Simulating Data

W. John Braun
University of British Columbia

September 7, 2024

Contents

1

Wildfire, Board Games and Statistics

Modelling and simulation is a fascinating interplay between computer science, mathematics and statistics. Applications are boundless. In this short chapter, we provide brief vignettes to begin to suggest some of the possibilities. Specifically, we consider applications in risk assessment, gaming and statistical analysis.

1.1 Wildfire risk: bootstrapping a deterministic fire model

Wildfire risk is of increasing importance in Canada and around the world, as the climate changes, and because of human disturbance of the natural landscape. Modelling and simulation play a crucial role in assessing the risk of wildfires to property and community infrastructure. An early example of a wildfire risk assessment is described by Braun et al. (2010). It provides an example of bootstrap simulation applied to a deterministic wildfire growth simulator, in this case, the Prometheus Wildland Fire Growth model (Tymstra et al., 2010).

Here, we consider output from a more accurate simulator called Dionysus which is described in the paper by Han and Braun (2014). Data on the areas and corresponding simulated areas for 99 fires in a subregion of the Muskoka District of Ontario that burned between 1980 and 2010 are plotted in the box plots displayed in Figure 1.1. The bold horizontal lines in the interior of each box represent the medians for the two datasets. Recall that the median is the middle or 50th percentile of a sample or distribution. Here we have an indication that the median of the true fire size distribution is somewhat lower than the median of the simulated fires. The whisker at the bottom of the simulated box plot indicates the possibility of very small simulated fires, while the lack of such a whisker in the box plot for the true fires indicates that there is a minimum size for the true fires. This reflects the fact that fires below a certain size might extinguish without intervention or with minimal human intervention so they do not enter the records.

Muskoka is home to a vast number of very expensive properties and is heavily forested but with many lakes and streams fragmenting the landscape from a wildfire perspective. The actual fires were often detected almost immediately, so the time of burning before suppression was much less than a day in most cases. The simulated fires were allowed to burn for 1 hour under the weather conditions that were present on the day that the fire actually burned.

Figure 1.1 shows results for the simulator for one of the fires. The different types of green areas represent different types of fuel (i.e. vegetation) and the blue areas represent bodies of water. The yellow areas represent the simulated burned areas according to the percentiles 0.1, 0.25, 0.5, 0.75, 0.9, read from left to right. For example, the probability that the fire is contained in the yellow region in the left-most panel is 10%, while the probability that it is contained in the yellow region in the right-most panel is 90%.

1.1.1 Simulating a popular long-standing board game

Monopoly is a game played by rolling a pair of dice and moving an object (called a token) around colored locations called properties according to the sum of the number of observed dots

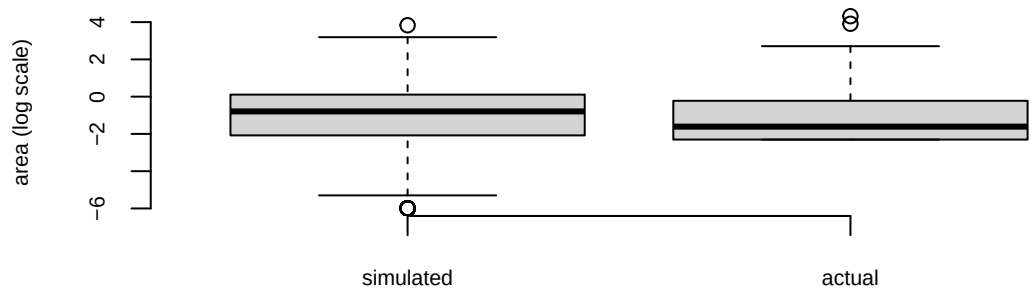


Figure 1.1: Simulated and actual burned areas in Muskoka, Ontario.

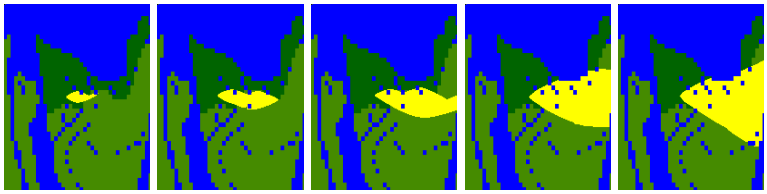
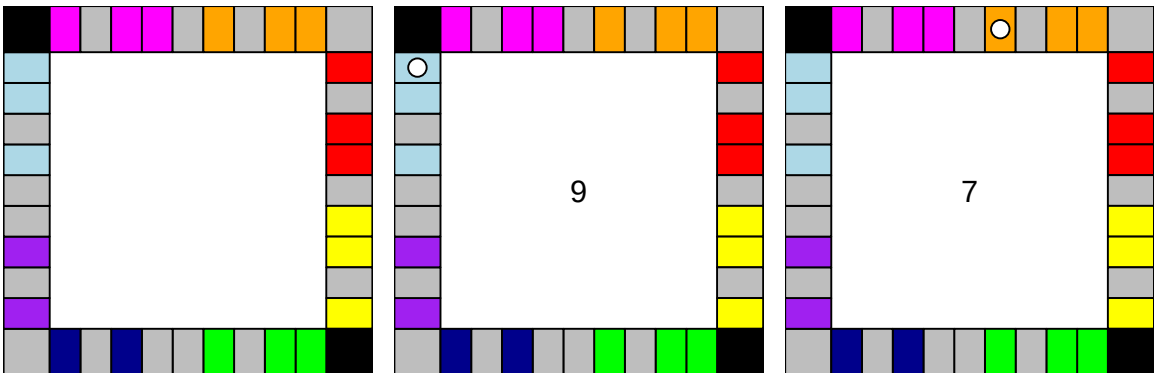


Figure 1.2: Simulated wildfire in the Muskoka region, burning for 1 hour. The yellow region in the first panel corresponds to the amount of fire burned at the 10th percentile; panel two gives the 25th percentile, panel three gives the 50th percentile, panel four gives the 75th percentile, and panel five gives the 90th percentile.



The left panel shows the various color-coded properties of the Monopoly board. The color coding has been chosen to match the colored properties used in the actual game. The grey bars represent non-colored areas of the board such as railroad, utilities, and so on. The black bar in the top left corner represents Jail. When a player lands on the bar in the lower right corner, they must immediately go to jail.

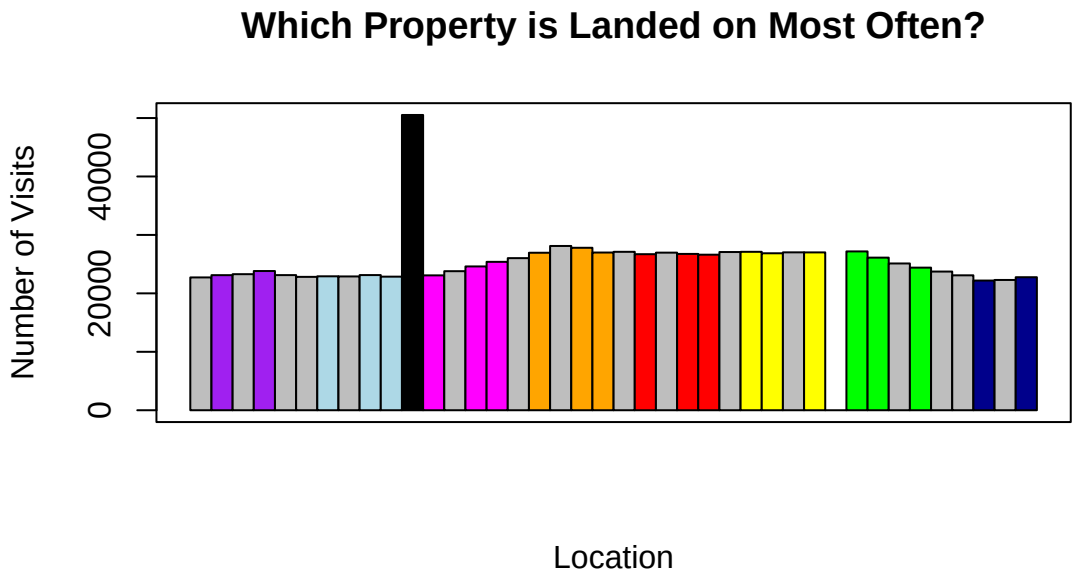
The results of the first two dice throws for a single player are shown in the second and third panels. The white circle (the token) represents the player's location on the board. The value at the center represents the sum of the values on the dice.

Having demonstrated a few moves on the Monopoly board, we are now ready to conduct a major simulation to determine which properties are most frequently landed on by a single player. The basic R code for the simulation is as follows:

```
Nplays <- 1000000
lands <- numeric(Nplays)
lands[1] <- 1                # The player starts at Go.
for (i in 2:Nplays){
    currentthrow <- sample(1:6, size=2, replace=TRUE)
    lands[i] <- (lands[i-1] + sum(currentthrow) - 1)%%40 + 1
    if (lands[i]==31) lands[i] <- 11  # "Go To Jail" is in position 31
}
```

On a 2 GHz laptop computer, this code takes less than 2 seconds to execute, resulting in the sequence of properties landed on by a single individual in 1000000 moves.

The following is a histogram whose bars are color-coded according to the property colors and whose bar heights are proportional to the frequencies with which the various properties are landed on.



Is it surprising that the dark blue properties are not landed on as frequently as the reds and oranges? Why is Jail so popular? Notice the increase in frequency as one proceeds from Jail through the light purple properties towards the orange properties; how is this increase related to the frequencies of the values of pairs of dice?

1.2 Comparing statistical estimators via simulation

A common problem in statistical research is to compare the quality of different estimators by simulation. In this section, we will compare the sample mean and sample median, using simulated random data.

1.2.1 Background

It is known that the median can sometimes be more appropriate than the mean when measuring central tendency. For symmetric distributions, both the mean and the median are unbiased. For normally distributed data, it is known that the variance of the mean is less than the variance of the median. This implies that the sample mean is a more precise estimator of the true mean, than the sample median.

When there are outliers, the median is often recommended over the mean, since the outliers can cause inaccuracies in the mean estimate. The median is said to be robust to such anomalies.

Sample Size: 10		
	variance of means	variance of medians
normal	0.10	0.14
t20	0.12	0.15
t10	0.13	0.15
t2	1.70	0.23
Sample Size: 30		
	variance of means	variance of medians
normal	0.04	0.05
t20	0.04	0.05
t10	0.04	0.05
t2	0.70	0.08
Sample Size: 100		
	variance of means	variance of medians
normal	0.01	0.01
t20	0.01	0.02
t10	0.01	0.02
t2	0.11	0.02

Table 1.1: Variances of averages and variances of medians for the samples coming from four different distribution types and three different sample sizes.

1.2.2 Simulation study

In our study, we

- simulated random data from several different distributions: normal, t_2, t_{10}, and t_{20} at sample sizes $n = 10, 30$ and 100.
- calculated the means and medians for each sample
- created a boxplot of the means and medians for visual comparison
- computed the variances of the means and medians for numerical comparison

1.2.3 Results: tabular variance comparisons

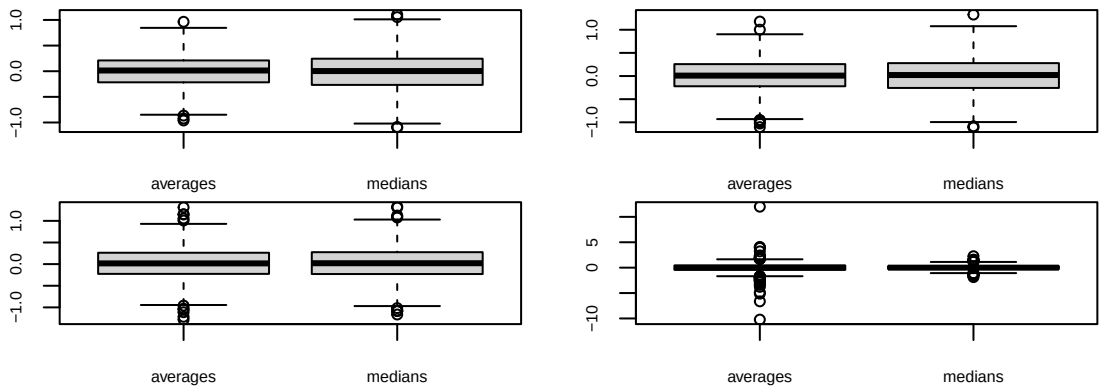


Figure 1.3: Mean versus median comparisons (sample size: 10)

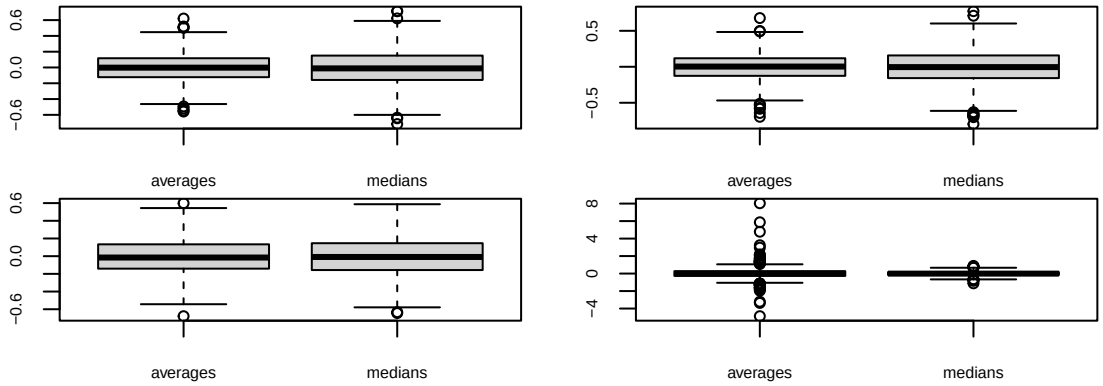


Figure 1.4: Mean versus median comparisons (sample size: 30)

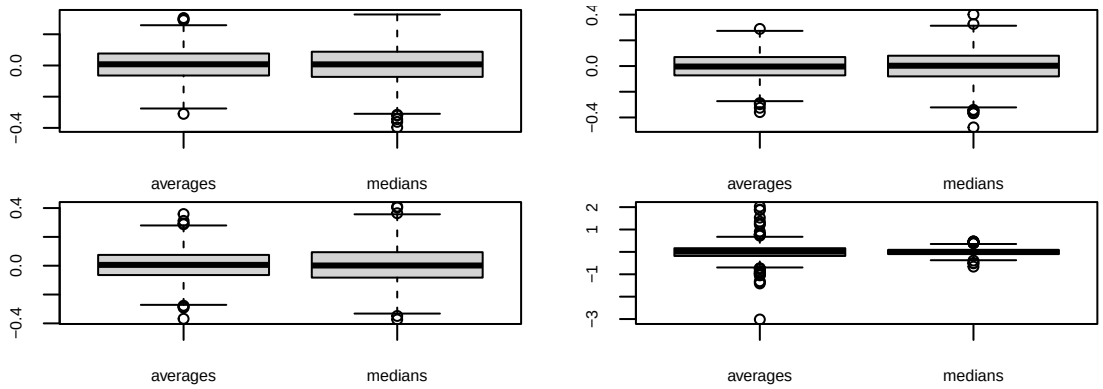


Figure 1.5: Mean versus median comparisons (sample size: 100)

1.2.4 Conclusions

For normal, t_{20}, and t_{10}, the mean has smaller variance than the median, independent of sample size. For t_2 data, the mean has a much larger variance. The variability of the mean and median tends to be similar except for the t_2 case, where the variability of the mean can be quite extreme.

1.3 Chapter Endnotes

We have chosen the R programming language (R Core Team, 2024) for our code demonstrations. This choice was made on the basis of the similarity R scripts share with generic pseudocode: simplicity. On the other hand, R is notoriously slow, so other programming languages will be preferred in practice. A knowledgeable programmer should be able to translate R code to the relevant language fairly easily.

There are many books and online references concerned with modelling and simulation. Some are focussed on the computational aspects, some on aspects related to theoretical physics, and some on statistical aspects. A very good introductory account, mainly from a probability and statistics perspective, can be found in the textbook by Ross (1990). A very comprehensive treatment which is more mathematical can be found in the work of Knuth (2014). Another useful book to consult is by Law et al. (2007).

Some of the material in the current work, particularly that on simulation of normal random variables and multivariate rejection sampling, has been taken from the Simulation chapter in the book by **?**.

2

Generating and Testing Pseudorandom Numbers

In this chapter, we first ...

2.1 Pseudorandom number generation

Simulation of realistic scenarios usually requires the incorporation of uncertain or unpredictable elements. This chapter describes how to simulate random numbers on a computer. For some applications, it is possible to use truly random sequences of numbers derived from atmospheric observations or emissions from radioactive substances. We will describe deterministic[1] methods of generating values which are then treated as though they are random. Simulated random numbers are called pseudorandom numbers, because they are usually not truly random. For some purposes, they can serve as a reasonable approximation. On the other hand, we will also see that caution is warranted under many circumstances. As was pointed out by Coveyou (1969), the generation of random numbers is too important to be left to chance.

The advantages of pseudorandom number generation over the use of these truly random sequences lie in the speed with which certain algorithms will produce large quantities of such numbers and in the fact that pseudorandom numbers can be reproduced. This latter fact turns out to be a double-edged sword: while reproducibility is important for checking scientific results, for example, this property renders the numbers vulnerable to attacks. For example, a lottery corporation will want to prevent their sequences from being correctly modelled, since future values are not supposed to be predicted by unauthorized individuals.

2.2 Predictable versus unpredictable sequences

The most common techniques for constructing pseudorandom variables on a computer are based on sequential generation. That is, previous values are used as inputs to some sufficiently complicated function whose output is the next value in the sequence.

2.2.1 Linearity is highly predictable

If we attempt to generate a sequence by using a linear function of the form

$$x_n = a + bx_{n-1}$$

where a and b are fixed constants, it will be too easy to predict successive members of our sequence. This would define, possibly, the worst of all pseudorandom number generators, since it would be perfectly predictable.

[1] i.e. non-random

Example 2.1 Suppose, for $n = 1, 2, \ldots,$

$$x_n = 3 + 2x_{n-1},$$

and $x_0 = 2$. Using the above formula with $n = 1, 2$ and 3, we obtain

$$x_1 = 7, x_2 = 17, x_3 = 37.$$

With or without the use of the formula, we can predict that the next numbers would 57, 77, and so on.

The kind of predictability coming from a linear mapping is obviously not a good characteristic of a random number simulator. The next question to consider is whether there are nonlinear mappings which are satisfactorily unpredictable.

2.2.2 *Nonlinearity does not always yield unpredictable sequences*

To obtain less predictable sequences, we will require a function that will variously lead to an increase or a decrease. Only nonlinear functions have such a property. Not all do.

Example 2.2 The cosine function is a well-known nonlinear function, so we let $f(x) = \pi \cos(x)$. We start with $x_0 = 2$ and generate 12 successive values from

$$x_n = f(x_{n-1}).$$

```
x <- 2; prnumbers <- numeric(12); f <- function(x) pi*cos(x)
for (n in 1:12) {
    x <- f(x)
    prnumbers[n] <- x
}
prnumbers
```

```
##  [1] -1.3073638  0.8180586  2.1477164 -1.7135662 -0.4470026  2.8329212
##  [7] -2.9931147 -3.1070269 -3.1397161 -3.1415871 -3.1415927 -3.1415927
```

The first few numbers produced by this function are certainly less predictable than the ones generated by the linear map, but eventually, this mapping converges to a single number.

The convergence observed in Example 2.2 occurs because the mapping $x = f(x)$ has a fixed point at $x = -\pi$, since $f(-\pi) = -1$, and this fixed point is stable, meaning that if x_{n-1} is larger than the fixed point, then $x_n = f(x_{n-1})$ will be smaller than x_{n-1}, and if x_{n-1} is smaller than the fixed point, then x_n will be larger than x_{n-1}, and in both cases, x_n will be closer to the fixed point than x_{n-1} was.

In general, the stability of a fixed point is related to the slope or derivative of the curve defined by any function $f(x)$ in a neighbourhood of the fixed point; if the slope is less than 1 in absolute value, the point is stable. Figure 2.1 illustrates the situation for the iteration described in Example 2.2. As the iterates approach the fixed point, the function slope is less than 1 (in absolute value) so the moves between iterates are small, and they become smaller as the fixed point is approached. This kind of stability is required if the goal is to numerically solve an equation of the form $f(x) = x$; it is not a desirable characteristic of a random number simulator. Instead, we need functions for which any fixed points would be unstable, that is, where the derivative at any fixed point is larger than 1 in absolute value.

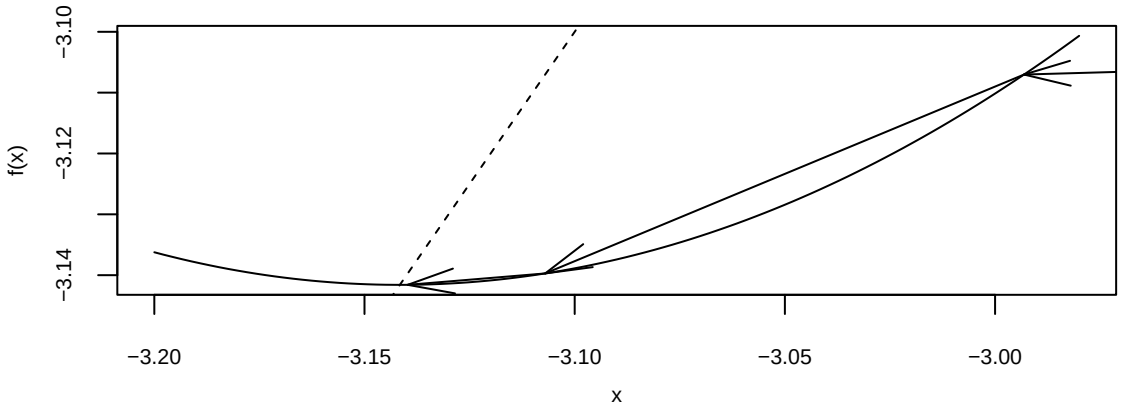


Figure 2.1: Convergence of $x_n = \pi \cos(x_{n-1})$ mapping. The arrows show the trajectory of the latter points in the sequence defined in Example 2.2 as they converge toward the fixed point of the function $f(x) = \pi \cos(x)$. Also shown is a dashed reference line of slope 1.

2.2.3 Nonlinear mappings with several unstable fixed points

By increasing the frequency of the waveform described by a periodic function, we can increase the number of possible fixed points in the interval $[-1, 1]$. Thus, derivative magnitudes of the fixed points will increase, thereby ensuring that they are not stable. In turn, we obtain sequences that appear less predictable and which do not converge to any single fixed value.

Example 2.3 The function plotted in Figure 2.2 is $f(x) = \cos(30x)$. The 45 degree line is overlaid, showing where $f(x) = x$. That is, fixed points occur at each intersection of these graphs. We can see a large number of fixed points. Note that the derivatives near the fixed points are also relatively large, inducing instability in the map defined by

$$x_n = \cos(30x_{n-1}).$$

The first 48 successive values from this map, starting with $x_0 = 2.0$ are certainly less predictable than before, as can be seen in the trace plot pictured in Figure 2.3.

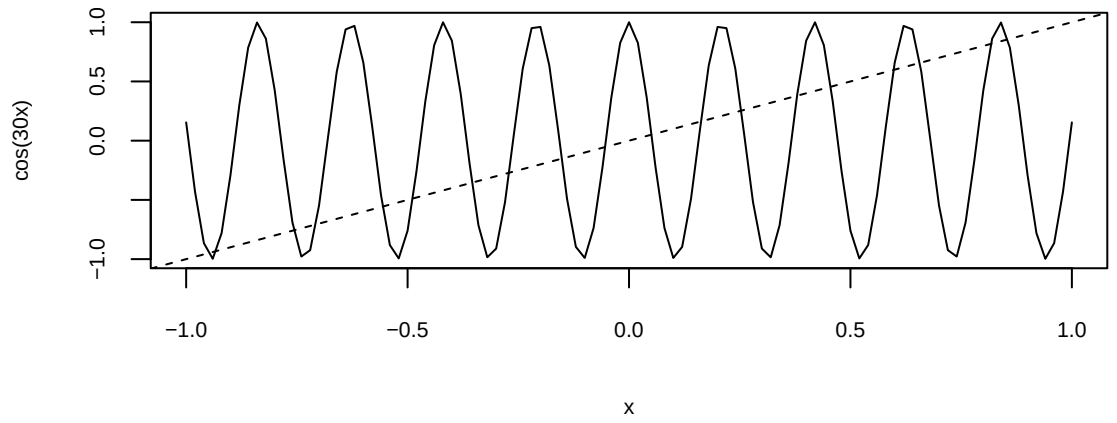


Figure 2.2: A mapping with more, unstable, fixed points. The dashed reference line has slope 1.

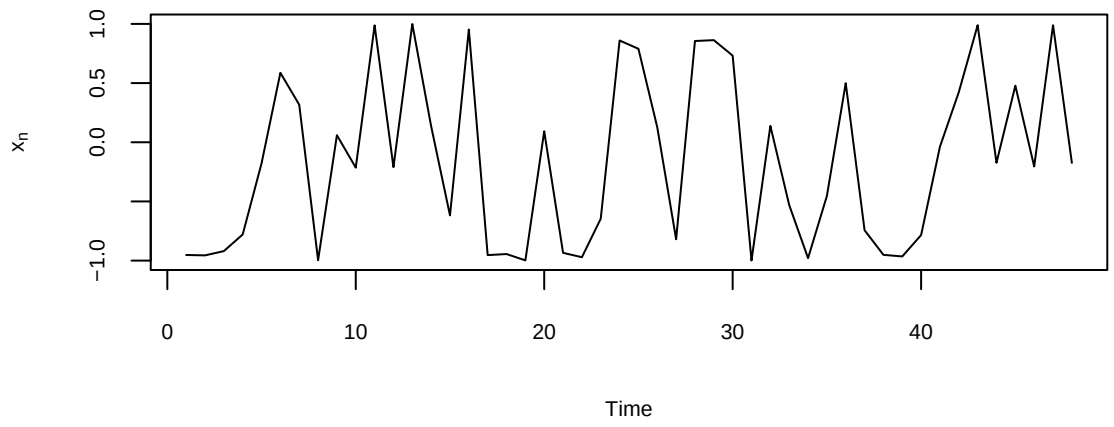


Figure 2.3: Trace plot of first 48 numbers sequentially generated from the cosine map, $x_n = \cos(30x_{n-1})$.

Repeated computations involving trigonometric functions can require more runtime than simpler computations. Therefore, most random number generators that have been implemented for general use do not use such functions. Instead, the operations carried out by random number generators are addition, multiplication, and division. In fact, an important operation associated with random number simulation is to find the remainder after division. This class of operations comprises modular arithmetic.

Finding the remainder after dividing a positive integer a by another positive integer m is also referred to as finding the congruence of a modulo (or "mod") m. We can always write $a = qm + b$ for integers q and b. Since b is the remainder of a after dividing by m, we write $b \equiv a \pmod{m}$ to indicate that b is congruent to a modulo m.

Functions of the form

$$f(x) = ax \pmod{m}$$

have jump discontinuities. Functions with jumps can sometimes induce mappings which seem to be unpredictable.

Example 2.4 Consider the function $f(x) = 32678x \pmod{33271}$:

```
modfun <- function(x)    (32678*x)%%33271
```

Its graph is plotted in Figure 2.4. Consider the first 40 pseudorandom numbers produced by this function via the mapping

$$x_n = 32678x_{n-1} \pmod{33271}$$

with $x_0 = 2$. These values are traced in Figure 2.5, showing a much less predictable pattern than any we have seen so far.

2.2.4 Congruential random number generators

The earliest pseudorandom number generators considered were of the form

$$\begin{aligned} x_n &= a\, x_{n-1} \pmod{m} \\ u_n &= x_n/m. \end{aligned}$$

Usually, m is a large integer, and a is another integer which is smaller than m.

To begin, an integer x_0 is chosen between 1 and m. x_0 is called the seed.

This is called a multiplicative congruential generator. Linear congruential generators are similar: $x_n = (a\, x_{n-1} + c) \pmod{m}$ for a given positive integer c.

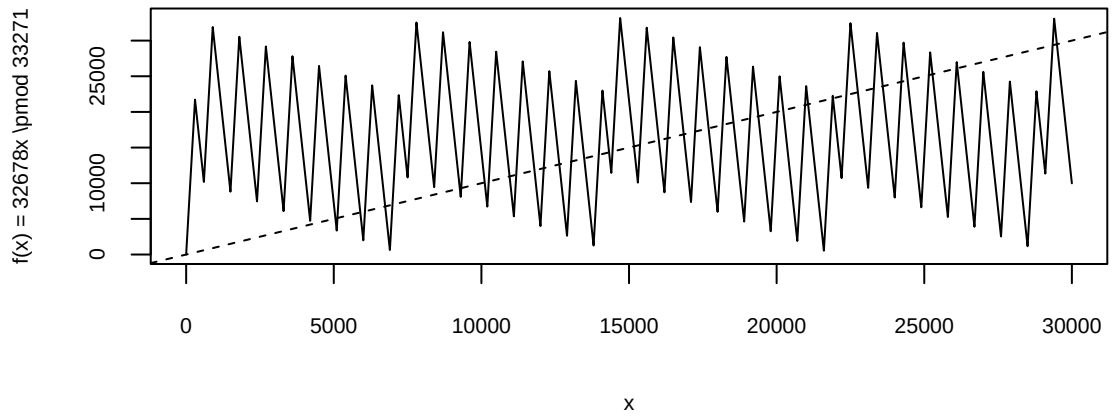


Figure 2.4: An example of a nonlinear function with jumps.

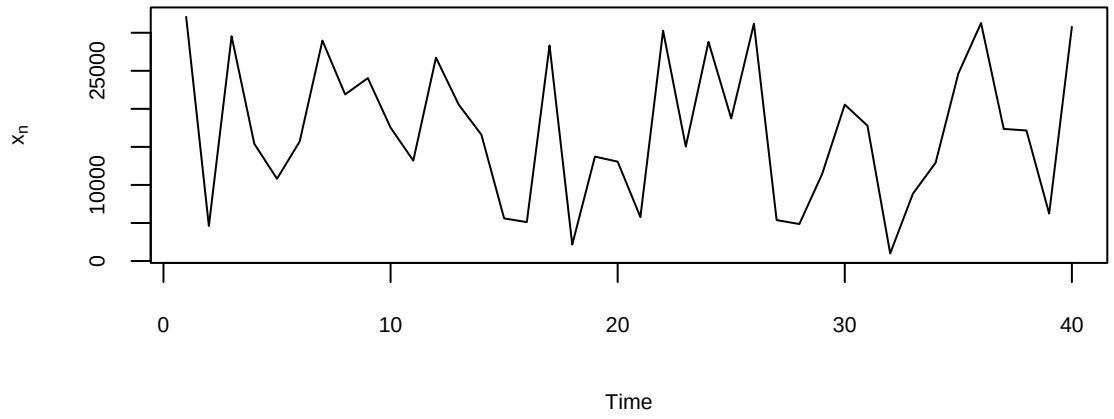


Figure 2.5: Trace plot of the first 40 numbers coming from the nonlinear function with jumps.

The quality of the generator depends on the values of the parameters a, c and m. Small values of m are not recommended, since there can be no more than m distinct values generated according to the method.

In fact, if the parameters are not chosen well, it is possible for the sequence of values to repeat or exhibit cycling before the full set of m values is obtained. Because of the modular arithmetic used in the computation, the longest cycle that could be hoped for is of length m. It is easy to cause cycling prematurely. Good generators have maximal cycle length; in other words, they do not cycle before the mth number is generated.

Example 2.5 *The code below produces 6 pseudorandom numbers based on the multiplicative congruential generator:*

$$x_n = 7x_{n-1} \ mod \ 32$$

$$u_n = x_n/32$$

with initial seed $x_0 = 2$.

```
random.number <- numeric(6)   # the output
                              # will be stored here
random.seed <- 2
for (j in 1:6) {
     random.seed <- (7 * random.seed) %% 32
```

```
     random.number[j] <- random.seed/32
 }
```

The first 6 values in the sequence are

```
random.number[1:6]

## [1] 0.4375 0.0625 0.4375 0.0625 0.4375 0.0625
```

Note that only 2 distinct values were obtained, since $7 \times 2 = 14$ *and* $14 \times 7 = 98$*, the latter being 2 modulo 32. The cycle length is 2, which is much less than the 32, we might have hoped for.*

The next example gives a generator with somewhat better behaviour.

Example 2.6 *The code below produces 30268 pseudorandom numbers based on the multiplicative congruential generator:*

$$x_n = 171x_{n-1} \; mod \; 30269$$

$$u_n = x_n/30269$$

with initial seed $x_0 = 27218$*.*

```
random.number <- numeric(30268)   # the output
                                  # will be stored here
random.seed <- 27218
for (j in 1:30268) {
     random.seed <- (171 * random.seed) %% 30269
     random.number[j] <- random.seed/30269
 }
```

The results, stored in the vector `random.number`*, are in the range between 0 and 1. These are the pseudorandom numbers,* $u_1, u_2, \ldots, u_{30268}$*.*

```
random.number[1:10]

##  [1] 0.7638508 0.6184876 0.7613730 0.1947867 0.3085335
##  [6] 0.7592256 0.8275794 0.5160725 0.2484060 0.4774191
```

```
length(unique(random.number))

## [1] 30268
```

The last calculation shows that this generator did not cycle before all possible numbers were computed.

The concept of relative primeness plays a role in ensuring that the cycle length for a generator is maximal. Two integers are said to be relatively prime if there is no integer greater than one that divides them both (that is, their greatest common divisor is one). For example, 8 and 9 are relatively prime, because the prime factors of 8 are 2 and the prime factors of 9 are 3 The numbers 10 and 12 are not relatively prime, because both numbers are divisible by 2.

2.2.5 Conditions which prevent premature cycling

An interesting mathematical result concerning cycling in linear congruential pseudorandom number generators is given by Knuth (1997). It states that the linear congruential sequence defined by m, a, c and x_0 has cycle length m if and only if the following hold:

1. c is relatively prime to m.
2. $b = a - 1$ is a multiple of p, for every prime p dividing m;
3. b is a multiple of 4, if m is a multiple of 4.

Example 2.7 *If $m = 8$, $a = 5$ and $x_0 = 3$, and $c = 3$. Then, $b = 4$ and the sequence is*

$$3, 2, 5, 4, 7, 6, 1, 0, 3$$

which has a cycle of length 8.

The theorem specifies a necessary and sufficient condition for obtaining a maximal cycle, so if the necessary condition does not hold, cycling will occur prematurely as in the next example.

Example 2.8 *If $m = 9$, $a = 5$ and $x_0 = 3$, and $c = 3$. Then the sequence is*

$$3, 0, 3$$

which has a cycle of length 2. If $x_0 = 1$, the sequence is

$$1, 8, 7, 2, 4, 5, 1$$

which has cycle length 6.

2.2.6 *Implementation*

The following R function implements the linear congruential method:

```
rlincong <- function(n, m = 2^16, a = 2^8+1 , c = 3, seed) {
    x <- numeric(n)
    xnew <- seed
    for (j in 1:n) {
        xnew <- (a*xnew + c)%%m
        x[j] <- xnew
    }
    x/m
}
```

Default settings are chosen to satisfy conditions of the theorem: $c = 3$ and m are relatively prime since they do not share prime factors, $a - 1$ is a multiple of 2 which is the only prime which divides m, and b is a multiple of 4 (m is a multiple of 4).

Example 2.9 *The following verifies the cycle length at the default settings and with seed 372737:*

```
u <- rlincong(2^16-1, seed = 372737)
u[1:4]

## [1] 0.6914673 0.7071381 0.7345276 0.7736359

length(unique(u)) - (2^16 - 1)

## [1] 0
```

A full cycle was achieved, which is what the theorem predicts.

2.2.7 Are the simulated numbers adequate for the problem at hand?

Generally, simpler problems will make fewer demands on the quality of the numbers generated, while complex problems such as those arising in theoretical physics or genomics may be too demanding for even the best of the currently available generators. Choice of seed turns out to be surprisingly critical.

Consider the generator based on

$$x_n = 7x_{n-1} \bmod 17.$$

Using $x_0 = 1$, we obtain the following values in the sequence before it begins to cycle:

```
##  [1]  7 15  3  4 11  9 12 16 10  2 14 13  6  8  5  1
```

(Warning! This is not a generator that should be seriouly considered in practice.)

2.2.8 Tossing two fair coins

We will use the rule that a 'Head' (H) is generated whenever the generated value is less than 9, and otherwise a 'Tail' (T) is generated. Thus, we could use the above sequence to generate the following pattern of heads and tails:

```
##  [1] "H" "T" "H" "H" "T" "T" "T" "T" "T" "H" "T" "T" "H" "H"
## [15] "H" "H"
```

We only require a single consecutive pair of coin tosses, not the entire sequence. Thus, if we request only 2 values from the generator and seed it with the value 1, we get an H-T outcome, while if we seed with the value 7, we get a T-H outcome, and so on.

Seeds and resulting outcomes (pairs of coin tosses):

```
##  [1]  7 15  3  4 11  9 12 16 10  2 14 13  6  8  5  1
```

```
##  [1] "H T" "T H" "H H" "H T" "T T" "T T" "T T" "T T" "T H"
## [10] "H T" "T T" "T H" "H H" "H H" "H H" "H H"
```

The frequency distribution for the outcomes is:

```
##
## H H H T T H T T
##   5   3   3   5
```

If one chooses the seed randomly from the set $\{1, 2, \ldots, 16\}$, a pair of heads will occur with probability 5/16 as is the case for a pair of tails. Thus, the generator will give a biased result for this simple problem.

Notice that if one seeds the generator with one of $\{4, 11, 9, 12, 2\}$, a T-T pair will result, while seeding with one of $\{13, 6, 8, 5, 15\}$ will yield an H-H outcome. Removing seeds at the extremes (i.e. either too large or too small) is a simple general strategy that often leads to improved performance. Thus, we could disqualify seeds 2, 4, 13 and 15. Choosing any other seed will result in a pair of coin toss outcomes that exactly follows the required probability distribution.

2.2.9 Tossing three fair coins

```
##  [1] "H T H" "T H H" "H H T" "H T T" "T T T" "T T T" "T T T"
##  [8] "T T H" "T H T" "H T T" "T T H" "T H H" "H H H" "H H H"
## [15] "H H H" "H H T"
```

Frequency distribution of outcomes:

```
##
## H H H H H T H T H H T T T H H T H T T T H T T T
##     3     2     1     2     2     1     2     3
```

Equally likely outcomes are assured if only one occurrence of each outcome is allowed. Restricting the possible seeds to the set $\mathbf{1}, 3, 6, 7, 9, 12, 15, 16\}$ will perfectly produce a set of three independent coin tosses.

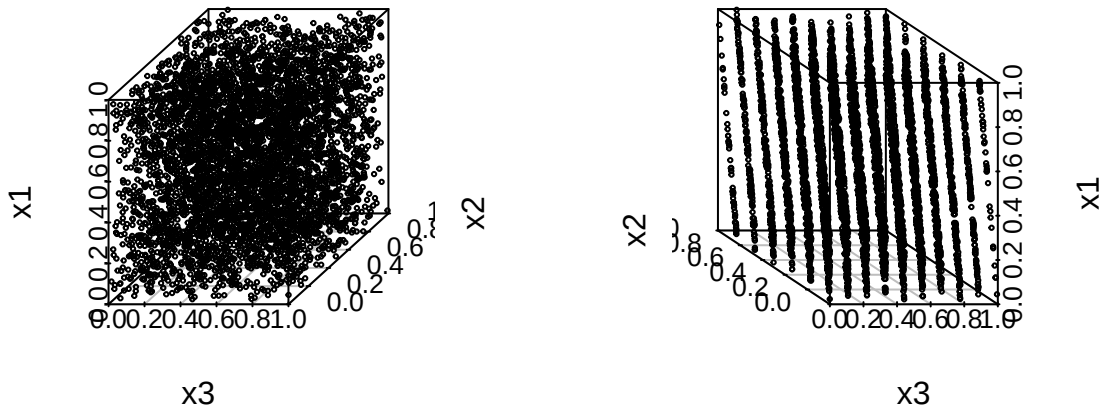
There is no way to produce a sequence of four independent coin tosses.

This example highlights the importance of choosing a seed appropriately. In practical situations, this is a difficult problem. The advice given by **?** remains useful.

2.2.10 "Random numbers fall mainly on the planes" (Marsaglia, 1968)

The RANDU pseudorandom number generator is a multiplicative congruential generator with $a = 65539$ and $m = 2^{31}$.

Scatterplots of consecutive triples of points:



All linear congruential generators have more or less severe forms of this property.

2.3 Testing generators

Traditionally, there were two essential requirements for a good sequence of pseudorandom numbers. First, the distribution of numbers should be uniform on the interval $[0, 1]$. Second, the successive values should be sequentially independent. That is, it should not be possible to predict values in the sequence from other values. In fact, the paper by O'Neill (2014) summarizes the modern goals of random number generators succinctly. Desirable features are

- Speed
- Statistical accuracy
- Long cycle length

- Efficient use of processor
- Portability
- Reproducibility
- Security; robustness against attacks

A number of tests have been developed to investigate the quality of pseudorandom number generators. These include

- Diehard battery of tests
- BigCrush
- TestU01

These are all collections of mainly statistical tests. One exception is the spectral test which examines the minimal distance between hyperplanes in successive dimensions (RANDU does poorly on this test in 3 dimensions.)

2.3.1 *Histogram*

The most straightforward check on the uniform distribution assumption is to plot a histogram of a sample coming from a random number generator. A perfect-looking rectangular distribution is often a symptom of too much order in the generator, while gaps in the histogram are also an indicator that the generator is failing to meet the uniform distribution requirement.

Example 2.10 *Sequences of pseudorandom numbers are stored in* `x1`, `x2` *and* `x3`*:*

```
x1 <- rng(1000, a = 32377, m = 32378); x2 <- rng(1000, a = 41, m = 2000)
    x3 <- runif(1000)
par(mfrow=c(1,3)); hist(x1); hist(x2); hist(x3)
```

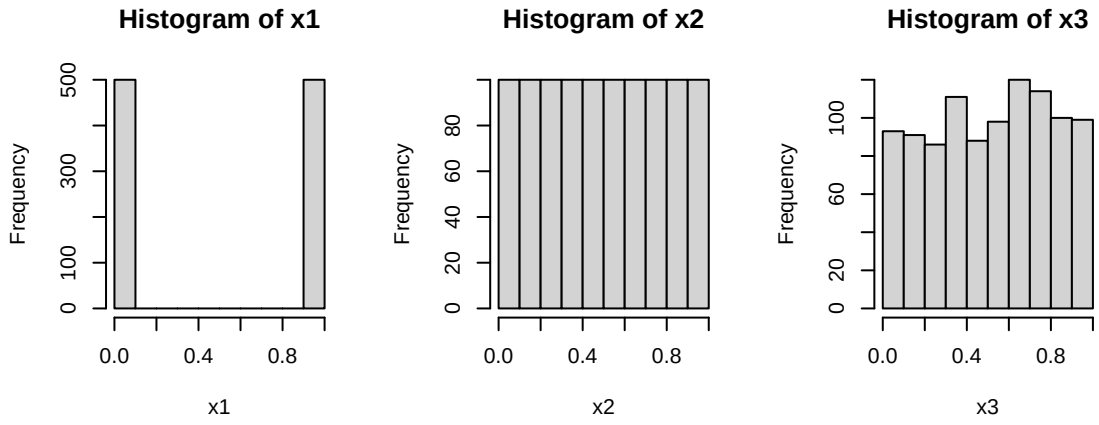


Figure 2.6: Histograms of three sets of pseudorandom numbers.

Figure 2.6 displays the histograms of the 3 sequences. `x1` *fails;* `x2` *and* `x3` *both appear to be uniform, though the middle histogram is actually too perfect to be believable.*

Sometimes the use of a chi-squared goodness-of-fit test is advocated, but this would usually be a waste of effort, since a glance at the histogram is all that is really needed to check for uniformity. Furthermore, the more important and difficult issue is related to the independence of the sequence of values. More care is required to assess this aspect of a generator.

2.3.2 *Visualizing RNG output with a lag plot*

We can get an idea of whether we are generating independent random numbers by plotting successive pairs of numbers - a lag 1 plot. Specific patterns indicate that the generator is not producing independent numbers. If you see points lining up, your generator is failing.

Example 2.11 *In the first example, we see points lying on clearly separated lines in the lag plot displayed in Figure 2.7. This generator is very poor.*

```
x <- rng(100,a=15, m=511,seed=12)
lag.plot(x, do.lines=FALSE)
```

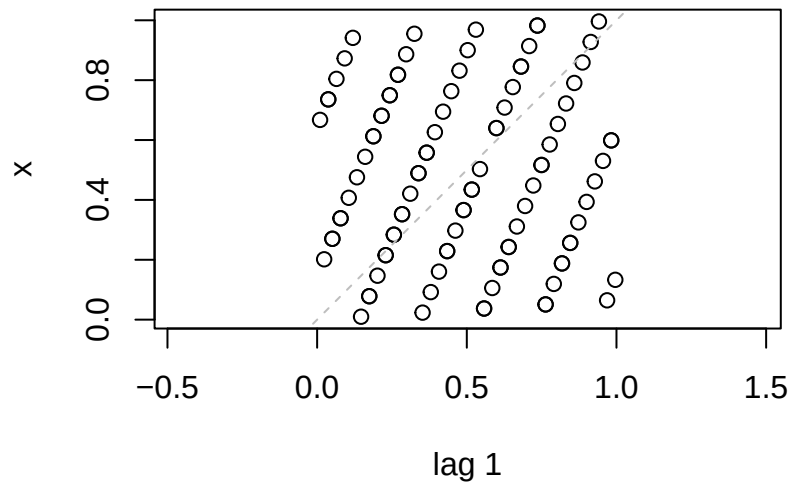


Figure 2.7: Lag plot for a pseudorandom number generator with bad properties.

Example 2.12 *By changing the a parameter, we get the result plotted in Figure 2.8.*

```
x <- rng(100,a=31, m=511,seed=12)
lag.plot(x, do.lines=FALSE)
```

Plotted points appear to be more random in this second example.

Example 2.13 *In our third example, we take a much larger value of m, together with a value of a which is in the order of the square root of m. This kind of combination can lead to better behaviour, and the lag plot picture in Figure 2.9 confirms this.*

```
x <- rng(100, a=171, m=30269, seed=12)
lag.plot(x, do.lines=FALSE)
```

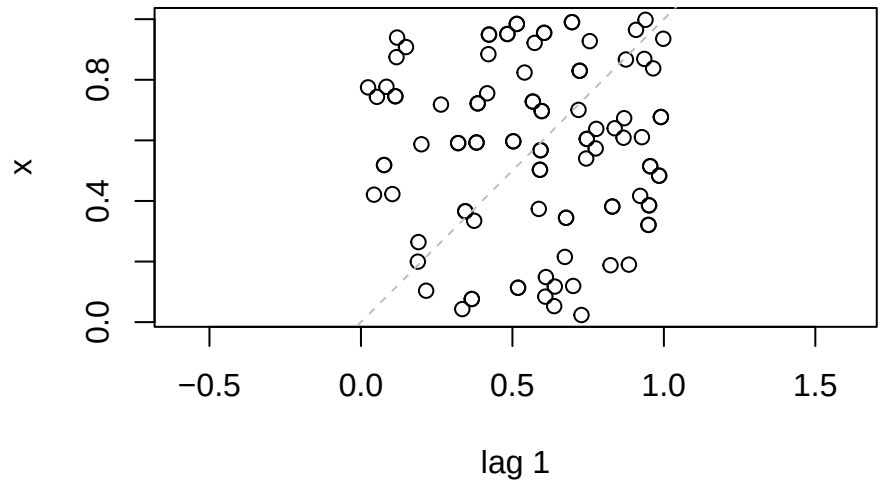


Figure 2.8: Lag plot for a pseudorandom number generator with small m but relatively good properties for very short sequences.

2.3.3 *Autocorrelation*

The autocorrelation function, ACF, begins to numerically summarize what can be observed graphically on a lag plot. The autocorrelation at a particular lag, say m, is the correlation between the $n+m$th value and the nth value, or more precisely, the correlation between the vector $(x_1, \ldots, x_{n-m})$ and the vector $(x_m, x_{m+1}, \ldots, x_n)$.

Autocorrelations, like correlations, can take values between -1 and 1, where positive values are indicative of the plotted points scattering about a line of positive slope, and negative values are indicative of the plotted points scattering about a line of negative slope.

Example 2.14 *Figure 2.10 shows the first 6 lag plots for one of the sequences generated earlier. In other words, the figures show plots of the $n+1$st elements versus the nth, $n+2$st versus the nth, $n+3$st versus the nth and so on.*

```
lag.plot(x2, lag=6)
```

At lags 1, 2, 3, 4 and 6, it would be hard to predict the current value of x2, but the lag 5 plot shows that the current value of x2 depends a lot on the value 5 time units earlier.

The autocorrelations for the first 5 lags are:

```
acf(x2)$acf[2:6]  #
```

```
## [1] -0.0328  0.0049 -0.1090 -0.1097  0.4575
```

The values of this autocorrelation function are plotted in Figure 2.11. Note the size of the 5th one. This says that every 5th value of the sequence is dependent.

The autocorrelations for the first 5 lags for values coming from `runif()` *are:*

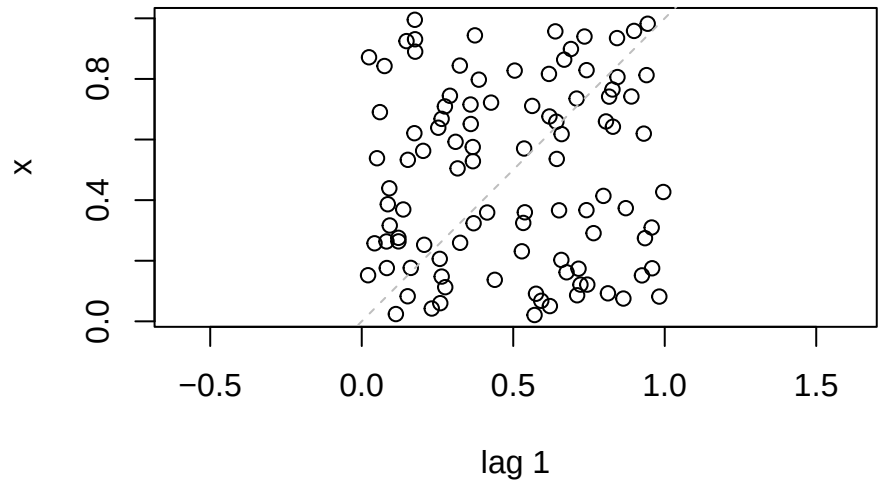


Figure 2.9: Lag plot for a pseudorandom number generator with moderate m and relatively good properties for short sequences.

```
acf(x3)$acf[2:6]   #
```

```
## [1] -0.012205 -0.001414 -0.012534  0.000379  0.003676
```

Again, the autocorrelation function is plotted in Figure 2.12. All ACF values checked are small. This sequence passes this test.

Note that a severe deficiency of checking the autocorrelations is that they only detect linear forms of dependence and can miss nonlinear dependencies. Thus, they really only serve as a quick way to check many lags at once; the lag plots have the advantage of highlighting nonlinear dependencies, if they are there. Both methods will fail to show more complex dependence structures, where, for example, values can be predicted nonlinearly by a combination of earlier values in the sequence.

Example 2.15 *In the 1960's, the RANDU pseudorandom number generator was very popular. It was a multiplicative congruential generator with* $a = 65539$ *and* $m = 2^{31}$.

We can generate numbers from this sequence using

```
n <- 5000
RANDU <- numeric(n)
x <- 123
for (i in 1:n) {
  x <- (65539*x) %% (2^31)
  RANDU[i] <- x / (2^31)
}
```

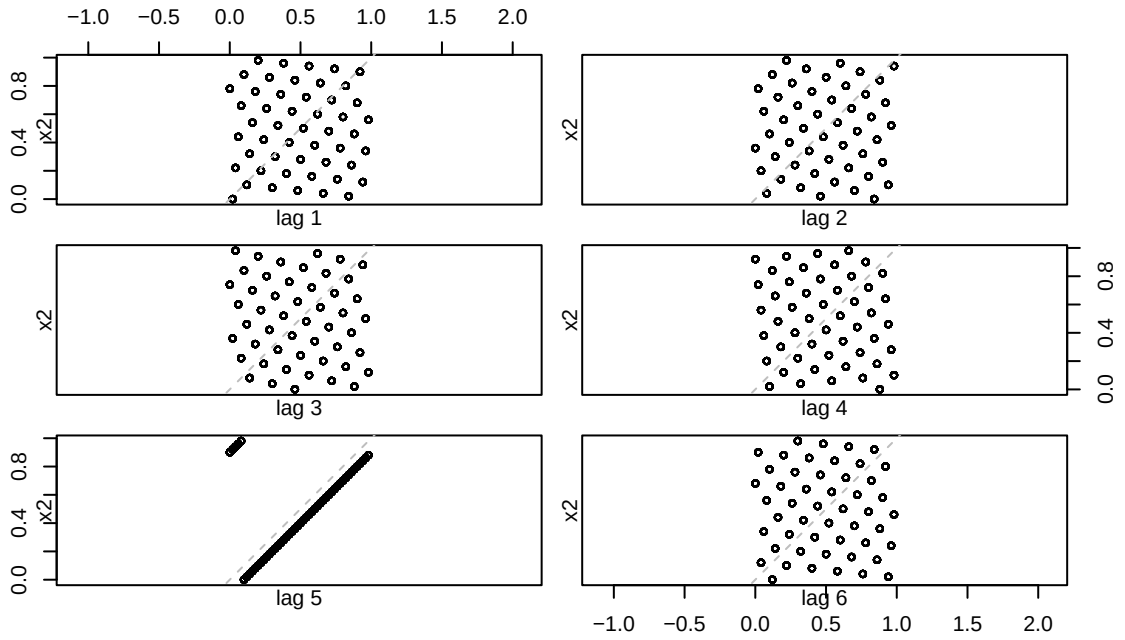


Figure 2.10: Lag plots for the first 6 lags.

```
par(mfrow=c(1,2))
hist(RANDU)
acf(RANDU)
```

According to these simple checks, there does not appear to be a problem, and this is a reason for its temporary popularity. However, upon more rigorous checking, a serious deficiency was discovered, when one plots a 3-dimensional version of the lag plot. In other words, plot u_{n+2} against u_{n+1} and u_n. The following code sets up the vectors that make up the 3d lag plot:

```
A <- diag(rep(1,n-1)); A <- rbind(rep(0, n-1), A)
A <- cbind(A, rep(0, n))
lagvectors <- matrix(RANDU, nrow=n)
m <- 2
for (j in 1:m) {
    lagvectors <- cbind(lagvectors, A%*%lagvectors[,j])
}
lagvectors <- data.frame(lagvectors)
names(lagvectors) <- c(paste("x", 1:(m+1), sep=""))
lagvectors <- lagvectors[-(1:m),]
```

Here, we have set up an $n \times n$ matrix A for which Ax yields a vector of length n whose first element is 0 and whose remaining elements are $x_1, \ldots, x_{n-1}$. Thus, A^2x is a vector whose first two elements are 0's and whose remaining elements are $x_1, \ldots, x_{n-2}$. The vectors $x_1 = x, x_2 = Ax, x_3 = A^2x$ are the basis of the 3-dimensional lag plot pictured, from two different orientations, in Figure 2.14. Here, we have used the `scatterplot3d` *function in the scatterplot3d package (Ligges and Mächler, (2003).*

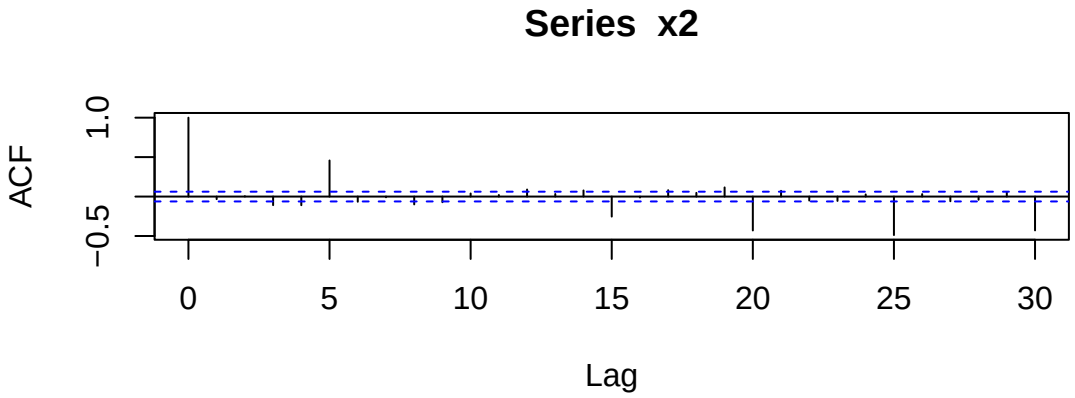


Figure 2.11: Autocorrelations for the x2 sequence.

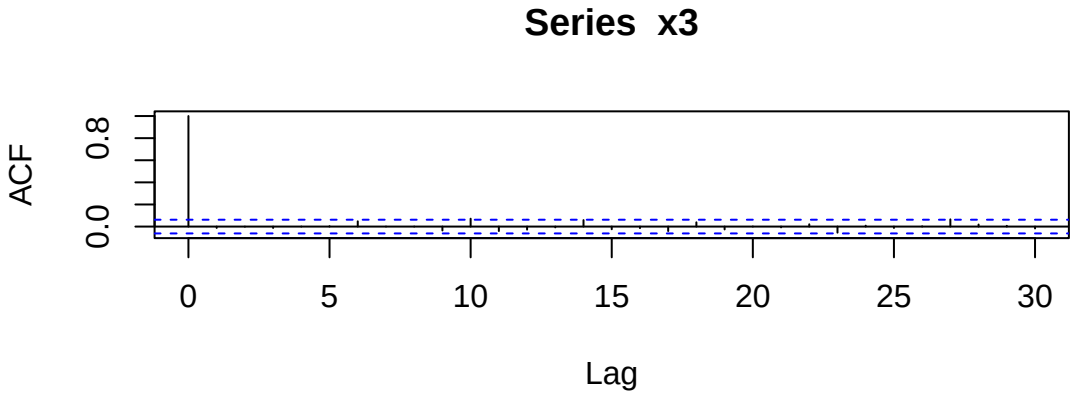


Figure 2.12: Autocorrelations for the x3 sequence.

```
library(scatterplot3d)
par(mfrow=c(1,2))
scatterplot3d(lagvectors$x3,lagvectors$x2, lagvectors$x1,
    xlab="x3", ylab="x2", zlab="x1", cex.symbols=.3)
scatterplot3d(lagvectors$x3,lagvectors$x2, lagvectors$x1,
    xlab="x3", ylab="x2", zlab="x1", angle=150, cex.symbols=.3)
```

The change in perspective is important, since most views of the data do not reveal anything other than an apparent random scatter of points - under the given 2-dimensional projection. The perspective taken in the right panel of Figure 2.14 reveals the difficulty with RANDU: it only produces u_n, u_{n+1}, u_{n+2} triples lying on a small number of planes in 3-dimensional space.

Better generators provide successive triples that do a better job of filling 3-dimensional space. In fact, one also desires a generator that will provide successive quadruples that leave only small gaps in 4-dimensional space, and more generally, successive m-tuples that leave only small gaps in m-dimensional space. The spectral test, which lies beyond the scope of this book, is designed to find these deficiencies at a given number of dimensions.

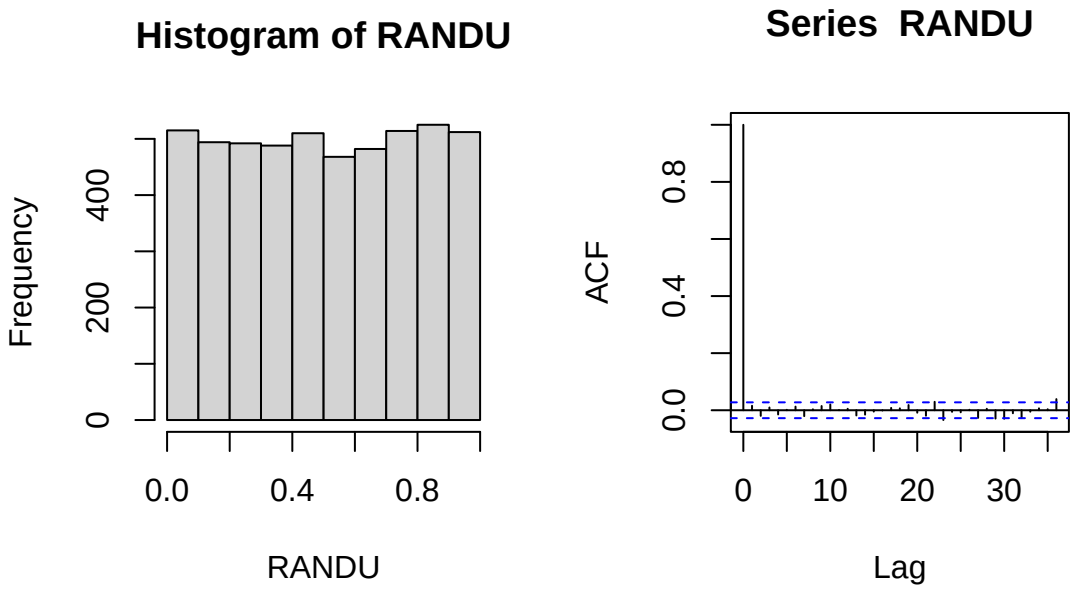


Figure 2.13: Histogram and ACF plots for a sequence of RANDU numbers.

2.3.4 Random forest testing of a pseudorandom number generator

The `randomForest` function in the *randomForest* package (Liaw and Wiener, 2002) can be used to set up a quick and simple approximation to the spectral test. The essential idea behind this test is to set up a flexible prediction model for successive elements of a sequence generated by a pseudorandom number generator, given m previous values. If the predictions are consistently inaccurate, the generator can be judged adequate; when the predictive model is sometimes successful, the generator should be judged a failure.

A quick review of the nature of regression trees and their extension as random forests is necessary before we describe an implementation of the random forest testing approach for pseudorandom numbers.

Regression trees and random forests

A regression tree is a flexible or nonparametric approach to predictive modelling which is particularly useful when there are a relatively large number of possible predictors and where the nature of the relationship between the response and the predictors is very much unknown and not likely linear. The *rpart* package (Therneau and Atkinson, 2018) implements a recursive partitioning technique which can produce both classification trees and regression trees. A classification tree is useful in the case where the response variable is categorical, for example, binary. Classification trees offer a flexible alternative to logistic regression. Regression trees offer a flexible alternative to multiple regression.

Example 2.16 *Consider the data in* `table.b3` *of the MPV package (Braun and MacQueen, 2019). This data set concerns gas mileage,* `y`*, for a number of cars, together with information on 11 other variables. In particular, we note that* `x10` *represents weight and* `x2` *represents horsepower. We can fit the default regression tree to these data using*

```
library(rpart); library(MPV)
mpg.tree <- rpart(y ~ ., data = table.b3)
```

The tree, plotted in Figure 2.15, is the result of executing the following code.

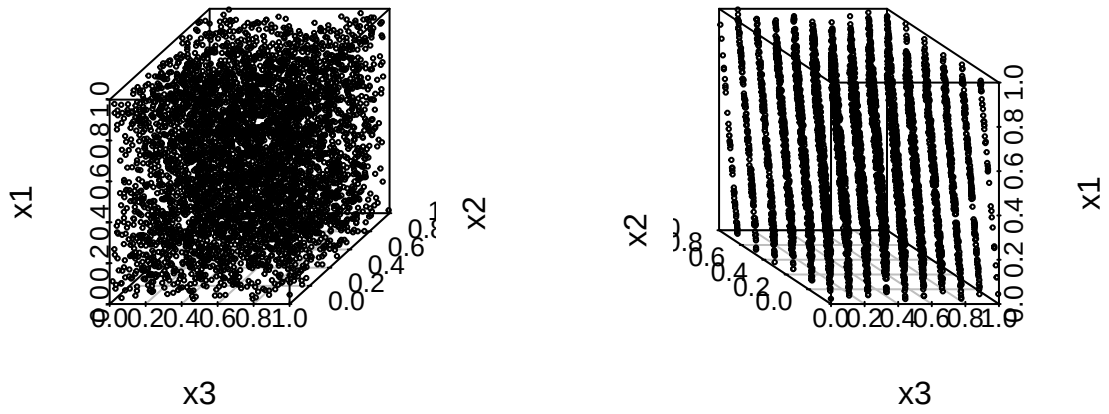


Figure 2.14: 3-dimensional lag plot of a sample from the RANDU simulator, from two different perspectives.

```
par(xpd = TRUE)
plot(mpg.tree, compress=TRUE)
text(mpg.tree, cex=.5, use.n = TRUE)
```

The tree indicates how the partitioning or splitting of the predictor space was undertaken. For weights (`x10`*) less than 2678 pounds, the gas mileage was predicted to be 29.76 miles per gallon. This split would have been chosen to minimize the prediction error. Then an additional split was made, for the data set where weights which are at least 2678 pounds. In this case, when the horsepower (*`x2`*) is less than 144, the gas mileage is predicted to be 20.3, and the prediction is 15.72 when the horsepower is at least 144. Again, this part of the predictor space was partitioned in order to minimize prediction error. The splitting process was terminated, based on a trade-off between predictive precision and model complexity. Models that have too many splits or branches tend to over-fit the data.*

As the name implies, a random forest is a random collection of trees. The trees are essentially constructed from random samples, taken with replacement, from the original sample of observations. This is an example of a technique called bootstrapping. By averaging the predictions over the entire set of trees, improved stability can often be achieved.

Example 2.17 *We can apply the random forest approach to the car gas mileage data as follows:*

```
library(randomForest)
mpg.rf <- randomForest(y ~ ., data = table.b3[,-4])
```

Figure 2.16 displays a plot of the actual gas mileages against predictions for both the random forest predictions and the tree predictions. Whether the random forest predictions are better is not necessarily clear. What is clear is that the predictions are somewhat finer grained and somewhat more flexible.

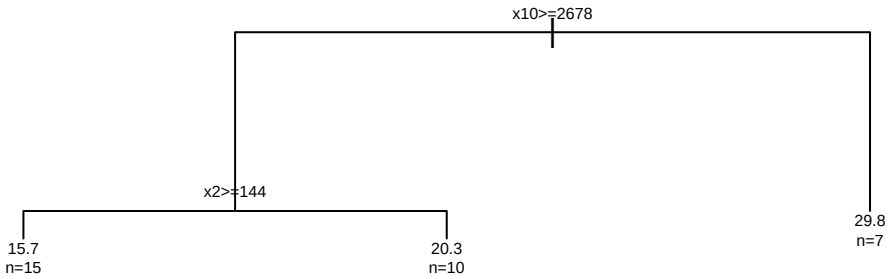


Figure 2.15: Default regression tree for car gas mileage data.

```
par(mfrow=c(1,2), mar=c(4, 4, 1, 1))
plot(table.b3$y ~ predict(mpg.rf), ylab="Actual mpg",
   xlab="Predicted mpg")
plot(table.b3$y ~ predict(mpg.tree), ylab="Actual.mpg",
   xlab ="Predicted mpg")
```

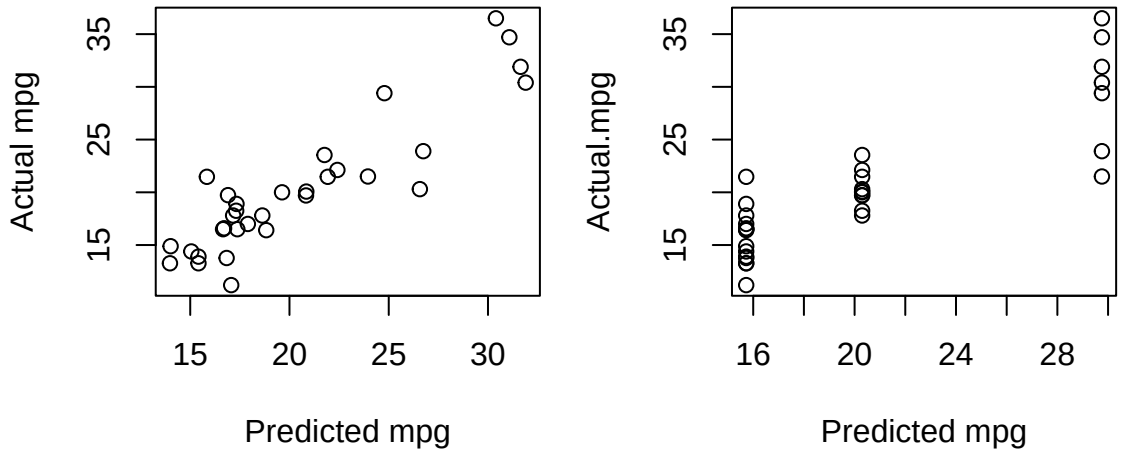


Figure 2.16: Random forest (left panel) and tree-based (right panel) predictions of car gas mileage.

Testing pseudorandom numbers with random forest prediction

The function `rftest()` can be used to carry out the test for a given pseudorandom number sequence, coming from the generator to be tested. Typically, as in the spectral test, one supplies a sequence of m values which represent the dimensionality of the space to be "filled" by the successive m-tuples of sequence values. The function constructs the m vectors as in the RANDU example of Section 2.3.3, and the random forest is then used to set up

a predictive model for values of x_{n+m}, given $x_{n+m-1}, x_{n+m-2}, \ldots, x_n$. The fitting is actual done on one-half of the data, the so-called training set. The remaining half of the data, the so-called test set, is plugged into the fitted model to obtain predictions. The actual values of x_{n+m} are plotted against the predictions, first using the training set – internal validation and then using the test set – external validation. In both cases, a scatter plot of the actual values against the predicted values is obtained, with a least-squares line overlaid. A line with positive slope, particularly on the second plot, is an indicator of failure for the generator. One would not expect a line with a substantial negative slope, since the poor predictivity from the random forest should only yield random predictions and not predictions that are negatively correlated with the actual values. Thus, a line with non-positive slope should be interpreted as a success for the generator. This is coded up in the function `rftest` which is available in the MPV package (Braun and MacQueen, 2019).

```
library(MPV) # The function rftest is in the MPV package
```

Example 2.18 *We start with the cosine sequence discussed in Section* **??** *to show why such a generator is not in practical use. The results of the random forest test are pictured in Figure 2.17 using* $n = 500$ *and the default of* $m = 5$ *dimensions.*

```
for (n in 1:500) {
    x <- cos(30*x)
    prnumbers[n] <- x
}
rftest(prnumbers)
```

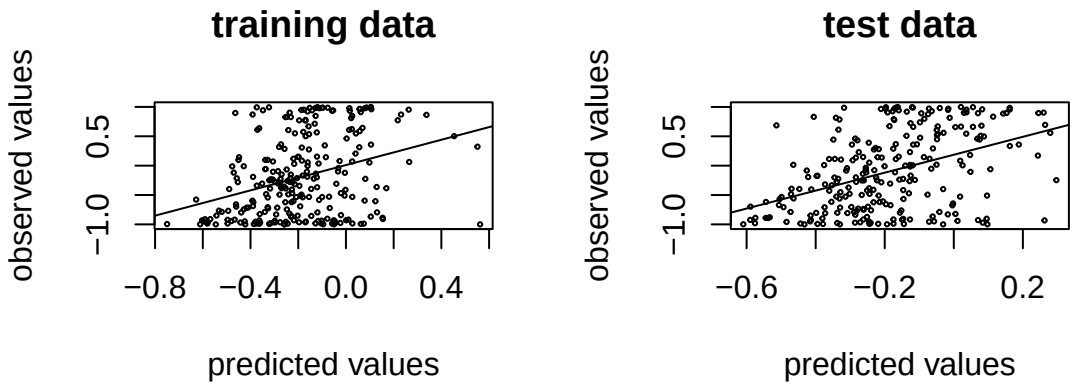


Figure 2.17: Random forest test for the cosine sequence of Section **??**.

Observe that in both plots, the least-squares line has a substantial positive slope. The random forest model is making excellent predictions of x_{n+5} *based on the previous 5 observations. This cosine mapping would not be useful in producing good approximations to random numbers.*

In practice, we should use as large a sample as is practical, and we should look for trouble over a sequence of m values, starting with 1. In the case of the spectral test, one does not normally go beyond 8 dimensions, but the random forest approximation can be run at higher dimensions without much difficulty.

Example 2.19 *We will check the quality of the default generator in R, using the random forest test, using* $n = 5000$*, and* $m = 1$ *and* $m = 10$*. It is an easy exercise to try intermediate values of* m*.*

```
u <- runif(5000)
```

Execution of the following scripts yields the plots displayed in Figures 2.18 and 2.19. In both cases, we see that the least-squares lines are effectively constant; in some cases, there is the appearance of a slightly negative slope, which we have discussed earlier as an indication that the random forest is really not able to deliver a predictive model. These plots confirm that, at least for small simulation problems, the default simulator in R is going to work well. Larger values of n should be considered before making conclusions about larger scale simulations.

```
rftest(u, m = 1)
```

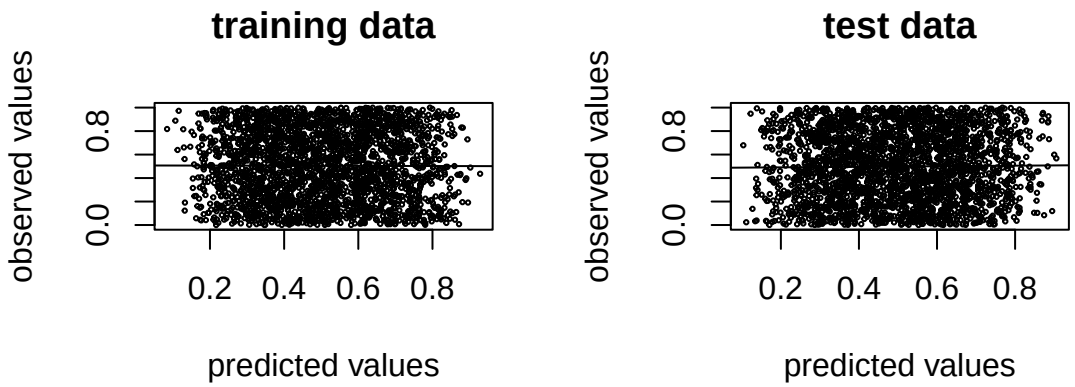


Figure 2.18: Random forest test for the default generator in R, using $m = 1$.

```
rftest(u, m = 10)
```

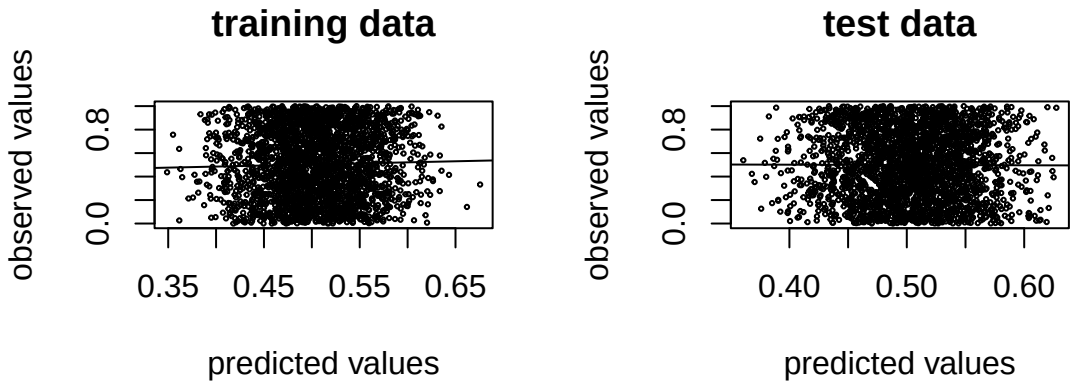


Figure 2.19: Random forest test for the default generator in R, using $m = 10$.

We saw earlier that the RANDU generator has a serious deficiency. Can the random forest test detect this problem?

Example 2.20 *Since the issue for RANDU occurs when $m = 2$, we will apply the random forest test using this value of m.*

```
rftest(RANDU,  m = 2)
```

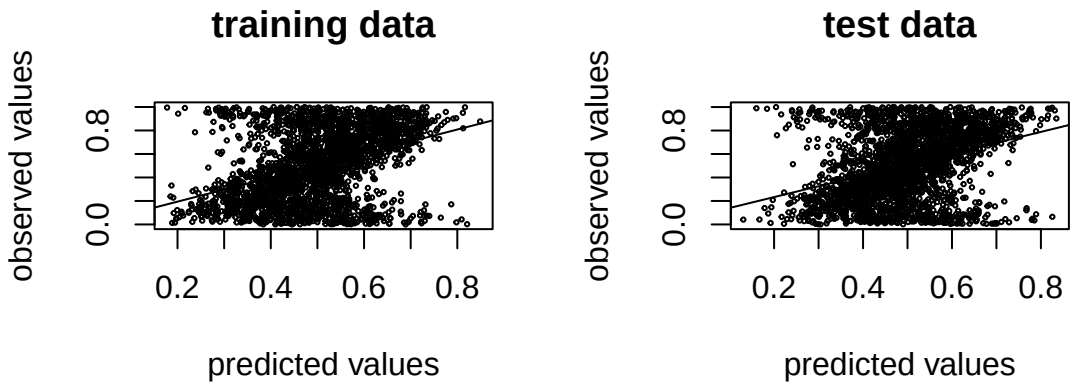


Figure 2.20: Random forest test for the RANDU sequence, using $m = 2$.

Figure 2.20 displays the results. As expected, the fitted least-squares line relating the actual sequence values with the random forest predictions has an obviously positive slope, indicating that the random forest is able to find a relatively good predictive model for values in this sequence, based on the preceding two values in the sequence. This result is consistent with the earlier analysis that indicates that the RANDU generator will not produce good unpredictable numbers.

2.4 Shuffling

Analogous to shuffling a deck of cards, there is a shuffling technique for reducing sequential dependence in a given sequence of pseudorandom numbers, x.

The shuffling algorithm uses an auxiliary table $v(1), v(2), \ldots, v(k)$, where k is some number chosen arbitrarily, usually in the neighborhood of 100. Initially, the v vector is filled with the first k values of the x sequence and an auxiliary variable y is set equal to the $(k+1)st$ value. The steps are:

Extract the index j . Set $j \leftarrow ky/m$, where m is the modulus used in the sequence x; that is, j is a random value, $0 \leq j < k$, determined by y.

Exchange. Set $y \leftarrow v[j]$, return y, and set $v[j]$ to the next member of the sequence x.

The following is an R implementation of shuffling, using the built-in generator to generate the auxiliary sequence.

```
shuffle <- function(n, k = 100, x = runif(n)) {
    v <- x[1:k]
    y <- x[k+1]
    xnew <- numeric(n - k)
    i <- 1
    while (n > k) {
        j <- floor(k*y)+1
        y <- v[j]
```

```
        xnew[i] <- y
        i <- i + 1
        v[j] <- x[k+1]
        x[k+1] <- x[n]
        n <- n-1
    }
    c(v, xnew)
}
```

Example 2.21 ***Shuffling a deck of 52 playing cards.***

We first define a vector containing the 52 different playing cards, using a factor called `cards` *with 52 levels:*

```
a <- 1:52
suits <- c("Spades", "Hearts", "Diamonds", "Clubs")
values <- c(2:10, "J", "Q", "K", "A")
cards <- factor(a)
levels(cards) <- as.vector(outer(values, suits, paste))
cards # sorted
```

```
##  [1] 2 Spades    3 Spades    4 Spades    5 Spades
##  [5] 6 Spades    7 Spades    8 Spades    9 Spades
##  [9] 10 Spades   J Spades    Q Spades    K Spades
## [13] A Spades    2 Hearts    3 Hearts    4 Hearts
## [17] 5 Hearts    6 Hearts    7 Hearts    8 Hearts
## [21] 9 Hearts    10 Hearts   J Hearts    Q Hearts
## [25] K Hearts    A Hearts    2 Diamonds  3 Diamonds
## [29] 4 Diamonds  5 Diamonds  6 Diamonds  7 Diamonds
## [33] 8 Diamonds  9 Diamonds  10 Diamonds J Diamonds
## [37] Q Diamonds  K Diamonds  A Diamonds  2 Clubs
## [41] 3 Clubs     4 Clubs     5 Clubs     6 Clubs
## [45] 7 Clubs     8 Clubs     9 Clubs     10 Clubs
## [49] J Clubs     Q Clubs     K Clubs     A Clubs
## 52 Levels: 2 Spades 3 Spades 4 Spades 5 Spades ... A Clubs
```

We can shuffle the cards using our `shuffle()` *function:*

```
a <- as.numeric(cards)/53
par(mfrow=c(3,3))
for (i in 1:9) {
    ts.plot(a)
    a <- shuffle(52, 10, a)
}
```

```
cards[a*53] # shuffled
```

```
##  [1] J Hearts    10 Spades   K Spades    K Diamonds
##  [5] 4 Hearts    6 Spades    3 Clubs     2 Hearts
##  [9] 5 Spades    7 Spades    8 Clubs     8 Hearts
## [13] 10 Diamonds 6 Clubs     A Spades    Q Clubs
## [17] 9 Diamonds  9 Spades    J Spades    A Clubs
## [21] J Diamonds  A Diamonds  9 Hearts    5 Hearts
```

```
## [25] 6 Hearts    7 Hearts    3 Diamonds  2 Diamonds
## [29] 10 Hearts   7 Clubs     9 Clubs     Q Spades
## [33] 7 Diamonds  4 Clubs     4 Diamonds  5 Diamonds
## [37] Q Diamonds  3 Spades    J Clubs     K Clubs
## [41] 6 Diamonds  4 Spades    8 Diamonds  2 Clubs
## [45] Q Hearts    3 Hearts    5 Clubs     8 Spades
## [49] 10 Clubs    A Hearts    2 Spades    K Hearts
## 52 Levels: 2 Spades 3 Spades 4 Spades 5 Spades ... A Clubs
```

Figure 2.21 shows how the original ordering of the numbers from 1 through 52 gradually takes on more of an unpredictable appearance as the number of shuffles increases.

Next, we can ask how many times we should shuffle to obtain a reasonably random ordering of the cards:

```
a <- as.numeric(cards)/53
par(mfrow=c(3,3))
for (i in 1:9) {
    acf(a)
    a <- shuffle(52, 10, a)
}
```

Figure 2.22 displays the autocorrelations for the shuffled numbers. The autocorrelations appear to die down after the numbers have been shuffled 6 or 7 times.

The cross-correlation between an n-vector x and another n-vector y, at lag m essentially measures the correlation between $(x_1, x_2, \ldots, x_{n-m})$ and $(y_m, y_{m+1}, \ldots, y_n)$.

Example 2.22 *We can use the cross-correlation function to see how much dependence occurs between "hands". Figure 2.23 contains graphs of the cross-correlations between the original ordering of the 52 numbers and the orderings following each of 9 successive shuffles of those numbers.*

```
b <- a # b contains the order for the original hand
# a will contain the order for the next 9 shuffles:
par(mfrow=c(3,3))
for (i in 1:9) {
    a <- shuffle(52, 10, a)
    ccf(a, b)
}
```

Comment about weaknesses - see Anderson

Anderson (1990) discusses the disadvantages of shuffling.

2.5 Fast computation

Anderson (1990) provides details on how to vectorize algorithm congruential generators. Use of the `modpower()` function in the *numbers* package (Borchers, 2021) enables us to do this in R.

```
library(numbers)  # loads the numbers package
```

A vectorized version of `rng()` is as follows:

```
rngV <- function(n, a, cc, m, seed, L) {
    x <- numeric(n)
    ai <- numeric(L+1)
    ell <- 0:L
    for (i in ell) {
        ai[i+1] <- modpower(a, ell[i+1], m)
}
    ci <- (cc*cumsum(ai[-(L+1)]))%%m
    ai <- ai[-1]
    x[1:L] <- (ai*seed  + ci)%%m
    M <- ceiling(n/L)
    for (j in 1:(M-1)) {
        seed <- x[j*L]
        x[(j*L+1):((j+1)*L)] <- (ai*seed + ci)%%m
    }
    return(x[1:n]/m)
}
```

The multiplicative congruential generator with modulus $m = 2^{19} - 1$ and $a = 701$ has a cycle length of $2^{19} - 2$ as shown below.

```
length(unique(rngV(524286, 701, 0, 524287, 1, 700)))

## [1] 524286
```

We can check that the vectorized generator produces the exact same sequence as the sequential version by computing the pairwise differences of the results and computing the mean of the absolute difference.

```
mean(abs(rngV(524286, 701, 0, 524287, 1, 700) - rng(524286, 701, 524287)))

## [1] 0
```

The time taken to generate the entire cycle with the vectorized version is about 25% of the sequential version.

```
system.time(rngV(524286, 701, 0, 524287, 1, 700))

##    user  system elapsed
##   0.024   0.000   0.025
```

```
system.time(rng(524286, 701,  524287))

##    user  system elapsed
##   0.132   0.000   0.133
```

2.6 Types of generators

1. Congruential (multiplicative, linear; recursive) generators
2. Multiple-recursive generators (these use $x_{n-1}, x_{n-2}, \ldots, x_{n-k}$ to generate x_n)
3. Modulo 2 (F_2, XOR) linear generators (numbers are produced bitwise)

4. Linear feedback shift register generators (e.g. Mersenne Twister(s)[2]) These are a special case of the F_2 generators
5. Multiply-with-carry generators
6. Combination generators

2.6.1 Combination Generators

Mathematical folklore, hinted at by Wichmann and Hill (1982): if $U_1, U_2, \ldots, U_n$ are independent uniform random variables on $(0, 1)$, then the fractional part of $V = \sum_{i=1}^{n} U_i$ is also uniformly distributed on $(0, 1)$. See also Miller and Nigrini (2006).

Sketch of the Proof:

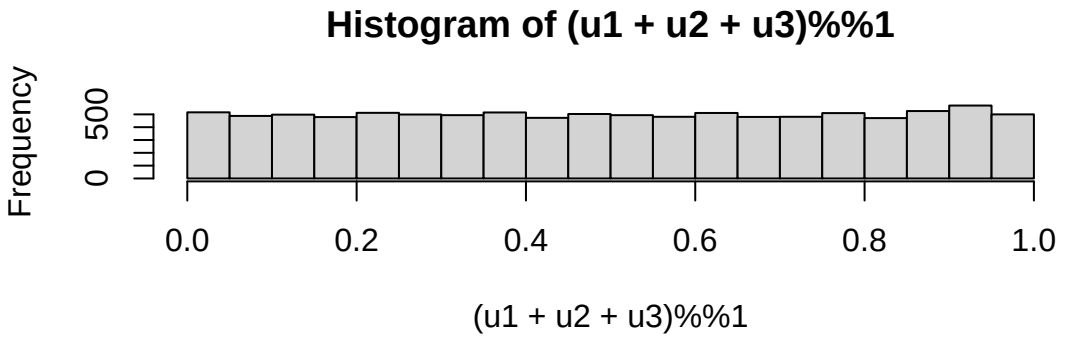
- When $n = 2$, calculate $P(V < v)$ by conditioning on the value of $[U_1 + U_2]$.
- When $n \geq 2$, use induction, with facts like $[v + [z]] = [v] + [z]$.

Details are provided in a later section of this chapter.
Examples of combination generators include Super-Duper and Wichmann-Hill.

2.6.2 Combination Generators

Numerical demonstration (n=3):

```
u1 <- runif(10000); u2 <- runif(10000); u3 <- runif(10000)
hist((u1+u2+u3)%%1)
```



2.6.3 Combination Generators: Wichman and Hill

For $i = 1, 2, \ldots,$

$$\begin{aligned} x_i &= (171 x_i) \bmod 30269 \\ y_i &= (172 y_i) \bmod 30307 \\ z_i &= (170 z_i) \bmod 30323 \\ U_i &= (x_i/30269 + y_i/30307 + z_i/30323) \bmod 1.0 \end{aligned}$$

Example of use:

[2] `runif` in R.

```
RNGkind("Wich")
runif(5)

## [1] 0.437343 0.318333 0.870639 0.953454 0.345549
```

2.6.4 Combined multiple recursive generator

The combined multiple recursive generator (cmrg) of L'Ecuyer (1996) is based on the difference of two underlying generators which are constructed from

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \mod m_1$$

$$y_n = (b_1 y_{n-1} + b_2 y_{n-2} + b_3 y_{n-3}) \mod m_2$$

where $a_1 = 0$, $a_2 = 63308$, $a_3 = -183326$, $b_1 = 86098$, $b_2 = 0$, $b_3 = -539608$, $m_1 = 2^{31} - 1 = 2147483647$ and $m_2 = 2145483479$.

The simulated numbers are then given by

$$z_n = (x_n - y_n) \bmod m_1.$$

Theory: the fractional part of $U_1 - U_2$ is uniformly distributed on $(0, 1)$.

2.6.5 "It's high time we let go of the Mersenne Twister" (Vigna, 2019)

The Mersenne Twister is actually a collection of generators, all of which are some form of shifted F_2 generator. The original version uses $k = 19937$, and since 2^{19937} is a Mersenne prime, it has maximal cycle length: $2^{19937} - 1$.

Problems with the Mersenne Twister have been evident since its inception.

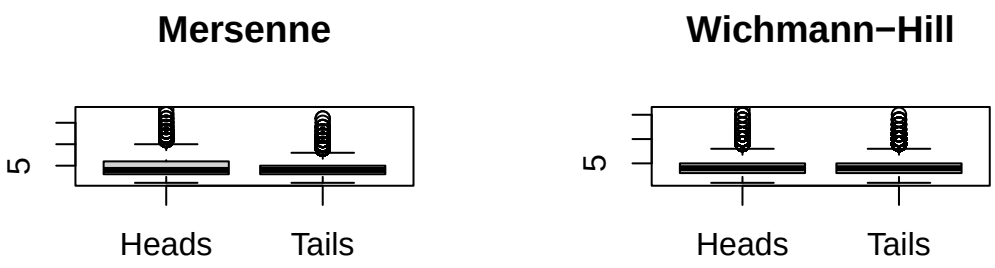
- It fails two statistical tests in the BigCrush test suite.
- It wastes space in the processor cache since k is unnecessarily excessive.
- Much faster generators are available now.
- These and other problems are described by Vigna (2019).

2.6.6 Seeing the problem for ourselves

Vigna (2019) describes a specific example involving the characteristic polynomial of an Erdös-Renyi graph to numerically demonstrate that the generator is producing too many 0's in the trailing bits. A more accessible example is as follows.

Define the random variable X to be the maximum runlength for Heads generated from a sequence of 32 fair and independent coin tosses. For each U generated by the Mersenne Twister, the following steps can be used to carry out the transformation defined by $X = g(U)$.

- Convert U to its binary representation and retain the leading 32 binary digits.
- Return the maximum runlength of 0's in the binary representation.
- Calculate $X_n = g(U_n)$ for $n = 1, 2, \ldots, 10000$ using R's Mersenne Twister and Wichmann-Hill
- Calculate $Y_n = g(1 - U_n)$ (maximum runlengths for Tails)



2.6.7 *Accessing better generators in R*

Wickham (2014) makes a compelling case for the use of the *Rcpp* facility in R to interface with C++ and the GSL library (The GSL Team, 2021) to speed up code, particularly random number generation.

To install *RcppGSL* (Eddelbuettel and Francois, 2022), you need to have a working version of GSL. On a computer running a Linux (Debian) operating system, this can be installed using

```
sudo apt install libgsl-dev
```

The package *gsl* (Hankin, 2006) provides a facility for accessing these generators without needing to program in C++.

Setting up the cmrg generator (L'Ecuyer, 1996) is as follows:

```
library(gsl)
r <- rng_alloc("cmrg")
rng_set(r, 100)

## [1] 100

rcmrg <- function(n) rng_uniform(r, n)
```

We can then use the function `rcmrg()` in the same way that we would use `runif()`. For example, generating 10 numbers proceeds as

```
rcmrg(10)

##  [1] 0.75100266 0.27632556 0.80290789 0.79234885 0.00991752
##  [6] 0.90312322 0.14127554 0.44023898 0.50391344 0.88495743
```

2.6.8 *Luxury generators*

The luxury random number generators or `ranlux` algorithms (James, 1994) are also available in GSL. One of the faster ones is `ranlxs0`.

```
r <- rng_alloc("ranlxs0")
rng_set(r, 100)

## [1] 100

rlxs0 <- function(n) rng_uniform(r, n)
```

```
rlxs0(5)

## [1] 0.8630 0.9370 0.1817 0.5500 0.4464
```

2.6.9 Permuted Congruential Generators (PCG)

O'Neill (2014) reconsidered the linear congruential generator but permuted the low order bits in the output to create a fast but more secure and statistically stronger set of generators. The PCG family of generators (O'Neill, 2014) has been ported into R using the Rcpp function through the *dqrng* package (Stubner, 2021).

```
library(dqrng)
```

The `pcg64` generator:

```
dqRNGkind("pcg64")
dqrunif(5)

## [1] 0.9442 0.6557 0.6191 0.8357 0.3454
```

2.6.10 XOR shift generators

The *dqrng* package also contains ports to the `Xoshiro256+` and `Xoroshiro128+` generators (Blackman and Vigna, 2021). The latter is the fastest generator available in the *dqrng* package.

```
dqRNGkind("Xoshiro256+")
dqrunif(4)

## [1] 0.6200 0.7356 0.6089 0.9021
```

```
dqRNGkind("Xoroshiro128+")
dqrunif(4)

## [1] 0.8794 0.9823 0.3808 0.9302
```

2.6.11 Timing comparisons

How does the speed of these methods compare with R's implementation of the Mersenne Twister?.

```
dqRNGkind("pcg64")
microbenchmark(runif(2e6), rcmrg(2e6), rlxs0(2e6), dqrunif(2e6))

## Unit: milliseconds
##            expr   min    lq  mean median    uq    max neval
##    runif(2e+06) 33.06 33.50 44.07  36.11 53.29 131.43   100
##    rcmrg(2e+06) 62.72 63.80 72.52  65.06 74.15 117.18   100
##    rlxs0(2e+06) 62.65 63.68 76.41  65.29 93.45 141.98   100
##  dqrunif(2e+06) 14.44 14.69 19.79  15.53 22.90  91.23   100

dqRNGkind("Xoshiro256+")
microbenchmark(dqrunif(2e6))
```

```
## Unit: milliseconds
##            expr   min    lq  mean median   uq   max neval
##  dqrunif(2e+06) 13.84 18.56 21.82  22.42 22.6 88.75   100

dqRNGkind("Xoroshiro128+")
microbenchmark(dqrunif(2e6))

## Unit: milliseconds
##            expr   min    lq  mean median    uq   max neval
##  dqrunif(2e+06) 13.36 13.56 15.94  13.69 14.21 67.98   100
```

2.6.12 Xorshift128 is fast but ...

Machine learning methods are now being used to crack more sophisticated generators (Hassan, 2021), such as the Xorshift128 generator

Security of generators is becoming a more important area of research, though there are early results on cracking generators[3]

Exercises

1. To see that computation of trigonometric functions is time-consuming is not completely obvious because of the way in which these functions are computed. The computation of $\sin(x)$ is often hard-coded on to the CPU and accessed from there by the C compiler, and the algorithm used to calculate $\sin(x)$ is not simply a straightforward Taylor expansion. Run the following code to compare the speed of the four functions below.

```
f1 <- function(x) ((32678*x)%%33271)/33271
f2 <- function(x) ((32679*x + 141)%%32768)/32768
f3 <- function(x) x - 0.166667*x^3 + 0.00833*x^5 - 0.00019*x^7 + x^9*2.6019e-6
f4 <- function(x) sin(x)
f5 <- function(x) x - x^3/6 + x^5/120 - x^7/5040 + 2.6019e-6*x^9
microbenchmark(f1(0.71), f2(0.71), f3(0.71), f4(0.71), f5(0.71))

## Unit: nanoseconds
##       expr min  lq  mean median    uq     max neval
##  f1(0.71) 481 511 32204  526.5 641.0 3156158   100
##  f2(0.71) 500 531 39453  546.5 626.5 3853827   100
##  f3(0.71) 561 591 52717  611.0 646.5 5186857   100
##  f4(0.71) 371 421 11860  431.0 471.0 1129438   100
##  f5(0.71) 551 601 54000  631.0 671.0 5322601   100
```

   `f3`(x) is a quick and accurate approximation to $\sin(x)$. [4]

2. Consider the following iteration scheme:

$$x_{n+1} = f(x_n) := \frac{2x_n}{3} + \frac{16}{x_n^2}$$

   where x_0 is the initial value, say, $x_0 = 28$.

   (a) Plot the graph of the function $f(x)$, for $x \in [1, 5]$ and overlay the straight line with intercept 0 and slope 1. Does $f(x)$ have a fixed point? Solve the equation $x = f(x)$ for x to determine its value.

[3] Marsaglia (2003) was aware of a simple method to crack LCGs already in the 1970s. His method requires only a few numbers and uses determinants of 2×2 matrices. **Provide Details**

[4] See (vegesm, 2024) for details.

(b) Using a `for` loop in R, run 10 steps of the proposed scheme, printing out the value of x_n at each step.

(c) Based on your analysis, could the proposed scheme lead to a useful pseudorandom number generator?

3. Consider the following iteration scheme:

$$x_{n+1} = f(x_n) := x_n + 7\mathrm{e}^{-x_n} - 1$$

where x_0 is the initial value, say, $x_0 = 2$.

(a) Plot the graph of the function $f(x)$, for $x \in [0, 5]$ and overlay the straight line with intercept 0 and slope 1. Does $f(x)$ have a fixed point? Solve the equation $x = f(x)$ for x to determine its value.

(b) Using a `for` loop in R, run 10 steps of the proposed scheme, printing out the value of x_n at each step.

(c) Based on your analysis, could the proposed scheme lead to a useful pseudorandom number generator?

4. Consider the following iteration scheme:

$$x_{n+1} = f(x_n) := \frac{x_n}{2} - \frac{24.5}{x_n}$$

where x_0 is the initial value, say, $x_0 = 0.5$.

(a) Plot the graph of the function $f(x)$, for $x \in [0.1, 10]$ and overlay the straight line with intercept 0 and slope 1. Does $f(x)$ have a fixed point?

(b) Using a `for` loop in R, run 10 steps of the proposed scheme, printing out the value of x_n at each step.

(c) Based on your analysis, could the proposed scheme lead to a useful pseudorandom number generator?

5. Consider the following iteration scheme:

$$x_{n+1} = f(x_n) := \frac{x_n}{2} - \frac{24.5}{\sqrt{x_n}} \mod 1$$

where x_0 is the initial value, say, $x_0 = 0.5$.

(a) Plot the graph of the function $f(x)$, for $x \in [0.01, 1]$ and overlay the straight line with intercept 0 and slope 1. Does $f(x)$ have a fixed point?

(b) Using a `for` loop in R, run 10 steps of the proposed scheme, printing out the value of x_n at each step.

(c) Based on your analysis, could the proposed scheme lead to a useful pseudorandom number generator?

6. Construct a function which takes `n` as an argument and implements the pseudorandom number generator iteration for the preceding question, returning, as output, a vector of length `n`. Conduct the basic checks on the quality of this generator? If it passes those basic checks, apply the random forest test to determine whether this generator could be used, at least in small simulations, using up to 2000 numbers.

7. It is possible to generate cycles of length 1 in a linear congruential generator. Consider the generator of Example 2.8 and observe its behaviour when $x_0 = 6$.

8. Which of the following linear congruential pseudorandom number generators have maximal cycle length?

(a) $a = 1025, c = 27, m = 2^{31}$

(b) $a = 1025, c = 54, m = 2^{31}$

(c) $a = 1025, c = 375, m = 2^{31}$

(d) $a = 10241, c = 375, m = 2^{31}$

9. Run basic checks on each of the generators from the previous question to determine whether any are obviously defective. For any that aren't, apply the random forest test for $m \leq 10$ as a further check. Do you think any of these generators could be used, at least for small problems?

10. Write an R function that implements the Fibonacci pseudorandom number generator, taking `n` and `x0` as the arguments and returning a vector of `n` pseudorandom numbers. Run the basic tests as well as the random forest test to assess the quality of this generator.

11. Consider the following random number generator which is based purely on shuffling an initially ordered sequence.

```
a <- (1:1000)/1001
for (i in 1:20) {
   a <- shuffle(1000, 50, a)
}
a
```

    (a) Test the sequence for uniformity. Write out your conclusion clearly.

    (b) Check the autocorrelations. Is there any indication of sequential linear dependence?

    (c) Construct a lag plot as a way of applying the spectral test in two dimensions. What do you conclude? What would you predict about the 51st number if you knew the 50th number was 0.5?

12. Repeat the previous question, but start with `a` being the first 1000 numbers generated from a multiplicative congruential generator with $b = 171$ and $m = 30269$. Also, check the autocorrelations for the multiplicative generator and compare with the performance after shuffling; does shuffling help?

13. Write an R function which applies the shuffling algorithm to output from the RANDU generator. Run the random forest test on sequences coming from the new generator to determine if you have an improved generator.

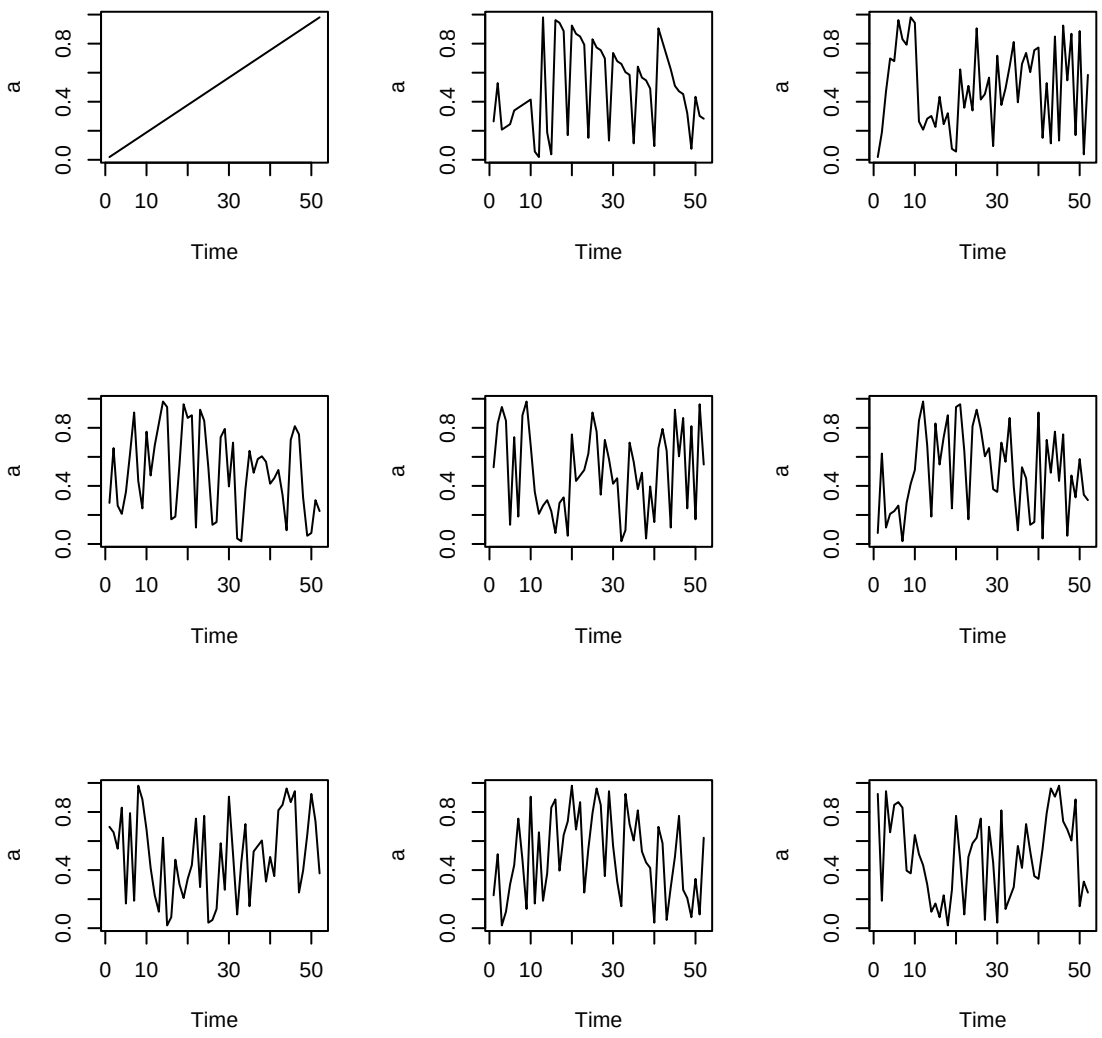


Figure 2.21: Trace plots for 9 successive shuffles of a deck of 52 numbers (representing playing cards in an ordinary deck).

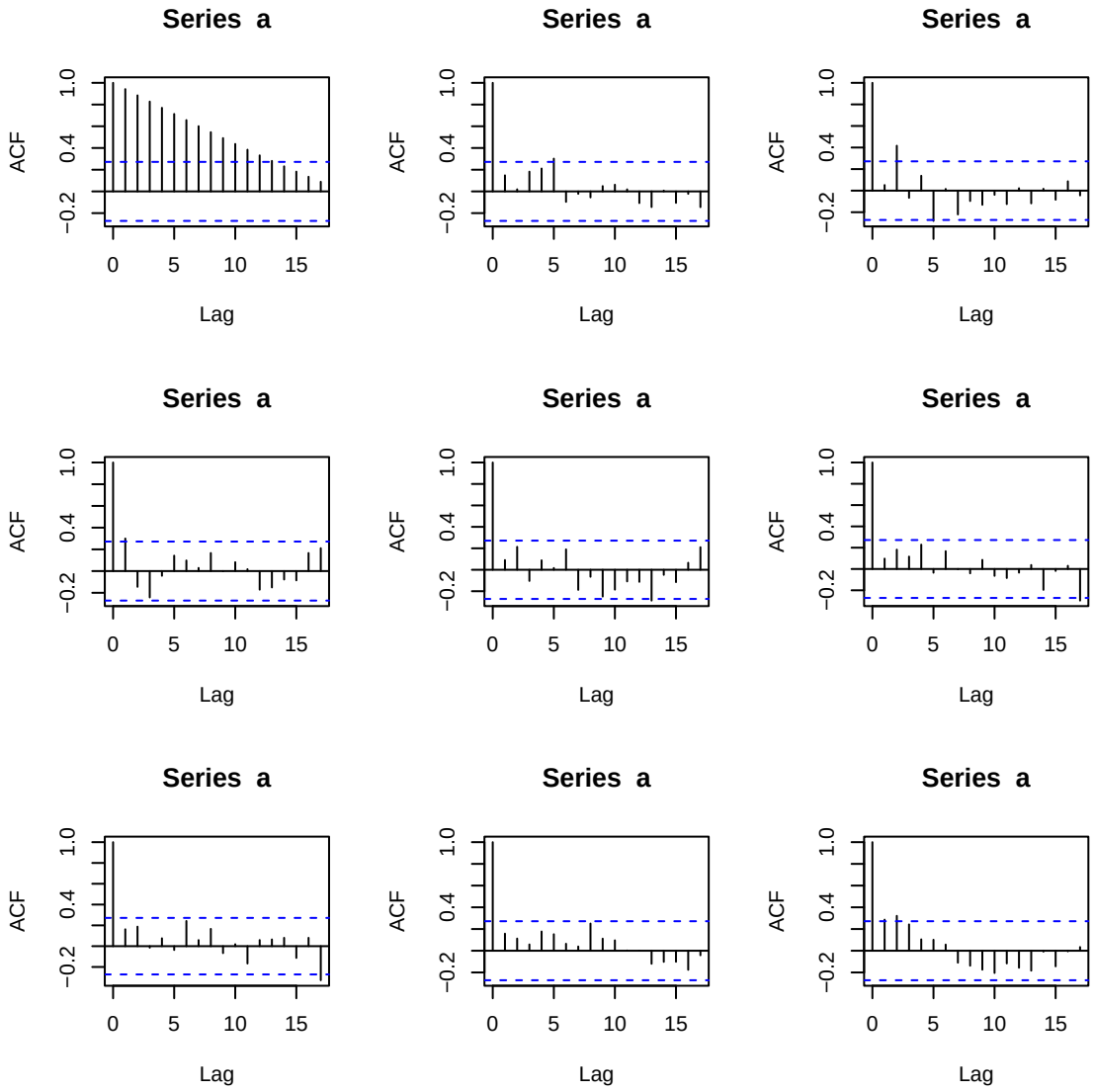


Figure 2.22: Autocorrelations for 9 successive shuffles of a deck of 52 numbers (representing playing cards in an ordinary deck).

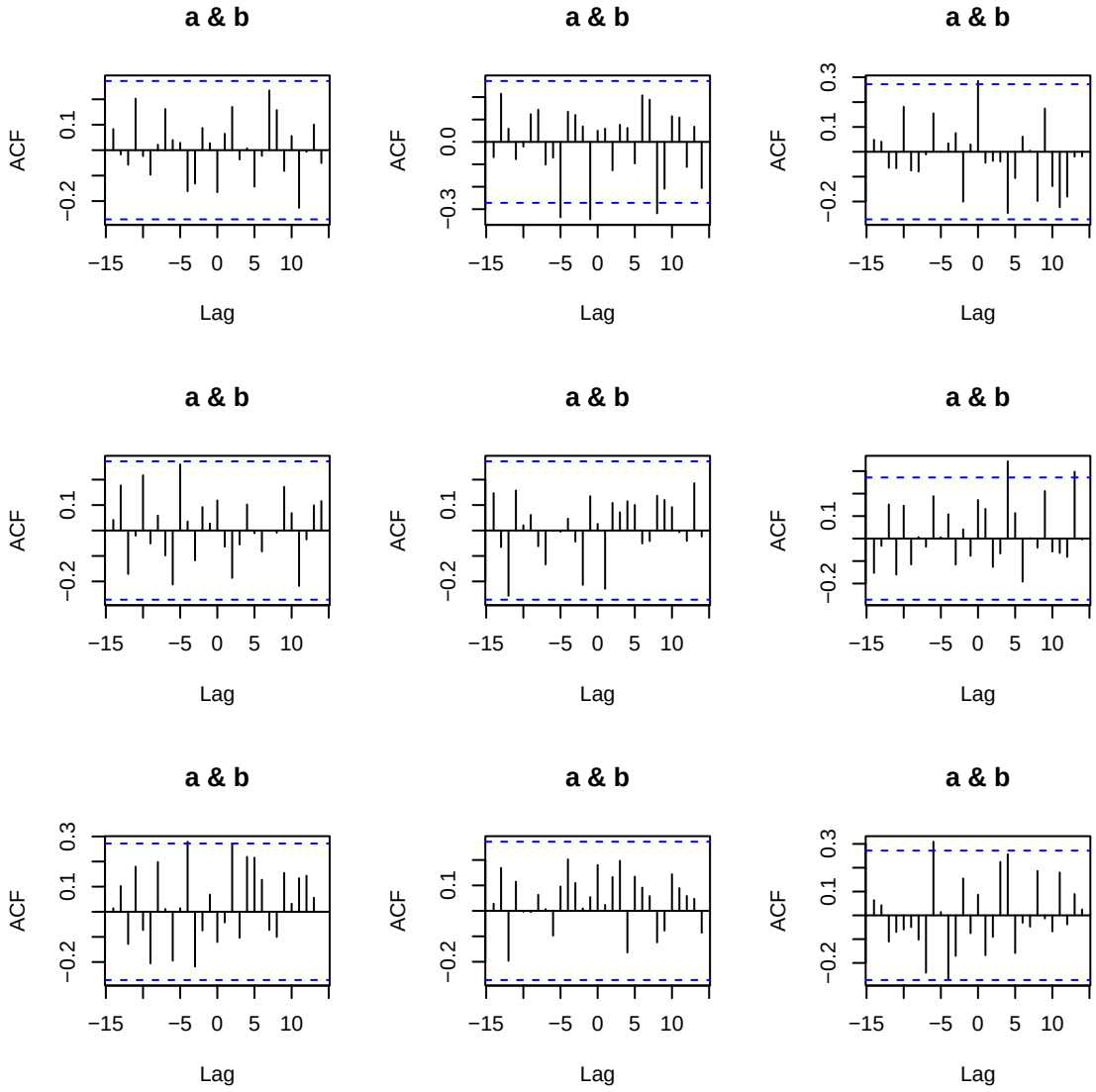


Figure 2.23: Cross-correlations between original ordering and first 9 orderings of a shuffled deck of 52 numbers (representing playing cards).

Bibliography

Anderson, S. L. (1990). Random number generators on vector supercomputers and other advanced architectures. *SIAM review*, 32(2):221–251.

Blackman, D. and Vigna, S. (2021). Scrambled linear pseudorandom number generators. *ACM Transactions Mathematical Software*, 47:1–32.

Borchers, H. W. (2021). *numbers: Number-Theoretic Functions*. R package version 0.8-2.

Braun, W. (2024). *mcODE: Monte Carlo Solution of First Order Differential Equations*. R package version 1.1.

Braun, W. J., Jones, B. L., Lee, J. S., Woolford, D. G., and Wotton, B. M. (2010). Forest fire risk assessment: an illustrative example from Ontario, Canada. *Journal of Probability and Statistics*, 2010(1):823018.

Braun, W. J. and MacQueen, S. (2019). MPV: Data sets from Montgomery, Peck and Vining. *R package version*, 1.

Cappé, O., Moulines, E., and Rydén, T. (2005). *Inference in Hidden Markov Models*. Springer, New York.

Coveyou, R. (1969). Random number generation is too important to be left to chance. *Applied Probability and Monte Carlo Methods and modern aspects of dynamics. Studies in applied mathematics*, 3:70–111.

Eddelbuettel, D. and Francois, R. (2022). *RcppGSL: 'Rcpp' Integration for 'GNU GSL' Vectors and Matrices*. R package version 0.3.12.

Han, L. and Braun, W. J. (2014). Dionysus: a stochastic fire growth scenario generator. *Environmetrics*, 25(6):431–442.

Hankin, R. K. S. (2006). Special functions in r: introducing the gsl package. *R News*, 6.

Hassan, M. (2021). *Cracking Random Number Generators Using Machine Learning - Part 1: xorshift128*.

James, F. (1994). Ranlux: A fortran implementation of the high-quality pseudorandom number generator of Lüscher. *Computer Physics Communications*, 79(1):111–114.

Knuth, D. E. (2014). *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional.

Law, A. M., Kelton, W. D., and Kelton, W. D. (2007). *Simulation Modeling and Analysis*, volume 3. Mcgraw-Hill, New York.

L'Ecuyer, P. (1996). Combined multiple recursive random number generators. *Operations research*, 44(5):816–822.

Liaw, A. and Wiener, M. (2002). Classification and regression by randomforest. *R News*, 2(3):18–22.

Marsaglia, G. (2003). Random number generators. *Journal of Modern Applied Statistical Methods*, 2(1):2.

Miller, S. J. and Nigrini, M. J. (2006). The modulo 1 central limit theorem and benford's law for products. *arXiv preprint math/0607686*.

Nair, U., Sankaran, P., and Balakrishnan, N. (2018). *Reliability Modelling and Analysis in Discrete Time*. Academic Press.

O'Neill, M. E. (2014). PCG: a family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Transactions on Mathematical Software*.

R Core Team (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Ross, S. M. (1990). *A Course in Simulation*. MacMillan, New York.

Stubner, R. (2021). *dqrng: Fast Pseudo Random Number Generators*. R package version 0.3.0.

The GSL Team (1996-2021). *Gnu Scientific Library*.

Tymstra, C., Bryce, R., Wotton, B., Taylor, S., Armitage, O., et al. (2010). Development and structure of prometheus: the canadian wildland fire growth simulation model. *Natural Resources Canada, Canadian Forest Service, Northern Forestry Centre, Information Report NOR-X-417.(Edmonton, AB)*.

vegesm (2024). How do calculators compute sine? *Algeo Calculator*.

Vigna, S. (2019). It is high time we let go of the Mersenne Twister. *arXiv preprint arXiv:1910.06437*.

Wichmann, B. A. and Hill, I. D. (1982). Algorithm as 183: An efficient and portable pseudo-random number generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2):188–190.

Wickham, H. (2014). *Advanced R*. CRC press.

Index