Scientific Method Implementation of Python Random Number Generator

Shaun Pritchard - PHY4060 Rasmussen College 11/1/2020

Abstract:

Python has built in random method in core library. This experiment will implement a random number generator from mathematical formulations and format using **Linear congruential generator** to code the random number generator in python based on the scientific method. I will then show how to optimize this random number generator asymptomatically.

Scientific Method

- Create A Model Describing Natural World
- Use Model to Develop Hypotheses
- Run Experiments to Validate Hypotheses
- Gather Results
- Refine Model and Repeat

Perspective:

Algorithm designer who does not experiment gets lost in abstraction
Software developer who ignores cost risks catastrophic consequences
Essentially as a Computer Scientist we follow the same pragmatic implications of the scientific method as other scientists to solve problems programmatically through abstraction.

From my experiment I will be developing a random number generator from scratch in python.

Problem: write a program to generate random numbers in python using linear recurrence

Model: classical probability and statistics

Hypothesis: frequency values should be uniform

Initial experiment:
• generate random numbers with linear congruential generator from scratch

Better experiment:
• Check for uniform frequencies and distribution between 0 and 1 using python program libraries.

First, we will need mathematical pseudocode to describe the algorithm we will use a method called a linear congruential generator, which was popular back in the old days.

The output of our program will take the form of a simple Uniform Distribution. Thereafter, we will use statistical theorems and transformations to generate random variables, corresponding to other distributions, based on this random generator.

Linear congruential generator

$$X_i = m.X_{i-1} + p$$

where m and p are two suitably chosen integers

The uniform random variates are obtained after it scales,

$$U_i = \frac{X_i}{p}$$

We will also need a seed number for starting the generation process. So, here is a simple Python function

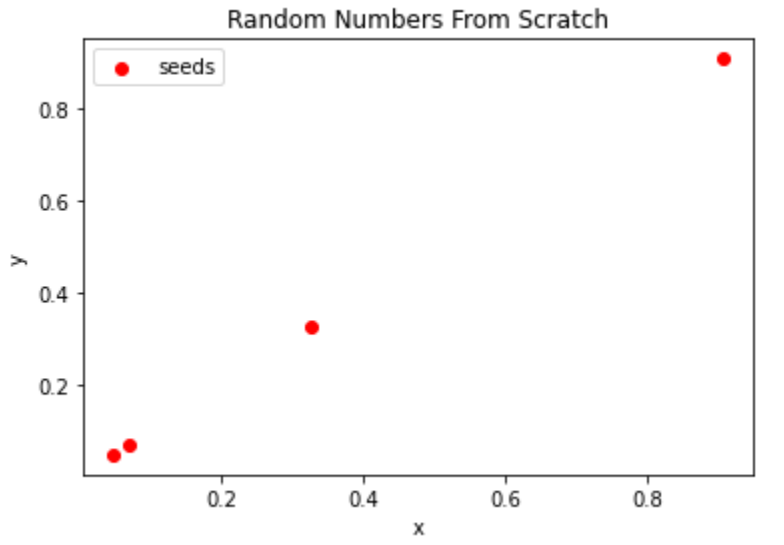
Pseudocode / Code Implementation

```
# Psuedocode for Random Number generator
# Only Generates 1 random number based on SEED value
def random_gen(m, mod, seed,size):
 U = np.zeros(size)
 x = (seed*m+1)%mod
 U[0] = x/mod
 for i in range(1,size):
   x=(x*m+1)%mod
   U[i] = x/mod
 return U
```

With this code we have a complete experiment, but when I run the code it only generates random numbers based on the seed value. In order to apply this as a working function we would need to constantly change the seed value.

This code generates and plots 4 different variations of seed values onto a scatter plot in python.

```
[0.04859086] [0.90593138] [0.07040717] [0.32586039]
```

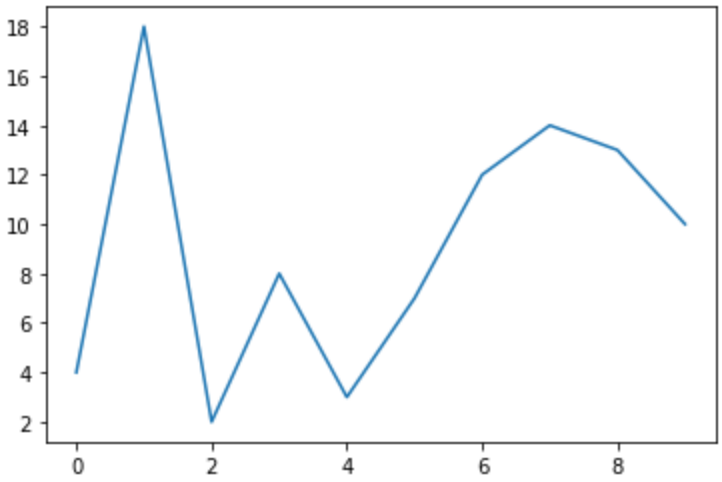


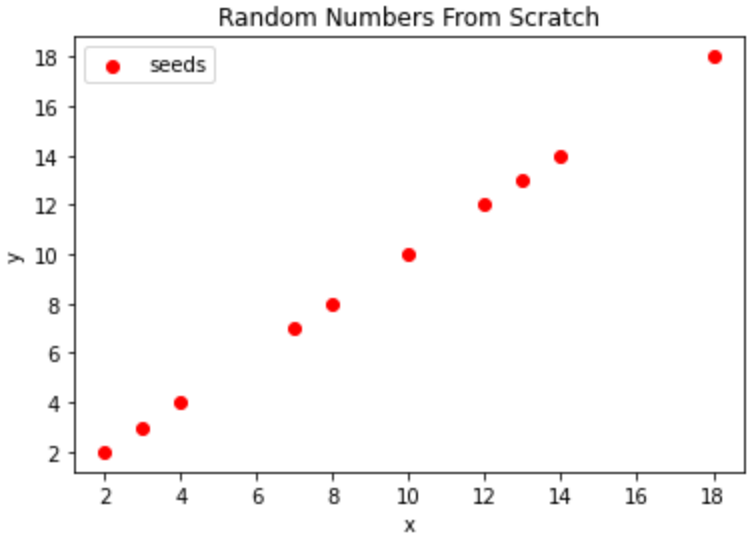
Although we are generating random numbers according to the **Linear congruential generator** pseudo code. We still have to change the seed values on at a time. We could set up another method to increment the seed value in a loop but python has

already done this for us. So for the 2nd experiment I will generate random numbers with the random method library provided by python and implement a random number generator via code that models the same principles as the code above in fewer lines of code.

```
#Simulate psudocode with python libary
# select a random sample without replacement
from random import seed
from random import sample
# seed random number generator
seed(1)
# prepare a sequence
seeds = [i for i in range(20)]
print(sequence)
# select a subset without replacement
subset = sample(seeds, 10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[4, 18, 2, 8, 3, 7, 12, 14, 13, 10]
```





Results:

With this code we can have randomized incrementation of the seed values into a list using linear recurrence. These next images show the iterative data and a line graph. Followed by a scatter plot of the randomized seed values.

We can see that both experiments meet the requirements of linear recurrence due to their linear slope patterns.

We can then continue to optimize this algorithm using techniques such as asymptotic Big-O notation and time complexity. Doing so we can get a number generator that is totally randomized implementing binomial distribution.

Here is the full working code I developed on Google Collab:

https://colab.research.google.com/drive/1A7mdZrJY2VFIaJwVVCHxVDV-G08mrqJd?usp=sharing