Pseudo Random Number Generators in Python, R and C++

Thomas Walker

Supervised by Dr Dean Bodenham

September 2022

Contents

Abstract

When conducting research involving computer simulations, often, a random function is used at some point within the code. These 'random' functions can be taken for granted and used without knowing exactly how it is 'random'. Consider the example of generating uniform random values in the interval $[0, 1)$; If one was to set the random generators in Python, R, or C++ to have the same seed then we would get different sequences of randomly generated observations. This can be particularly frustrating for someone who wants to reproduce someone else's research within a different programming language, or for collaborators who have different programming language preferences. In this work, we explore why these differences occur, and what we can do to align the generators to facilitate reproducible research to occur across the languages. We focus our attention on the generation of random uniform values in the interval $[0, 1)$ as they form the basis of most other random functions, allowing the results of this investigation to be utilised in aligning other random functions. For example, a popular method for generating non-uniform variates involves passing uniform observations through the inverse CDF of these distributions. Interestingly the default pseudo-random number generator used in these languages is the same, the Mersenne Twister. The discrepancy in the randomly generated sequences arises in how the initial state of the generator is set and how the outputs are processed to generate the random observations. Knowing this we can construct functions to reproduce results seen in R or C++ within a Python environment.

1 Introduction

In this work, we investigate pseudo-random number generators (PRNG) in Python, R, and C++. The aim of the investigation is to understand how the languages go about generating random numbers, and how they then go on to generate samples from a uniform distribution. The intention of this work is to be able to provide frameworks to allow us to align the generation of uniform variates across the languages. This can facilitate research endeavours across different programming languages and can help reproduce the results of investigations involving random simulations. For example, if we are working in Python and R we note that even if we set the seed using the same integer when we go to generate uniform observations we get different outputs.

```
import numpy.random as rng
from rpy2 import robjects

rng.seed(1)
print(rng.uniform(size=5))

robjects.r(’’’
set.seed(1)
print(runif(5))
’’’)
```

Similarly, if we run the following code in C++ we obtain discrepancies in the samples observed, despite setting the seed to be 1 in each of the languages.

```
#include <iostream>
#include <random>

int main()
{
    std::mt19937 gen(1);
    for (int n = 0; n < 5; n++) {
        std::cout << std::generate_canonical<double,
        std::numeric_limits<double>::digits>(gen)<< ’\n’;;
    }
}
```

The results of the investigations conducted are available in the following GitHub repository. Using the ’Aligning Pseudo Random Number Generators Notebook’ one can see implementations on which this report makes remarks. The repository establishes a connection from R and C++ random number generators to a Python environment. For Python we will concern ourselves with random functions from the NumPy library, for R we will deal with the standard random functions, and for C++ we will look both at its standard random functions and those instantiated by the boost library.

2 The Mersenne Twister

This is an algorithm for generating pseudo-random numbers as is the default for each of our considered languages. The reason for its popularity is its large period and the strong results it provides under the k-distribution test.

2.1 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 the w-dimensional row vector over $\mathbb{F}_2$ that is the binary representation of the integer with the least significant bit appearing to the right. To denote these word vectors we will use bold vector notation, $\mathbf{x}$.

2.2 Distribution test

The k-distribution test is a test that attempts to quantify the 'randomness' of an algorithm. The outline is as follows:

- Start with a pseudo-random sequence, (x_i), of w-bit integers of period P.
- 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 $\mathrm{trunc}_c(x)$, then the vectors we are considering are
    $$(\mathrm{trunc}_v(x_i), \dots, \mathrm{trunc}_v(x_{i+k+1})$$
    for $0 \leq i \leq P$.
- There are 2^{kv} possible vectors, so we 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.

In [1] there is an illustrative geometric intuition for the test.

2.3 The Algorithm

The Mersenne Twister algorithm as described in [1] 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

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

where

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

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

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

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:

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

Where u, s, t, and l are integers, $\ll k$ denotes a left bit shift by k and $\gg k$ denotes a right bit shift by k. $\mathbf{b}$ and $\mathbf{c}$ 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. The period parameters of the MT19937 instance are

- $w = 32$
- $n = 624$
- $r = 31$
- $m = 397$
- $a = 0\text{x}9908\text{b}0\text{df}$

Whilst the tempering parameters are

- $u = 11$
- $s = 7$
- $t = 15$
- $l = 18$
- $b = 0\text{x}9\text{d}2\text{c}5680$
- $c = 0\text{xefc}60000$

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.

3 Setting the Generator

As noted in the previous section we must provide an initial key and a starting position to allow us to start generating pseudo-random integers. If we want to have each programming language generate the same sequence of random numbers then it is important we understand how they each set up the generators. As we will see there are methods to set up the generator directly or in-directly.

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 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. For Python, we will consider the function `numpy.random.uniform()` which according to [2] samples uniformly over the half-open interval [low, high), where by default `low=0` and `high=1`. For R we will look at the standard `runif()` function [3] which, differing to Python will sample from the interval $(\min, \max)$, where by default `min=0` and `max=1`. For C++ we will investigate multiple methods for generating uniform variates. From the standard library [4], 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 [5] which samples from the interval $[\min, \max)$, with `min=0` and `max=1`. What we see is the following.

- 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 PRNGs Across Programming Languages

We are now in a position to be able to align the generation of pseudo-random integers across the languages. There are multiple ways of doing this, each of which is explored in the associated notebook mentioned in the introduction to this report.

6 Generating Exponential Random Variates

Python uses the inverse-CDF method by default, despite having an implementation of the Ziggurat method within the source code and stating within the documentation that the Ziggurat method is used. R uses an algorithm attributed to Jh Ahrens and U Dieter to generate its exponential random variates. The standard library in C++ has an `exponential_distribution` function to generate exponential variates. It uses the inverse-CDF method in order to generate its variates. The boost library has a similar `exponential_distribution` function to generate variates from the standard exponential. However, it generates the variates using the Ziggurat method.

7 Generating Normal Variates

7.1 Python

Currently, we are missing the implementation to replicate the normal variates from NumPy's normal function. At the moment I have a function that uses the same methods (I believe) as NumPy's normal function to generate a normal variate from the outputs of the generator. However, upon running my implementation the observations I find are different despite the generators being set in the same way. The algorithm which I have written I believe to be correct as running it a large number of times I see that the output resembles that of a normal distribution, so the discrepancy I think lies in the random aspect of the algorithm. The particular algorithm we are implementing is the Ziggurat method, where the random element comprises of generating a 64-bit integer from our generator and uses this to generate our normal observation. Therefore, I think the issue lies in the 64-bit integer I am using in the algorithm rather than the algorithm itself and I haven't found a way to check that this is really the case. Despite this, I will give the coded implementation that I have achieved so far, with the intention that the underlying algorithm can still be analyzed despite it not being able to replicate the observations we see in practice.

7.2 R

To generate its normal variates R uses the inverse-CDF method. However, by looking through the source code it is clear that it has the ability to use a vast array of other algorithms. The algorithms are implemented such that they sample from a standard normal, the variates are subsequently scaled if we are considering non-standard normal distributions.

7.3 C++

The boost library generates its variates using the Ziggurat method.

8 Ziggurat Method

This is a method to generate samples from decreasing densities. It uses a form of rejection sampling to generate its variates and works in the following way. We choose a covering set $(\mathcal{Z})$ for the area $(\mathcal{C})$ under a density $f(x)$ which consists of a set of rectangles of equal area (v) stacked on top of each other, with the bottom strip tailing off to infinity. We will label the rightmost coordinate of rectangle i (R_i) by x_i. So we have that $0 = x_0 < x_1 < x_2 < \dots$. If a random rectangle R_i is selected then a random point in R_i is Ux_i with U uniform $(0, 1)$, and if $x < x_{i-1}$ then our random point (x, y) must be in $\mathcal{C}$ and so we can confirm the point x without having to calculate y. Let r be the rightmost x_i. We may generate from the base strip as follows:

- generate $x = \frac{vU}{f(r)}$, with U uniform $(0, 1)$
- If $x < r$, return x
- Else return x from the tail

So we get an x from the base rectangle with probability $\frac{rf(r)}{v}$, the same as generating an x from one of the other rectangles. This ensures that we can easily sample an x from $\mathcal{Z}$ as we can randomly choose a rectangle according to a uniform distribution (as they can be chosen with equal probability) and then we can easily sample from the corresponding rectangle. Python implements a version of this algorithm that uses 255 rectangles, and a base strip as the covering set. To apply the algorithm in its entirety we can use the following procedure, which uses the output of a 32-bit word generator for maximum efficiency.

1. Generate a random 32-bit word j, let i be the index provided by the rightmost 8 bits of j.
2. Set $x = jw_i$. If $j < k_i$ return x
3. If $i = 0$ return an x from the tail
4. If $[f(x_{i-1} - f(x_i))]U < f(x) - f(x_i)$, return x
5. Go to step 1

Here $w_i = 2^{31}x_i$ and $k_i = \lfloor 2^{32}(\frac{x_{i-1}}{x_i})\rfloor$ for $1 \leq i \leq 255$ and for $i = 0$ $k_0 = \lfloor 2^{32}\frac{rf(r)}{v}\rfloor$ and $w_0 = 2^{31}\frac{v}{f(r)}$ So for each density, we consider we need to find the appropriate values of r and v, and consequently, x_i, to form our rectangles for our covering set $\mathcal{Z}$. We can then sample from $\mathcal{Z}$ and reject samples according to our algorithm to generate variates of our desired distribution. The rejection rate for this algorithm is very low making it efficient (for most distributions a sample x is accepted around 98% of the time).

9 Randomly Sampling and Shuffling 1-D Lists

We can now how the programming languages go about sampling from a list of elements. We will cover multiple cases, including sampling with and without replacement as well as generating samples of different sizes. Furthermore, we will show how programming languages shuffle the elements of a list. Some of these cases may overlap, for example, if we take a sample without a replacement that is the same in size as the size of the list then we are effectively permuting the elements of the list. However, we ought to be careful as there may be different functions to perform these operations within the programming and they may yield different results. In the proceeding sections, we will only consider Python and R.

9.1 Python

Python shuffles a list by considering each element in turn and sampling uniformly from the set of indices before this element and then swapping the element we are at with the element at the index we have sampled. Python's NumPy library also has a function `numpy.random.choice` which allows us to sample elements from a 1-D array. We can specify whether this sampling is with or without replacement as well as the size of the sample to generate. We note that we get different results when we specify to sample from the discrete uniform, which is equivalent to not specifying a discrete distribution. Generating a sample without replacement with a size equal to the size of the list will give the same output as if we called our function directly (provided we do not specify the probabilities)

9.2 R

In R we have a similar function, `sample()`, which allows us to generate a sample from a list either with or without replacement. Furthermore, we can specify the size of the sample and the probability distribution against which we sample. The sample function in R lets us set the probabilities for the discrete distribution we wish to sample from. We note that when we set the probabilities all equal to each other we get a different sample to the sample where we do not specify the probabilities. By default R still samples uniformly when the probabilities are not given, however, it uses a different algorithm to do so and yields different samples. We note that for the case where we are not specifying the probabilities, the samples obtained when we sample with or without replacement are the same up to the point where we sample a duplicate element in the list.

References

[1] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: 8.1 (Jan. 1998), pp. 3–30.

[2] Numpy Developers. *numpy.random.uniform*. 2022. URL: https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.html.

[3] K Gerald van den Boogaart. *runif*. 2023. URL: https://www.rdocumentation.org/packages/compositions/versions/2.0-6/topics/runif.

[4] C++ Developers. *generate_canonical*. 2023. URL: https://cplusplus.com/reference/random/generate_canonical/.

[5] *Boost Random*. URL: https://www.boost.org/doc/libs/1_83_0/doc/html/boost_random/reference.html.