Simulation I

W. John Braun, UBC

COSC/DATA 405/505

Outline

1. Introduction - nonlinearity and predictability
2. Desirable properties of generators
3. Multiplicative congruential generators and cycling
4. Seed issues - an illustrative example
5. Basic checks: histogram and autocorrelation function
6. Shuffling
7. Statistical testing, the spectral test and a random forest test
8. Linear congruential generators and cycling
9. Combination generators
10. It's "high time"
11. Accessing better generators in R

Introduction

1. **Random and pseudorandom numbers**

2. **Predictable and unpredictable numbers**

Nonlinearity can sometimes lead to less predictable 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.

An example of a nonlinear function is the cosine function. We start with $x_0 = 2$ and generate 12 successive values from

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

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

Nonlinearity - Example

```
prnumbers

##  [1] -1.3073638  0.8180586  2.1477164 -1.7135662
##  [5] -0.4470026  2.8329212 -2.9931147 -3.1070269
##  [9] -3.1397161 -3.1415871 -3.1415927 -3.1415927
```

The first few numbers produced by this function seem to be unpredictable, but eventually this mapping converges to a single number.

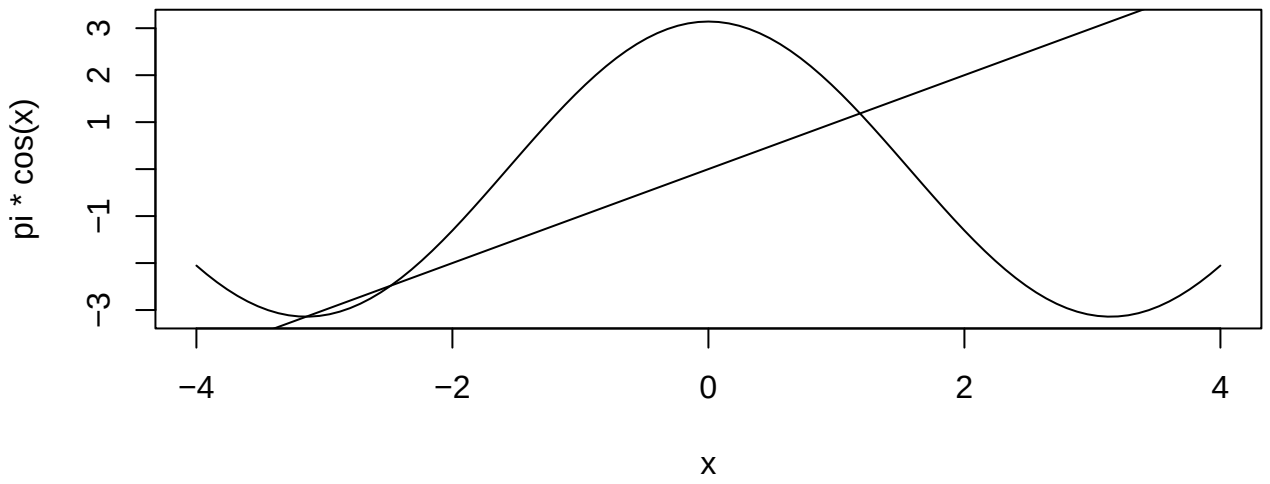
The convergence in this example occurs because the mapping $x = \pi \cos(x)$ has a stable fixed point at $x = -\pi$.

This fixed point is stable, meaning that if x_{n-1} is larger than the fixed point, then $x_n = \pi \cos(x_{n-1})$ 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.

Nonlinearity - Example

The stability of a fixed point is related to the slope of the curve $f(x)$ in a neighbourhood around the fixed point; if the slope is less than 1 in absolute value, the point is stable.

```
curve(pi*cos(x), -4, 4)
abline(0, 1)
```



Nonlinearity - Example

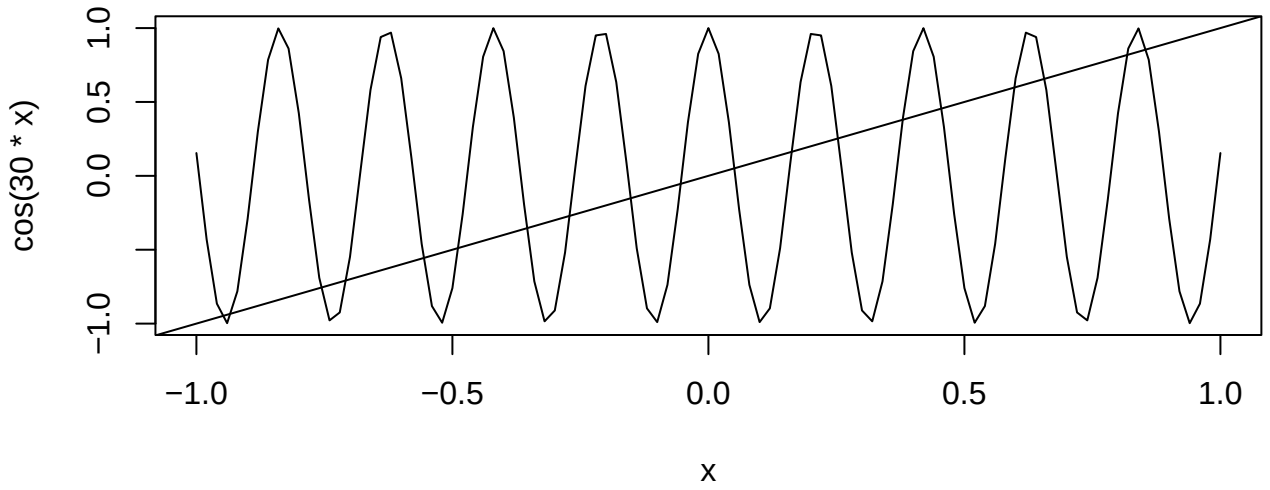
A mapping for a pseudorandom number generator should not have a stable fixed point.

We can increase the frequency of the waveform described by the cosine function increasing the number of possible fixed points in the interval $[-1, 1]$ while also assuring that they are not stable.

This mapping is plotted on the next slide, together with the function $f(x) = x$ overlaid, so we can see a large number of fixed points.

Nonlinearity - Example

```
curve(cos(30*x), -1, 1)
abline(0, 1)
```



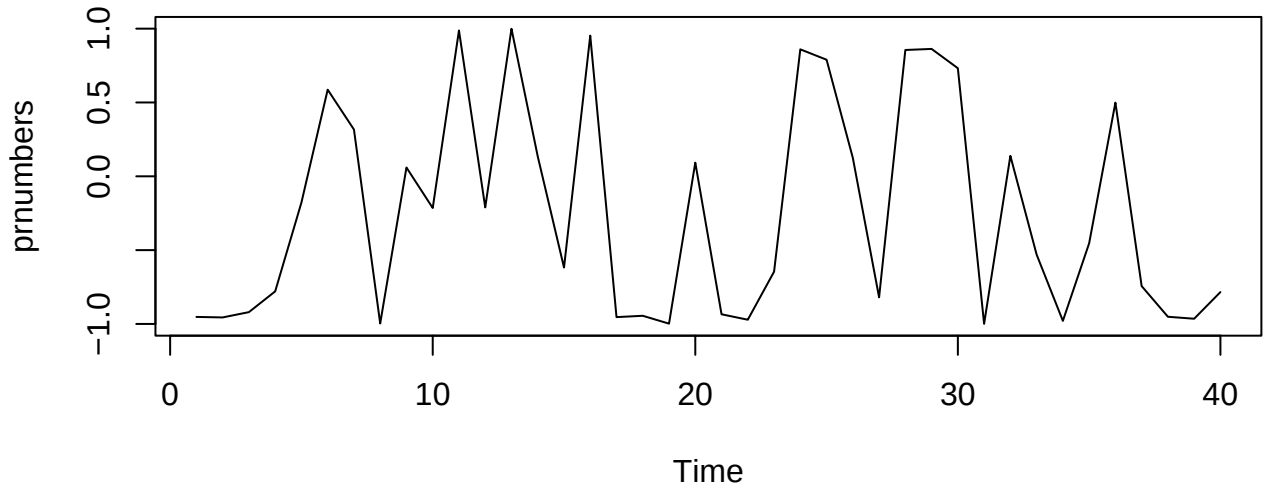
Note that the slopes near fixed points (points of intersection between the overlaid line and the curve) are also relatively large, inducing instability.

Nonlinearity - Example

Illustration:

Start with $x_0 = 2$ and generate 40 successive values from

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

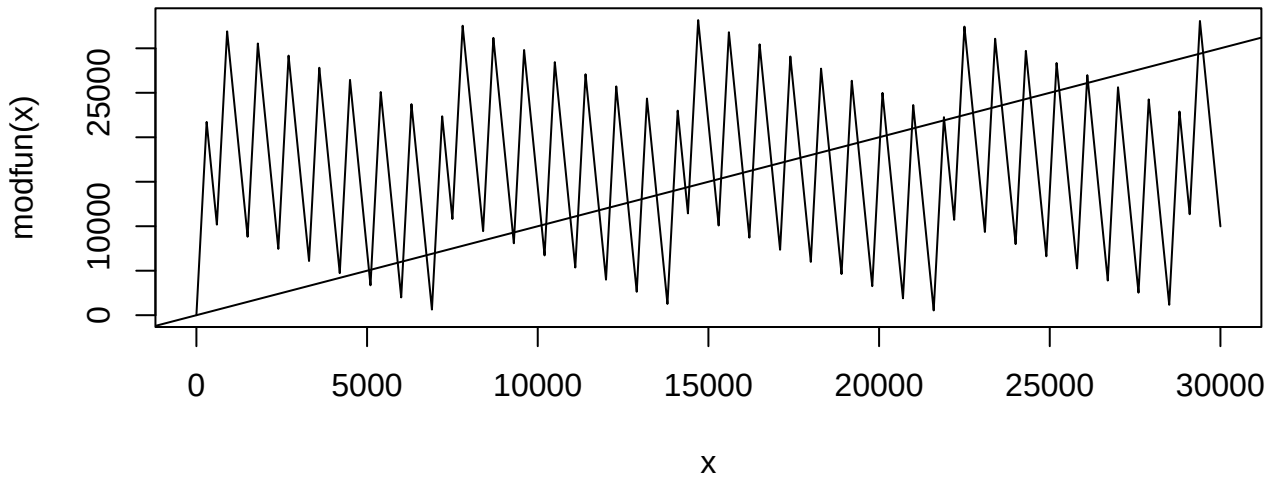


The numbers produced by this function are certainly less predictable than before, as can be seen in the trace plot above.

Nonlinearity

Functions with jumps can also provide mappings which are very unpredictable.

Example: consider the function $f(x) = 32678x$ mod 33271:

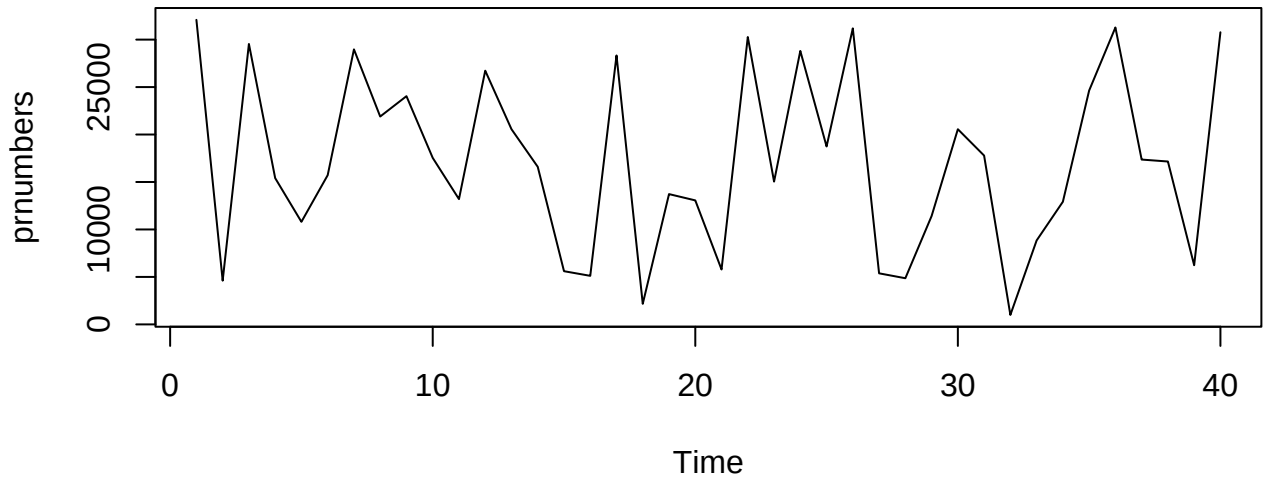


Nonlinearity - Example

Illustration:

Start with $x_0 = 2$ and generate 40 successive values from

$$x_n = 32678 x_{n-1} \textbf{ mod } 33271.$$



Desirable properties of a pseudorandom number generator

- **Speed**
- **Statistical accuracy**
- **Long cycle length**
- **Efficient use of processor**
- **Portability**
- **Reproducibility**
- **Security; robust against attacks**

Multiplicative Congruential Random Number Generators

The earliest pseudorandom number generators considered were of the form*

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

m is a large integer, and a is another integer which is smaller than m. a and m are usually relatively prime.

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

x_0 is called the seed.

*Linear congruential generators are similar: $x_{n+1} = (ax_n + c)$ mod m for a positive integer c.

Example

Take $m = 7$ and $a = 3$. Also, take $x_0 = 2$. Then

$$
\begin{aligned}
x_1 &= 3 \times 2 \bmod 7 = 6, & u_1 &= 0.857 \\
x_2 &= 3 \times 6 \bmod 7 = 4, & u_2 &= 0.571 \\
x_3 &= 3 \times 4 \bmod 7 = 5, & u_3 &= 0.714 \\
x_4 &= 3 \times 5 \bmod 7 = 1, & u_4 &= 0.143 \\
x_5 &= 3 \times 1 \bmod 7 = 3, & u_5 &= 0.429 \\
x_6 &= 3 \times 3 \bmod 7 = 2, & u_6 &= 0.286
\end{aligned}
$$

Cycling

It should be clear that the iteration will set $x_7 = x_1$ and cycle x_i through the same sequence of integers, so the corresponding sequence u_i will also be cyclic.

An observer might not easily be able to predict u_2 from u_1, but since $u_{i+6} = u_i$ for all $i > 0$, longer sequences are very easy to predict.

Cycling

In order to produce an unpredictable sequence, it is desirable to have a very large cycle length so that it is unlikely that any observer will ever see a whole cycle.

The maximal cycle length is m, so m would normally be taken to be very large.

Caution

Care must be taken in the choice of a and m to ensure that the cycle length is actually m.

Note, for example, what happens when $a = 171$ and $m = 29241$. Start with $x_0 = 3$, say.

$$x_1 = 171 \times 3 = 513$$

$$x_2 = 171 \times 513 \textbf{ mod } 29241 = 0$$

All remaining x_n's will be 0.

Choosing a and m

To avoid this kind of problem, we should choose m so that it is not divisible by a; thus, prime values of m will be preferred.

The next example gives a generator with somewhat better behaviour.

Example

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

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

$$u_n = x_n/30269$$

with initial seed $x_0 = 27218$**.**

Example

```
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,

$u_1, u_2, \ldots, u_{30268}$.

Output

```
random.number[1:50]
```

```
##  [1] 0.763851 0.618488 0.761373 0.194787 0.308533
##  [6] 0.759226 0.827579 0.516073 0.248406 0.477419
## [11] 0.638673 0.213122 0.443920 0.910238 0.650732
## [16] 0.275133 0.047739 0.163302 0.924708 0.125145
## [21] 0.399716 0.351416 0.092074 0.744722 0.347517
## [26] 0.425452 0.752255 0.635568 0.682084 0.636361
## [31] 0.817668 0.821269 0.437048 0.735175 0.714857
## [36] 0.240510 0.127226 0.755625 0.211801 0.217946
## [41] 0.268724 0.951766 0.751957 0.584724 0.987743
## [46] 0.904093 0.599954 0.592091 0.247547 0.330536
```

Output

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

## [1] 30268
```

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

A Multiplicative Congruential Generator Function

The following function will produce `n` simulated random numbers on the interval $[0, 1]$, using a multiplicative congruential generator:

```
rng <- function(n, a=171, m=30269, seed=1) {
    x <- numeric(min(m-1,n))
    x[1] <- seed
    for (i in 1:min(m-1,n)){
        y <- x[i]
        x[i+1] <- (a*y)%%m
    }
    x[2:(n+1)]/m
}

rng(5)  # simple example of use of rng

## [1] 0.0056493 0.9660379 0.1924741 0.9130794 0.1365754
```

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

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

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.)

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"
## [13] "H" "H" "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.

Tossing two fair coins

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"
##  [9] "T H" "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.

Obtaining the correct solution by restricting the choice of seed

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.

Tossing three fair coins

```
##   [1] "H T H" "T H H" "H H T" "H T T" "T T T" "T T T"
##   [7] "T T T" "T T H" "T H T" "H T T" "T T H" "T H H"
##  [13] "H H H" "H H H" "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 $1, 3, 6, 7, 9, 12, 15, 16\}$ **will perfectly produce a set of three independent coin tosses.**

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

Starting Seeds - What to do in practice

If the goal is to make an unpredictable sequence, then a random value is desirable.

For example, the computer might determine the current time of day to the nearest millisecond, then base the starting seed on the number of milliseconds past the start of the minute.

To avoid predictability, this external randomization should only be done once, after which the formula above should be used for updates.

Starting Seeds

The second strategy for choosing x_0 is to use a fixed, non-random value, e.g. $x_0 = 1$.

This makes the sequence of u_i values predictable and repeatable.

This would be useful when debugging a program that uses random numbers, or in other situations where repeatability is needed.

The way to do this in R is to use the `set.seed()` function.

Example

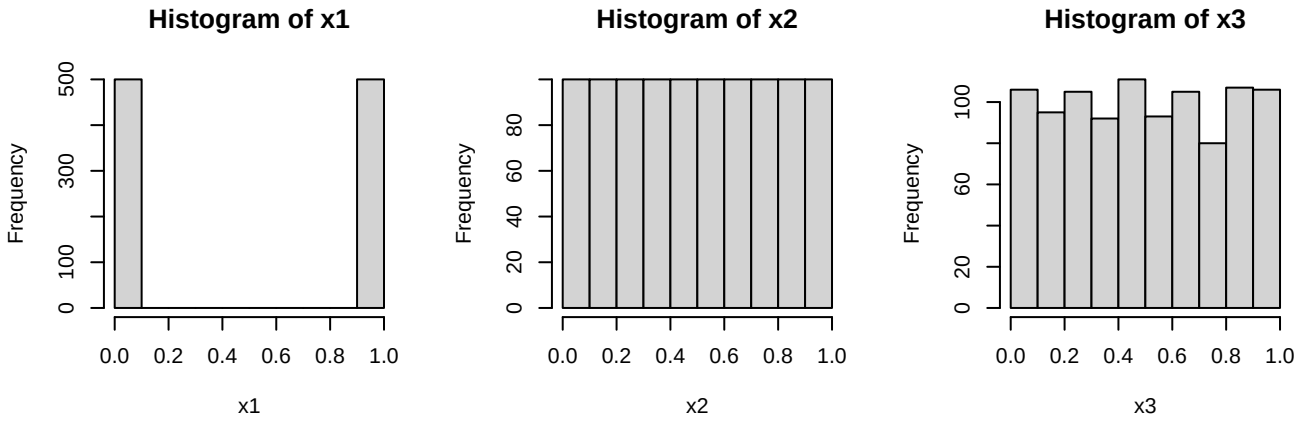
```
set.seed(32789)   # this ensures that your
                  # output will match ours
runif(5)

## [1] 0.35752 0.35376 0.26723 0.99693 0.13174
```

Basic checks - Histogram

The distribution of numbers should be uniform on the interval $[0, 1]$.

Example:

```
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)
```

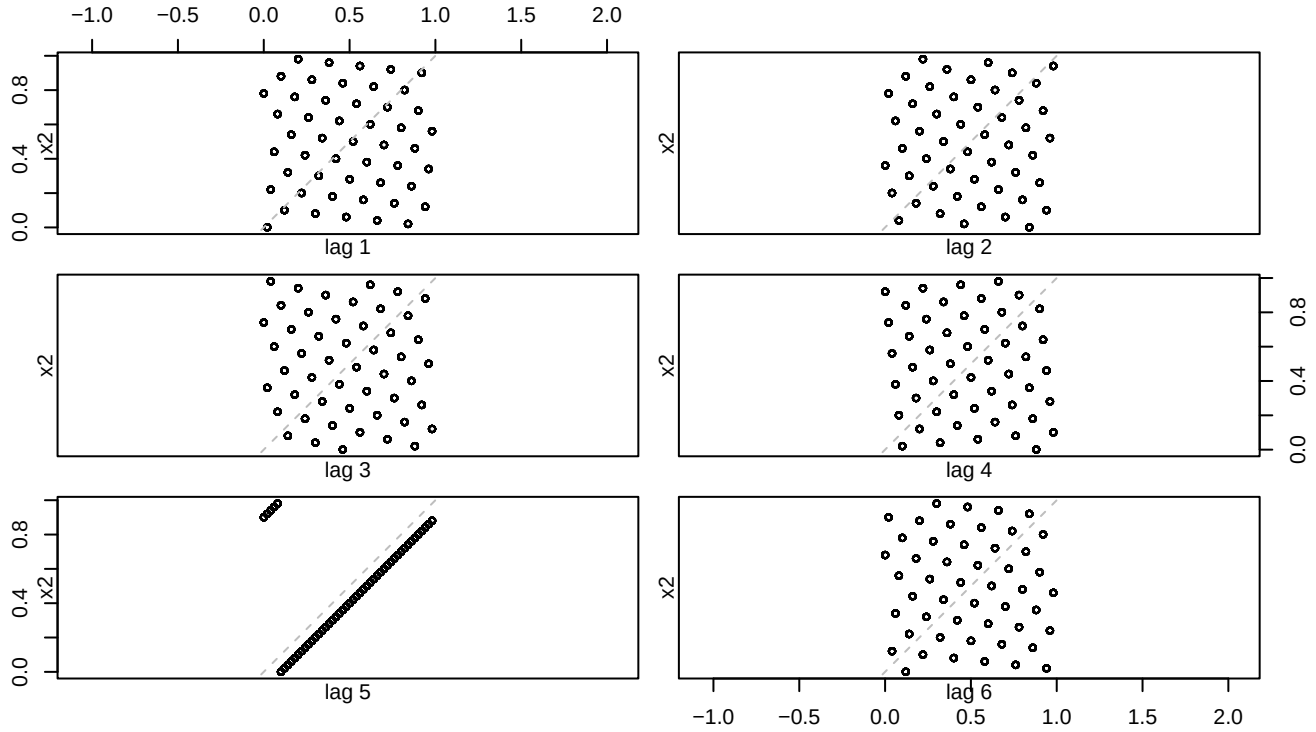


x1 fails; x2 and x3 both appear to be uniform.

Basic checks - autocorrelation

The autocorrelation function, ACF, numerically summarizes what can be observed graphically on a lag plot:

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

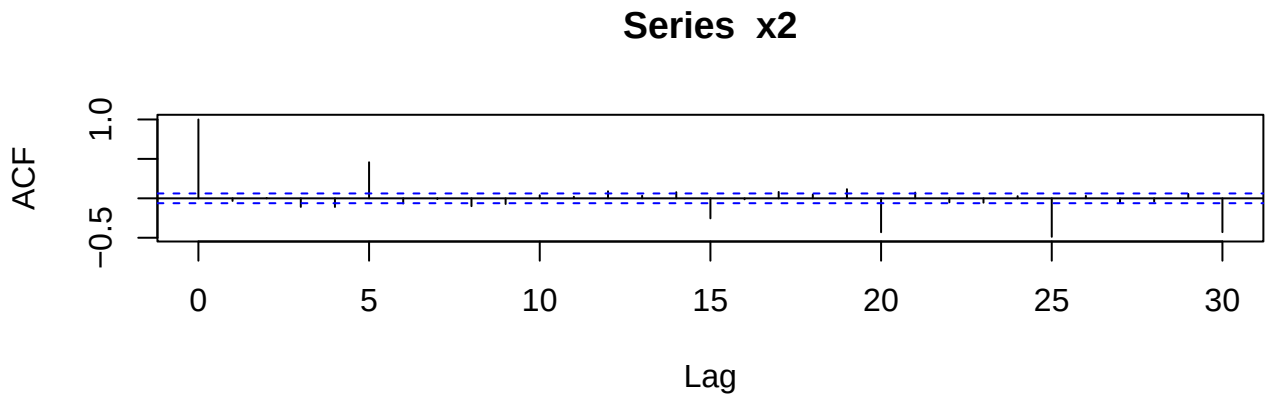


At lags 1, 2, 3, 4 and 6, it would be hard to predict the current value of x2, but the lag 5 plot shows that the current value of x2 depends a lot on the value 5 time units earlier.

Basic checks - autocorrelation

The autocorrelations for the first 5 lags are:

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



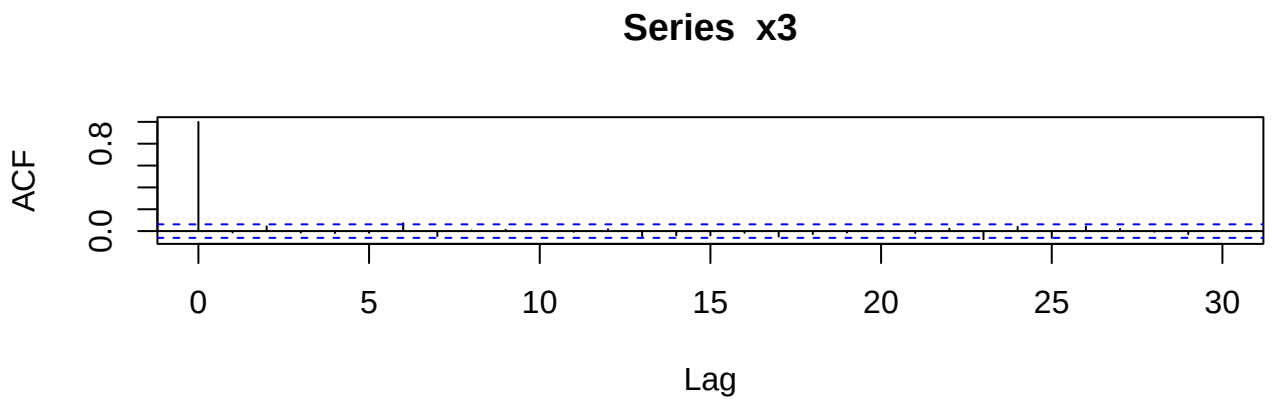

```
## [1] -0.0328  0.0049 -0.1090 -0.1097  0.4575
```

Note the size of the 5th one. This says that every 5th value of the sequence is dependent.

Basic checks - autocorrelation

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

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




```
## [1] -0.0131  0.0434 -0.0137 -0.0185 -0.0155
```

All ACF values checked are small. This sequence passes this test.

Basic checks - autocorrelation

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.

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.

Shuffling

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.

Shuffling

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)
}
```

Shuffling - Example

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:

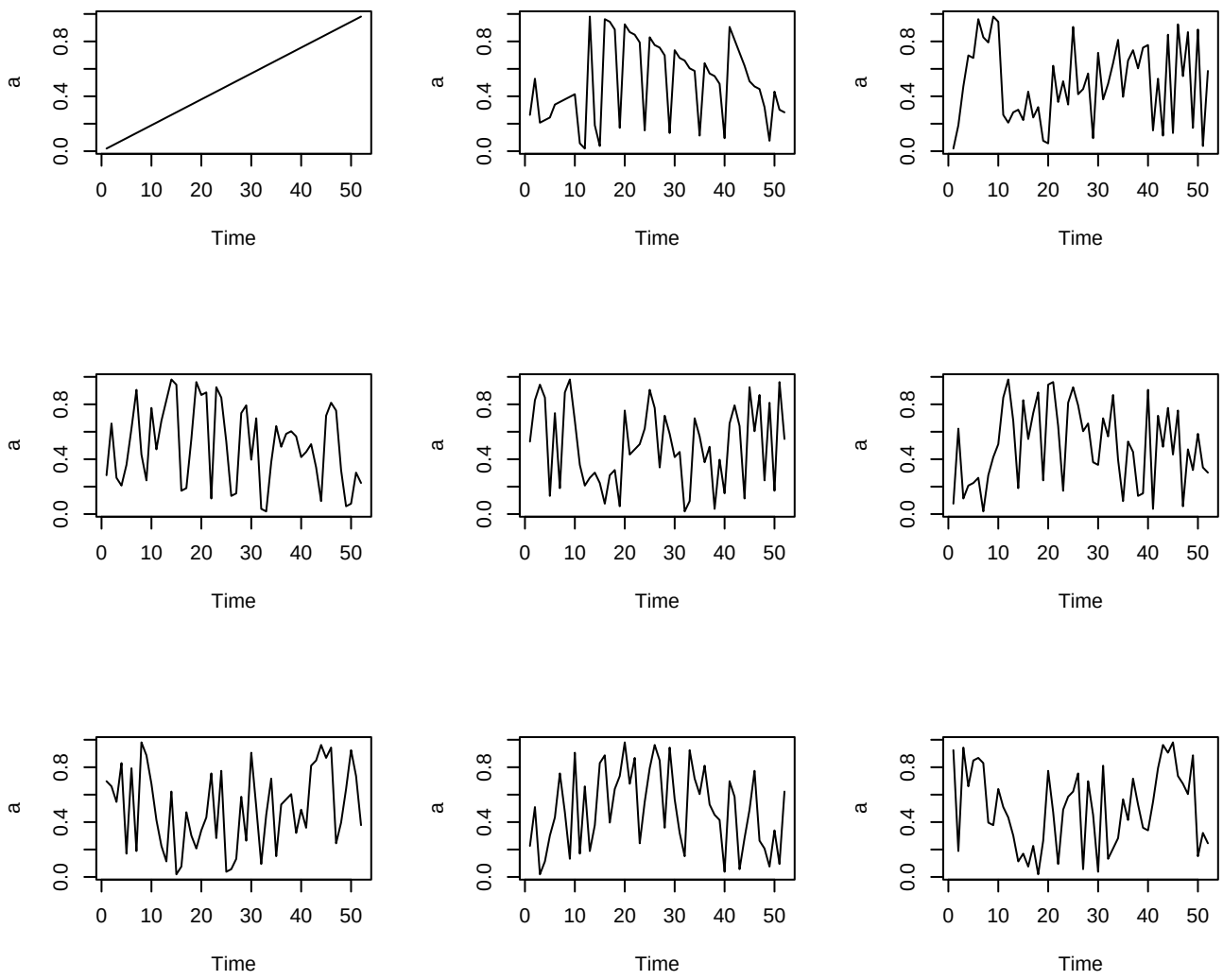
```
a <- 1:52
suits <- c("Spades", "Hearts", "Diamonds", "Clubs")
values <- c(2:10, "J", "Q", "K", "A")
cards <- factor(a)
levels(cards) <- as.vector(outer(values, suits, paste))
cards # sorted

##  [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 ... A Clubs
```

Shuffling

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)
}
```



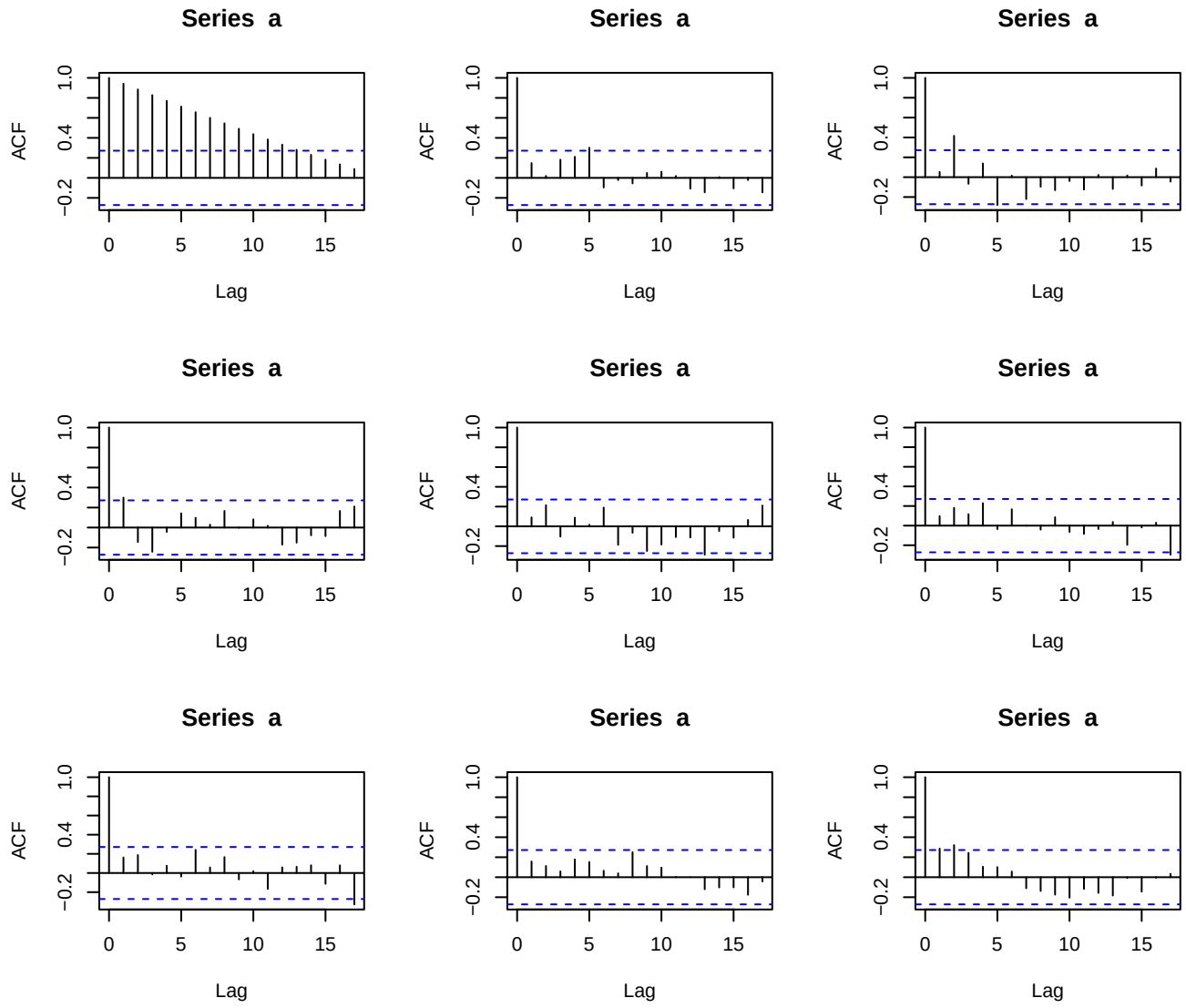

```
cards[a*53] # shuffled

##  [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 ... A Clubs
```

Shuffling - Autocorrelations

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)
}
```



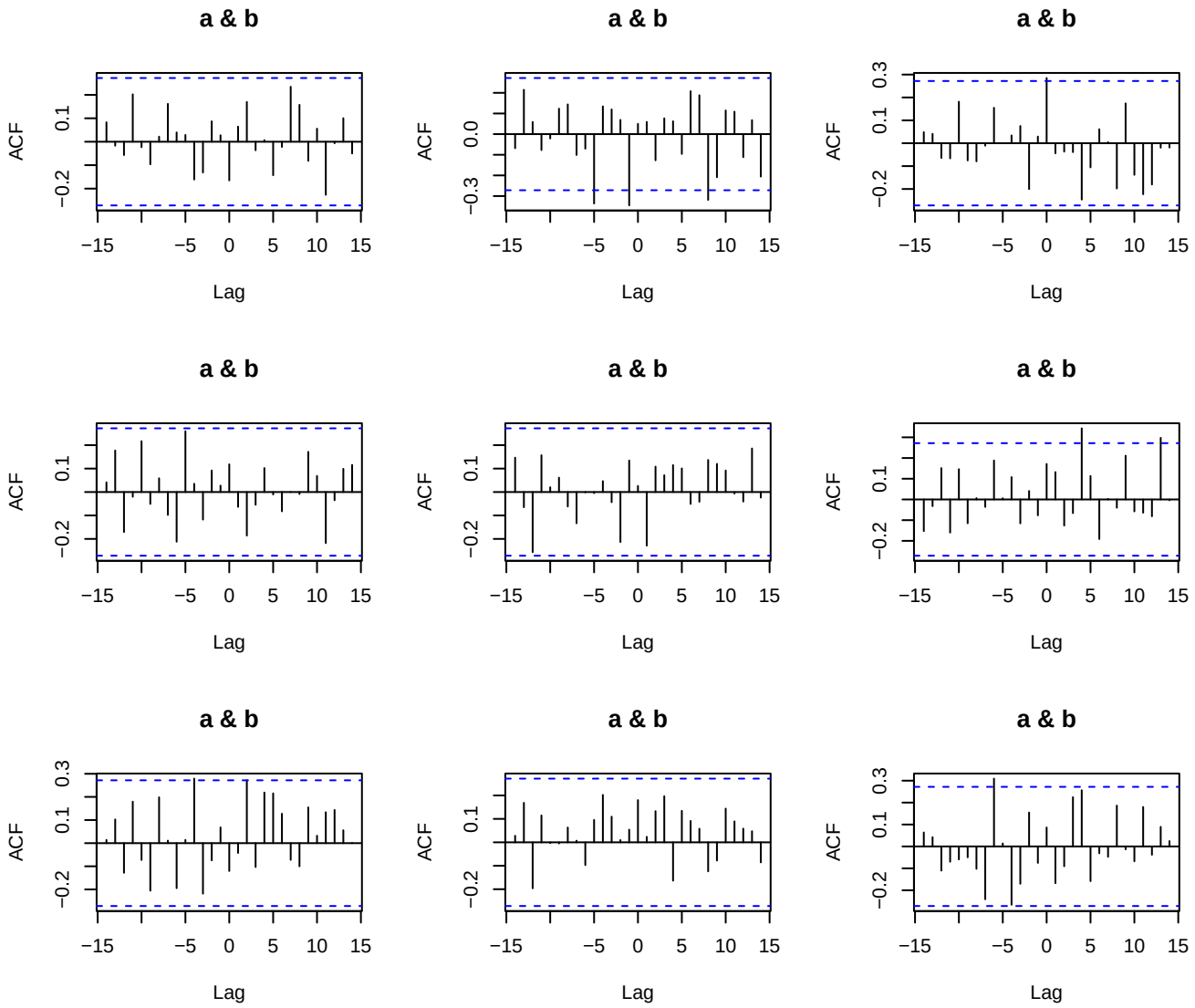
The autocorrelations appear to die down after the numbers have been shuffled 6 or 7 times.

Shuffling - Cross-correlations

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 $(y_m, y_{m+1}, \ldots, y_n)$.

We can use the cross-correlation function to see how much dependence occurs between "hands".

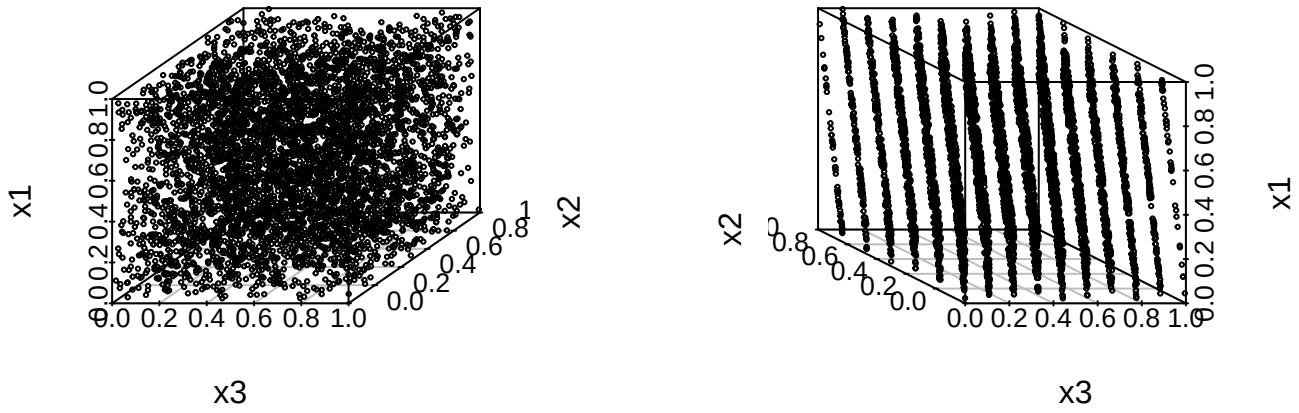
```
b <- a # b contains the order for the original hand
# a will contain the order for the next 9 shuffles:
par(mfrow=c(3,3))
for (i in 1:9) {
    a <- shuffle(52, 10, a)
    ccf(a, b)
}
```


a & b
ACF
Lag
−15
−5
0
5
10
15
−0.2
0.1
−0.3
0.0
0.3

"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.

Testing generators

- **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.)

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.

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.

Random forest

A quick digression into 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

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

Consider the data in `table.b3` of the *MPV* package.

This data set concerns gas mileage, `y`, for a number of cars, together with information on 11 other variables.

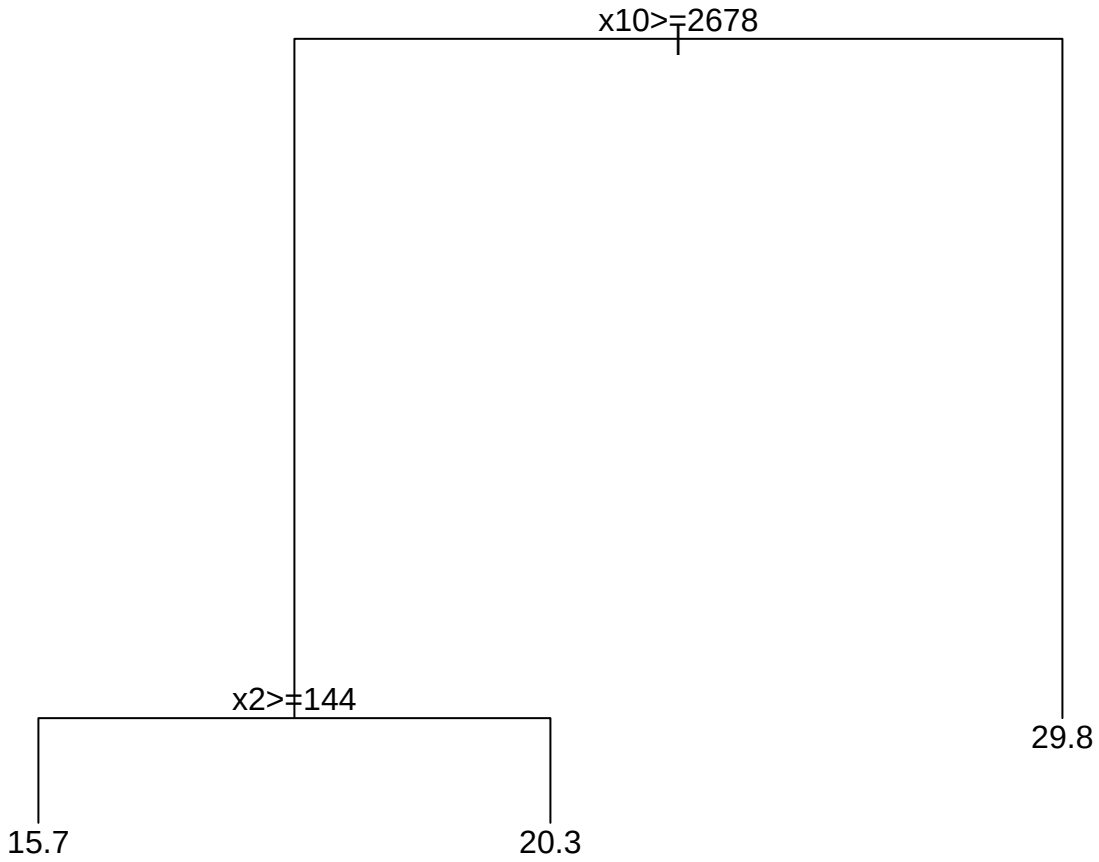
In particular, we note that `x10` represents weight and `x2` represents horsepower.

We can fit the default regression tree to these data using

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

Example

```
plot(mpg.tree)
text(mpg.tree)
```



Regression trees

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.

Regression trees

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.

Random forests

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

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])
```

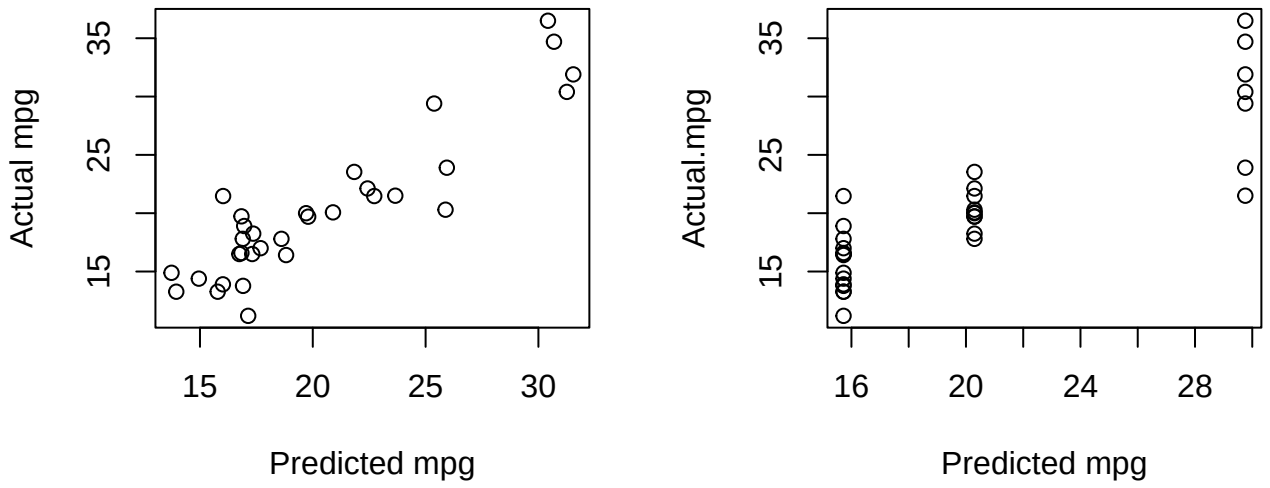
The plots on the next slide are of the actual gas mileages against predictions for both the random forest predictions and the tree predictions.

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.

Example

```
par(mfrow=c(1,2))
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")
```



Testing pseudorandom numbers with random forest prediction

The function below 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, 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.

Testing pseudorandom numbers with random forest prediction

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.

Testing pseudorandom numbers with random forest prediction

```
rftest <- function(u, m=5) {
    n <- length(u)-1
    A <- diag(rep(1,n))
    A <- rbind(rep(0, n), A)
    A <- cbind(A, rep(0, n+1))
    xy <- matrix(u, nrow=n+1)
    for (j in 1:m) {
        xy <- cbind(xy, A%*%xy[,j])
    }
    xy <- data.frame(xy)
    names(xy) <- c("y", paste("x", 1:m, sep=""))
    xy <- xy[-(1:m),]
    xytrain <- xy[1:(n/2),]
    xytest <- xy[-(1:(n/2)),]
    xy.rf <- randomForest(y ~ ., data = xytrain)
    par(mfrow=c(1,2))
    plot(predict(xy.rf), xytrain$y, cex=.3,
```

```
    xlab="predicted values", ylab="observed values",
    main = "training data")
  abline(lm(xytrain$y ~predict(xy.rf)))
  require(randomForest)
    plot(predict(xy.rf, newdata=xytest), xytest$y, cex=.3,
    xlab="predicted values", ylab="observed values",
    main="test data")
  abline(lm(xytest$y ~ predict(xy.rf, newdata=xytest)))
}
```

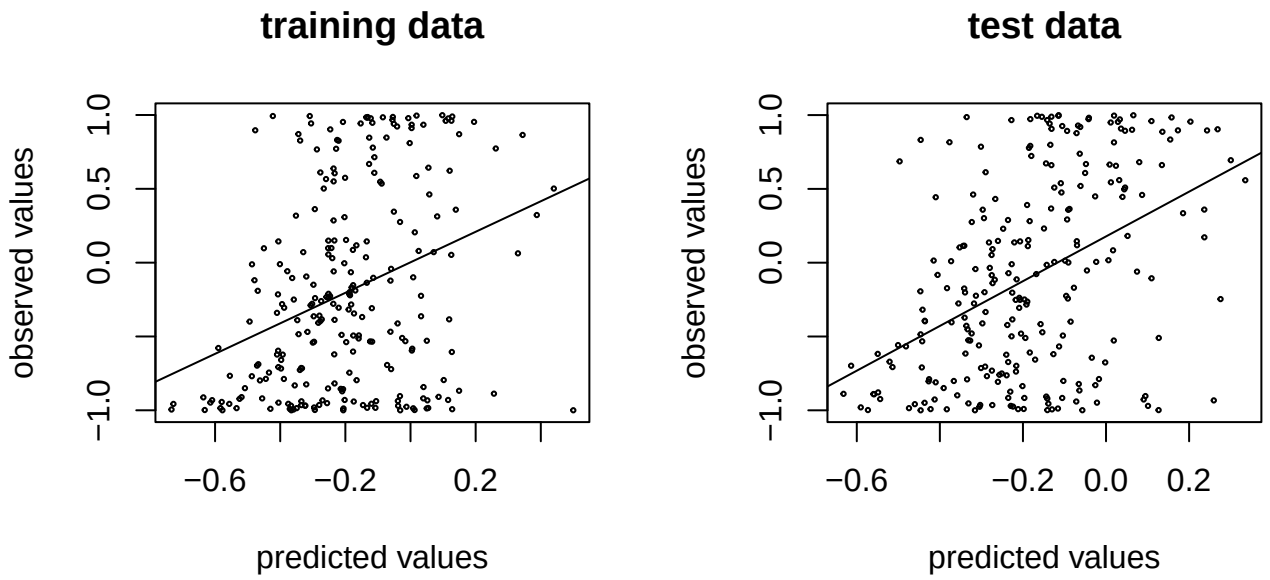
Example

We start with the cosine sequence discussed earlier to show why such a generator is not in practical use.

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

Example

```
rftest(prnumbers)
```



Example

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.

Testing pseudorandom numbers with random forest prediction

The function `MPV::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 cosine example, 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.

Testing pseudorandom numbers with random forest prediction

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.

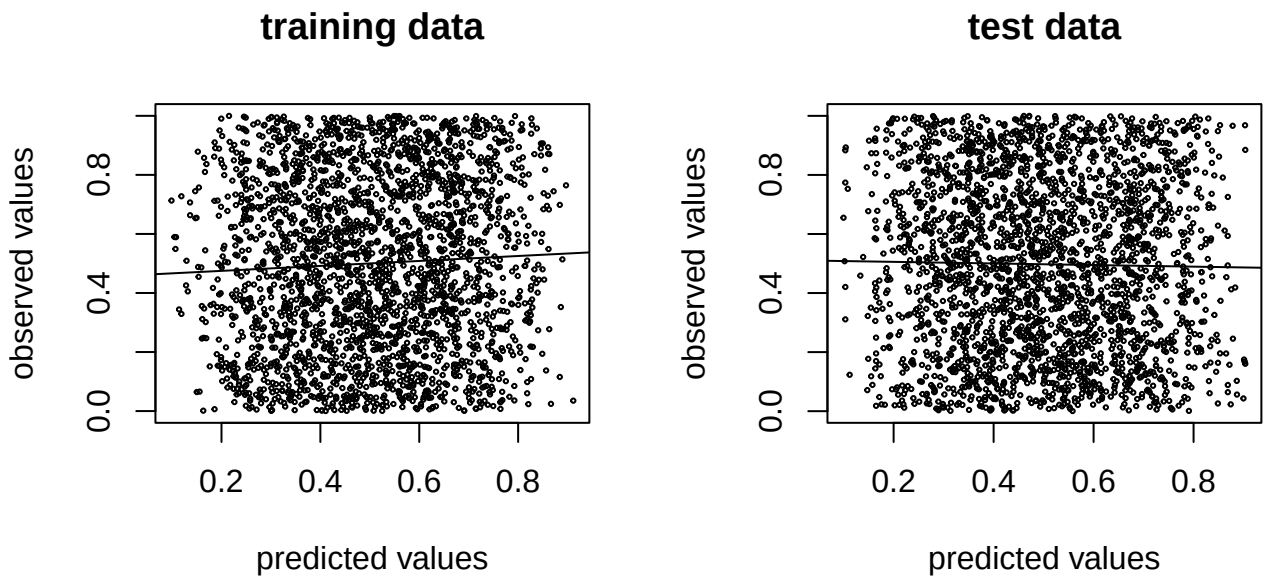
Example - testing the default R generator

We will check the quality of the default generator in R, using the random forest test, using $n = 5000$, and $m = 1, 2, \ldots, 10$:

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

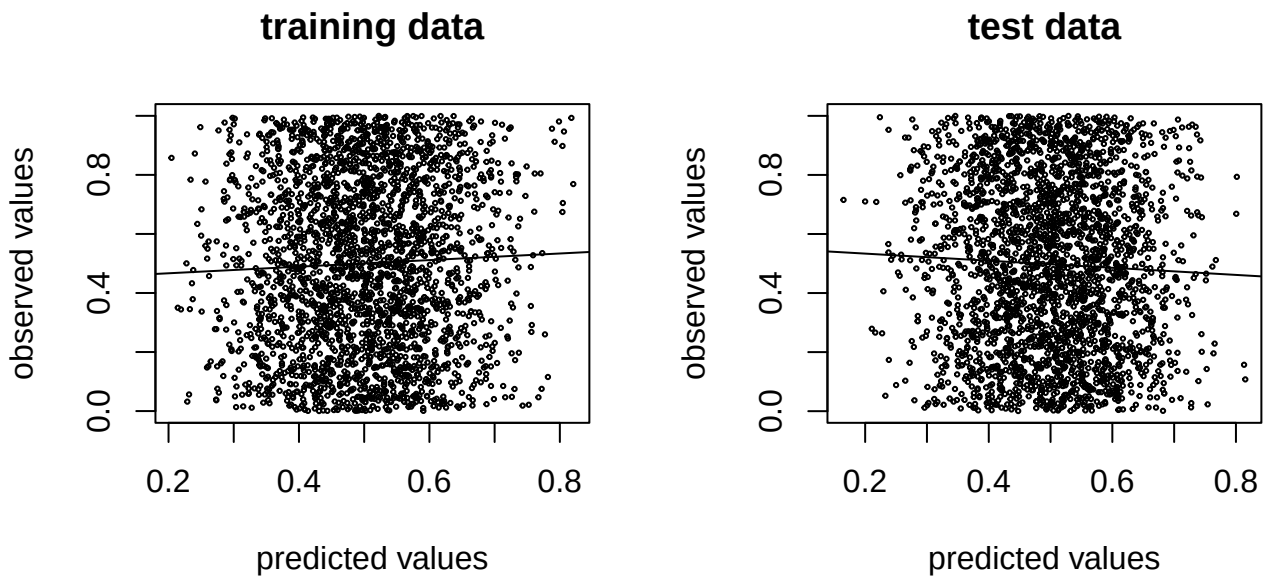
Random forest test for the default generator in R, using $m = 1$

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



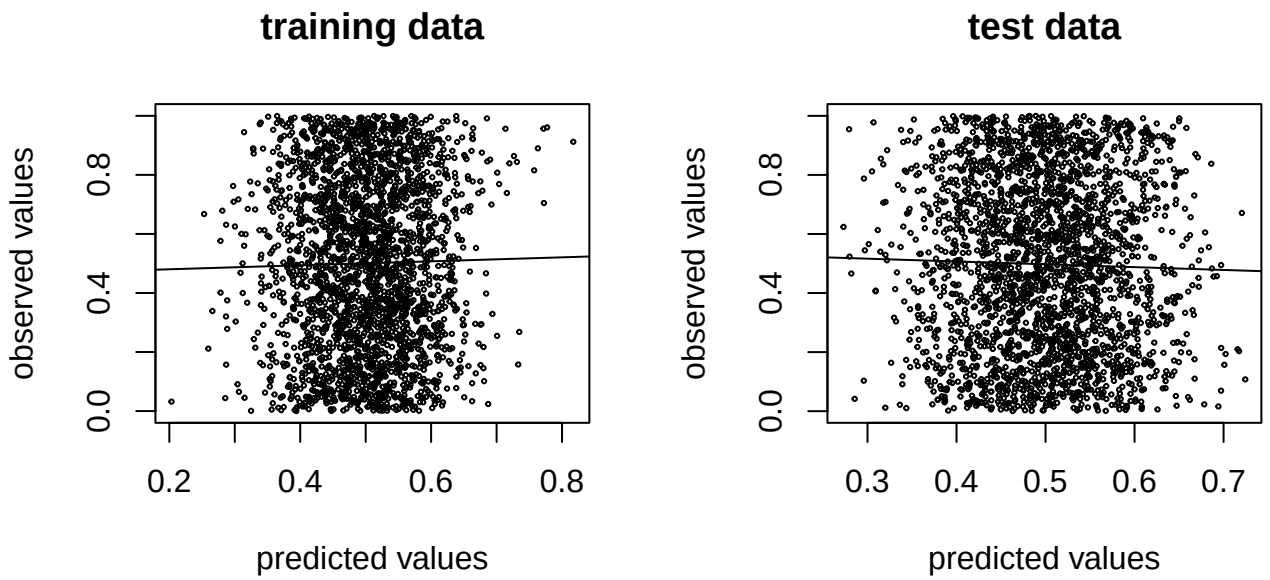
Random forest test for the default generator in R, using $m = 2$

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



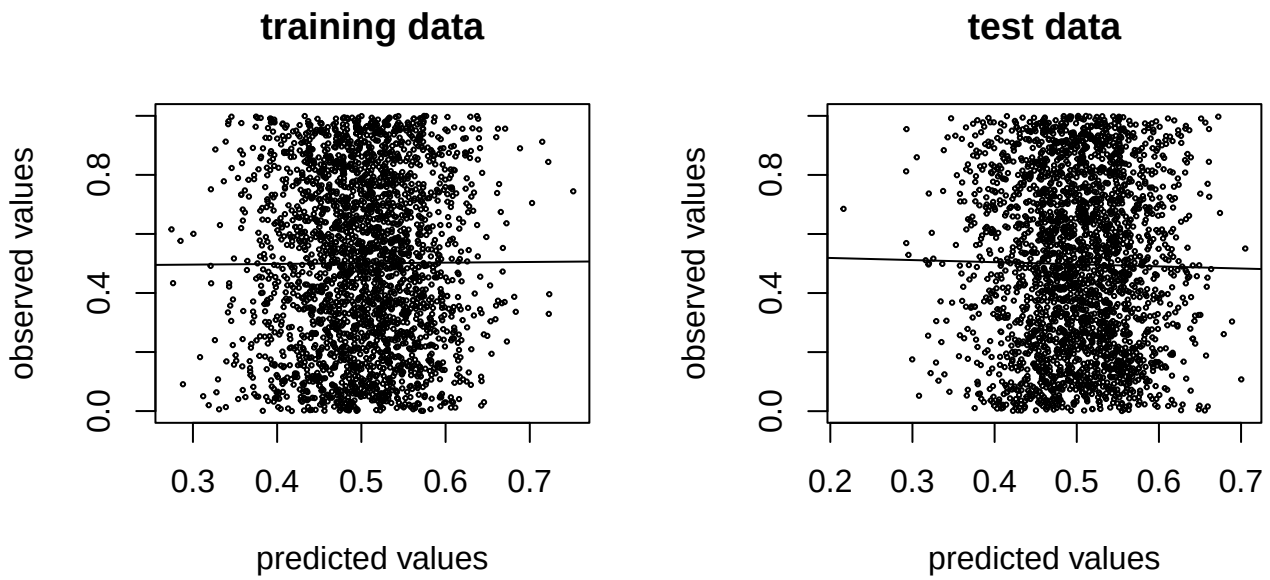
Random forest test for the default generator in R, using $m = 3$

```
rftest(u, m = 3)
```



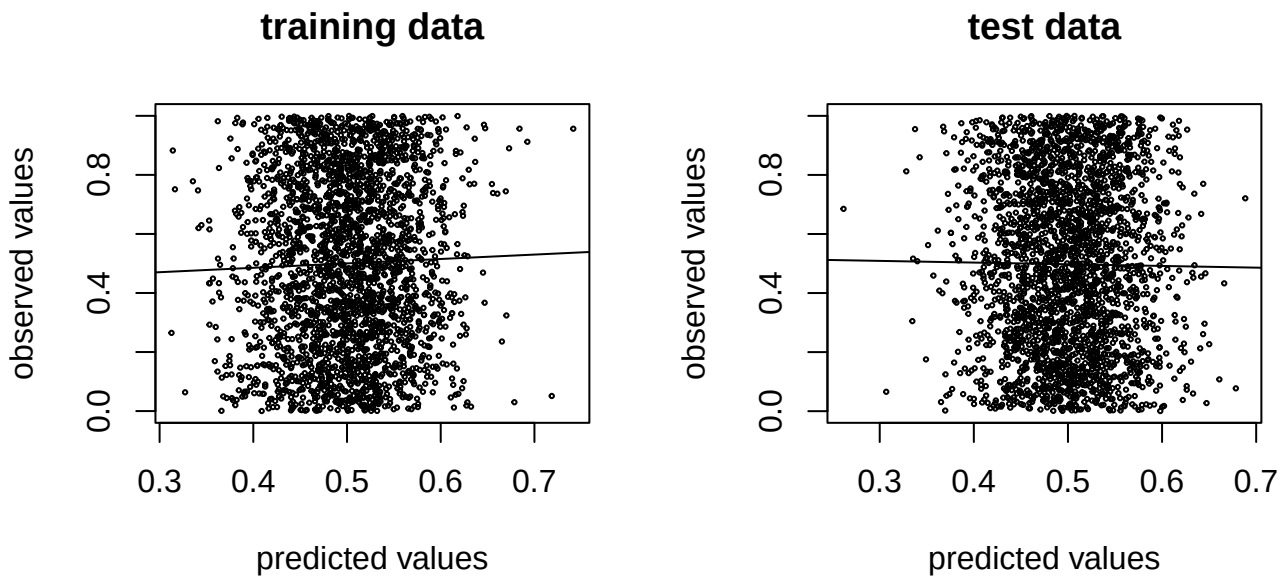
Random forest test for the default generator in R, using $m = 4$

```
rftest(u, m = 4)
```



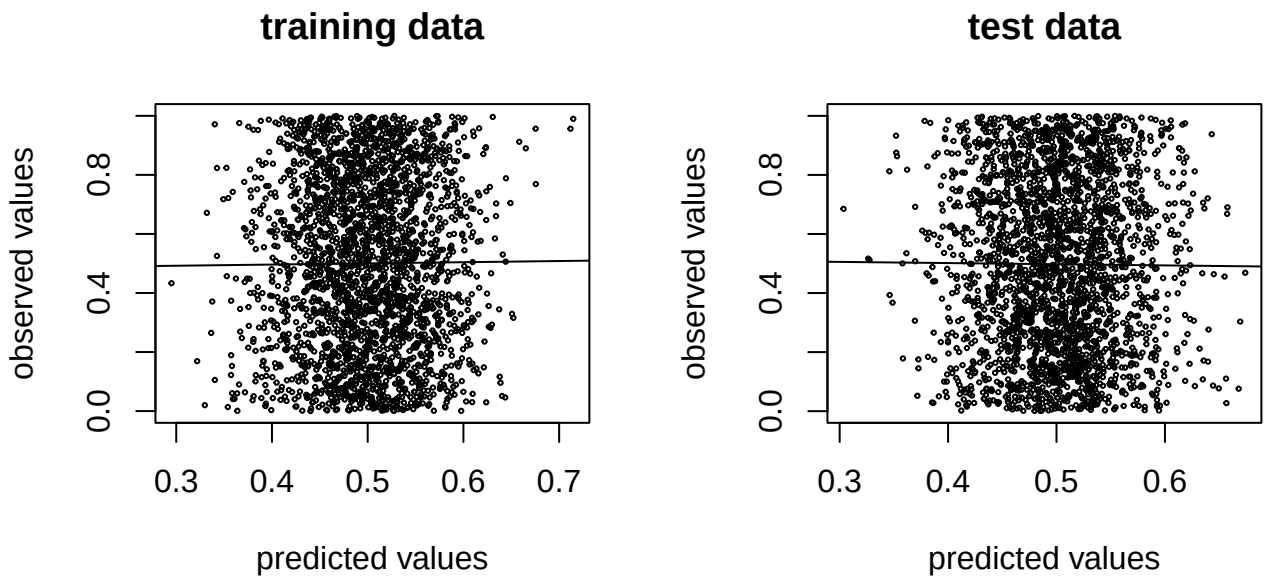
Random forest test for the default generator in R, using $m = 5$

```
rftest(u, m = 5)
```



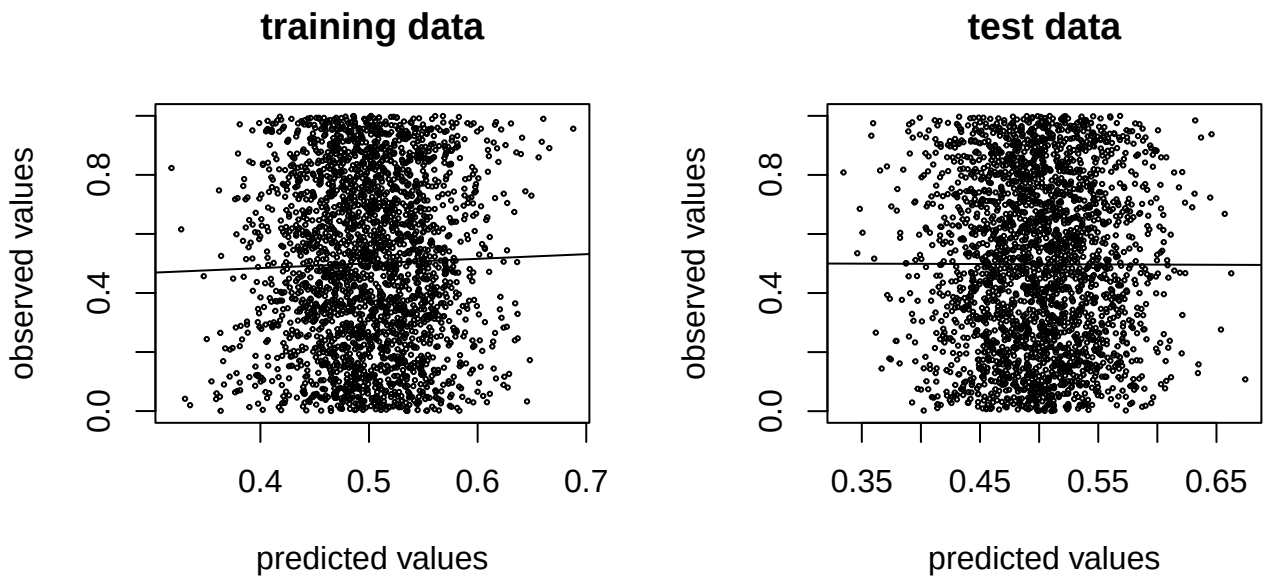
Random forest test for the default generator in R, using $m = 6$

```
rftest(u, m = 6)
```



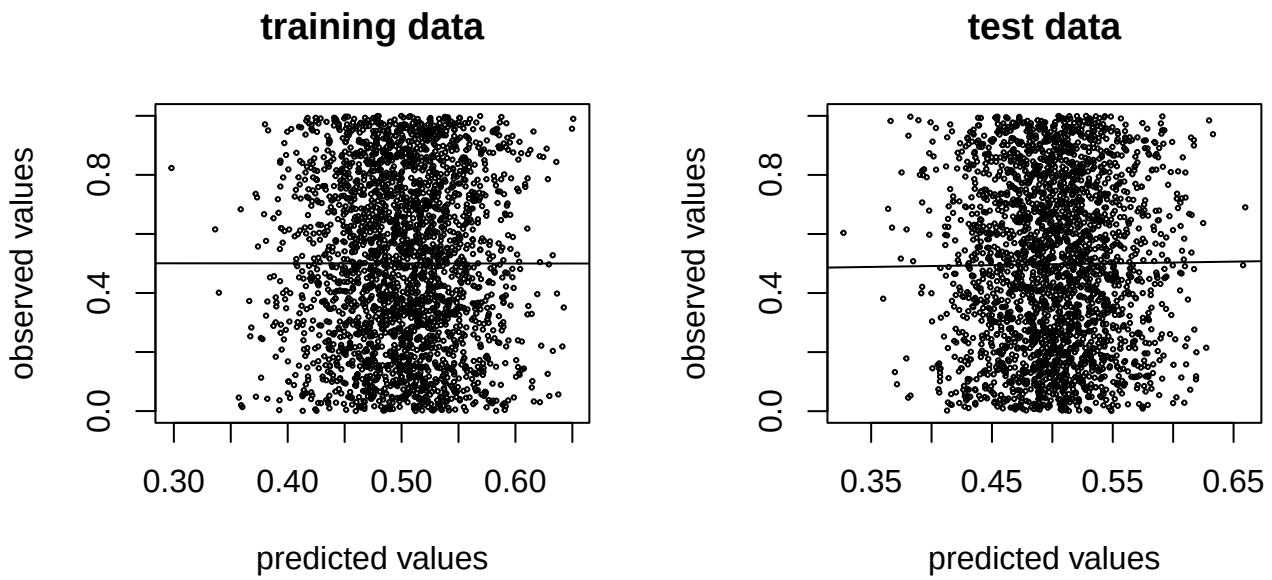
Random forest test for the default generator in R, using $m = 7$

```
rftest(u, m = 7)
```



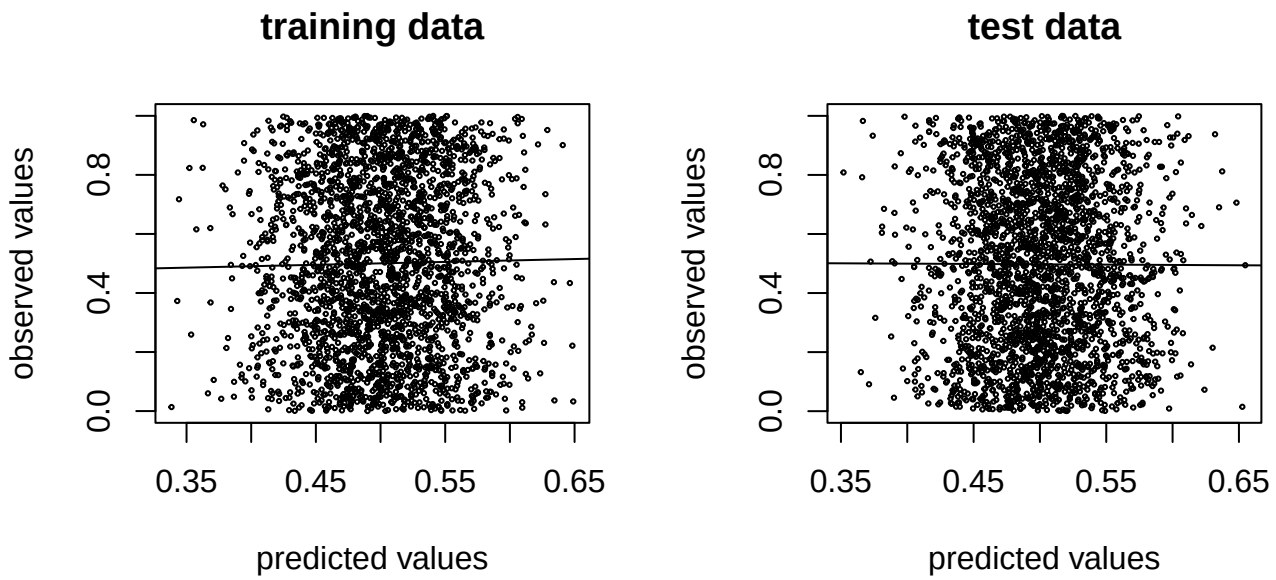
Random forest test for the default generator in R, using $m = 8$

```
rftest(u, m = 8)
```



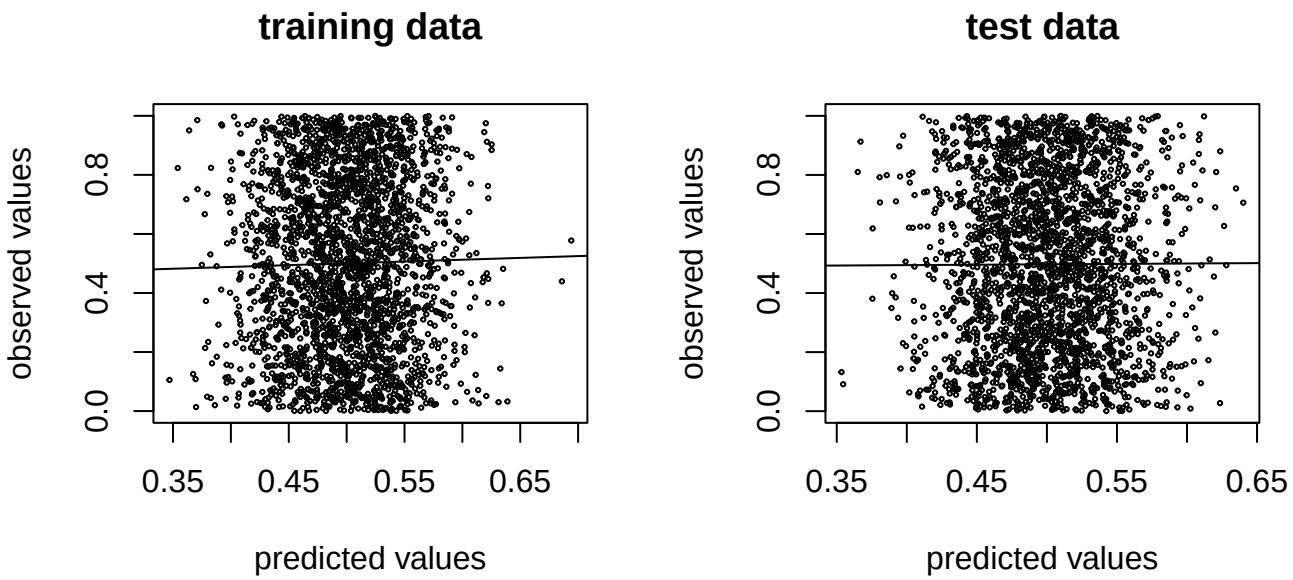
Random forest test for the default generator in R, using $m = 9$

```
rftest(u, m = 9)
```



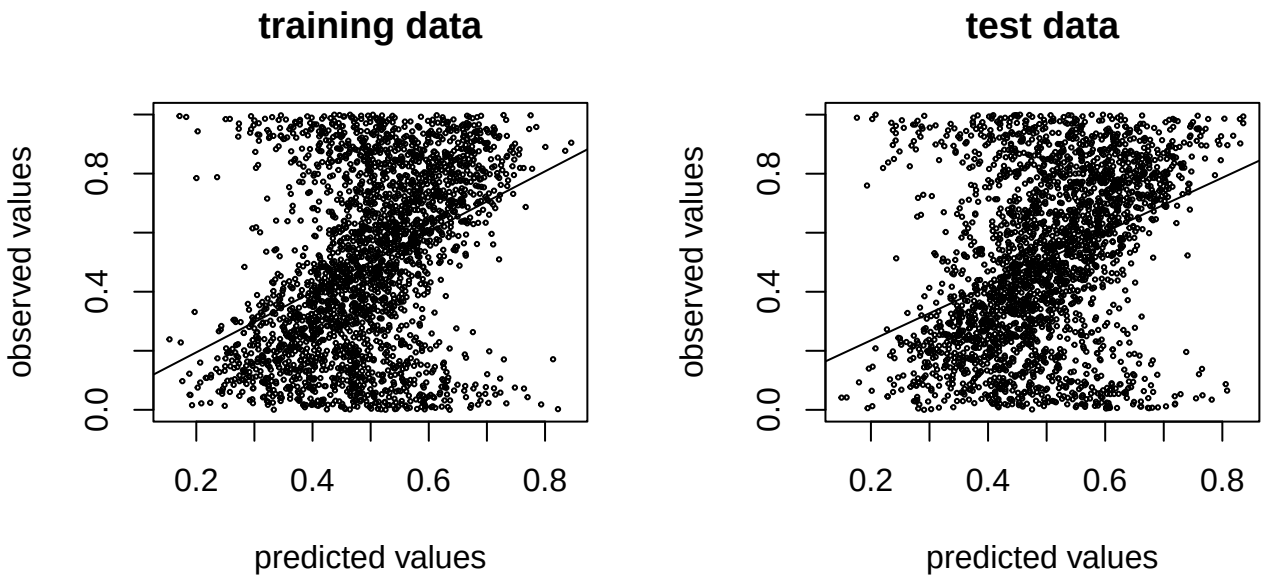
Random forest test for the default generator in R, using $m = 10$

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



Applying the random forest test with $m = 2$ to RANDU

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



This result is consistent with the earlier analysis that indicates that the RANDU generator will not produce good unpredictable numbers.

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)*) These are a special case of the F_2 generators**

5. **Multiply-with-carry generators**

6. **Combination generators**

*`runif` in R.

The Linear Congruential Method

This method is an extension of the Multiplicative Congruential Generator.

Ingredients:

m**: the modulus;** $m > 0$**.**

a**: the multiplier;** $0 \leq a < m$**.**

c**: the increment;** $0 \leq c < m$**.**

x_0**: the starting value, or seed;** $0 \leq x_0 < m$**.**

A linear congruential sequence of random numbers is generated using

$$x_{n+1} = (ax_n + c) \textbf{ mod } m.$$

Conditions Which Prevent Premature Cycling

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**

   (Two integers are relatively prime if there is no integer greater than one that divides them both (that is, their greatest common divisor is one). For example, 12 and 13 are relatively prime, but 12 and 14 are not.);

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.**

For example, if $m = 8$, $b = 4$ and $x_0 = 3$, and $c = 3$. Then, $a = 5$ and the sequence is

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

which has a cycle length 8.

Implementation

```
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).

Implementation

Run example with default settings and seed 372737:

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

## [1] 0.691467 0.707138 0.734528 0.773636

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

## [1] 0
```

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

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).

Proof:

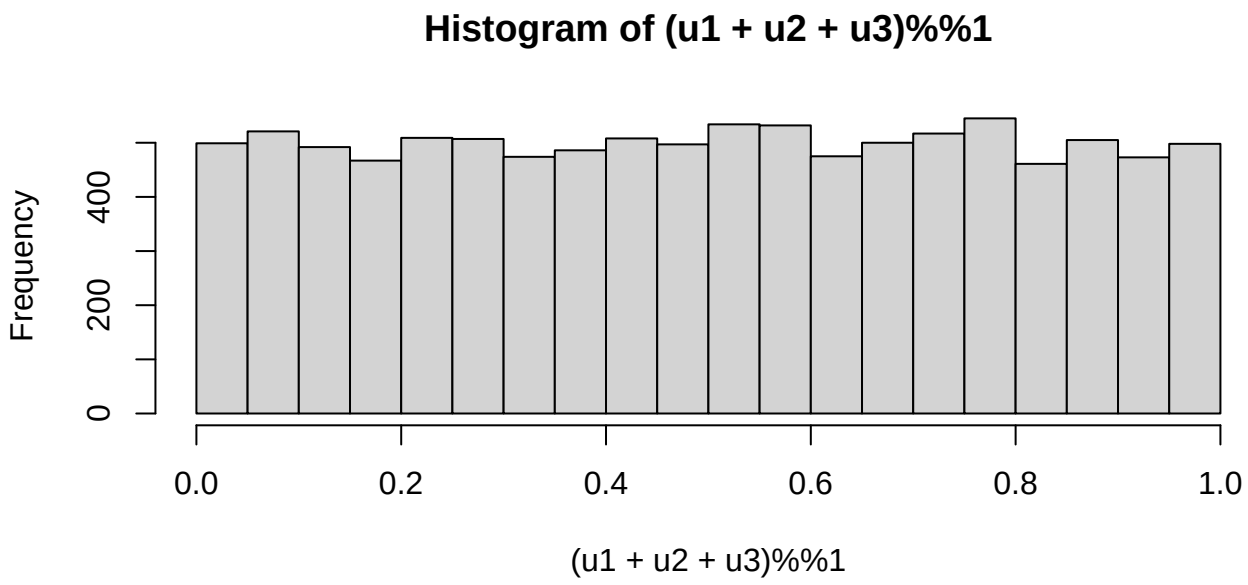
- **When $n = 2$, calculate $P(V < v)$ by conditioning on the value of $[U_1 + U_2]$.**

- **When $n \geq 2$, use induction, with facts like $[v + [z]] = [v] + [z]$.**

Examples: Super-Duper, Wichmann-Hill

Numerical demonstration (n=3):

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



Combination Generators: Wichman and Hill

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

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

Example of use:

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


## [1] 0.285345 0.895050 0.545943 0.736370 0.972083
```

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

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

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

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

$$z_n = (x_n - y_n) \textbf{ mod } m_1.$$

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

"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).**

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 $X = g(U)$.

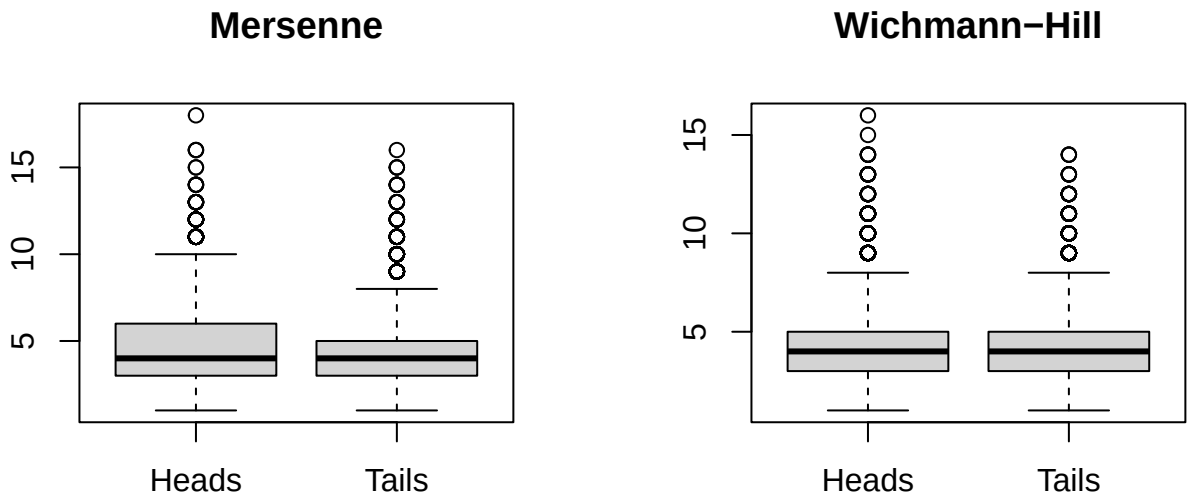
- **Convert U to its binary representation and retain the leading 32 binary digits.**

- **Return the maximum runlength of 0's in the binary representation.**

Seeing the problem for ourselves

- **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)**



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++.

Accessing better generators in R

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
##  [5] 0.00991752 0.90312322 0.14127554 0.44023898
##  [9] 0.50391344 0.88495743
```

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
```

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] 0.4168 0.8758 0.1363 0.6713 0.6155
```

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
```

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) 32.51 33.00 40.77  37.54 42.42  89.72   100
##    rcmrg(2e+06) 60.06 61.56 69.49  65.14 70.56 126.64   100
##    rlxs0(2e+06) 58.40 62.05 73.41  68.43 76.72 170.83   100
##  dqrunif(2e+06) 14.13 14.47 18.65  15.75 19.93  67.85   100

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

## Unit: milliseconds
##             expr   min    lq  mean median   uq   max neval
##  dqrunif(2e+06) 13.55 13.78 16.39  14.11 15.7 63.29   100

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

## Unit: milliseconds
##             expr   min    lq mean median    uq  max neval
##  dqrunif(2e+06) 13.14 13.32 15.5  13.65 13.81 61.1   100
```

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*

*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.

***References**

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

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

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.

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

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*.

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*.

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

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

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.**