improvement.

II. Tests

Three tests were selected for implementation: a basic uniformity test, a triplet correlation test, and a squeeze test. The uniformity test and triplet correlation test come from the project starter materials, and the squeeze test was chosen as a representative test from the Diehard set of PRNG quality tests.

The uniformity test checks whether the randomly generated values are uniform. Given a set of N random numbers on the interval [0,1), the number of random numbers smaller than a given number p is defined as r(p). If M such sets are generated, the variance between sets of each r(p) is given by Np(1-p).

The triplet correlation test checks whether successive randomly generated values are correlated. 3L3 random integers are generated and sets of three successive integers from this list are taken to be a point in 3D space within a cube of side length L. After all points from the set are chosen, the number of unchosen points should be L3/e. If another set of L3 points are chosen, the number of unchosen points is further reduced by a factor of e. In general, then, after k iterations, L3/e-k unchosen points remain.

The squeeze test is an alternative test of uniformity. A variable is initialized to 232, then it is multiplied by a random number and the result is rounded up to the nearest integer. The result is multiplied by another random number and rounded up, then this process is repeated until the value of the variable is 1. The number *n* of random numbers needed to reduce the variable to 1 through this process is recorded. Once 1 is reached, the variable is reset to 232 and the algorithm is repeated. The distribution of values of *n* should follow a normal distribution (see section V for more detail). The distribution is quantitatively measured by performing a chi-squared test, which returns a low p-value if the distribution is normal.

III. Generators

Six generators were chosen for use in this project. The first is mt19937ar, an implementation of the Mersenne Twister algorithm. This is the default PRNG in MATLAB, Python, and many other languages; it is of particular importance to us due to its prevalence in PHYS 580 lab work and homework. The Mersenne Twister algorithm is complex and involves several bit shift, bit mask, and modulus operations.

The second is IBM’s RANDU generator, which was widely used in the 1970s and was quickly identified as a low-quality generator. It is an instance of a linear congruential generator, where

For the purposes of this project, the integer result is divided by 231 so the final result is in the range [0,1).

The third is the Blum-Blum-Shub (BBS) algorithm, where

where p and q are large prime numbers. For this project, p and q were chosen to be 20549 and 35671, respectively. For the purposes of this project, the integer result of this process is divided by pq.

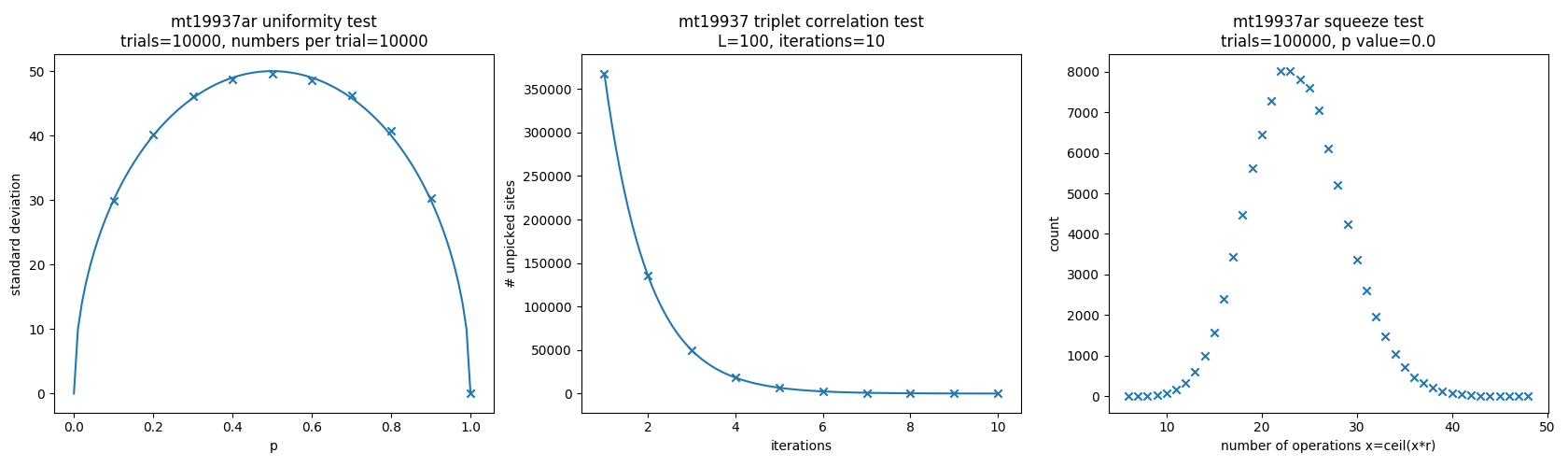
The fourth is the middle-squares algorithm, which is simple to calculate but can easily be shown to be flawed. To generate N digit random numbers, a seed value (padded on the left with zeros if necessary to reach N digits) is squared, and the middle N digits are taken to be the random number to output (and the seed for the next iteration of the generator). For the purposes of this project, the integer result of this process is divided by 10N-1.

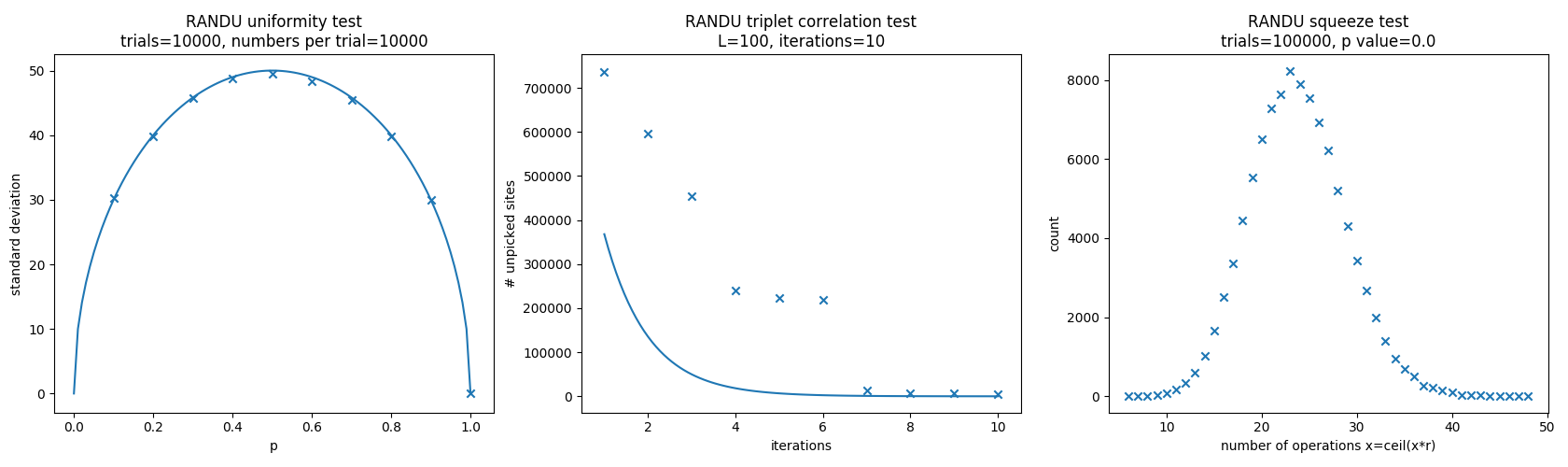
The fifth is file reads from /dev/random, a file present on many UNIX-based systems. This file is filled by the operating system with random binary data generated from the system’s physical state; the generation method varies from system to system. 2000 bytes were read at a time and split into 1000 2-byte sections, which were then interpreted as integers. For the purpose of this project, the integer results of this process were divided by 216-1.

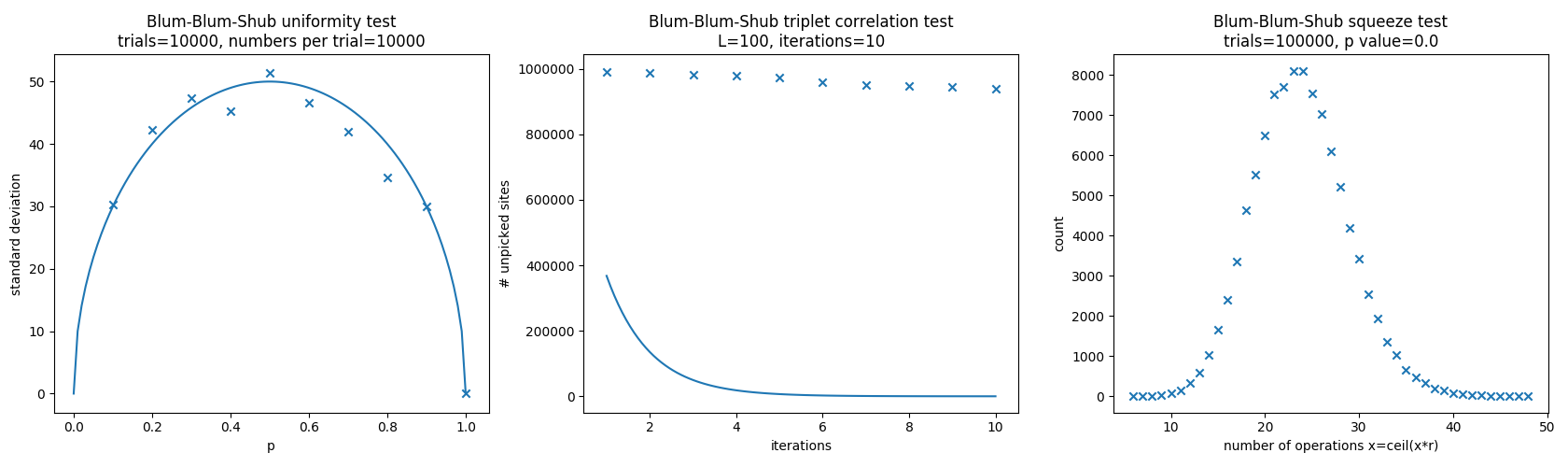
The final is file reads from a text file containing digits of pi. A similar file read strategy was used, but the data read were strings consisting of 5 integers. Each string was directly converted to an integer between 0 and 99999. For the purpose of this project, the integer results were divided by 99999.

IV. Results

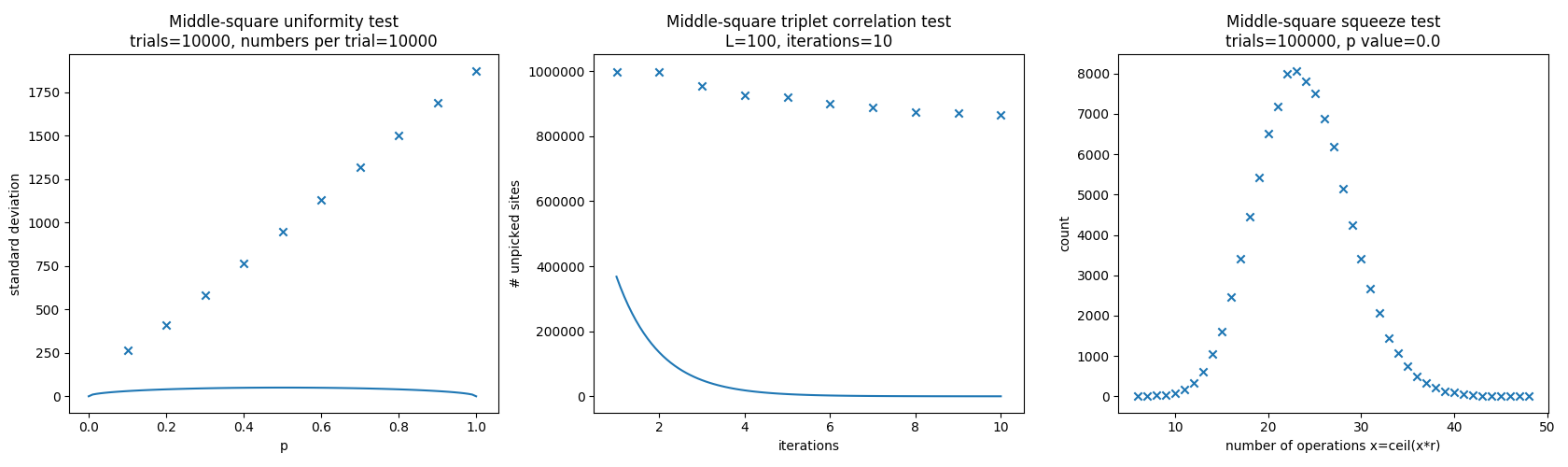
Plots of each generator’s test results are given below with a brief discussion of notable results.



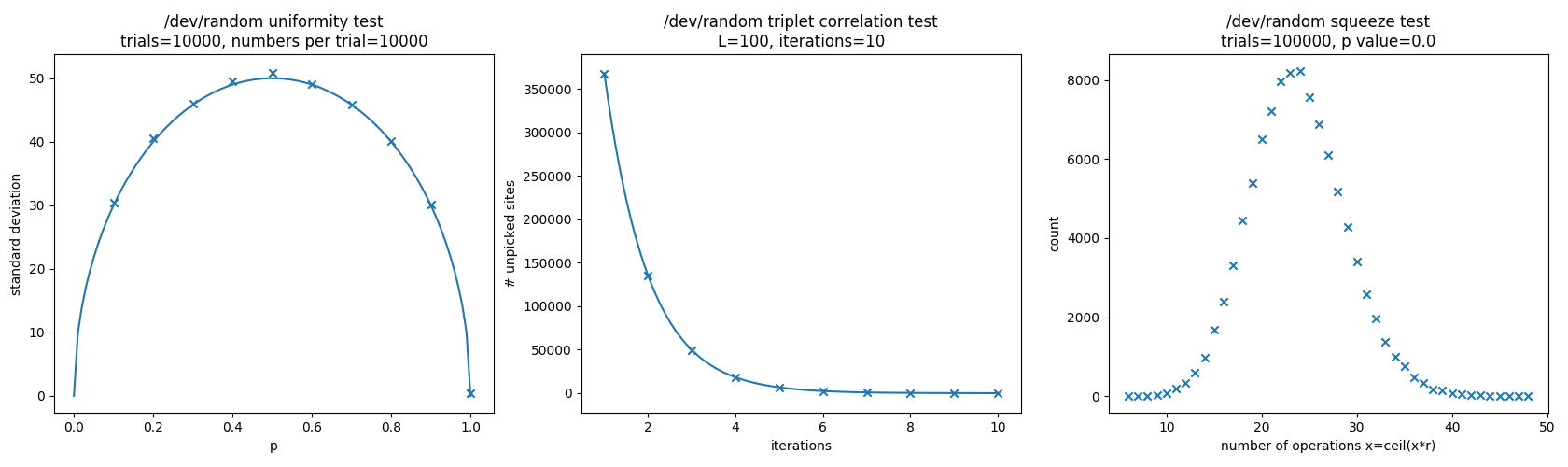


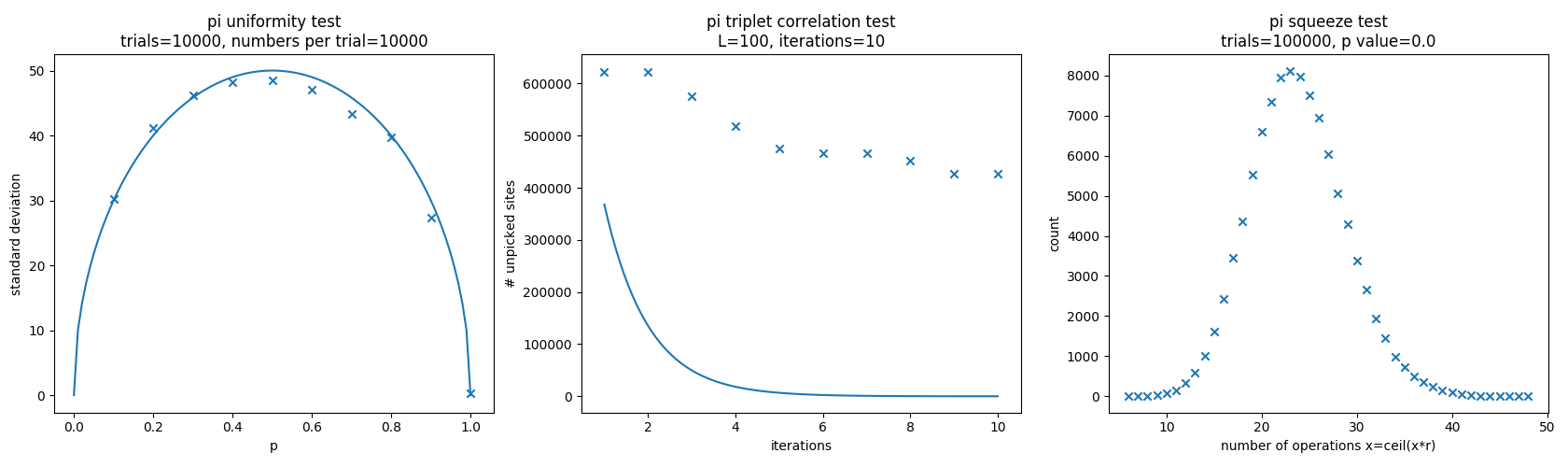


The BBS results for triplet correlation is extremely poor. This is understandable based on the very simple algorithm, where the result is essentially based only on the square of the seed.

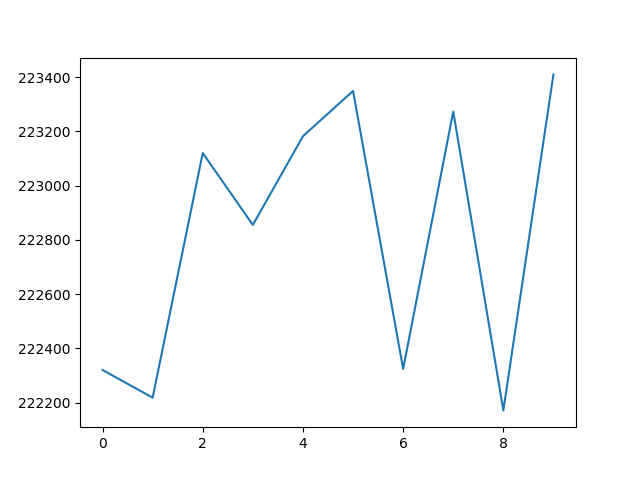


Again, the result of the middle-squares method is based only on the square of the seed, so the results are highly correlated. In addition, the uniformity suffers for long sets of numbers as the generator is prone to falling into loops. The loops’ periods become shorter as the values in the loop become smaller, which may explain the linear trend in the uniformity test data. The squeeze test results are unaffected because the algorithm generates at most 48 values before randomly re-seeding the generator.

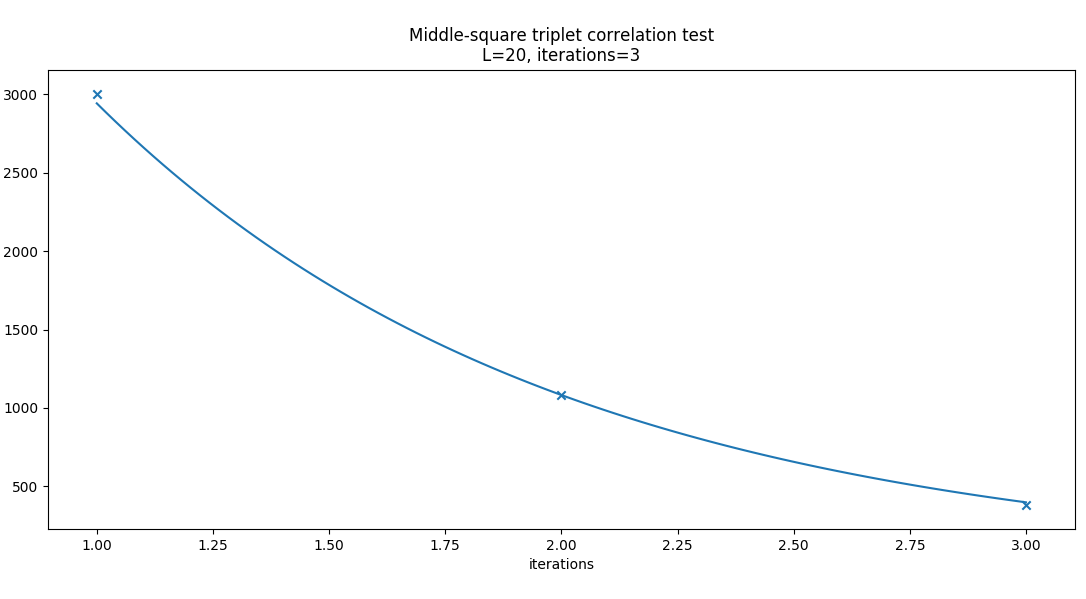




The correlation test results for pi are poor, but presumably this should not be the case since pi is an irrational number with no correlations present. It is likely that the correlation issues are at least partially caused by generating too many random numbers from the text file, which has a finite length and therefore must eventually repeat values exactly if a sufficiently high number of values are requested. To test the assumption that pi is uncorrelated and the flaw lies in the finite length of pi used, two further tests were performed on pi. The first reads each digit of the file and counts the frequency of each, and the second repeats the triplet correlation test but for a smaller number of random numbers generated to ensure that the values can be read without repeating.



The frequencies of each digit of pi are given above, with the digit on the x-axis and the frequency on the y-axis. Note that the y-axis does not start at zero; these values are essentially equal. On a digit-by-digit basis, then, pi is apparently uniform.



Repeating the triplet correlation test on digits of pi where L=20 and iterations=3 (instead of L=100 and iterations=10) results in a much closer match to theoretical values. Thus, the conclusion is that the initial test was flawed by the file size limit.

Finally, runtimes were calculated for programs that

V. Suggestions for Future Improvement

Unfortunately, other obligations interfered with an exhaustive investigation of the topics presented in this report. This work may be built upon and improved in the future primarily by finding a theoretical distribution for the squeeze test results. It is known that the results are normally distributed, but expressions for the mean and standard deviation remain unknown, as a search of the literature did not produce values. An attempt was therefore made to derive a theoretical solution.

Intuition suggests that the mean should be 32 random numbers, as the expectation value of each random number is 0.5 and the initial number is 232. However, the results of each test indicate that there is a flaw in this reasoning, as the mean in each case is clearly not 32 from inspection of the plots.

Based on the data collected from tests, the mean was numerically found to be approximately 23, while the standard deviation was approximately 4.5. These values are apparently uniform across the generators tested, so this is likely a universal behavior.

Additionally, after performing the tests it became clear that the squeeze test was not a useful differentiator of the generators chosen. It did provide some insight into the ways in which generated numbers were non-uniform in the cases when the squeeze test was successful while the uniformity test failed. However, in future tests it may be more instructive to use some alternate test instead of or in addition to the squeeze test.

VI. Software

This project was completed using Python 3.6.1. Each script, given in full below, is self-contained. prng\_evaluation.py must be modified to use a given generator, while prng\_timing.py and count\_pi.py are finalized.

#

# prng\_evaluation.py

# PRNG Evaluation - PHYS 580

# Nathan Glotzbach

#

# Generators - normalized to [0,1)

# mt19937

# RANDU

# Blum Blum Shub

# middle-square

# /dev/random

# digits of pi

# Tests

# uniformity

# triplet correlation

# squeeze

import matplotlib.pyplot as plt

import random

import numpy as np

from scipy.stats import chisquare

def mt(num=10000):

i = 0

while i < num:

i += 1

yield random.random()

def randu(num=10000):

i = 0

r = random.random() \* 10\*\*10

while i < num:

i += 1

r = np.mod(65539\*r, 2\*\*31)

yield r / (2\*\*31)

def bbs(num=10000):

i = 0

p = 20549

q = 35671

r = random.random() \* 10\*\*10

while i < num:

i += 1

r = np.mod(r\*r, p\*q)

yield r / (p\*q)

def mid\_sq(num=10000):

i = 0

n = 10

r = random.random() \* 10\*\*n

while i < num:

i += 1

sq = int(r\*r)

sqstr = str(sq)

while len(sqstr) < n:

sqstr = '0'+sqstr

cut = len(sqstr) - n

if cut == 0:

r = int(sqstr)

else:

r = int(sqstr[int(np.floor(cut/2)):int(np.floor(-cut/2))])

yield r / (10\*\*n)

def dev\_rand(num=10000):

i = 0

num\_bytes = 2

num\_read = 1000

read = np.zeros(num\_read)

while i \* num\_read < num:

i += 1

with open('/dev/random','rb') as f:

for j in range(num\_read):

f.seek(num\_bytes \* j)

byte = f.read(num\_bytes)

read[j] = int.from\_bytes(byte, byteorder='big')

for k in range(num\_read):

yield read[k] / (2\*\*(8\*num\_bytes))

def pi\_rand\_multidigit(num=10000):

i = 0

num\_digits = 5

num\_read = 1000

read = np.zeros(num\_read)

ind = int(random.random() \* 2 \* 10\*\*6) + 2

while i \* num\_read < num:

i += 1

with open('/Users/npglotzbach/Desktop/pi-billion.txt') as f:

for j in range(num\_read):

f.seek(ind)

r = f.read(num\_digits)

read[j] = int(r)

ind += num\_digits

if ind >= 2 \* 10\*\*6:

ind = 2

for k in range(num\_read):

yield int(read[k]) / (10\*\*num\_digits)

plt.figure(1,figsize=(18,5))

# uniformity test

trials = 10000

rs = 10000

count = np.zeros((10,trials))

stdev = np.zeros(10)

for trial in range(trials):

for r in mid\_sq(rs):

c = int(np.ceil(10 \* r))

for i in range(c):

count[-(i+1)][trial] += 1

if np.mod(trial, 1000) == 0:

print('.', end='')

for i,ct in enumerate(count):

stdev[i] = np.std(ct)

plt.subplot(131)

X = np.linspace(0,1,100)

Y = np.sqrt(-rs \* X \* (X - 1))

plt.plot(X,Y)

x = np.linspace(0.1,1,10)

plt.scatter(x,stdev,marker='x')

plt.title('Middle-square uniformity test\ntrials=' + str(trials) + ', numbers per trial=' + str(rs))

plt.xlabel('p')

plt.ylabel('standard deviation')

print('Completed uniformity test.')

# triplet correlation test

L = 20

iterations = 3

N = L\*\*3

num\_gen = 3 \* N

gen = -np.ones(num\_gen \* iterations)

unpicked = np.zeros(iterations)

for iteration in range(iterations):

for i,r in enumerate(pi\_rand\_multidigit(num\_gen)):

gen[i + num\_gen \* iteration] = np.ceil(r \* L)

print('.', end='')

sites\_picked = np.reshape(gen[gen >= 0], (N \* (iteration + 1), 3))

unique\_sites = np.unique(sites\_picked, axis=0)

unpicked[iteration] = N - len(unique\_sites)

#plt.subplot(132)

X = np.linspace(1,iterations,100)

Y = N \* np.exp(-X)

plt.plot(X,Y)

x = np.linspace(1,iterations,iterations)

plt.scatter(x,unpicked,marker='x')

plt.xlabel('iterations')

plt.ylabel('# unpicked sites')

plt.title('Middle-square triplet correlation test\nL=' + str(L) + ', iterations=' + str(iterations))

print('Completed autocorrelation test.')

# squeeze test

iterations = 100000

counts = np.zeros(43)

for i in range(iterations):

val = 2\*\*32

count = 0

for r in mt(48):

val = (val \* r)

count += 1

if val <= 1:

break

if count >= 48:

count = 48

break

if count <= 6:

counts[0] += 1

else:

counts[count - 6] += 1

if np.mod(i, 10000) == 0:

print('.', end='')

cs = chisquare(counts)

#plt.subplot(133)

X = np.linspace(6,48,43)

plt.scatter(X,counts,marker='x')

plt.xlabel('number of operations x=ceil(x\*r)')

plt.ylabel('count')

plt.title('Middle-square squeeze test\ntrials=' + str(iterations) + ', p value=' + str(cs[1]))

print('Completed squeeze test.')

plt.tight\_layout()

plt.show()

#

# prng\_timing.py

# PRNG Timing - PHYS 580

# Nathan Glotzbach

#

# Generators - normalized to [0,1)

# mt19937

# RANDU

# Blum Blum Shub

# middle-square

# /dev/random

# digits of pi

import random

import numpy as np

from time import clock

def mt(num=10000):

i = 0

while i < num:

i += 1

yield random.random()

def randu(num=10000):

i = 0

r = random.random() \* 10\*\*10

while i < num:

i += 1

r = np.mod(65539\*r, 2\*\*31)

yield r / (2\*\*31)

def bbs(num=10000):

i = 0

p = 20549

q = 35671

r = random.random() \* 10\*\*10

while i < num:

i += 1

r = np.mod(r\*r, p\*q)

yield r / (p\*q)

def mid\_sq(num=10000):

i = 0

n = 10

r = random.random() \* 10\*\*n

while i < num:

i += 1

sq = int(r\*r)

sqstr = str(sq)

while len(sqstr) < n:

sqstr = '0'+sqstr

cut = len(sqstr) - n

if cut == 0:

r = int(sqstr)

else:

r = int(sqstr[int(np.floor(cut/2)):int(np.floor(-cut/2))])

yield r / (10\*\*n)

def dev\_rand(num=10000):

i = 0

num\_bytes = 2

num\_read = 1000

read = np.zeros(num\_read)

while i \* num\_read < num:

i += 1

with open('/dev/random','rb') as f:

for j in range(num\_read):

f.seek(num\_bytes \* j)

byte = f.read(num\_bytes)

read[j] = int.from\_bytes(byte, byteorder='big')

for k in range(num\_read):

yield read[k] / (2\*\*(8\*num\_bytes))

def pi\_rand\_multidigit(num=10000):

i = 0

num\_digits = 5

num\_read = 1000

read = np.zeros(num\_read)

ind = int(random.random() \* 2 \* 10\*\*6) + 2

while i \* num\_read < num:

i += 1

with open('/Users/npglotzbach/Desktop/pi-billion.txt') as f:

for j in range(num\_read):

f.seek(ind)

r = f.read(num\_digits)

read[j] = int(r)

ind += num\_digits

if ind >= 2 \* 10\*\*6:

ind = 2

for k in range(num\_read):

yield int(read[k]) / (10\*\*num\_digits)

print('time to generate 1000000 random numbers')

start = clock()

for r in mt(1000000):

continue

mt\_time = np.round(clock() - start, decimals=6)

print('mt19937ar: ' + str(mt\_time) + ' seconds')

start = clock()

for r in randu(1000000):

continue

randu\_time = np.round(clock() - start, decimals=6)

print('randu: ' + str(randu\_time) + ' seconds')

start = clock()

for r in bbs(1000000):

continue

bbs\_time = np.round(clock() - start, decimals=6)

print('blum-blum-shub: ' + str(bbs\_time) + ' seconds')

start = clock()

for r in mid\_sq(1000000):

continue

mid\_sq\_time = np.round(clock() - start, decimals=6)

print('middle-squares: ' + str(mid\_sq\_time) + ' seconds')

start = clock()

for r in dev\_rand(1000000):

continue

dev\_rand\_time = np.round(clock() - start, decimals=6)

print('/dev/random: ' + str(dev\_rand\_time) + ' seconds')

start = clock()

for r in pi\_rand\_multidigit(1000000):

continue

pi\_time = np.round(clock() - start, decimals=6)

print('digits of pi: ' + str(pi\_time) + ' seconds')

#

# count\_pi.py

# Nathan Glotzbach

#

import matplotlib.pyplot as plt

import numpy as np

def read\_pi():

with open('/Users/npglotzbach/Documents/Purdue/18F/PHYS 580/project/pi-billion.txt') as f:

f.seek(2)

num = f.read(1)

while num:

yield int(num)

num = f.read(1)

count = np.zeros(10)

for num in read\_pi():

count[num] += 1

plt.plot(count)

plt.show()