Random Number Generation on a Decimal Computer

Ronald Mak

Department of Computer Engineering
Department of Computer Science
Department of Applied Data Science
San José State University
ron.mak@sjsu.edu
http://www.cs.sjsu.edu/~mak

November 15, 2019
[Updated November 20, 2019]

I propose a rapid algorithm to generate uniformly distributed random numbers that is suitable for a decimal computer, specifically the IBM 1401 computer. I employ simple statistical methods to demonstrate the viability of this algorithm. Then I show that it is easy to convert the uniformly distributed random numbers to normally distributed random numbers., i.e., numbers that fit a bell curve.

1. Introduction

The IBM 1401 was an early 1960s era transistor-based small-business computer. It supported a maximum of 16 KB of core memory, and it had an 87 KHz machine cycle – extremely miniscule and painfully slow compared to today's computers. It had a decimal-based architecture and could store a single digit 0 – 9 as an 8-bit character. Each digit took 6 bits of the character, plus there was a parity bit and a "word mark" bit. Numbers were arbitrarily long; a number was addressed by its rightmost digit and the leftmost digit had its word mark bit set. Today, we would call each character a "byte", each number a "string decimal", and say that the IBM 1401 performed "string decimal arithmetic".

The IBM 1401 had basic add and move machine instructions, but no bitwise shift or "and" or "or" instructions. Multiplication and division were slow operations, and floating-point operations were done by software routines. Therefore, an efficient algorithm designed to run on that machine should rely primarily on adds and moves. Arithmetic operations on numbers and moves of numbers from one memory location to another were delimited by word marks. There were machine instructions to set and clear a word mark as needed at any particular memory address.

2. An algorithm to generate uniformly distributed random values

My random number algorithm relies on rotating 7-digit numbers to the right by 3 digits and by 5 digits. Note that 3, 5, and 7 are prime numbers.

To rotate a 7-digit number to the right by 3 digits on the IBM 1401, store the number in memory at the left end of a 10-character buffer. For example, to rotate the number `1234567`:

1234567xxx

Move the number right 3 characters to the right:

1231234567

Move the rightmost 3 characters of the buffer to the beginning of the buffer:

5671234567

Then the leftmost 7 characters of the buffer contain the rotated number **5671234**. Therefore, each rotation requires two moves.

Rotating a 7-digit number to the right is similar, except that you start with the number at the left end of a 12-character buffer. For example, rotating the number **1234567** yields **3456712**.

Steps of the algorithm:

1. Start with two nonzero 7-digit seed values, call them **r1** and **r2**.
2. Rotate **r1** to the right by 3 characters and rotate **r2** to the right by 5 characters.
3. Add the two rotated numbers to product a 7-digit **sum** (drop any overflowed digits at the left).
4. Digits at the left end of **sum** constitute a random value. If, for example, you want 2-digit random numbers, take the leftmost two digits.
5. Set **r1** to the value of **r2**, and set **r2** to the value of **sum**.
6. Go back to step 2 and iterate as many times as desired. Each iteration produces one random value.

To the right is an example of the first three iterations to generate 2-digit random values.

```
r1 = 1234567  # seed
r2 = 8901234  # seed

ITERATION #1

r1  = 5671234  # rotated right 3
r2  = 0123489  # rotated right 5
sum = 5794723  # sum = r1 + r2

*** random value = 57

r1  = 0123489  # r1 = r2
r2  = 5794723  # r2 = sum

ITERATION #2

r1  = 4890123  # rotated right 3
r2  = 9472357  # rotated right 5
sum = 4362480  # sum = r1 + r2

*** random value = 43

r1  = 9472357  # r1 = r2
r2  = 4362480  # r2 = sum

ITERATION #3

r1  = 3579472  # rotated right 3
r2  = 6248043  # rotated right 5
sum = 9827515  # sum = r1 + r2

*** random value = 98

r1  = 6248043  # r1 = r2
r2  = 9827515  # r2 = sum
```

3. Python implementation

I implemented the algorithm in Python and generated a million 2-digit random values. I used Python's standard integer datatype, so I did the rotations with integer multiplication, division, and modulo operations with powers of 10. As mentioned above, on the IBM 1401, only moves are required to do the rotations.

```
rvs = []  # list to contain the random values, initially empty

r1  = 1234_567  # seed
r2  = 89_01234  # seed

# Loop to generate a million random values.
for i in range(1_000_000):
    r1 = 10000*(r1%1000) + r1//1000    # rotate right 3 digits
    r2 = 100*(r2%100000) + r2//100000  # rotate right 5 digits

    sum = (r1 + r2)%10000000   # add, and keep the sum to 7 digits
    rvs.append(sum//100000)    # append the leftmost 2-digit number to list rvs

    r1 = r2
    r2 = sum
```

```
print(f'The first 20 random values:')
print(f'{rvs[:20]}')
```

```
The first 20 random values:
[57, 43, 98, 31, 47, 30, 81, 92, 80, 25, 65, 4, 31, 30, 48, 60, 40, 4, 16, 5]
```

4. Basic statistical confirmation

We need to confirm that we did indeed generate uniformly distributed random values. First, some basic statistics:

```
import statistics
import numpy as np

print(f'           mean = {statistics.mean(rvs):.2f}')
print(f' first quartile = {np.percentile(rvs, 25)}')
print(f'second quartile = {np.percentile(rvs, 50)}')
print(f' third quartile = {np.percentile(rvs, 75)}')
```

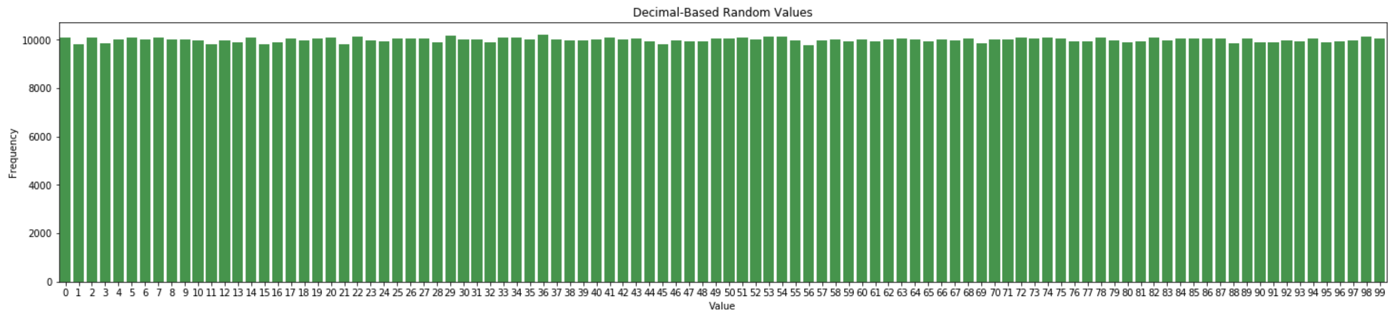
```
           mean = 49.50
 first quartile = 25.0
second quartile = 50.0
 third quartile = 74.0
```

It's indeed a good sign that the mean (arithmetic average) is very close to 50, the first and second quartiles hit their marks, and the third quartile is off by only 1.

Here’s a frequency bar chart of all the values. They do appear to be uniformly distributed!

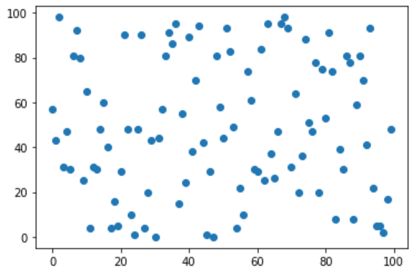
```
import matplotlib.pyplot as plt
import seaborn as sns

def show_chart(title, x_label, y_label, x_values, y_values, size=(10, 5)):
    """
    Display a bar chart.
    @param title the chart title.
    @param x_label the label for the x axis
    @param y_label the label for the y axis
    @param x_values the x values to plot
    @param y_values the y values to plot
    @param size the size (width, height) of the chart
    """
    figure = plt.figure(figsize=size)

    axes = sns.barplot(x_values, y_values, color='green', alpha=0.75)
    axes.set_title(title)
    axes.set(xlabel=x_label, ylabel=y_label)
```

```
freq = [0]*100
for v in rvs:
    freq[v] += 1

show_chart('Decimal-Based Random Values', 'Value', 'Frequency',
           list(range(100)), freq, size=(25, 5))
```

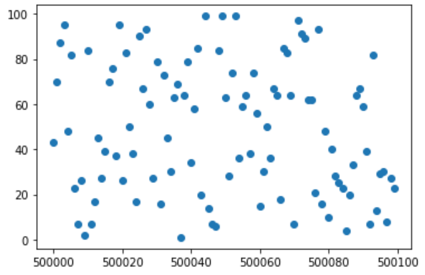


Even though the final result appears valid, let’s make sure that the values were generated in random order not in clumps or in a pattern. Here are scatter plots that show the generation order of the first 100 generated values, 100 values near the middle, and 100 values near the end:

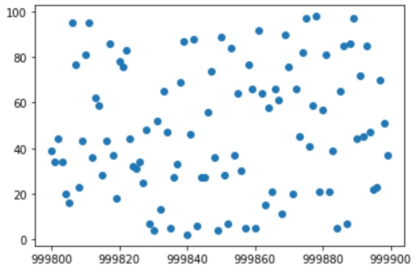
```
xs = [list(range(100))]
ys = rvs[0:100]

plt.scatter(xs, ys)
plt.show()
```



```
xs = list(range(500_000, 500_100))
ys = rvs[500_000:500_100]

plt.scatter(xs, ys)
plt.show()
```



```
xs = list(range(999_800, 999_900))
ys = rvs[999_000:999_100]

plt.scatter(xs, ys)
plt.show()
```



The values in these three samples do appear to have been generated in random order.

5. Generate normally distributed random values

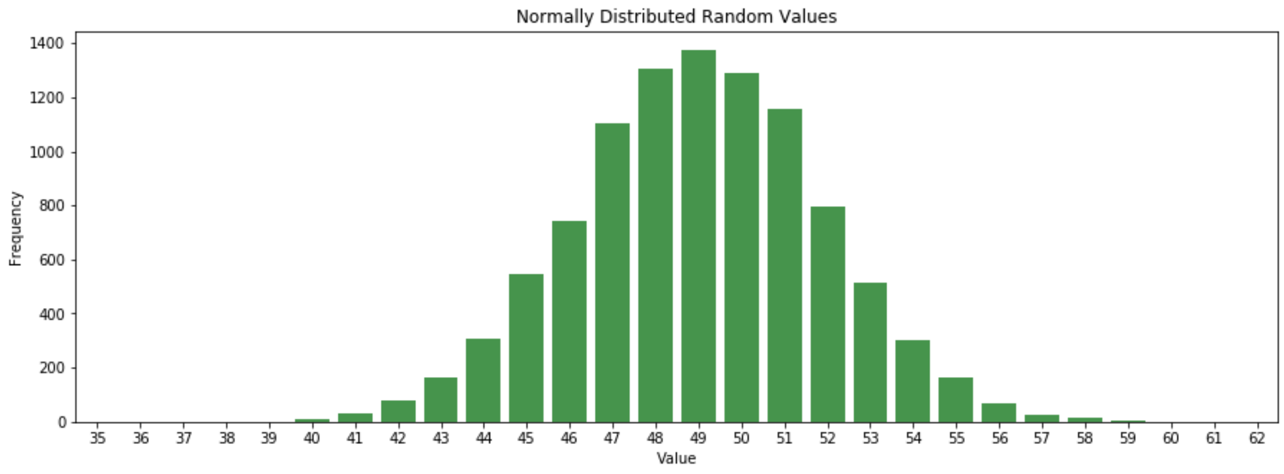
It is very simple and efficient to extend the above algorithm to also generate normally distributed random values, values that fit a bell curve.

As the uniformly distributed random values are being generated, add them together in groups of one hundred – the first through 100th values, the 101st through 200th values, etc. Keep all the digits of each sum. Compute the average of each group by dividing its sum by 100. Of course, on the IBM 1401, that "division" is done by not including the last two digits of the sum.

According to the Central Limit Theorem of statistics, these group averages are normally distributed random values.

In the Python implementation, I computed group averages separately at the end using the uniformly distributed random values stored in list **`rvs`**:

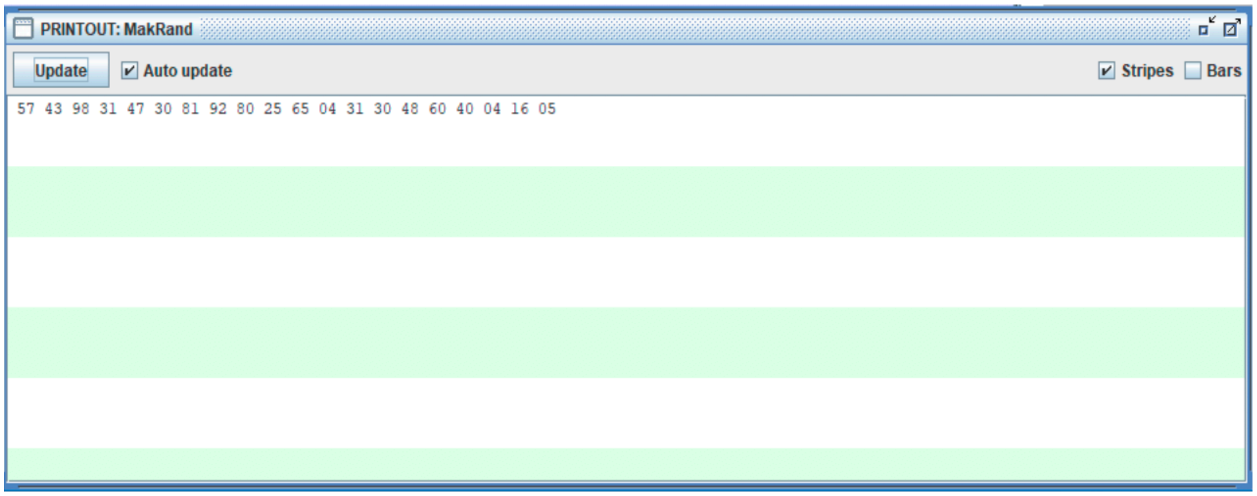
```
import statistics

stdev  = int(statistics.stdev(rvs))
offset = int(statistics.mean(rvs)) - stdev//2;
limit  = offset + stdev

freq = [0]*stdev

# Loop to compute the average of each group of 100 random values.
for lo in list(range(0, 1_000_000, 100)):
    hi = lo + 100
    mean = int(statistics.mean(rvs[lo:hi]))

    if (mean >= offset) and (mean < limit):
        freq[mean - offset] += 1

show_chart('Normally Distributed Random Values', 'Value', 'Frequency',
           list(range(offset, limit)), freq, size=(15, 5))
```



6. IBM 1401 Autocoder implementation

[Added November 20, 2019]

Autocoder program by Michael Albaugh. Assembled and run under ROPE 1401 simulation.



```
57 43 98 31 47 30 81 92 80 25 65 04 31 30 48 60 40 04 16 05
```

```
          job  1401 PRNG FROM RON MAK
* Assemble with command
* autocoder -l MakRand.lis -o MakRand.obj -bV ctload.ac
          CTL  6111
* INDEX REGISTERS
          ORG  87
X1        DCW  @000@  POSITION IN LP BUFFER
          ORG  92
X2        DCW  @000@
          ORG  97
X3        DCW  @000@
          ORG  333
* WORKING STORAGE
ACC       DCW  @0000000@
SUM       DCW  @0000000@
*
* USES TWO 7-DIGIT SEEDS
R1        DCW  @1234567@
R2        DCW  @8901234@
*
* TEST CODE
          ORG  400
START     MCW  @000@,X1
LLOOP     B    MAKRND
          MCW  SUM-5,203&X1
          SBR  X1,3&X1
          C    @060@,X1
          BU   LLOOP
          W
          CS   332
          CS
          H
* PRNG PROPER
          ORG  500
MAKRND    SBR  MRRTN&3   SAVE RETURN ADDRESS
* ROTATE R1 RIGHT BY THREE CHARACTERS.
          MCW  R1-3,ACC     MOVE NUM MSDS TO ACC LSDS
          MCW  R1           MOVE NUM LSDS TO ACC MSDS (CHAIN)
          MCW  ACC,R1
* ROTATE R2 RIGHT BY FIVE CHARACTERS
          MCW  R2-5,ACC     MOVE NUM MSDS TO ACC LSDS
          MCW  R2           MOVE NUM LSDS ACC MSDS (CHAIN)
          MCW  ACC,R2
          MCW  R2,SUM
          A    R1,SUM
          MZ   @0@,SUM-6    IGNORE OVERFLOW
          MCW  R2,R1
          MCW  SUM,R2
MRRTN     B    MRRTN     RETURN TO CALLER
          END  START
```