Pseudo Random Number Generators in Python, R, and C++ With Applications to Generating Uniform Variates

Thomas Walker

September 2022

1 Introduction

In this notebook, I will outline the main method used to generate pseudo-random numbers in programming languages. Leading on from this we will then explore how the programming languages use these numbers to generate uniform variates. As a consequence, we will be able to construct methods such that we can align pseudo-random number generation across the programming languages. This research only concerns itself with Python, R , and C++. For python we will consider the random functions within the NumPy library, for R we will only concern ourselves with the standard random functions it employs, and for C++ we will look into both its standard random functions and those facilitated by the boost library.

2 The Mersenne Twister

This is a very popular pseudo-random number generator (PRNG) used by programming languages. Although many offer the option to use other PRNG, often the default is an instance of The Mersenne Twister. The reason for its popularity is its large period (which we will discuss in due course, as it's dependent on the particular instance of The Mersenne Twister algorithm we implement) and the strong result it provides for the k-distribution test.

2.1 k-distribution test

This is a test that attempts to provide an indication of how 'random' a PRNG is. The general outline of the test is as follows:

- We have a pseudo-random sequence, (x_i), of w-bit integers of period P.
- We consider the vectors formed by the leading v bits of k of these w-bit integers
  - If we denote the leading v bits of x as $trunc_v(x)$, then the vectors we are considering are

$$(trunc_v(x_i), trunc_v(x_{i+1}), \ldots, trunc_v(x_{i+k-1}), \quad \text{for } 0 \leq i \leq P$$

- There are 2^{kv} possible vectors, so we say that the random sequence (x_i) is k-distributed to v-bit accuracy if each combination of bits occurs the same number of times within a period, except for the all-zero combination that occurs once less often.

[1] provides an illustrative geometric intuition for the test.

2.2 Terminology

To understand how the algorithm works we should clear up some terminology. We will be dealing with *words* of size w, that is we are considering integers between 0 and $2^w - 1$. For a given word we can associate with it a *word vector*, this is simply a w-dimensional row vector over $\mathbb{F}_2$. We associate the two by the binary representation of the word, the least significant bit appears to the right. To denote these word vectors we will use bold vector notation, $\mathbf{x}$.

2.3 The Algorithm

The Mersenne Twister algorithm employs a linear recurrence to generate 32-bit integer words, and then to improve its performance on the k-distribution test it employs what is called a *tempering* stage. Essentially, this involves transforming the outputted 32-bit integer word by right multiplying its associated word vector by a $w \times w$ invertible matrix. The specific linear recurrence employed is the following:

$$\mathbf{x}_{k+n} := \mathbf{x}_{k+m} \oplus (\mathbf{x}_k^u | x_{k+1}^l)\mathbf{A}, \quad (k = 0, 1, \dots)$$

Where:

- n: The degree of the recurrence (Integer)
- r (hidden in the definition of $\mathbf{x}_k^u$): $0 \leq r \leq w - 1$ (Integer)
- m: $1 \leq m \leq n$ (Integer)
- $\mathbf{A}$: An element of $\mathbf{M}_{w \times w}(\mathbb{F}_2)$ (Matrix)

With $\mathbf{x}_k^u$ denoting the upper $u(= w - r)$ bits of $\mathbf{x}_k$ and $\mathbf{x}_k^l$ denoting the lower r bits of $\mathbf{x}_k$ and $(\mathbf{x}_k^u | \mathbf{x}_{k+1}^l)$ simply denoting their concatenation.

To make this linear recurrence easy to implement, the matrix $\mathbf{A}$ is often taken to have the simpler form

$$\begin{pmatrix} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ a_{w-1} & a_{w-2} & \dots & \dots & a_0 \end{pmatrix}$$

This allows the calculation $\mathbf{xA}$ to be reduced to

$$\mathbf{xA} = \begin{cases} \text{shiftright}(\mathbf{x}) & \text{if } x_0 = 0 \\ \text{shiftright}(\mathbf{x}) & \text{if } x_0 = 1 \end{cases}$$

As a consequence of this, when implementing a specific instance of this algorithm the matrix $\mathbf{A}$ is simply given as a word of size w (as $a_i \in \mathbb{F}_2$, the concatenation of the a_i s forms a binary value).

We then move onto the tempering stage of the algorithm, which as briefly described above involves a transformation. More specifically, we have a sequence of transformations, $\mathbf{x} \mapsto \mathbf{z}$. The

transformations are carried out in the following way:

$$\begin{aligned} \mathbf{y} &:= \mathbf{x} \oplus (\mathbf{x} >> u) \\ \mathbf{y} &:= \mathbf{y} \oplus ((\mathbf{y} << s) \text{ AND } \mathbf{b}) \\ \mathbf{y} &:= \mathbf{y} \oplus ((\mathbf{x} << t) \text{ AND } \mathbf{c}) \\ \mathbf{z} &:= \mathbf{y} \oplus (\mathbf{x} >> l) \end{aligned}$$

Where u, s, t, and l are integers, $<< k$ denotes a left bit shift by k and $>> k$ denotes a right bit shift by k. **b** and **c** are called bitmasks.

2.4 MT19937

Python, R, and C++ each employ the MT19937 instance of the Mersenne Twister algorithm, which is simply a version of the algorithm where the parameters take on specific values. The choice of these parameters is important as it affects the size of the period for the sequence of random numbers generated, as well as their distribution amongst the bits.
MT19937 period parameters:

- $w = 32$
- $n = 624$
- $r = 31$
- $m = 397$
- $a =$ 0x9908b0df

MT19937 tempering parameters:

- $u = 11$
- $s = 7$
- $t = 15$
- $l = 18$
- $b =$ 0x9d2c5680
- $c =$ 0xefc60000

With these specific set of parameters, the algorithm has a period of $2^{19937} - 1$ and is 623-distributed amongst the full 32-bits. Along with the property of being easy to implement sufficiently in code, the MT19937 algorithm is a popular option as a PRNG.

2.5 Implementation

We use the following method to implement this algorithm:

1. We set the generator with an initial list of 624 words, which we are going to call the key.
2. Using this initial key we update the 624 words using the linear recurrence to generate a new key of 624 words.
3. We sample this key at its first value and temper this word to generate an output.

4. After this we sample the key at its next element.
5. We temper this value to generate our next integer output.
6. Repeating this we will eventually reach the end of our key, at which point we use the key and the linear recurrence to generate a new key and start sampling again from the beginning.
7. This is repeated to keep generating a sequence of random integers.

Throughout this report, the 'state' will refer to the tuple (position, key), where position gives the index at which we are sampling the key at.

[]:
```
import numpy as np
import numpy.random as rng
from rpy2 import robjects
import matplotlib.pyplot as plt
```

[]:
```
# The following code block defines a set of functions so that we can work with␣
 ↪signed and unsigned integers
# R works with signed integers whereas Python and C++ work with unsigned integers

def int32(num):
    while num>2**31 or num<-2**31:
        if num > 2 ** 31:
            num=-(2 ** 32 - num)
        elif num < -2 ** 31:
            num = 2 ** 32 + num
    return num

def int32_rev(num):
    if num<0:
        return 2**32+num
    return num

def state32(state):
    pos, key = state
    return pos, [int32(x) for x in key]
```

[]:
```
n = 624
m = 397
matrix_a = 0x9908b0df
upper_mask = 0x80000000
lower_mask = 0x7fffffff

def mt19937(key):
    '''
    Here we are using the linear recurrence to generate a new key, based on the␣
 ↪previous key (or any set of 624 words)
    '''
```

```
    for i in range(n - m):
        y = (key[i] & upper_mask) | (key[i + 1] & lower_mask)
        key[i] = key[i + m] ^ (y >> 1) ^ (-(y & 1) & matrix_a)
    for i in range(n - m, n - 1):
        y = (key[i] & upper_mask) | (key[i + 1] & lower_mask)
        key[i] = key[i + (m - n)] ^ (y >> 1) ^ (-(y & 1) & matrix_a)
    y = (key[n - 1] & upper_mask) | (key[0] & lower_mask)
    key[n - 1] = key[m - 1] ^ (y >> 1) ^ (-(y & 1) & matrix_a)
    return key

def update_state(state):
    '''
    This function updates the state, that involves incrementing the position␣
 ↪variable and if necessary (i.e. we reach the end of the key) we update the key
    '''
    pos = state[0]
    key = state[1]
    if pos>=623:
        return (0, mt19937(key))
    else:
        return (pos+1, key)

def mt19937_next(state):
    '''
    The tempering function, takes as input the state so that it can be sampled,␣
 ↪it then takes this sample and tempers it to generate the final output
    '''
    pos, key=update_state(state)
    y=key[pos]
    y=int32_rev(y)
    y ^= (y >> 11)
    y ^= (y << 7) & 0x9d2c5680
    y ^= (y << 15) & 0xefc60000
    y ^= (y >> 18)
    return y, (pos, key)
```

3 Setting the Generator

As stated above, we must provide an initial key and a starting position to allow us to start generating pseudo-random integers.

3.1 Python

In Python, we can do this directly by using `numpy.set_state()`, which takes as input the type of generator we are using (in our case it is 'MT19937', however, NumPy does offer the ability to use other generators), the key (taken to be a list of 624 32-bit integers) and the position to sample the key. Or we can use `numpy.seed()` which simply takes an integer as an input. From this integer, it

will generate a list of 624 and set the initial position to be at the end of the list. This is so that when we want to generate our first output the key is updated according to the linear recurrence and we take our sample to be the first word in this new key.

3.2 R

Similarly, in R we set the state directly by defining `.Random.seed`. We do so by supplying a list with the following format, [type of generator (10403 in our case), index to take sample, key]. Or we can simply supply `set.seed()` with an integer and the key is generated from this integer, and again the position is set to the end of this key.

3.3 C++

In C++ we set the seed when we define the generator. If we are working with the standard random functions, we would include the random header, `#include <random>`, and then define the generator, `std::mt19937 gen(seed)`. If we were to use the boost the library we would again have to import the header, `#include <boost/random.hpp>`, and then define the generator, `boost::random::mt19937 gen{seed}`.

3.4 Code Implementation

We can replicate each of these methods of instantiation by re-writing their underlying algorithms as python functions.

[]:
```
def python_state_from_seed(seed):
    key = []
    seed &= 0xffffffff
    for i in range(624):
        key.append(seed)
        seed = (1812433253 * (seed ^ (seed >> 30)) + i + 1) & 0xffffffff
    return (624, key)

def r_state_from_seed(seed):
    for j in range(51):
        seed=int32(69069*seed+1)
    key = []
    for i in range(624):
        seed=int32(69069*seed+1)
        key.append(seed)
    return (624, key)

def cplusplus_state_from_seed(seed):
    key = []
    seed &= 0xffffffff
    for i in range(624):
        key.append(seed)
        seed = (1812433253 * (seed ^ (seed >> 30)) + i + 1) & 0xffffffff
    return (624, key)
```

3.5 Remarks

There are a few points we should note about these particular functions and methodologies for setting up the generator.

1. Python and C++ generate their initial 624 words, in the same way, meaning that given an initial integer seed the MT19937 generator will be set to the same state.
2. R uses a different algorithm to generate its state from an initial integer seed.
   (a) The method R implements is similar to that described in the initial paper for the MT19937 algorithm, published in 1998. However, in 2002 issues were raised in regard to the algorithm yielding nearly shifted sequences when two different integer seeds were provided. The concerns were in relation to how this affected the 'randomness' of the generator. This issue arose when the two supplied integers were close in terms of their Hamming distance. Therefore, an updated method for generating the key from the integer seed was given and this is the one that python and C++ adopt. However, R's implementation of the initial method is a slight variant of that proposed in the original paper and people believe that it doesn't suffer the same issues that forced the update in 2002. I believe the slight variant that R employs is the initial looping of the integer before starting to generate the key.

4 Generating Uniform Variates

We have seen how each language generates a 32-bit word and analyzed some of the differences in the methodologies they adopt for doing this. Now we will move on to seeing how each language takes those 32-bit words to generate a uniform variate.

4.1 Python

We will consider the function `numpy.random.uniform()`, which according to the documentation it samples uniformly over the half-open interval [low, high), where by default `low=0` and `high=1`.

4.2 R

For R we will look at the standard `runif()` function which, differing to python will sample from the interval (min, max), where by default `min=0` and `max=1`.

4.3 C++

Finally, for C++ we will investigate multiple methods for generating uniform variates. From the standard library, we will look into `generate_canonical` which produces variates from the interval $[0, 1)$. Then we will look at `uniform_real_distribution` from the boost library which samples from the interval [min, max), with `min=0` and `max=1`.

4.4 Code Implementation

[]:
```
def python_random_uniform(state, low=0, high=1, size=1):
    output=[]
    for i in range(size):
        integer_one, state = mt19937_next(state)
        integer_two, state = mt19937_next(state)
        variate = ((integer_one>>5)*67108864.0+(integer_two>>6))/
 ↪(9007199254740992)
        output.append(low+(high-low)*variate)
    return output, state

def r_random_uniform(state, n=1, min=0, max=1):
    output = []
    for i in range(n):
        state=state32(state)
        integer_output, state = mt19937_next(state)
        output.append(integer_output/(2**32-1))
    return output, state

def cplusplus_generate_canonical(state, size=1):
    output = []
    def cplusplus_next_float(state):
        integer, state = mt19937_next(state)
        return integer/(2**32), state
    while len(output)!=size:
        next_float, state = cplusplus_next_float(state)
        if next_float != 1:
            output.append(next_float)
    return output, state

def cplusplus_boost_uniform_real_distribution(state, a=0, b=1, size=1):
    output = []
    for i in range(size):
        integer_one, state = mt19937_next(state)
        integer_two, state = mt19937_next(state)
        output.append(((integer_two>>5)*67108864.0+(integer_one>>6))/
 ↪(9007199254740992-1))
    return output, state
```

4.5 Remarks

- The combination implemented by python and C++ may seem arbitrary at first glance, however, it is used to help solve a prominent issue in floating point operations. Due to how we store floating point numbers, when we divide our integer output the effects of rounding will cause some bits to be over-represented and others to not be present at all in the final output space. Therefore, to try and increase the uniformity of the distribution of bits amongst the output space, Python and C++ use two outputs from the generator to generate a larger 53-bit

integer which is then divided to get our variate
  - I will only explain Python's process for forming this larger integer as C++'s method is *almost* identical. We draw two 32-bit words from our generator. We apply a bitshift of 5 to the first (effectively dividing it by 2^5) making it a 27-bit integer. Then it is multiplied by $67108864 = 2^{26}$, making it a 53-bit integer. However, in binary this new integer will have 26 zeros at the right, this is where we use our second integer. We take our second integer and perform a bitshift of 6, which forms a new 26-bit integer. We take this and add it to the 53-bit integer so that we populate its remaining bits. We can now divide by 2^{53} to get our variate between $[0, 1)$
- We must note as well that the effects of rounding may cause 1 to be an output of this process despite the MT19937 having a range of $[0, 2^{32} - 1)$ and we are dividing by 2^{32}
- R simply divides the output of the generator by $2^{32} - 1$ to generate its variates.
  - In this case, if 0 or 1 is to be returned it has predetermined values that it will output instead. If 0 was to be outputted $\frac{1}{2(2^{32}-1)}$ is outputted instead and if 1 were to be outputted the function will output $1 - \frac{1}{2(2^{32}-1)}$
- Now in regards to the way C++ generates its variates
  - We notice that for `generate_canonical` we simply divide by 2^{32}, like in R and we deal with the potential of returning 1 by calling the function again in the event that 1 is to be returned. This has minor effects on the uniformity of the output space.
  - Boost's `uniform_real_distribution` works in a similar but yet different way to Python's `numpy.uniform()`. Note that it divides by $2^{53} - 1$ to reduce the risk of returning 1.
  - We note that the standard library also has a `uniform_real_distribution` function, which works in the same way as `generate_canonical`.
  - Boost also has a function `uniform_01` which is just the special case of `boost::uniform_real_distribution` we are considering.

5 Aligning PRNG Across Languages

It would be beneficial if we had methods to align the generation of pseudo-random numbers across the language such that we can replicate any random simulations or models across the languages. There are multiple ways we could go about doing this which I will explore now. For convenience, I will aim to align C++ and R to Python, as this will then provide capabilities to align C++ and R if desired.

5.1 Aligning R and Python

There is one simple solution to this, which is to run R code in Python. This can be done by simply downloading the 'rpy2' library and using its 'robjects' module.

[]:
```
robjects.r('''
set.seed(1)
print(runif(2))
''')
```

```
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819
```

Or we can use the functions we defined above to reproduce random variates generated in R in

Python.

[ ]:
```
r_random_uniform(r_state_from_seed(1), n=2)[0]
```

[ ]:
```
[0.2655086632039185, 0.372123899723432, 0.5728533634852743, 0.9082077902062349,␣
↪0.20168193108441354]
```

5.2 Aligning C++ and Python

Similarly, we can use the functions we defined above to reproduce random variates generated in C++ in Python.

[ ]:
```
cplusplus_generate_canonical(cplusplus_state_from_seed(1))[0]
```

[ ]:
```
[0.4170219984371215, 0.99718480813317, 0.720324489288032, 0.9325573612004519, 0.
↪00011438108049333096]
```

[ ]:
```
cplusplus_boost_uniform_real_distribution(cplusplus_state_from_seed(1),size=1)[0]
```

[ ]:
```
[0.997184808679089, 0.9325573647046425, 0.1281244456776557, 0.9990405164832814,␣
↪0.23608897735741655]
```

6 References

[1] Matsumoto M & Nishimura T (1998) Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator

[2] Goualard, F. (2020). Generating Random Floating-Point Numbers by Dividing Integers: A Case Study.

[3] https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.html

[4] https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/Uniform

[5] https://cplusplus.com/reference/random/generate_canonical/

[6] https://cplusplus.com/reference/random/uniform_real_distribution/

[7] https://www.boost.org/doc/libs/1_36_0/libs/random/random-distributions.html