Pseudorandom Numbers in R

Jack Baumruk

October 31, 2017

Introduction

Humans have been trying to generate random numbers for over 4000 years, dating back to the earliest uses of dice carved from bone [1]. Every since, scientists, gamblers, and statisticians have been searching for the best ways to generate random numbers for use in simulations, random sampling, and fair games. With the advent of computers, we have a tool that can use random numbers to do incredible tasks and research, but came across a fundamental problem: computers are entirely deterministic systems. Instead, we must try to produce "pseudorandom" numbers which are generated deterministicly but, ideally, are indistinguishable from true random numbers. In this post, We will briefly discuss the history of pseudorandom number generation and then dive into how these numbers are generated and used in R.

A Brief History of Psuedorandom Numbers

The basic idea of a pseudorandom number generator is to start with some seed and iterively apply an algorithm to create a sequence of numbers independent from each other and uniformly distributed on some interval. We want to have our seed be from some random source (or as random as possible) like the low bits of the current time, or a mixture of user input, or have a simple seed and run the algorithm for several iterations before using the numbers. We then want an algorithm that can iterate through a sequence of numbers uniformly distributed in some range that takes a long time to return to a previous state, and thus repeat (that is, having a long period), and won't converge to a particular value over time.

One of the earliest examples of a pseudorandom number generator, proposed by John von Neumann, was the Middle-sqaure method [2]. The algorithm is quite simple, in which one takes a 4-digit number, squares it to get an 8-digit number, then takes the middle 4 digits of that number as the next value, and repeat to generate a sequence of pseudorandom 4-digit numbers, or be generalized to any n-digit numbers. This method had several problems if trying to create a sequence indistinguishable from random such as short periods and convergence to zero. For this method, however, von Neumann discussed that he felt it was important for pseudorandom number sequences to be distinguishable from truly random numbers, for one attempting to arithmetically produce random digits is "in a state of sin." However, most 20th-century statisticians weren't particularly interested in the philosphical or religious consequences of their algorithms, and rather were just trying to find a way to create sufficient randomness for practical use in games and simulations. An attempt to fix von Neumann's method and make it less distinguishable from random was proposed in which one adds a Weyl Sequence to the squares to extend the period, which proved effective and is an interesting read for those of you who are curious [3].

Throughout the 20th century, statisticians have been trying to find better and better pseudorandom number generators, and have been increasingly successful. Now there are dozens of very effective algorithms being used across various machines and software which can be read about in more detail in Donald Knuth's *The Art of Computer Programming: Seminumerical Algorithms, Volume 2* for the interested [4]. In this post, however, we will focus on the algorithms used by R, in particular the "Mersenne-Twister" algorithm used by default and abundantly common amongst many other software packages.

I will primarily using R's provided documentation for the base package as reference for the following sections, cited at the end of this post as [5].

Random Seeds in R

First, we will look at R creates a stores random seeds. In this and future sections, we will be exclusively looking at the case in which "Mersenne-Twister" is the RNG used.

When the user wants to execute an operation that requires random numbers, such as `runif()` which returns random numbers uniformly from [0,1], R will check for the object `.Random.seed` which stores the current state of the software's random number generator (RNG). This object is stored in the User's work environment, and therefore can be saved and restored as needed. If there is no `.Random.seed` found in the work environment, R will generate one using the current time and process ID, thus will be different every time a user starts a new session. However, a user can specify a `.Random.seed` value for R to use by using the `set.seed()` function to ensure their results are reproducible.

The `.Random.seed` Object

The `.Random.seed` object is a vector of 626 integers. The first two are used to indicate the properties of the object, while the other 624 are a string of pseudorandom integers. They are 32-bit integers in the range $[-2^{31}, 2^{31}]$ and are meant to be a simulation of a uniform distribution on that range. We look at the first few of these numbers in the seed generated by `set.seed(1)`:

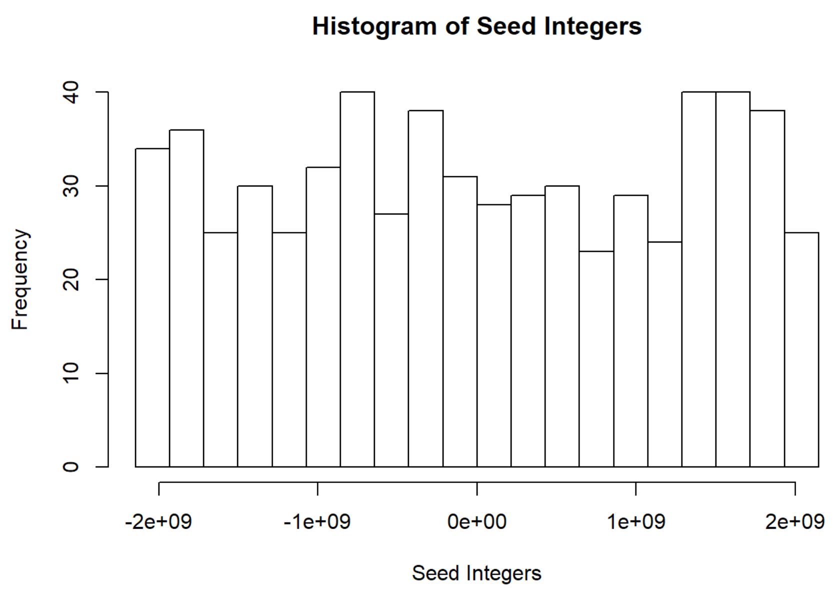
```r
#generate a .Random.seed object
set.seed(1)

#look at the first 5 32-bit integers in the seed
head(.Random.seed[-(1:2)], 5)
```

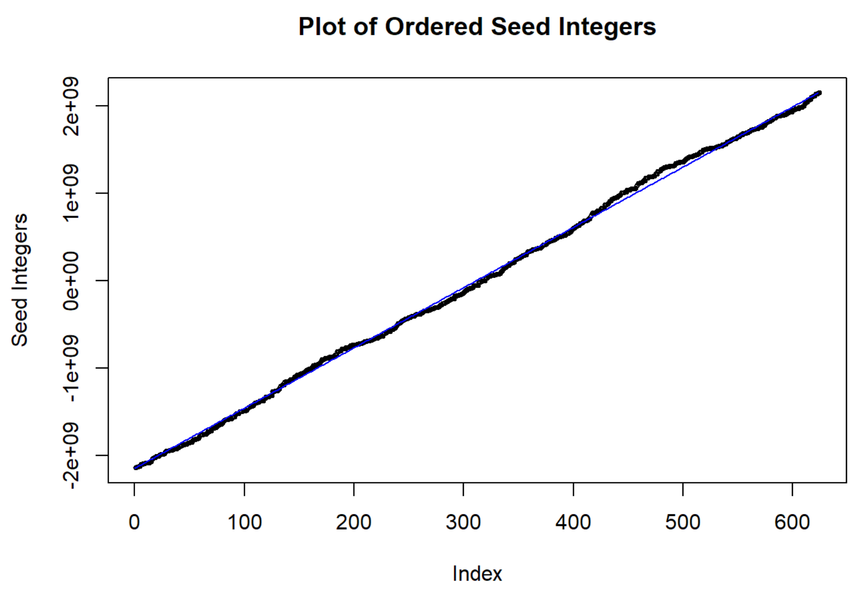
```
## [1] -169270483 -442010614 -603558397 -222347416 1489374793
```

We can look at a histogram and plot of the ordered integers to see how closely this seed is to a uniform distribution:

```r
# Display a histogram of the integers with 21 bins spanning the range:
hist(.Random.seed[-(1:2)], breaks = (-10:10)*(2^31)/10,
     main = 'Histogram of Seed Integers', xlab = 'Seed Integers')
```



```
# Plot the ordered integers with a line showing the uniform distribution:
plot(.Random.seed[order(.Random.seed[-(1:2)])+2],
     main = 'Plot of Ordered Seed Integers', xlab = NULL,
     ylab = 'Seed Integers', cex = 0.5)
points(0:623, (0:623)*(2^32)/623 - (2^31), type = 'l', col = 'blue')
```



We see by these plots that the seed integers do appear to have a uniform distribution as the histogram is reasonably flat and the plot of ordered values is close to the straight diagonal line shown.

The first two values of `.Random.seed` indicate the type of RNG used and the current position in the seed, respectively. Let's look at these values for our object:

```
.Random.seed[1:2]
```

```
## [1] 403 624
```

The first number tells us two things. The first digit indicates the normal generator used (which we will not discuss here), and the last two digits indicate the kind of RNG used. Here, `03` indicates that we are using the fourth RNG in the list (indexed from zero), which in our case is 'Mersenne-Twister.' We can also see this indicated by the function `RNGkind()`:

```
RNGkind()
```

```
## [1] "Mersenne-Twister" "Inversion"
```

Here, the first entry indicates the RNG used and the second, the normal generator used.

The second number in `.Random.seed` indicates the current position in the sequence, as R will continue to use the same seed object for number generation until it has gone through the entire list, after which it generates new integers. We see by the above that after calling `set.seed()`, R sets the position to 624, which is the end of the sequence indicating that upon calling for a random number, R will generate a new sequence of integers and start back at 1 for that sequence. We can see how R iterates through the seed with a simple example:

```r
# Starting position and first integer
.Random.seed[2:3]
```

```
## [1]        624 -169270483
```

```r
# Generate a random number, and check the new position and first integer
x <- runif(1)
.Random.seed[2:3]
```

```
## [1]          1 1654269195
```

```r
# Generate 5 more random numbers, check the new position and first integer, repeat
x <- runif(5)
.Random.seed[2:3]
```

```
## [1]          6 1654269195
```

```r
x <- runif(5)
.Random.seed[2:3]
```

```
## [1]         11 1654269195
```

```r
# Generate 624 random numbers and check the position and first integer
x <- runif(624)
.Random.seed[2:3]
```

```
## [1]         11 1980538363
```

```r
# Some functions use more than one integer from .Random.seed when generating an output
x <- rnorm(5)
.Random.seed[2:3]
```

```
## [1]         21 1980538363
```

We see that every time R generates a uniform random number, it increments the position by one. If R generates a vector of 5 uniform random numbers, it must increment the position by 5. Some functions, such as `rnorm()`, will used more than one integer from `.Random.seed` to generate the output, but will increment the position counter appropriately to indicate how many it used so that they will not be used again. Once the position reaches 624, it resets to one and generates a new set of seed integers. We can also see that the first integer in the seed (and the rest of the integers) stay the same as the position is incremented, until it passes 624 after which it changes.

The `set.seed()` Function

One of the most important functions for random numbers in R is the `set.seed()` function. This function allows the user to specify a value to be used to deterministically generate the `.Random.seed` object so that every time a particular batch of code is run, the same results will be returned. The `set.seed()` function takes a single integer argument that will be used to initialize the RNG as specified by the Mersenne-Twister algorithm.

We show the first few seed integers of `.Random.seed` after using different initial seeds:

```r
set.seed(1)
head(.Random.seed[-(1:2)], 5)
```

```
## [1] -169270483 -442010614 -603558397 -222347416 1489374793
```

```r
set.seed(2)
head(.Random.seed[-(1:2)], 5)
```

```
## [1] -1619336578  -714750745  -765106180   158863565 -1093870294
```

```r
set.seed(3)
head(.Random.seed[-(1:2)], 5)
```

```
## [1] 1225564623 -987490876 -926653963  540074546  617851915
```

```r
set.seed(123)
head(.Random.seed[-(1:2)], 5)
```

```
## [1] -983674937  643431772 1162448557 -959247990 -133913213
```

We see that each seed we set creates completely different seed integers.

Since each time R generates random numbers from the seed integers it does so deterministically and the Mersenne-Twister algorithm generates new seed integers from the previous ones deterministically, we can call as many random numbers as we want after setting a seed and we will still be able to know we will end up at the same final seed everytime we run our code.

We demonstrate this by generating thousands of random nunmbers after setting the seed, check the new seed, and repeat the process to ensure we end up at the same place and generate the same numbers:

```
# Set the initial seed
set.seed(1)

# Generate 100,000 random numbers and check the seed values
x <- rnorm(100000)
head(.Random.seed[-(1:2)],5)
```

```
## [1] -1413073199 -1962465615   119760528  -141269000   132924947
```

```
# We repeat to show consistency
set.seed(1)
y <- rnorm(100000)
head(.Random.seed[-(1:2)],5)
```

```
## [1] -1413073199 -1962465615   119760528  -141269000   132924947
```

```
# We see that we generated the same numbers
sum(abs(x - y))
```

```
## [1] 0
```

Thus, `set.seed()` does exactly what we want by ensuring that we can always replicate our results for a particular set of code.

The Mersenne Twister Pseudorandom Number Generator

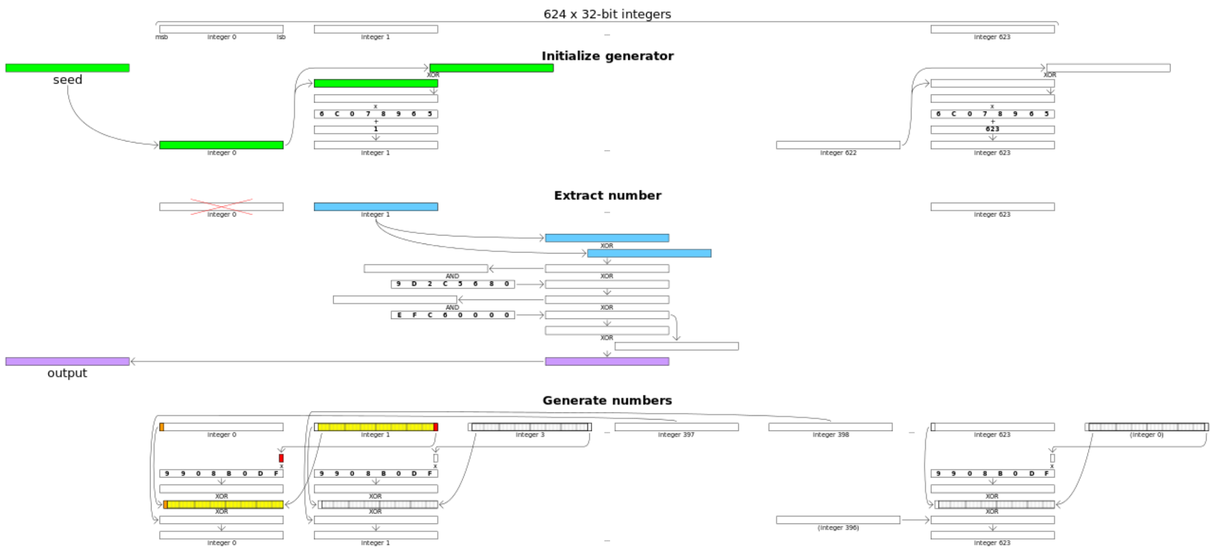
The Mersenne Twister is a pseudorandom number generator (MT) developed in 1997 by Makoto Matsumoto and Takuji Nishimura [6]. The name comes from the fact that the algorithm involves using a Mersenne prime number (primes of the form $2^n - 1$) and the process of "twisting" which is the process of generating a new set of integers by *twisting* the integer value by using an *xor* mask of a given parameter on some number of bits of that value and some number of bits of another value some distance away in the list of integers. If that last sentence made no sense to you, that's okay, as MT is actually a fairly involved and complicated algorithm. For that reason, we will not go into too much depth on the actual algorithm, but at least take a look at it and see why the values seem so random. For a full understanding of the algorithm for more advanced readers, I would encourage you to read the originial 1997 paper [6] and the wikipedia article on MT [7].

The most commonly used form of this alrogithm, and the one used by R, is MT19937, which is what we will be looking at here. The period of this MT is the Mersenne prime $2^{19937} - 1$ and it generates 624 32-bit integers. A 64-bit version called MT19937-64 also exists.

First one must initialize the MT by taking the seed as x_0 to generate x_1 via a formula involving bit-wise Xor, bit-shifts, and constants. The same process is then repeated on x_1 to generate x_2 and so forth up to $x_6$24.

Next, to extract our initial numbers, we than have to apply another formula involving several Xors, bit-shifts, concatenations, and constants on each of our x_i. These numbers are then sufficiently distributed to be statistically independent of the seed, passing the Diehard tests [8], and being indistinguishable from a truly random uniform distribution.

From there, another complicated, but less so, algorithm is used to generate a new set of integers from the previous. See the below image for an outline of the process.



Mersenne Twister Algorithm Visualization

Summary: Why are Pseudorandom Numbers Important?

Psuedorandom numbers allow statisticians to simulate statistical models and distributions with excellent accuracy and are one of the most important tools for research and application in Statistics. Further, by not using truly random numbers, one can use a pre-established seed, allowing peers, clients, and students to reproduce the results in order to understand, learn, and debug studies and projects. Understanding how these numbers are produced and thereby the advantages and disadvantages they carry is very important for doing statistical work and research, and should not be ignored. Once you have a grasp of random numbers and their use in R, you can use them to their fullest extent without losing rigor or understanding.

References:

[1] Tashion, Carl. “A Brief History of Random Numbers.” *FreeCodeCamp*, 10 Mar. 2017, https://medium.freecodecamp.org/a-brief-history-of-random-numbers-9498737f5b6c.

[2] von Neumann, John. “Various Techniques Used in Connection with Random Digits”. In: A.S. Householder, G.E. Forsythe, and H.H. Germond, eds., Monte Carlo Method, *National Bureau of Standards Applied Mathematics Series 12* (1951), pp. 36-38 (cit. on p. 1).

[3] Widynski, Bernard. “Middle Square Weyl Sequence RNG.” arXiv preprint arXiv:1704.00358 (2017).

[4] Knuth D.E.. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1997. ISBN 0-201-89684-2.

[5] R Core Team (2016). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

[6] Matsumoto, M.; Nishimura, T. (1998). “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator” *ACM Transactions on Modeling and Computer Simulation.* 8 (1): 3-30. doi:10.1145/272991.272995. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf

[7] Mersenne Twister. (2017, October 27). In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Mersenne_Twister&oldid=807413974

[8] Davis, Chris. “A Better Random Number Generator” *Small Talk Newsletter*. Society of Actuaries.
/newsletters/small-talk/2006/june/stn-2006-iss26-davis.pdf