Random Number Generators

Written by:

- Steve Mrenna (University of Cincinnati)
- Philip Ilten (University of Cincinnati)

For any Monte Carlo (MC) computational technique, random numbers (or pseudo-random numbers) are critical, not only in the sampling characteristics of the generator but also the speed of the sampling. For example, simulating a Large Hadron Collider (LHC) collission can require upwards of 2.5×10^6 random number calls.

This worksheet is largely based on chapter 7 of Numerical Recipes, and associated primary papers. The following is a list of references for further reading.

- Hammersley, J.M. and Handscomb, D.C. 1964, Monte Carlo Methods (London: Methuen)
- Kalos, M.H. and Whitlock, P.A. 1986, Monte Carlo Methods (New York: Wiley)
- Bratley, P., Fox, B.L., and Schrage, E.L. 1983, A Guide to Simulation, 2nd ed. (New York: Springer)
- Lepage, G.P. 1978, A New Algorithm for Adaptive Multidimensional Integration, Journal of Computational Physics, vol. 27, pp. 192-203
- Lepage, G.P. 1980, VEGAS: An Adaptive Multidimensional Integration Program, Publication CLNS-80/447, Cornell University
- Numerical Recipes Software 2007, Complete VEGAS Code Listing, Numerical Recipes Webnote No. 9
- Press, W.H., and Farrar, G.R. 1990, Recursive Stratified Sampling for Multidimensional Monte Carlo Integration, Computers in Physics, vol. 4, pp. 190-195
- Numerical Recipes Software 2007, Complete Miser Code Listing, Numerical Recipes Webnote No. 10

Requirements

This notebook requires a few external dependencies which are imported here. First, it is useful to have interactive plotting via `matplotlib`. We need this in Exercise: choosing better parameters. To enable this, we need to install the `ipympl` module and restart the kernel. To make sure this only gets run once, we also create global `init` variable.

```
# Check if we are initialized.
try:
    init
except:
```

```
except:
    init = False

# Install only if we are not initialized.
if init == False:
    !pip install -q ipympl
    get_ipython().kernel.do_shutdown(restart=True)
```

Next, we configure `matplotlib` and allow widgets in Colab.

```
# Set the we are initialize.
init = True

# Allow for interactive plots.
%matplotlib widget
from matplotlib import pyplot as plt

# Allow for widgets in Colab.
try:
    from google.colab import output

    output.enable_custom_widget_manager()
except:
    pass
```

Finally, we import both `numpy` and `scipy` for matrix operations which are used in the XORshift RNGs section. We also need `sys` to determine the size of types when performing bit shifts.

```
import numpy as np
import scipy as sp
import sys, math, inspect
```

Introduction

This tutorial will give an introduction to Random Number Generators (or RNGs, for short). RNG algorithms are really pseudo-random (pRNG). This means they are deterministic. Randomness (Jaynes, Probability Theory: The Logic of Science) is a statement about knowledge, not about causes. pRNGs have causes, but you might not know them, or you might use them in a way so that they appear to be random, or random "enough".

What are the basic properties of RNs? On any interval, there is an equal probability to obtain any value. For convenience, we will take this interval to be $(0, 1)$. Note, we will generally exclude 0 from the interval when considering RNs on a computer, since many algorithms will fail if 0 is ever encountered.

At the very least, we expect our computer RNs to have the properties of abstract ones.

Mean:

Mean:

$$\langle x \rangle = \int_0^1 \mathrm{d}x\, x = \frac{1}{2}$$

Variance:

$$\langle x^2 \rangle - \langle x \rangle^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}$$

Since it is useful to check the mean and variance, let us write simple functions which calculate these values given an iterable object like a list or array.

```
# Note, these functions are available in `numpy` as `numpy.mean` and
# `numpy.var`.
def mean(xs):
    """
    Return the mean for a list of numbers given by `xs`.

    xs: list of numbers to find the mean for.
    """
    return sum(xs) / float(len(xs))


def variance(xs):
    """
    Return the variance for a list of numbers given by `xs`.

    xs: list of numbers to find the variance for.
    """
    return mean([x**2 for x in xs]) - mean(xs) ** 2
```

In practice, computers represent only a finite range of numbers. This has implications for RNGs. Any algorithm that is deterministic and shuffles through the numbers is bound to repeat itself. Further, if our algorithm uses one number as the seed for the next, this repetition will mean that the sequence of pRNs repeats itself. This hardly looks random. Much of the art in developing good RNGs is knowing how to characterize or calculate the length of these cycles. If the cycle is long enough, much longer than any sequence of RNs we ever use in practice, then there is hope that this sequence will have the desired properties of random numbers. Identifying an algorithm with a long cycle is a priority, and possibly led to the focus on modular arithmetic, to be discussed shortly.

Middle square method

The need for such algorithms only came about with the development of practical computers and Monte Carlo (MC) techniques. Historically, we are placing ourselves in the 1940-1950s. Von Neumann (who else?) used a method called the middle square. It can give us some insight to the problem we are

racing.

The middle square algorithm is short and strange:

1. input a number of length n
2. square the number and zero-pad if not of length $2n$
3. output the "middle" number of length n
4. use this number as the random variate and the input for step 1

For example:

$$14^2 = 196 \to 0196 \to 19; 19^2 = 0361 \to 36; 36^2 = 1296 \to 29; 29^2 = 0841 \to 84, \text{ etc.}$$

Von Neumann used larger n than this, but you get the point.

Before we practically explore the middle square method, let us first consider some technical realities behind RNGs. Perhaps the most critical aspect is that all RNGs require a definition of a state; for the next RN to be generated, the RNG must have saved information stored on how the previous RN was generated. In the context of Python, this means that we need to use a class to define an RNG.

```
# Here, we define a simple base class which can hold the state for an RNG.
class RNG:
    """
    This is a simple base class for random number generation.
    """

    def __init__(self, seed):
        """
        Constructor for this class, e.g. `RNG(seed)`.

        seed: seed used to initialize the generator.
        """
        self.seed = seed
        # In derived classes, further state can be initialized here.

    def __call__(self):
        """
        Returns a uniform RN between 0 and 1. This defines the `()`
        for this class. For example:
        ```
        rng = RNG(seed)
        rng()
        ```
        """
        # In derived classes, the actual algorithm should be implemented here.
        return 0.5
```

We are also interested in the periodicity for the RNG, so we can define a function, which given an RNG, generates RNs until the sequence repeats.

```
# Since it is useful to determine the period of an RNG, let us define a method
# that does just this. Since we are storing the RNs we need to be careful not
# to store a very long list.
def rng_sequence(rng, n=int(1e6)):
    """
    Given an RNG, return its sequence after its first repetition or until
    `n` RNs have been generated.

    n: maximum number of RNs to generate.
    """
    # Start with no RN and create the RN sequence.
    rn, rns = None, []

    # Loop while the next RN is not in the sequence.
    while rn not in rns:
        # Append the RN to the sequence.
        if rn != None:
            rns.append(rn)
        # Generate the next RN.
        rn = rng()
        # Return if `n` is reached.
        if len(rns) >= n:
            return rns

    # Return the sequence.
    return rns
```

Exercise: implement the middle square method

For n=2, try all seeds (inputs) and identify the longest sequence (before a repeat). For this longest sequence, calculate the mean and variance. Then, identify the worst seeds (0 period).

```
# First we define the middle square class.
class RNGMiddleSquare(RNG):
    # For our state, we need the previously generated integer and what
    # `n` we are using.
    def __init__(self, seed, n=2):
        super().__init__(seed)
        self.state = seed
        self.n = n

    # Here, we define the actual RNG.
    def __call__(self):
        # The `zfill` method pads the string with 0s, this gives us a string.
        self.state = str(self.state**2).zfill(2 * self.n)
        # Next, we extract the middle of the string as an integer.
        i = int(self.n / 2)
        self.state = int(self.state[i:-i])
        # We divide by 10^n - 1 so our RN is between 0 and 1.
```

```
        return self.state / float(10**self.n - 1)


# Now we can try it out for n = 2.
# Input the random number seed and define the sequence.
seed = 10

# Create the RNG.
rng = RNGMiddleSquare(seed, 2)

# We store the current RN in `rn` and the sequence of RNs in `rns`.
rn, rns = None, []

# Start throwing random numbers.
while rn not in rns:
    # Append RN to the sequence.
    if rn != None:
        rns.append(rn)
    # Generate the new random number using the middle square method.
    rn = rng()
    # Print the result.
    print(f"#{len(rns)}: {rn}")

# Print the result.
print(
    f"We began with {seed} and"
    f" have repeated ourselves after {len(rns)} steps"
    f" with {rn}."
)

# We can now calculate the mean and the variance of the sequence.
# Numpy allows us to easily calculate the mean and variance.
print("mean = ", mean(rns))
print("variance = ", variance(rns))

     #0: 0.10101010101010101
     #1: 0.10101010101010101
     We began with 10 and have repeated ourselves after 1 steps with 0.10101010101010101
     mean =  0.10101010101010101
     variance =  0.0


# We can also try it for n = 4.
# Input the random number seed and define the sequence.
seed = 1234

# Create the RNG.
rng = RNGMiddleSquare(seed, 4)

# We store the current RN in `rn` and the sequence of RNs in `rns`.
rn, rns = None, []

# Start throwing random numbers.
while rn not in rns:
```

```
    # Append RN to the sequence.
    if rn != None:
        rns.append(rn)
    # Generate the new random number using the middle square method.
    rn = rng()
    # Print the result.
    print(f"#{len(rns)}: {rn}")

# Print the result.
print(
    f"We began with {seed} and"
    f" have repeated ourselves after {len(rns)} steps"
    f" with {rn}."
)
```

```
 #0: 0.5227522752275228
 #1: 0.3215321532153215
 #2: 0.33623362336233625
 #3: 0.30303030303030304
 #4: 0.18091809180918092
 #5: 0.27242724272427243
 #6: 0.42014201420142017
 #7: 0.6484648464846484
 #8: 0.04220422042204221
 #9: 0.178017801780178
 #10: 0.16841684168416843
 #11: 0.8358835883588359
 #12: 0.8561856185618562
 #13: 0.2907290729072907
 #14: 0.45064506450645064
 #15: 0.30403040304030404
 #16: 0.24162416241624163
 #17: 0.8370837083708371
 #18: 0.05690569056905691
 #19: 0.32373237323732373
 #20: 0.47814781478147816
 #21: 0.857985798579858
 #22: 0.5992599259925993
 #23: 0.9040904090409041
 #24: 0.7216721672167217
 #25: 0.0706070607060706
 #26: 0.49844984498449846
 #27: 0.8402840284028403
 #28: 0.5936593659365936
 #29: 0.23602360236023603
 #30: 0.5696569656965697
 #31: 0.4444444444444444
 #32: 0.7491749174917491
 #33: 0.115011501150115
 #34: 0.32253225322532253
 #35: 0.4006400640064006
 #36: 0.048004800480048
 #37: 0.2304230423042304
 #38: 0.3084308430843084
 #39: 0.5110511051105111
 #40: 0.11211121112111211
 #41: 0.25662566256625663
```

```
    #42: 0.5843584358435844
    #43: 0.1406140614061406
    #44: 0.976897689768977
    #45: 0.4138413841384138
    #46: 0.12301230123012301
    #47: 0.512951295129513
    #48: 0.30663066306630665
    #49: 0.40034003400340035
    #50: 0.024002400240024
    #51: 0.0576057605760576
    #52: 0.3317331733173317
    #53: 0.0024002400240024004
    #54: 0.0005000500050005
    #55: 0.0
    #56: 0.0
    We began with 1234 and have repeated ourselves after 56 steps with 0.0.


# Now, we try out all the seeds for n = 2.
# Create a dictionary of periods.
periods = {}

# For all seeds we iterate between 1 and 99.
# We don't use 0, since this has a period of 0.
for seed in range(1, 100):
    # Create the RNG.
    rng = RNGMiddleSquare(seed, 2)

    # Determine the period.
    rns = rng_sequence(rng)

    # Include the period and sequence in the dictionary.
    period = len(rns)
    if not period in periods:
        periods[period] = [[seed, rns]]
    else:
        periods[period] += [[seed, rns]]

# Print out a sorted list of the periods.
for period, seeds in sorted([(key, val) for key, val in periods.items()]):
    print("-" * 10)
    print(f"period = {period}")
    print("-" * 10)
    for seed, rns in seeds:
        print(f"seed = {seed}")
        if len(rns) > 0:
            print("  mean = ", mean(rns))
            print("  variance = ", variance(rns))

    ----------
    period = 1
    ----------
    seed = 1
      mean =  0.0
      variance =  0.0
    seed = 2
```

```
seed = 2
  mean =  0.0
  variance =  0.0
seed = 3
  mean =  0.0
  variance =  0.0
seed = 10
  mean =  0.10101010101010101
  variance =  0.0
seed = 40
  mean =  0.6060606060606061
  variance =  0.0
seed = 50
  mean =  0.5050505050505051
  variance =  0.0
seed = 51
  mean =  0.6060606060606061
  variance =  0.0
seed = 60
  mean =  0.6060606060606061
  variance =  0.0
seed = 90
  mean =  0.10101010101010101
  variance =  0.0
seed = 98
  mean =  0.6060606060606061
  variance =  0.0
----------
period = 2
----------
seed = 4
  mean =  0.005050505050505051
  variance =  2.5507601265177026e-05
seed = 5
  mean =  0.010101010101010102
  variance =  0.0001020304050607081
seed = 6
  mean =  0.015151515151515152
  variance =  0.00022956841138659323
seed = 20
  mean =  0.5050505050505051
  variance =  0.010203040506070793
seed = 24
  mean =  0.4090909090909091
  variance =  0.02777777777777779
seed = 30
  mean =  0.5050505050505051
  variance =  0.16324864809713285
seed = 32
  mean =  0.010101010101010102
  variance =  0.0001020304050607081
seed = 45
```

RNGs based on modular arithmetic

The middle square can have short periods (bad) and are not even very random (really bad).

These types of RNGs were quickly overshadowed by those based on modular arithmetic. Two numbers are of the same modular class if they differ by only multiples of an integer m. Thus,

$$1 \equiv 4 \pmod 3 \equiv 7 \pmod 3 \equiv 82 \pmod 3.$$

In Python, the mod operator is given by the `%` symbol. Modular arithmetic has some nice properties that allow us to calculate or estimate periods, and it can be done quickly on a computer (a feature that could be relevant when sampling millions of RNs). However, the equivalences can also lead to some unexpected consequences.

First, we will explore linear congruential generators. These have the form,

$$x_i = ax_{i-1} + c \mod m, 0 < x_i < m \in \mathbb{Z},$$

where a is the *multiplier*, a positive integer, m is the *modulus*, also an integer, and c is the *increment*, a non-negative integer. As with all such algorithms, you can convert this into a value between 0 and 1 by dividing by the largest element m. In general, a should be relatively prime to m. When $c \neq 0$ this type of RNG is called a linear congruential genenerator (LCG) and when $c = 0$ this RNG is called a multiplicative LCG (MLCG). In the examples below, we will focus on the special case of the MLCG.

```
# We can now define a linear congruential generator.
class RNGLCG(RNG):
    # For our state, we only need the previously generated integer.
    # We give a default value of `c` so this is an MLCG.
    def __init__(self, seed, a=65593, m=2**31, c=0):
        super().__init__(seed)
        self.state = seed
        self.a = a
        self.m = m
        self.c = c

    # Here, we define the actual RNG.
    def __call__(self):
        self.state = (self.a * self.state + self.c) % self.m
        return self.state / (self.m - 1)
```

Now, let us test this RNG by first checking its period, mean, and variance.

```
# Create the RNG.
rng = RNGLCG(9, 3, 31)

# Find the period.
rns = rng_sequence(rng)

# Print the period, mean, and variance.
print(f"period = {len(rns)}")
print(f"mean = {mean(rns)}")
print(f"variance = {variance(rns)}")
```

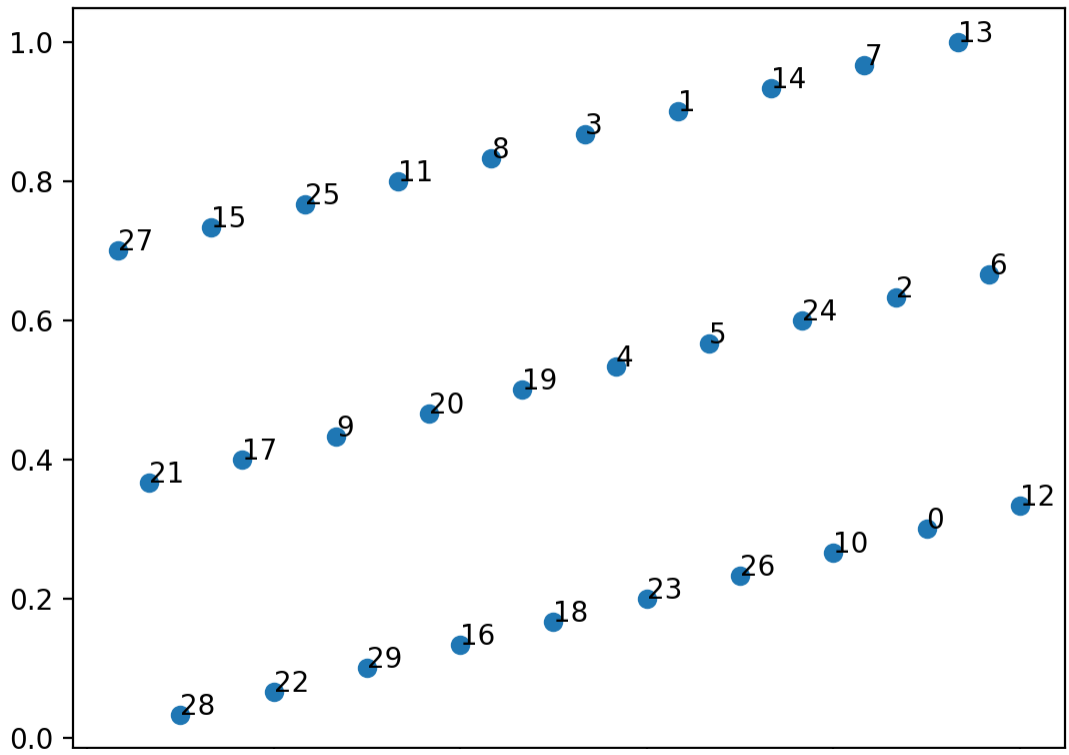
```
period = 30
mean = 0.5166666666666666
variance = 0.08324074074074073
```

Exercise: patterns in multiplicative congruential generator

Let's investigate some other properties of this RNG. The pairs (x_i, x_{i-1}) lie on a plane. Plot their pattern.

```
# Our x values are just the random numbers.
xs = rns

# Our y values are shifted to the left by one.
ys = rns[-1:] + rns[:-1]
# This can also be accopmlished with `numpy.roll`.
# ys = numpy.roll(rns, 1)

# Create the plot.
fig, ax = plt.subplots()
ax.scatter(xs, ys)

# Label each RN by its number in the sequence.
for i, rn in enumerate(rns):
    ax.annotate(str(i), (xs[i], ys[i]))
```

Figure 1



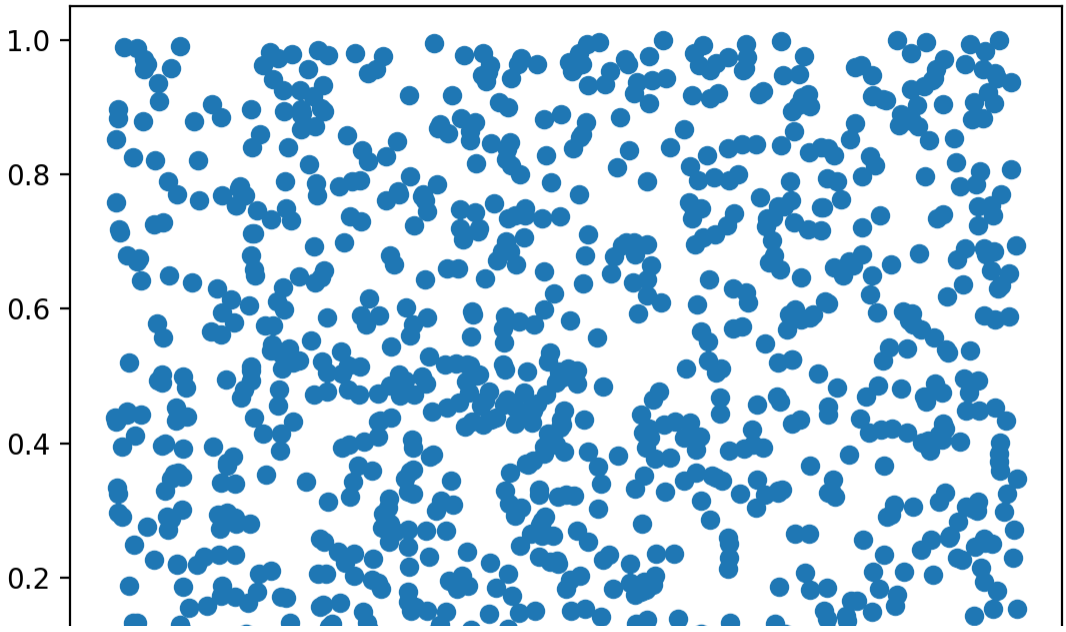
Exercise: choosing better parameters

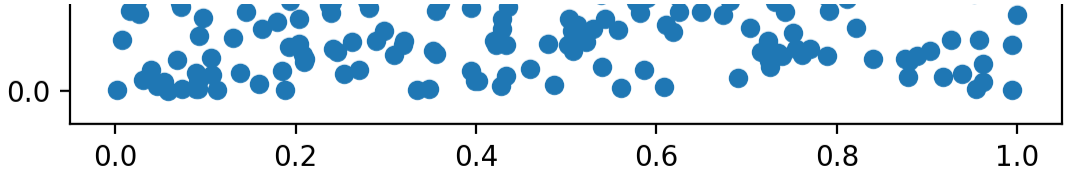
A better choice of a and m can give our RNG better properties. Create a new RNG with $a = 65539$ and $m = 2^{31}$. Generate a sequence of 1000 RNs using seed = $2^8 - 1$. Make a sequence of points (x_i, x_{i-1}) and plot them. Do any patterns emerge?

```
# Create the random number generator.
rng = RNGLCG(2**8 - 1, 65539, 2**31)

# Generate the sequence.
rns = rng_sequence(rng, 1000)


# Create the pairs.
xs = rns
ys = rns[-1:] + rns[:-1]

# Create the plot.
fig, ax = plt.subplots()
ax.scatter(xs, ys)
```

```
<matplotlib.collections.PathCollection at 0x785cb80189d0>
```

Figure 2



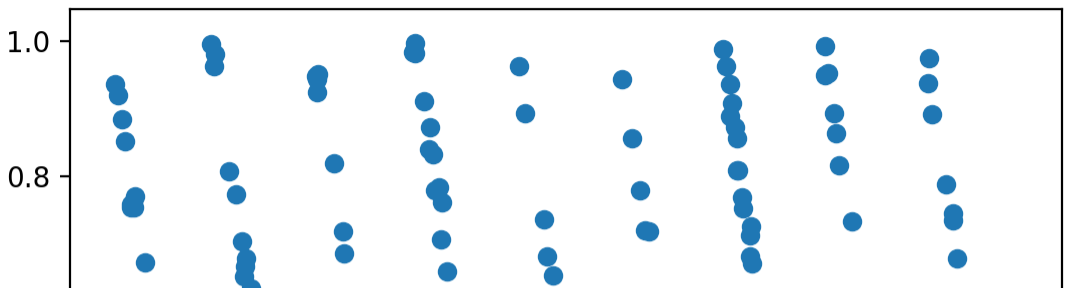


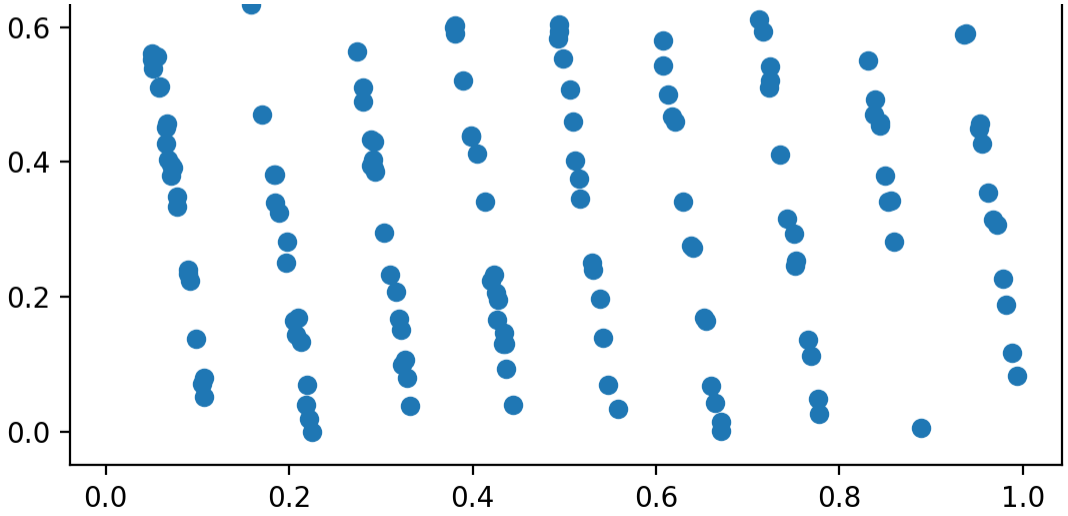
Now, given a little more advanced knowledge, let us see if we can find a pattern. Using the same RNG and seed, generate 20002 RNs and make a sequence of points (x_{i-1}, x_i, x_{i+1}). Filter these points for $0.50 < x_i < 0.51$. Make a 2-D scatter plot of (x_{i-1}, x_{i+1}) for these filtered points. Do any patterns emerge?

```
# Create the random number generator.
rng = RNGLCG(2**8 - 1, 65539, 2**31)

# Generate the sequence.
rns = rng_sequence(rng, 20002)


# Create the pairs.
xs = rns[-1:] + rns[:-1]
ys = rns[1:] + rns[:1]

# Filter the pairs.
xs = [x for x, rn in zip(xs, rns) if 0.50 < rn < 0.51]
ys = [y for y, rn in zip(ys, rns) if 0.50 < rn < 0.51]

# Create the plot.
fig, ax = plt.subplots()
ax.scatter(xs, ys)

# Indeed, we do see a pattern here!
```

```
<matplotlib.collections.PathCollection at 0x785ca061de10>
```

Figure 3

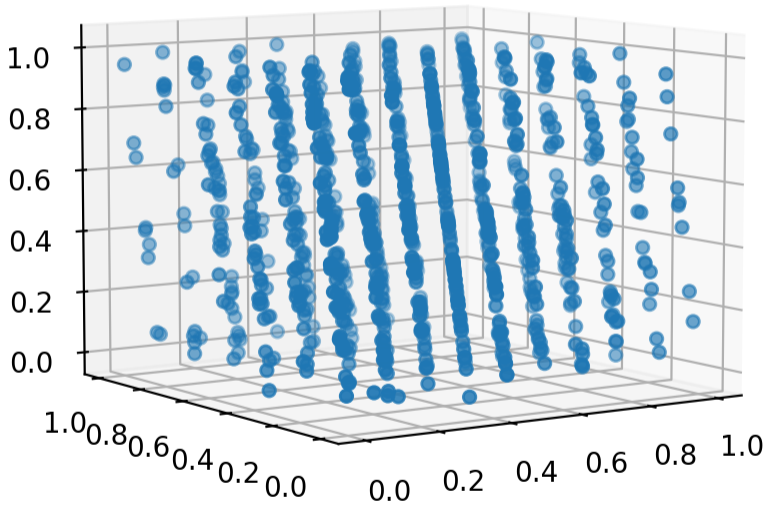




So far, we have only been looking in two dimensions, what about in three? Generate 1000 RNs using the same RNG and seed. Then make a 3-D scatter plot with the points (x_{i-1}, x_i, x_{i+1}).

```
# Create the random number generator.
rng = RNGLCG(2**8 - 1, 65539, 2**31)

# Generate the sequence.
rns = rng_sequence(rng, 1000)


# Create the triplets.
xs = rns[-1:] + rns[:-1]
ys = rns
zs = rns[1:] + rns[:1]

# Create the figure.
fig = plt.figure()
ax = fig.add_subplot(projection="3d")
ax.scatter3D(xs, ys, zs)

# Rotate the view so we can see the pattern.
ax.view_init(elev=7, azim=-124)
```

Figure 4



This example is not just pedagogical. In fact, this RNG is called `RANDU`. The authors of Numerical Recipes (3rd ed), share this anecdote:

> Even worse, many early generators happened to make particularly bad choices for m and a. One infamous such routine, `RANDU`, with a = 65539 and m = 2^{31}, was widespread on IBM mainframe computers for many years, and widely copied onto other systems. One of us recalls as a graduate student producing a "random" plot with only 11 planes and being told by his computer center's programming consultant that he had misused the random number generator: "We guarantee that each number is random individually, but we don't guarantee that more than one of them is random." That set back our graduate education by at least a year!

XORshift RNGs

Modern RNGs still combine two algorithms to remove undesired correlations. However, they use an independent algorithm for the first sequence, so as not to unwittingly combine two correlations that arise from the same class of algorithm.

One such popular algorithm is called `XORshift`. Its properties are understood by studying the multiplication of 3 special kinds of binary matrices: the identity matrix I, a right-shift matrix R, and a left-shift matrix L. However, this is type of RNG is typically programmed more efficiently using bit-shift and exclusive OR (XOR) operations (true if exactly one of two inputs is true, false otherwise).

The resulting algorithm does not look anything like matrix multiplication, but it really is. This is because a bit shift can be represented on an n-bit vector (typically 32-bit or 64-bit) by a matrix with only ones on a sub-diagonal. Thus, to right-shift a bit sequence $\beta = (b_1, b_2, \cdots, b_n) \to \beta' = (0, b_1, b_2, \cdots, b_{n-1})$, you would multiply β by an $n \times n$ matrix R,

$$\beta' = \beta R$$

with only 1s above the diagonal.

$$R \equiv \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Similarly, a left-shift matrix has only 1s on a subdiagonal below the diagonal.

$$L = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Finally, since these are binary matrices, all operations use integer arithmetic modulo 2, *i.e.* all arithmetic operations are defined as normal, but with the result taken as $\pmod 2$.

It is important to note here that the right-shift defined above is a *logical* right shift and not an *arithmetic* right shift. In a logical right shift, the left-most bit is set to zero, while for an arithmetic right-shift the left-most bit does not change. This is important for signed numbers, where the left-most bit gives the sign of the number. In most programming languages, the right-shift operator is defined as the arithmetic right shift and not the logical right shift.

Let us define a transformation matrix T in terms of XOR operations and bit-shifts.

$$T = (1 + L^a)(1 + R^b)(1 + L^c)$$

This transformation matrix is an $n \times n$ binary matrix with (a, b, c) bit-shifts left-right-left. Here, 1 is an $n \times n$ identity matrix. This identity matrix, in combination with the modulo 2 arithmetic, corresponds to an XOR operator.

The claim is that the series of operations $\beta T, \beta T^2, \cdots \beta T^{2^n - 1}$, every possible β is produced. This means that the cycle has length $2^n - 1$. Only certain triplets of (a, b, c) fulfill the ideal cycle length of $2^n - 1$. A minimal test that T has the desired properties is to check that

$$T^{2^n} = T$$

holds.

Exercise: implement bit shifting

Before we implement a full `XORshift` RNG, we need to build the necessary matrices to define the transformation T. Using the `sparse` sub-module from `scipy`, define functions that return R and L matrices. We use the `sparse` module to make the calculation more efficient than just full matrix multiplication.

```
# First, we define the method for the right-shift matrix.
def build_rshift(n, dtype=np.int32):
    """
    Return a right-shift matrix of dimension `n`.

    n:     dimension of the shifting matrix.
    dtype: type of entries to use in the matrix.
    """
    return sp.sparse.diags([1], [1], shape=(n, n), dtype=dtype)


# Second, we define the method for the left-shift matrix.
def build_lshift(n, dtype=np.int32):
    """
    Return a left-shift matrix of dimension `n`.

    n:     dimension of the shifting matrix.
    dtype: type of entries to use in the matrix.
    """
    return sp.sparse.diags([1], [-1], shape=(n, n), dtype=dtype)


# For completeness, we define the identity matrix as well.
def build_i(n, dtype=np.int32):
    """
    Return an identity matrix of dimension `n`.

    n:     dimension of the shifting matrix.
    dtype: type of entries to use in the matrix.
    """
    return sp.sparse.eye(n, dtype=dtype)


# Print an example R.
r = build_rshift(4)
print(f"R =\n{r.toarray()}")

# Print an example L.
l = build_lshift(4)
print(f"L =\n{l.toarray()}")
```

```
# We can also check to see if the shift actually works.
vec = [1, 1, 1, 1]
print(f"original      = {np.array(vec)}")
print(f"right shifted = {vec * r}")
print(f"left shifted  = {vec * l}")
```

```
R =
[[0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]
 [0 0 0 0]]
L =
[[0 0 0 0]
 [1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]]
original      = [1 1 1 1]
right shifted = [0 1 1 1]
left shifted  = [1 1 1 0]
```

In Python, the right-shift operator is given by `>>` followed the number of positions, and the left-shift operator is given by `<<`, also followed by the number of positions. Check that your shift matrices are correctly built using these built in operators, remembering that the right-shift operator is an arithmetic shift.

As a brief, but relevant aside, the Python `int` type does not have a fixed size and instead uses "arbitrary precision arithmetic" (also known as `bignum`). This means that you should use the fixed size-types from `numpy` instead. Note, real number types like `float` in Python are fixed size. Why is `float` fixed size but `int` variable size?

```
# It is useful to first write a little utility that creates
# a vector of bits from a decimal number.
def dec_to_bits(dec, dtype=int):
    """
    Return a vector of bits with type `dtype` given decimal value `dec`.

    dec:   decimal value to convert to bits.
    dtype: type of elements in the returned vector.
    """
    # Determine the storage size of the value type in bytes. Multiply by 8
    # to convert from bytes to bits.
    n = np.dtype(dec).itemsize * 8
    # Convert it into a binary string, strip the leading `0b`, and zero pad.
    chars = bin(dec).lstrip("0b").zfill(n)
    # Split the string into a vector and return.
    return [dtype(char) for char in chars]


# It is also useful to got the opposite direction, convert a vector of bits
# into a decimal value
```

```
# into a decimal value.
def bits_to_dec(bits, dtype=int):
    """
    Return a decimal value of type `dtype` given `bits`, a vector of bits.

    bits:  vector of bits to convert to decimal value.
    dtype: type of the returned decimal value.
    """
    # Convert the vector to a binary string.
    chars = "".join([f"{int(bit)}" for bit in bits])
    # Read the binary string into a decimal value.
    return dtype(chars, 2)


# Define an integer value to shift. We work with unsigned 8-bit integers
# so that the right-shift is the same between logical and arithmetic.
val = np.uint8(120)
# Convert this decimal value into a vector of bits.
beta = dec_to_bits(val)

# We now need our operators. Here, `n` is given by the length
# of `beta`. Here, `n` should be 8.
n = len(beta)
r = build_rshift(n)
l = build_lshift(n)

# Multiply the vector by the right-shift operator.
beta_p = beta * r

# Convert back to decimal form, print, and check.
dec = bits_to_dec(beta_p)
print(f"matrix right-shifted = {dec}")
print(f"operator right-shifted = {val >> 1}")

# We do the same for left-shifting.
beta_p = beta * l
dec = bits_to_dec(beta_p)
print(f"matrix left-shifted = {dec}")
print(f"operator left-shifted = {val << 1}")

     matrix right-shifted = 60
     operator right-shifted = 60
     matrix left-shifted = 240
     operator left-shifted = 240



# Let us now address integers can use `bignum`, while real numbers cannot.

# The size of a finite integer is fixed. The following
# gives the size of `val`. The `-2` accounts for the leading "0b".
val = 278811
print("-" * 10)
print(f"val: {val}")
print(f"bits of val: {len(bin(val)) - 2}")
```

```
print(f"bits of val: {len(bin(val)) - 2}")

# We can check this against the internal Python representation.
print(f"Python storage: {sys.getsizeof(val)*8}")

# Interesting, these values are not the same. It turns out that Python stores
# additional information for an integer.
# * reference count (64 bits): Stores how many times this integer is
#   referenced in memory.
# * data type (64 bits): Tells Python what data type this object is.
# * size (64 bits): Allocates the memory where the actual value is stored.
# * value (? bits): The actual memory where the value is stored.
val = 1
print("-" * 10)
print(f"val: {val}")
print(f"bits of val: {len(bin(val)) - 2}")
print(f"Python storage: {sys.getsizeof(val)*8}")

# From the above, see that 32 bits are used as the actual memory allocation
# for the integer value. Why, when only 1 bit is needed? This has to do with
# memory allocation. This value must be stored in a contiguous block of memory.
# Whenever the value changes and more memory is needed, a memory block must
# be allocated, and the value must be copied. Turns out that 32 bits covers
# most integers users will use. What happens if we go really big?
val = 123848585883838383838383838383
print("-" * 10)
print(f"val: {val}")
print(f"bits of val: {len(bin(val)) - 2}")
print(f"Python storage: {sys.getsizeof(val)*8}")

# Here we see that we now need 97 bits, and Python has allocated an additional
# 96 bits (128 bits total).

# So why doesn't this work for real numbers? Real numbers can require an
# infinite precision rather than a fixed size, so this expansion is no longer
# possible.
```

```
----------
val: 278811
bits of val: 19
Python storage: 224
----------
val: 1
bits of val: 1
Python storage: 224
----------
val: 123848585883838383838383838383
bits of val: 97
Python storage: 320
```

We can also consider the general form of L^a and R^b. Is there a more efficient way to write L^a then just L**a ?

```
# First, we can see what happens when we take L^a.
n = 4
l = build_lshift(n)
print("-" * 10 + "\nmatrix multiplication\n" + "-" * 10)
for a in range(1, n + 1):
    la = l**a
    print(f"L^{a} =\n{la.toarray()}")

# We see that the diagonal is just shifted down.
# It turns out that we can replicate this with the `diag` method.
print("-" * 10 + "\nusing `diags` method\n" + "-" * 10)
for a in range(1, n + 1):
    la = sp.sparse.diags([1], [-a], shape=(n, n), dtype=np.int32)
    print(f"L^{a} =\n{la.toarray()}")
```

```
----------
matrix multiplication
----------
L^1 =
[[0 0 0 0]
 [1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]]
L^2 =
[[0 0 0 0]
 [0 0 0 0]
 [1 0 0 0]
 [0 1 0 0]]
L^3 =
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [1 0 0 0]]
L^4 =
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
----------
using `diags` method
----------
L^1 =
[[0 0 0 0]
 [1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]]
L^2 =
[[0 0 0 0]
 [0 0 0 0]
 [1 0 0 0]
 [0 1 0 0]]
L^3 =
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [1 0 0 0]]
```

```
 [1 0 0 0]]
L^4 =
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

```
# We can use this property to construct L^a and R^b with
# more general `build_lshift` and `build_rshift` methods.


# First, redefine R.
def build_rshift(n, a=1, dtype=np.int32):
    """
    Return a right-shift matrix R^a of dimension `n`.

    n:     dimension of the shifting matrix.
    a:     number of right shifts to perform.
    dtype: type of entries to use in the matrix.
    """
    return sp.sparse.diags([1], [b], shape=(n, n), dtype=dtype)


# Second, redfine L.
def build_lshift(n, a=1, dtype=np.int32):
    """
    Return a left-shift matrix L^a of dimension `n`.

    n:     dimension of the shifting matrix.
    a:     number of left shifts to perform.
    dtype: type of entries to use in the matrix.
    """
    return sp.sparse.diags([1], [-a], shape=(n, n), dtype=dtype)
```

Exercise: determine suitable triplets

Construct T for the triplets $(1, 3, 10)$, $(5, 17, 13)$, and $(2, 5, 14)$ using 32-bit precision. Which of these are suitable triplets, using the test $T^{2^n} = T$?

```
# First we create the `build_tshift` method.
def build_tshift(n, a, b, c, reverse=False, dtype=np.int32):
    """
    Return a shift matrix of dimension `n` with the form:
    T = (1 + L^a)(1 + R^b)(1 + L^c)
    If `reverse` is `True`, than the operator directions are swapped.
    T = (1 + R^a)(1 + L^b)(1 + R^c)

    n:       dimension of the shifting matrix.
    a:       number of first shifts.
```

```
    b:       number of second shifts.
    c:       number of third shifts.
    reverse: reverse from left-right-left to right-left-right.
    dtype:   type of entries to use in the matrix.
    """
    if reverse:
        return (
            (build_i(n, dtype) + build_rshift(n, a, dtype))
            * (build_i(n, dtype) + build_lshift(n, b, dtype))
            * (build_i(n, dtype) + build_rshift(n, c, dtype))
        )
    else:
        return (
            (build_i(n, dtype) + build_lshift(n, a, dtype))
            * (build_i(n, dtype) + build_rshift(n, b, dtype))
            * (build_i(n, dtype) + build_lshift(n, c, dtype))
        )


# Define our triplets.
ts = ((1, 3, 10), (5, 17, 13), (2, 5, 14))

# We are testing for 32-bits.
n = 32

# Loop over the triplets.
for a, b, c in ts:
    # Construct the shift matrix.
    t = build_tshift(n, a, b, c)
    # Test the condition.
    t = t.todense()
    check = (t ** (2**n) % 2) - t
    print(f"non-zero elements for T{a, b, c} = {np.count_nonzero(check)}")
```

```
 non-zero elements for T(1, 3, 10) = 0
 non-zero elements for T(5, 17, 13) = 0
 non-zero elements for T(2, 5, 14) = 524
```

Exercise: implement XORshift with matrices

Now that we have all the components, define an XORshift RNG. From the exercise above, choose valid default values for a, b, and c.

```
# We can now define an XORshift generator using matrices.
class RNGXORShiftMatrix(RNG):
    """
    RNG using an three XOR operations in combination with bit shifts.
    """

    # The `dtype` determines how many bits we use
```

```
    # The `dtype` determines how many bits we use.
    def __init__(self, seed, a=5, b=17, c=13, reverse=False, dtype=np.uint32):
        """
        Constructor for this class, e.g. `RNGXORShiftMatrix(seed)`.

        seed: seed used to initialize the generator.
        n:       dimension of the shifting matrix.
        a:       number of first shifts.
        b:       number of second shifts.
        c:       number of third shifts.
        reverse: reverse from left-right-left to right-left-right.
        dtype:   type of entries to use in the matrix.
        """
        super().__init__(seed)
        # The state is just our beta vector.
        self.state = dec_to_bits(dtype(seed))
        # It's useful to store the number of bits we use.
        self.n = len(self.state)
        # We also need the shift matrix.
        self.t = build_tshift(self.n, a, b, c, reverse, dtype)

    # Here, we define the actual RNG.
    def __call__(self):
        """
        Return a uniform RN.
        """
        self.state = (self.state * self.t) % 2
        return bits_to_dec(self.state) / (2**self.n - 1)


# We can now generate some numbers and test the generator.
# The implementation of this RNG is not efficient, and so generation
# is slow. Luckily, we can reimplement with shift operators which is
# siginficantly faster.
rng = RNGXORShiftMatrix(20)
rns = rng_sequence(rng, 10000)
print(f"period = {len(rns)}")
print(f"mean = {mean(rns)}")
print(f"variance = {variance(rns)}")
```

```
period = 10000
mean = 0.5000721137264446
variance = 0.08428034641603971
```

Exercise: implement XORshift with operators

It turns out, not suprisingly, that the `XORshift` RNG we implemented with matrices is not particularly efficient. How many operations per RNG call are required for the matrix implementation? Assume the matrices are dense and ignore the operations necessary to convert to from a vector of bits to a real number.

We only need to build the transformation matrix T once, so this means that for each call we need to peform the following.

$$\beta' = \beta T$$

This consists of multiplying a vector of length n with a matrix of size $n \times n$ which is $2n^2$ operations.

So, let us now implement a more efficient version of this generator where we use the built in Pythonic operators for XOR and bit shifting.

```
# Class to perform XORshift with Python bit-wise operators.
class RNGXORShiftOperator(RNG):
    # Use the same default `a`, `b`, and `c` as RNGXORShiftMatrix.
    # Use the same default `dtype` as RNGXORShiftMatrix.
    def __init__(self, seed, a=5, b=17, c=13, reverse=False, dtype=np.uint32):
        super().__init__(seed)
        # Store the state.
        self.state = dtype(seed)
        # Store the bit size.
        self.n = self.state.itemsize * 8
        # Store the `a`, `b`, and `c` configuration.
        self.a = a
        self.b = b
        self.c = c
        self.reverse = reverse

    # Define the actual RNG call.
    def __call__(self):
        if self.reverse:
            # Bit-shift `state` right by `a`, then take XOR with non-shifted state.
            self.state ^= self.state >> self.a
            # Bit-shift `state` left by `b`, then take XOR with non-shifted state.
            self.state ^= self.state << self.b
            # Bit-shift `state` right by `c`, then take XOR with non-shifted state.
            self.state ^= self.state >> self.c
            # Normalize and return.

        else:
            # Bit-shift `state` left by `a`, then take XOR with non-shifted state.
            self.state ^= self.state << self.a
            # Bit-shift `state` right by `b`, then take XOR with non-shifted state.
            self.state ^= self.state >> self.b
            # Bit-shift `state` left by `c`, then take XOR with non-shifted state.
            self.state ^= self.state << self.c
            # Normalize and return.
        return self.state / (2**self.n - 1)
```

Compare this `XORshift` RNG with the one you wrote using matrices. Do the two match? Should they?

```
# Define the two RNGs.
rng_operator = RNGXORShiftOperator(20)
rng_matrix = RNGXORShiftMatrix(20)
# Loop over the first 5 RNs generated.
for i in range(0, 5):
    print(rng_operator(), rng_matrix())

# They match!
```

```
0.0012590037661741962 0.0012208521368961903
0.31494838495621186 0.3125047731009556
0.6018479994036835 0.07241821290748618
0.3318541944799605 0.7962083818382137
0.8610716305815316 0.12702384594060104
```

Combined RNGs

We have explored three different RNG methods: Middle Square Method, LCGs, and `XORshift`. While only the final RNG method is currently considered to be sufficient, another possibility is to combine RNGs together to produce a better RNG. In Numerical Recipes (3rd ed), a number of guidelines are given for what constitutes a bad RNG, which is then followed by this "received wisdom of the present".

> An acceptable random generator must combine at least two (ideally, unrelated) methods. The methods combined should evolve independently and share no state. The combination should be by simple operations that do not produce results less random than their operands.

In Numerical Recipes, an RNG is recommended that combines four different RNGs. The first RNG is a an `XORshift`, the second is an LCG, the third is another `XORshift`, and the fourth is a Multiply with Carry (MWC). These RNGs are combined as follows.

$$(\mathrm{XOR}_1(\mathrm{LCG}_2) + \mathrm{XOR}_3) \wedge \mathrm{MWC}_4$$

The notation here of $\mathrm{XOR}_1(\mathrm{LCG}_2)$ is called a succesor relation, where the state of LCG_2 is still iterated as normal, but is taken as the state of XOR_1.

Exercise: implement an MWC RNG

The idea behind an MCW is similar to an LCG with $m = ab - 1$, but is implemented using a bit-shift with an AND operation. The algorithm is as follows, considering an n-bit state.

1. Right-shift x_i by $n/2$.
2. Take the AND operation between x_i and $2^{n/2} - 1$.

3. Multiply the result of (2) by a and add to the result of (1). Implement an MCW for $n = 64$ and $a = 4294957665$.

```
# Class to perform MWC.
class RNGMWC(RNG):
    def __init__(self, seed, a=4294957665, dtype=np.uint64):
        super().__init__(seed)
        # Store the state.
        self.state = dtype(seed)
        # Store the bit size.
        self.n = self.state.itemsize * 8
        # Store the `a` configuration.
        self.a = a
        self.b = dtype((2 ** (self.n / 2) - 1))
        self.c = int(self.n / 2)

    # Define the actual RNG call.
    def __call__(self):
        # Bit-shift `state` right by `n/2`.
        x = self.state >> self.c
        # Take bit-wise AND with 2^(`n`/2) - 1, multiply by `a`.
        y = self.a * (self.state & self.b)
        # Add the results.
        self.state = x + y
        # Normalize and return.
        return self.state / (2**self.n - 1)


# Now we can test the generator.
rng = RNGMWC(13)
rns = rng_sequence(rng, 10000)
print(f"period = {len(rns)}")
print(f"mean = {mean(rns)}")
print(f"variance = {variance(rns)}")
```

```
period = 10000
mean = 0.4973903249805164
variance = 0.08349778027608351
```

Exercise: combine RNGs

We now have all the ingredients needed to create the recommended RNG combination from Numerical recipes. However, we need to know the parameters used for each RNG.

- XOR_1: $a = 21, b = 35, c = 4$, and left-right-left shifts
- LCG_2: $a = 2862933555777941757$, $c = 7046029254386353087$, and $m = 2^n$
- XOR_3: $a = 17, b = 31, c = 8$, and right-left-right shifts
- MWC_4: $a = 4294957665$

For all the generators we have $n = 64$. There is also some additional initialization that is required that is described in the comments for the exercise code.

To check your solution, the first 5 random numbers should be the following.

```
 0.40573455184711105
 0.4626566077784852
 0.06435855239385686
 0.5209198360653725
 0.034725647607092826
```

```
# Class to create a combined RNG.
class RNGCombine(RNG):
    def __init__(
        self,
        seed,
        seed3=4101842887655102017,
        seed4=1,
        a1=21,
        b1=35,
        c1=4,
        a2=2862933555777941757,
        c2=7046029254386353087,
        a3=17,
        b3=31,
        c3=8,
        reverse3=True,
        a4=4294957665,
        dtype=np.uint64,
    ):
        super().__init__(seed)
        # Determine the seeds for the RNGs. Any value for `seed1` is fine,
        # since this seed is never used in RNG1.
        seed1 = dtype(seed)
        # The seed for RNG2 is the XOR of `seed1` and `seed3`.
        seed2 = int(dtype(seed1) ^ dtype(seed3))
        # We already have `seed3` and `seed4`.

        # Store the bit size and data type.
        self.n = seed1.itemsize * 8
        self.dtype = dtype

        # Create the 4 RNGs.
        self.rng1 = RNGXORShiftOperator(seed1, a1, b1, c1, False, dtype)
        self.rng2 = RNGLCG(seed2, a2, 2**self.n, c2)
        self.rng3 = RNGXORShiftOperator(seed3, a3, b3, c3, True, dtype)
        self.rng4 = RNGMWC(seed4, a4, dtype)
        self.rngs = [self.rng1, self.rng2, self.rng3, self.rng4]
```

```
        # Perform some additional initialization.
        # Call the RNG with the current states.
        self()
        # Update the state of RNG3 to that of RNG2.
        self.rng3.state = self.dtype(self.rng2.state)
        # Call the RNG with the current states.
        self()
        # Update the state of RNG4 to that of RNG3.
        self.rng4.state = self.rng3.state
        # Call the RNG with the current states.
        self()

    # Define the actual RNG call.
    def __call__(self):
        # Independently call RNG2, RNG3, and RNG4.
        self.rng2()
        self.rng3()
        self.rng4()
        # Set the RNG1 state to the RNG2 state. Make sure this is the correct
        # data type.
        self.rng1.state = self.dtype(self.rng2.state)
        # Call RNG1.
        self.rng1()
        # Combine the states. We could use the following line.
        # state = (self.rng1.state + self.rng3.state) ^ self.rng4.state
        # But, this generates an overflow warning. The behavior is correct,
        # since this just performs modulo 2^64. However, we can avoid the
        # warning by using the `bignum` Python integer type, and then
        # performing modulo 2^64. Note, this is less efficient.
        state = (
            self.dtype((int(self.rng1.state) + int(self.rng3.state)) % 2**64)
            ^ self.rng4.state
        )
        # Normalize and return.
        return state / (2**self.n - 1)


# Now we can test the generator.
rng = RNGCombine(13)
# Loop over the first 5 RNs generated.
for i in range(0, 5):
    print(rng())

# Calcuate the mean and variance.
rns = rng_sequence(rng, 10000)
print(f"period = {len(rns)}")
print(f"mean = {mean(rns)}")
print(f"variance = {variance(rns)}")

     0.40573455184711105
     0.4626566077784852
     0.06435855239385686
     0.5209198360653725
     0.034725647607092826
```

```
     period = 10000
     mean = 0.5006965927025085
     variance = 0.08308961209839943



# It turns out that the `RNGCombine` method can be a little slow, mainly
# because of how we use the Python `int` class in our LCG. Below, we implement
# the same RNG but following more closely the implementation of Numerical
# Methods.
class RNGCombineNM(RNG):
    def __init__(
        self,
        seed,
        seed3=4101842887655102017,
        seed4=1,
        a1=21,
        b1=35,
        c1=4,
        a2=2862933555777941757,
        c2=7046029254386353087,
        a3=17,
        b3=31,
        c3=8,
        a4=4294957665,
        dtype=np.uint64,
    ):
        super().__init__(seed)

        # Set the states for the RNGs.
        self.state3 = dtype(seed3)
        self.state2 = dtype(seed) ^ self.state3
        self.state4 = dtype(seed4)

        # Set the parameters.
        self.n = self.state2.itemsize * 8
        self.a1, self.b1, self.c1 = a1, b1, c1
        self.a2, self.c2 = dtype(a2), dtype(c2)
        self.a3, self.b3, self.c3 = a3, b3, c3
        self.a4 = dtype(a4)
        self.b4, self.c4 = dtype((2 ** (self.n / 2) - 1)), int(self.n / 2)

        # Initialize the states further.
        self()
        self.state3 = self.state2
        self()
        self.state4 = self.state3
        self()

    # Define the actual RNG call.
    def __call__(self):
        # Call RNG2. Here, modulo 2^64 is handled by the overflow.
        # print(self.a2)
        # print(self.state2)
```

```
        # print(self.state2)
        # print(self.c2)
        self.state2 = self.a2 * self.state2 + self.c2
        # Call RNG3.
        self.state3 ^= self.state3 >> self.a3
        self.state3 ^= self.state3 << self.b3
        self.state3 ^= self.state3 >> self.c3
        # Call RNG4.
        self.state4 = self.a4 * (self.state4 & self.b4) + (self.state4 >> self.c4)
        # Call RNG1.
        state1 = self.state2 ^ (self.state2 << self.a1)
        state1 ^= state1 >> self.b1
        state1 ^= state1 << self.c1
        # Combine the states.
        state = (state1 + self.state3) ^ self.state4
        return state / (2**self.n - 1)


# We can compare this to `RNGCombine`. This second RNG will produce warnings
# since we are utilizing the overflow behavior of `numpy`.
rng_combine = RNGCombine(13)
rng_methods = RNGCombineNM(13)
# Loop over the first 5 RNs generated.
for i in range(0, 5):
    print(rng_combine(), rng_methods())
```

```
0.40573455184711105 0.40573455184711105
0.4626566077784852 0.4626566077784852
0.06435855239385686 0.06435855239385686
0.5209198360653725 0.5209198360653725
0.034725647607092826 0.034725647607092826
/tmp/ipython-input-38-1682420623.py:50: RuntimeWarning: overflow encountered in sca
  self.state2 = self.a2 * self.state2 + self.c2
/tmp/ipython-input-38-1682420623.py:62: RuntimeWarning: overflow encountered in sca
  state = (state1 + self.state3) ^ self.state4
/tmp/ipython-input-38-1682420623.py:50: RuntimeWarning: overflow encountered in sca
  self.state2 = self.a2 * self.state2 + self.c2
```

Tests of Random Number Generators

At the begining of this notebook, we defined a good RNG based on its mean and variance matching an expected uniform distribution. However, as we saw with various RNGs, the period of a generator can be small, so it begins repeating, or patterns can even arise within a single period. However, as we saw for LCGs that even if we chose better parameters these patterns could continue to emerge. So, how do we catch these patterns, and what defines a good RNG?

There is not a simple answer to this question, but instead there are standard series of tests that can be applied to RNGs that attempt to quantify how good an RNG is. Perhaps the best known is the `diehard` test suite developed by George Marsaglia which was extended by Robert Brown into the

dieharder test suite, and by Pierre L'Ecuyer and Richard Simard into the TestU01 test suite. We will explore a few tests from the original `diehard` suite of tests.

Exercise: timing

Timing does not assess the quality of an RNG, but is necessary from a practical point of view to determine whether an RNG is usable. Evaluate the timing per RNG call for all the RNGs implemented in this notebook.

```
# We could create a list of the classes here.
# classes = [RNGCombine, ...]

# However, we can also inspect our global environment for relevant classes
# to make sure we catch 'em all.
rng_classes = []
for name, obj in list(globals().items()):
    # Check the object is a class and is in the main namespace.
    if inspect.isclass(obj) and obj.__module__ == "__main__":
        # Use only the classes with a name starting with RNG.
        if name.startswith("RNG") and name != "RNG":
            rng_classes += [obj]

# Now we needs to create our RNGs.
rngs = [rng_class(13) for rng_class in rng_classes]

# Next, we can time each RNG for 1000 calls.
calls = 1000
timings = []
import time

for rng in rngs:
    start = time.time()
    for call in range(0, calls):
        rng()
    stop = time.time()
    timings += [(stop - start, rng)]

# Finally, we print the timing for each RNG, sorted low to high.
timings.sort()
for i, (timing, rng) in enumerate(timings):
    print(f"({i:{int(len(rngs)/10)}}) {timing/calls:.2e} " f"{rng.__class__.__name__}")
```

```
 (0) 4.75e-07 RNGLCG
 (1) 8.93e-07 RNGMWC
 (2) 1.22e-06 RNGXORShiftOperator
 (3) 1.26e-06 RNGMiddleSquare
 (4) 4.07e-06 RNGCombineNM
 (5) 6.64e-06 RNGCombine
 (6) 6.88e-05 RNGXORShiftMatrix
 /tmp/ipython-input-38-1682420623.py:50: RuntimeWarning: overflow encountered in sca
```

```
    self.state2 = self.a2 * self.state2 + self.c2
/tmp/ipython-input-38-1682420623.py:62: RuntimeWarning: overflow encountered in sca
    state = (state1 + self.state3) ^ self.state4
/tmp/ipython-input-38-1682420623.py:50: RuntimeWarning: overflow encountered in sca
    self.state2 = self.a2 * self.state2 + self.c2
```

Exercise: Kolmogorov-Smirnov test

The general idea behind the `diehard` tests is to generate a distribution produced from an algorithm using the RNG and compare this to the asymptotic limit of the distribution for the algorithm. To do this, we need to be able to quantify how well a discrete generated distribution agrees with a continuous distribution. The Kolmogorov-Smirnov (KS) test does this by finding the maximum distance between the cumulative distribution functions of the two distributions. Try to implement this below.

```
def ks_distance(xs, cdf):
    """
    Returns the KS test statistic.

    xs:  distribution of generated points.
    cdf: cumulative distribution function for comparison. This should be a
         function which takes an `x` and returns the CDF.
    """
    # Sort the values.
    xs.sort()
    # Set the initial maximum distance.
    dmax = 0
    # Determine the normalization.
    norm = len(xs)
    # Loop over the values.
    for i, x in enumerate(xs):
        # Set the number of values.
        n = i + 1
        # Find the distance.
        d = abs(cdf(x) - n / norm)
        # Update the maximum distance.
        dmax = max(d, dmax)
    # Return the maximum distance.
    return dmax
```

Now, let us test a uniform distribution with the KS test.

```
# Create the uniform distribution.
rng = RNGCombine(13)
xs = [rng() for i in range(0, 1000)]


# Define the CDF for the uniform distribution.
```

```
# Define the CDF for the uniform distribution.
def cdf_uniform(x):
    return x


# Run the KS test.
d = ks_distance(xs, cdf_uniform)
```

However, this number is not so useful unless we know the expected distribution for KS distances. The asymptotic CDF for the distance d as $n \to \infty$ for the generated distribution is given by the following.

$$\mathrm{CDF}(d) = \frac{\sqrt{2\pi}}{\sqrt{n}d} \sum_{k=1}^{\infty} e^{-(2k-1)^2\pi^2/(8nd^2)}$$

Implement a function which calculates the p-value for the KS test.

```
def ks_pvalue(d, n, kmax=100):
    """
    Returns the asymptotic KS p-value.

    d:    KS distance from a generated distribution.
    n:    points in the generated distribution.
    kmax: maximum number of iterations in the sum.
    """
    p = 0
    a = math.pi**2 / (8 * n * d**2)
    for k in range(1, kmax):
        p += math.exp(-((2.0 * k - 1) ** 2) * a)
    return 1 - p * (2 * math.pi) ** 0.5 / (n**0.5 * d)
```

We can test our implementation against the `scipy` implementation.

```
p = ks_pvalue(d, len(xs))
s = sp.stats.kstest(xs, cdf_uniform)
print(f"local distance: {d}")
print(f"scipy distance: {s.statistic}")
print(f"local distance: {p}")
print(f"scipy distance: {s.pvalue}")
```

```
local distance: 0.02069399865145033
scipy distance: 0.02069399865145033
local distance: 0.7851638037164712
scipy distance: 0.7770261716974947
```

Exercise: minimum distance test

There are quite a few tests in the `diehard` including tests such as the birthdays, craps, car parking,

and monkeys. We will focus on a test with a slightly more boring name, the minimum distance test. In this test, n random points (x_i, y_i) are generated where both x_i and y_i are uniformly distributed between 0 and 10^4. The minimum distance between all point combinations, $d^2_{\min}$, is then returned. Implement this test below.

```
def diehard_minimum_distance(rng, n):
    # Create the points.
    # Make sure they are sampled between 0 and 1e4
    xs = [(rng() * 1e4, rng() * 1e4) for i in range(0, n)]

    # Calculate the distance.
    dmin = 1e8
    for i, (xi, yi) in enumerate(xs):
        for j, (xj, yj) in enumerate(xs[i + 1 :]):
            d = (xi - xj) ** 2 + (yi - yj) ** 2
            dmin = min(d, dmin)

    # Return the minumum distance.
    return dmin
```

This value itself, or an ensemble of these values, is not useful unless we know the corresponding distribution. Create a distribution from 100 $d^2_{\min}$ using $n = 8000$ and the `RNGCombine` RNG. **This will take some time.**

```
# Create the RNG.
rng = RNGCombine(13)

# Run the test 100 times with 8000 points each.
tests = 100
n = 8000
xs = []
for test in range(0, tests):
    print(f"test {test}")
    x = diehard_minimum_distance(rng, n)
    print(f"    d = {x:.2e}")
    xs += [x]
```

```
test 0
    d = 5.00e-01
test 1
    d = 1.25e-01
test 2
    d = 2.08e+00
test 3
    d = 1.46e+00
test 4
    d = 1.04e+00
test 5
    d = 6.45e-01
test 6
```

```
    d = 7.29e-01
test 7
    d = 8.81e-01
test 8
    d = 1.01e-01
test 9
    d = 7.64e-02
test 10
    d = 4.26e-01
test 11
    d = 7.60e-01
test 12
    d = 2.06e+00
test 13
    d = 4.81e-02
test 14
    d = 1.54e+00
test 15
    d = 1.31e-01
test 16
    d = 1.79e+00
test 17
    d = 2.44e-01
test 18
    d = 2.64e+00
test 19
    d = 1.16e+00
test 20
    d = 3.17e+00
test 21
    d = 1.34e+00
test 22
    d = 8.56e+00
test 23
    d = 3.74e+00
test 24
    d = 1.73e+00
test 25
    d = 1.34e+00
test 26
    d = 1.17e+00
test 27
    d = 5.31e-01
test 28
    d = 1.47e-01
test 29
    d = 1.70e+00
test 30
    d = 6.36e-01
test 31
    d = 5.87e-01
test 32
    d = 6.98e-01
test 33
    d = 1.29e-01
test 34
    d = 1.20e+00
test 35
    d = 1.04e-01
```

d = 1.04e-01
test 36
d = 4.05e-01
test 37
d = 1.24e+00
test 38
d = 9.98e-01
test 39
d = 1.81e-01
test 40
d = 1.82e-01
test 41
d = 1.52e-01
test 42
d = 5.22e-01
test 43
d = 1.35e+00
test 44
d = 8.76e-01
test 45
d = 9.27e-01
test 46
d = 1.36e+00
test 47
d = 1.66e+00
test 48
d = 3.07e+00
test 49
d = 1.92e+00
test 50
d = 6.96e-01
test 51
d = 1.26e+00
test 52
d = 3.64e+00
test 53
d = 2.95e+00
test 54
d = 4.94e-02
test 55
d = 3.34e-01
test 56
d = 2.20e-01
test 57
d = 5.38e-01
test 58
d = 3.06e-01
test 59
d = 2.76e-01
test 60
d = 1.78e-01
test 61
d = 1.82e+00
test 62
d = 2.13e+00
test 63
d = 6.44e-02
test 64
d = 6.85e-01

    d = 6.85e-01
test 65
    d = 1.25e+00
test 66
    d = 8.26e-01
test 67
    d = 8.31e-01
test 68
    d = 1.68e+00
test 69
    d = 1.57e+00
test 70
    d = 1.60e-01
test 71
    d = 1.21e-02
test 72
    d = 5.78e-01
test 73
    d = 2.77e+00
test 74
    d = 1.25e+00
test 75
    d = 4.14e-01
test 76
    d = 7.36e-01
test 77
    d = 2.19e+00
test 78
    d = 6.50e-01
test 79
    d = 7.30e-01
test 80
    d = 1.09e+00
test 81
    d = 9.89e-02
test 82
    d = 3.20e-01
test 83
    d = 2.00e+00
test 84
    d = 9.57e-01
test 85
    d = 2.74e-01
test 86
    d = 9.23e-01
test 87
    d = 2.57e+00
test 88
    d = 3.13e-01
test 89
    d = 7.34e-01
test 90
    d = 1.59e+00
test 91
    d = 1.13e+00
test 92
    d = 1.98e+00
test 93
    d = 3.94e+00

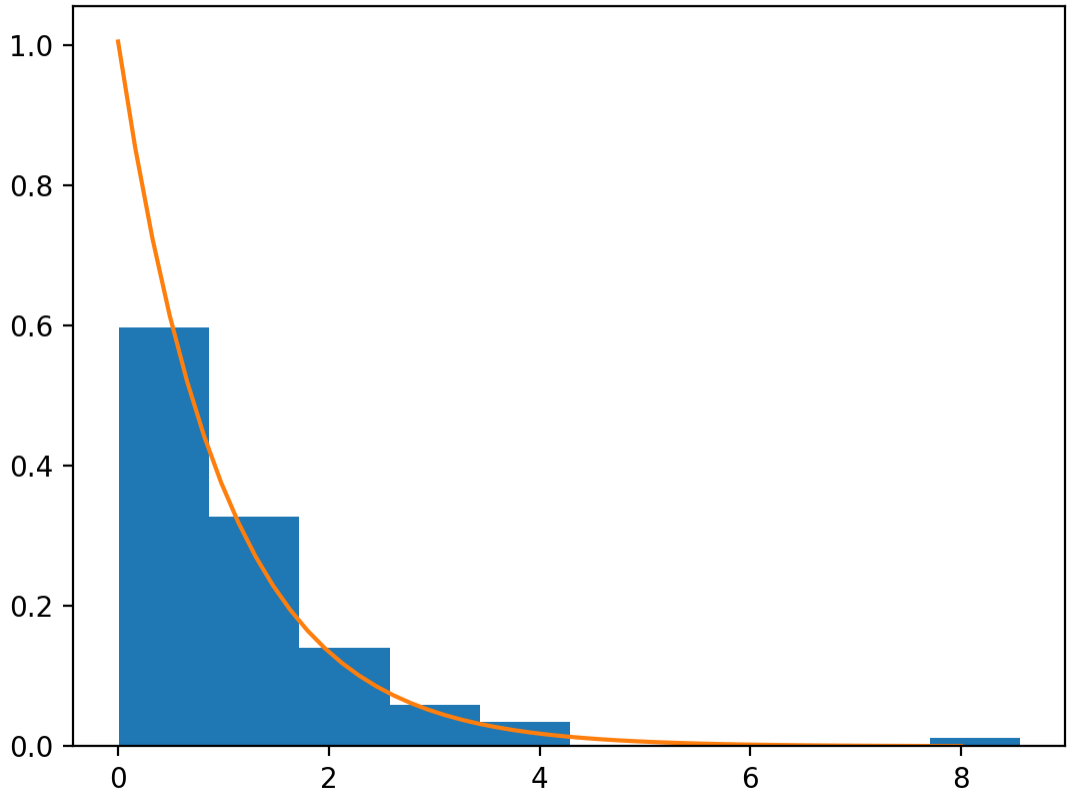
```
    d = 3.94e+00
test 94
    d = 3.65e-01
test 95
    d = 2.70e-01
test 96
    d = 2.22e-01
test 97
    d = 1.75e+00
test 98
    d = 1.32e+00
test 99
    d = 5.56e-01
```

Plot the histogram for these $d^2_{\min}$. What kind of distribution does this look like?

```
# Create the figure.
fig, ax = plt.subplots()
# Use the `ax.hist` to create a histogram.
ax.hist(xs, density=True)
# With a little foreknowledge we know the PDF (see below).
pdf = np.linspace(0, 8)
mu = 1 / 0.995
ax.plot(pdf, [mu * math.exp(-mu * x) for x in pdf])
```

```
[<matplotlib.lines.Line2D at 0x785c89ea3010>]
```

Figure 5



It turns out the distribution is an exponential.

$$\mathrm{PDF}(x) = \mu e^{-\mu x}$$

The CDF is then given by the following.

$$\mathrm{CDF}(x) = 1 - e^{-\mu x}$$

For this particular test, $\mu = 1/0.995$. Implement the expected CDF for this test and calculate the KS p-value.

```
# First, we define the CDF.
def cdf_exp(x, mu=1 / 0.995):
    """
    Return the CDF for an exponential.

    x:  value to determine the CDF for.
    mu: rate parameter.
    """
    return 1 - math.exp(-mu * x)


# Now we perform the test.
d = ks_distance(xs, cdf_exp)
p = ks_pvalue(d, len(xs))
print(f"KS p-value = {p:.2e}")

     KS p-value = 5.07e-01
```

We start to get worried if the p-value is close to 0 or 1. Normally, something less than 0.05 would be considered problematic. However, there are a sufficient number of `diehard` tests that failures at the 0.05 level can still occur for an RNG that is ideal. However, when p-values are less than 1×10^{-5} this is typically indicative of an RNG that has some serious shortcomings.