Elements of Probability and Statistics

Lecture 04: Pseudorandom Numbers Generation, Simulations of Random Experiments

IAI, TCG-CREST August 18, 2023

4.1 Pseudorandom Number Generators

Computers are useful for simulating complex probability problems to estimate the probabilities of different events. However computers are deterministic systems, and hence are only capable of generating specific outputs when given specific inputs. Generating truly random numbers solely from computers is not possible. This motivated the development of procedures which generate a sequence of numbers starting from an initial number called a *seed*. The generated numbers should not follow any discernible pattern and should be statistically as close to random as possible. These procedures are called pseudorandom number generators. Two methods are discussed next for their historical significance and their simplicity. Modern methods are often based on such methods, but are modified so as to be able to generate random numbers that are statistically as close to random as possible.

1. Middle-square method, by John von Neumann [1946]: This method produces a sequence of n-digit random numbers, where n is even. The method starts with an n-digit number Z_0, called the *initial seed*. This number is squared, obtaining Z_0^2 as a $2n$-digit number by left-padding the number with zeros. Finally, the middle n digits from the $2n$-digit Z_0^2 are selected as Z_1, the next random number in the sequence.

The following is an example of the method. Let the initial seed be $Z_0 := 1234$, a 4-digit number. Squaring, we get $Z_0^2 = 1522756$. The number is left-padded with zeros to make the 8-digit number: 01522756. The middle 4 digits are selected to form the next number in the sequence $Z_1 = 5227$. Continuing this method, we get the sequence of numbers $Z_2 = 3215$, $Z_3 = 3362$, $Z_4 = 3030$, $Z_5 = 1809$, $Z_6 = 2724$, $Z_7 = 4201$, $Z_8 = 6484$, $Z_9 = 0422$, $Z_{10} = 1780$, ..., and so on.

The middle-square method is fast, but it can easily get stuck in a loop, generating a sequence of numbers in a period. For a while it used to be studied only for historical reasons.

2. Linear Congruential Generator (LCG), by W. E. Thomson and A. Rotenberg [1958]: This method generates numbers in a sequence by the following equation,

$$Z_{n+1} = (aZ_n + c) \bmod m \ .$$

The parameters are chosen in ways so that the sequence produces numbers that are close to random. Here m is chosen to be a prime (e.g., a Mersenne prime) or a power of a prime (often a power of 2). The parameters c and Z_0 are selected so as to be coprime to m. The parameter a is chosen so that either $a - 1$ is divisible by all prime factors of m, or $a - 1$ is divisible by 4 if m is divisible by 4.

This method is a generalization of the Lehmer generator, by D. H. Lehmer [1951].

One more fundamental method is the **Linear-feedback shift register (LFSR)** by R. C. Tausworthe [1965]. Currently there are several standard pseudorandom number generation procedures, some of which are noted below.

1. Mersenne Twister [1998], based on LFSR. It is currently used by the Python `random` library.

2. Permuted Congruential Generator [2014], based on LCG. It is currently used by the Python `numpy` library to generate random numbers.

3. Recently, methods have been proposed that are based on the middle-square method: the **Middle-Square Weyl Sequence RNG** [2017], and the **Squares RNG** [2020]. Both were proposed by B. Widynski.

While pseudorandom methods try to generate sequence of numbers that are statistically close to random, there are also approaches to generate hardware-based *true* random generators. Some interesting examples of true random generators are the following:

1. The company `Cloudflare` has in their premises a wall of lave lamps; the complex real-time movement of the 'lava' in the lamps are captured by a camera to generate random numbers.

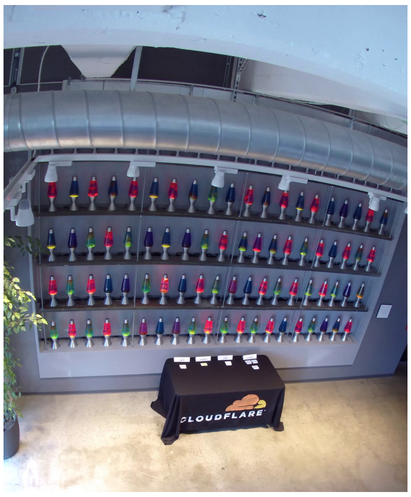


Figure 1: The wall of lava lamps in the `Cloudflare` company premises. Image Source: `https://blog.cloudflare.com/lavarand-in-production-the-nitty-gritty-technical-details/`.

2. `random.org` uses radio receivers placed at different geogrphical locations to pick up atmospheric noise, which is then used to generate random numbers (`https://www.random.org/`).

3. The Linux operating system utilizes information from numerous sources such as random number generators provided in the hardware, system interrupts, CPU execution time jitters, etc., to provide what is called *cryptographically secure pseudorandom numbers*.

4.2 Running simulations in `Python`

(The next section contains introductory discussions on `Python`, `NumPy` and `Matplotlib`, which are used here.)

Prob. 1: Simulate the toss of a fair coin 10^4 times and observe the ratio of outcomes.

```
n_tosses = 10 ** 4
obs = np.random.randint(low=0, high=2, size=(n_tosses,))
print(obs[0:10])

n_heads = (obs == 0).sum()
n_tails = (obs == 1).sum()

p_heads = n_heads / (n_heads + n_tails)
p_tails = n_tails / (n_heads + n_tails)

print(p_heads, p_tails)

> [0 1 1 0 1 0 1 1 1 0]
0.4923 0.5077
```

Estimating for lower (100) or higher (10^8) number of tosses:

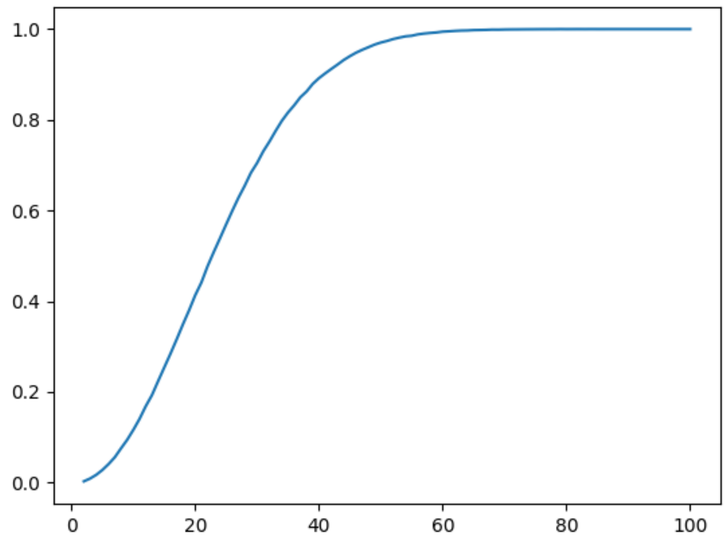
```
# For n_tosses = 100
> 0.53 0.47

# For n_tosses = 10 ** 8
> 0.50007616 0.49992384
```

Prob. 2: Simulate the birthday problem for $k = 23$ people.

```
k = 23
n_exps = 10 ** 5
obs = np.random.randint(
    low=0, high=365, size=(n_exps, k)
)
success = 0
for i in range(obs.shape[0]):
    _, counts = np.unique(obs[i], return_counts=True)
    success += (counts > 1).sum() > 0
prob = success / obs.shape[0]
print(prob)

> 0.50761
```

Extend the above approach to simulate the birthday problem for the range of $k = 2$ to $k = 100$, and plot the results.

```
n_exps = 10 ** 5
results = np.zeros((100-2+1))
for k in range(2, 100+1):
    obs = np.random.randint(
        low=0, high=365, size=(n_exps, k)
    )
    success = 0
    for i in range(obs.shape[0]):
        _, counts = np.unique(obs[i], return_counts=True)
        success += (counts > 1).sum() > 0
    results[k-2] = success / obs.shape[0]
plt.plot(np.arange(2,100+1), results)
plt.show()
```



4.3 Selected elementary Python commands

We begin by discussing some commonly used and important to know concepts around Python, after which we discuss essential commands in NumPy and Matplotlib.

1. The `print()` command displays any arguments passed to it.

```
# Comments are written with a preceding #
''' Three single quotation marks
allow multi-line comments'''

print('Hello')

> Hello
```

Multiple arguments can be passed to `print()` by separating them by commas.

2. Five commonly used built-in data types in Python are: (i) integer (`int`), (ii) numeric `float`, (iii) string (`str`), (iv) boolean `bool`, and (v) the 'none' type (`NoneType`). Data of any of these types can be stored in a variable following the format `variable_name = data`.

```
print(2, type(2))

> 2 int
```

```
x = 3.14
print(x, type(x))

> 3, float
```

```
x = 'Hello'
print(x, type(x))

> Hello, str
```

```
# The boolean type has only 2 possible values
x = True
y = False
print(x, type(x))
print(y, type(y))

> True bool
False bool
```

```
# The NoneType type is used to indicate an absence of data
x = None
print(x, type(x))

> None NoneType
```

3. The following are examples of some simple arithmetic operations. Some points to remember: (i) For the numeric type, addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**) operations are defined. Only for integers, division quotient (//) and division remainder (+) operations are defined. If any arithmetic operations involve boolean data, then True values are implicitly converted to 1, and False values are implicitly converted to 0.

```
a = 3.4
b = 2
print(a + b)

> 5.4
```

```
a = 3
print(a ** 3)

> 27
```

```
a = 10
b = 3
print(a//b, a%b)

> 3 1
```

```
a = 10
b = True
print(a + b)

> 11
```

4. *Relational* operators (<,<=,>,>=,==,!=) can be used to compare the values of numeric, integer or boolean data types. *Logical* operators (and, or, not) can be used on them as well.

```
# Example of a use of a relational operator
a = 3.14
b = 2
print(a <= b)

> False
```

```
# Example of a use of a logical operator
# numeric / integer 0 is converted to False
# non-zero values are converted to True
a = 3.14
b = False
print(a or b)

> True
```

5. The if-else commands are called *conditional* statements, as they allow the conditional execution of commands, i.e., based on whether a condition is true, different sets of commands can be executed.

```
a = 3
if a % 2 == 0:
    print(a, 'is even')
else:
    print(a, 'is odd')

> a is even
```

```
a = 3
b = 4.4
c = 3.5
print('The maximum value:')
if a > b:
    if a > c:
        print(a)
    else:
        print(c)
else:
    if b > c:
        print(b)
    else:
        print(c)

> The maximum value:
4.4
```

6. The `for` loop allows a sequence of numbers to be generated.

```
for i in range(5): # end value of sequence
    print(i, ' ')

for i in range(4,9): # start and end value of sequence
    print(i, ' ')

for i in range(0,10,2): # start, end and increment values
    print(i, ' ')

> 0 1 2 3 4
4 5 6 7 8
0 2 4 6 8
```

7. Functions in `Python` help to reuse the same code for different possible inputs.

```
def function_name(a ,b):
    return a + b

print(function_name(2,3))
print(function_name(10,20))

> 5
30
```

4.4 Selected elementary NumPy commands

1. The basic NumPy data type we use is called an array, which can store several data values under the same variable name. Therefore vectors can be represented using 1-dimensional arrays, matrices using 2-dimensional arrays, 3-dimensional tensors by 3-dimensional arrays, and so on.

```
import numpy as np # Python libraries need to be imported

arr1 = np.array([1,2,3,4])
arr2 = np.array([[1,2,3,4], [5,6,7,8]])

print(arr1)
print(arr2)

> [1 2 3 4]
[[1 2 3 4]
 [5 6 7 8]]
```

2. The shape and dimensions of NumPy arrays can be easily retrieved. NumPy arrays can be reshaped to any desired configuration as well.

```
print(arr1.ndim, arr1.shape)
print(arr2.ndim, arr2.shape)

> 1 (4,)
2 (2,4)
```

```
print(np.reshape(a1, (2,2)))

> [[1 2]
[3 4]]
```

3. NumPy arrays filled with random values can be created by specifying their shapes.

```
r1 = np.random.rand(2,2) # Uniform random values in [0,1]
print(r1)

# Samples from a normal distribution
r2 = np.random.normal(loc=0, scale=1, size=(2,3))
print(r2)

# Random integers between low and high
r3 = np.random.randint(low=0, high=10, size=(5,))
print(r3)
```

```
> [[0.84781    0.20967248]
 [0.3954045  0.18252528]]
[[-1.29531559 -1.20132127 -2.30156091]
 [-0.04031745  0.14404613  0.97561405]]
[2 6 0 5 9]
```

4. Some often used `NumPy` array creation methods are shown below.

```
z1 = np.zeros((2,2)) # Creates a zero-filled array
print(z1)

I = np.eye(3) # Creates an identity matrix
print(I)

ar1 = np.arange(1,10)
print(ar1)

> [[0. 0.]
 [0. 0.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[1 2 3 4 5 6 7 8 9]
```

5. One can easily obtain various statistics from arrays.

```
print(np.amax(ar1), np.argmax(ar1))
print(np.amin(ar1), np.argmin(ar1))
print(np.mean(ar1), np.std(ar1))

> 9 8
1 0
5.0 2.581988897471611
```

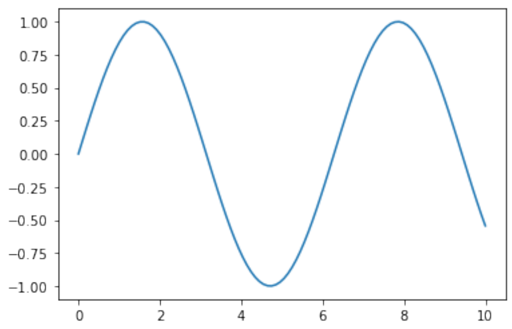
```
# For 2-dim and higher dim arrays,
# the operations can be performed along specifc axes

r1 = np.random.rand(2,2)
print(r1)
print(np.amax(r1))
print(np.amax(r1, axis=0))
print(np.amax(r1, axis=1))

> [[0.267623   0.59266045]
 [0.18233888 0.22373945]]
0.5926604476850343
[0.267623   0.59266045]
[0.59266045 0.22373945]
```

6. *Boolean* arrays are extremely useful for selecting specific values from arrays.

```
r1 = np.random.rand(4)
print(r1)

b1 = r1 > 0.5 # Boolean array
print(b1)

print(r1[b1])
# selects array elements
# that satisfy the previous condition

> [0.97739676 0.44292792 0.54389469 0.75951689]
[ True False  True  True]
array([0.97739676, 0.54389469, 0.75951689])
```

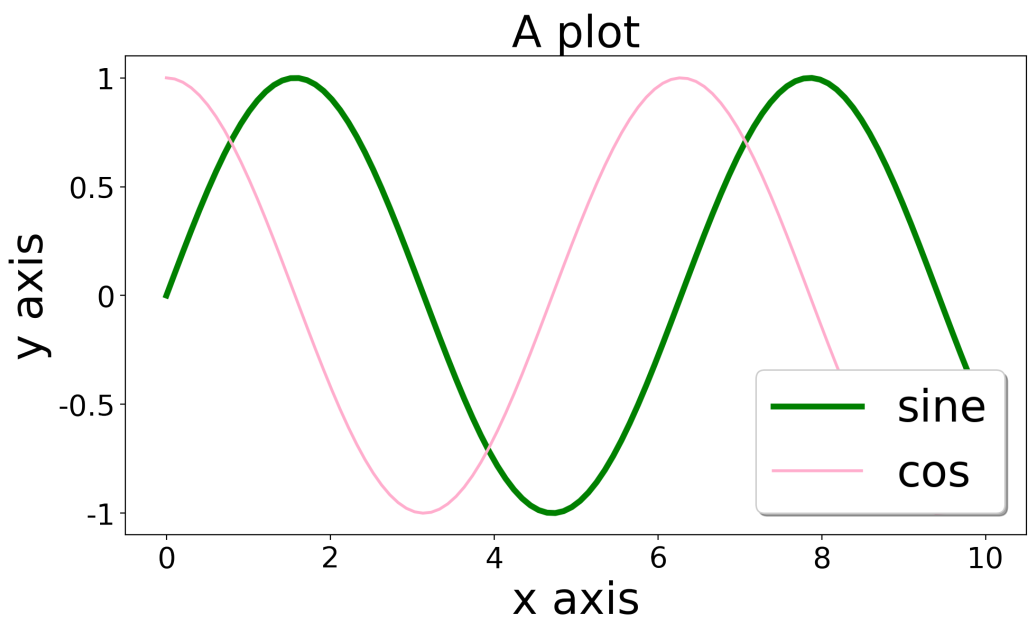
4.5 An introductory usage of `Matplotlib` for plotting

```
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

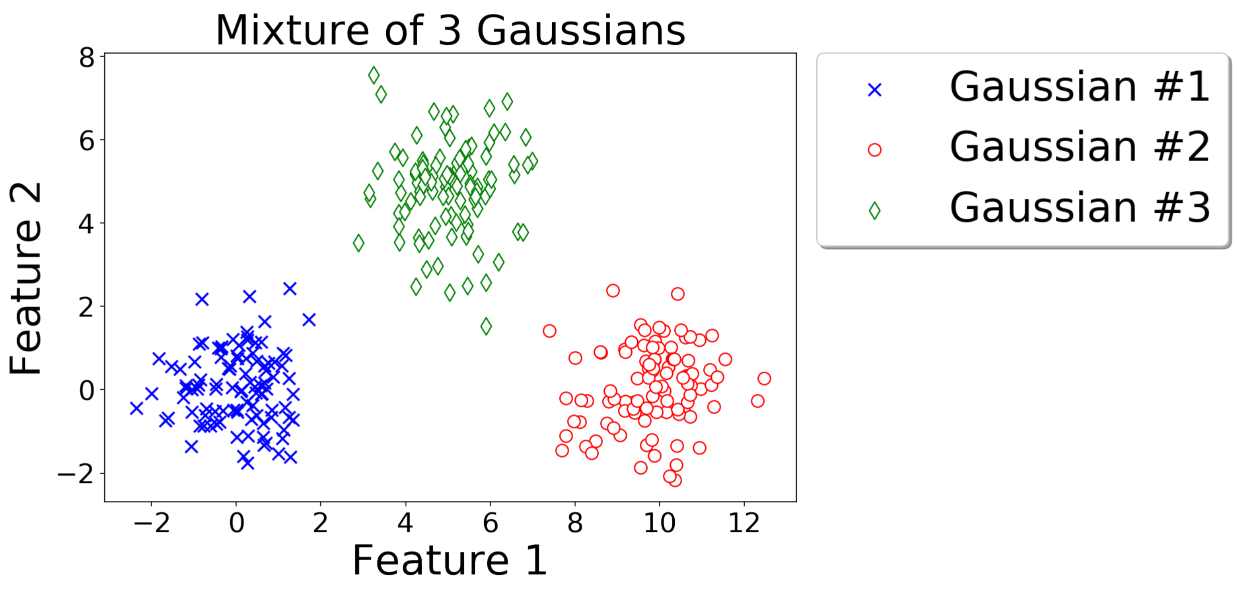


```
# Customizing a plot

import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.figure(figsize=(11,6), dpi=200)
[line_handle] = plt.plot(
x, y, c='g', linewidth=4, label='sine'
)
y2 = np.cos(x)
[line_handle2] = plt.plot(
x, y2, c='#ffadcd', linewidth=2, label='cos'
)
```

```
plt.legend(
handles=[line_handle, line_handle2], fontsize=30,
loc='lower right', shadow=True
)
plt.xlabel('x axis', fontsize=30)
plt.ylabel('y axis', fontsize=30)
plt.title('A plot', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(
[-1, -0.5, 0, 0.5, 1], [-1, -0.5, 0, 0.5, 1],
fontsize=20
)
plt.show()
```



```
# A scatter plot with a customized legend
import matplotlib.pyplot as plt

x1 = np.random.normal(loc=[0,0], scale=1, size=(100,2))
x2 = np.random.normal(loc=[10,0], scale=1, size=(100,2))
x3 = np.random.normal(loc=[5,5], scale=1, size=(100,2))
plt.figure(figsize=(9,6), dpi=150)
g1 = plt.scatter(
    x1[:,0], x1[:,1], marker='x', s=80, c='b',
    label='Gaussian #1'
)
g2 = plt.scatter(
    x2[:,0], x2[:,1], marker='o', s=80, c='w',
    edgecolor='r', label='Gaussian #2'
)
g3 = plt.scatter(
    x3[:,0], x3[:,1], marker='d', s=80, c='w',
    edgecolor='g', label='Gaussian #3'
)
plt.legend(
    handles=[g1, g2, g3], fontsize=30, shadow=True,
    bbox_to_anchor=(1.03, 1), borderaxespad=0
)
plt.xlabel('Feature 1', fontsize=30)
plt.ylabel('Feature 2', fontsize=30)
plt.title('Mixture of 3 Gaussians', fontsize=30)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.show()
```


Mixture of 3 Gaussians
Feature 1
Feature 2
Gaussian #1
Gaussian #2
Gaussian #3