Random Thoughts About Pseudorandom Numbers

W. John Braun, UBC

MATH 590

February 26, 2024

1. Introduction
2. Desirable properties of generators
3. Types of generators
4. Seed issues - an illustrative example
5. RNG testing
6. It's "high time"
7. Accessing better generators in R

Introduction

Random number generation is too important to be left to chance (Coveyou, 1969).

Introduction: Nonlinearity and unpredictable sequences

To obtain unpredictable 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
}
```

Introduction: 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.

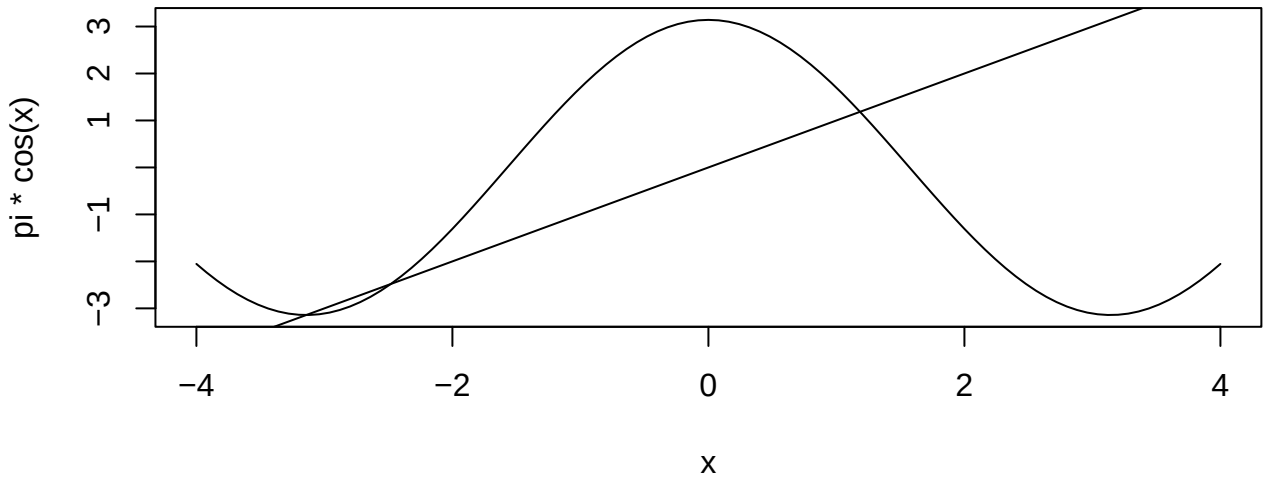
The convergence in this example occurs because the mapping $x = \pi \cos(x)$ has a stable fixed point at $x = -\pi$.

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.

Introduction: 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)
```



*Any numerical analyst should recognize that this is a statistical prank being played on them.

Introduction: Nonlinearity - Example

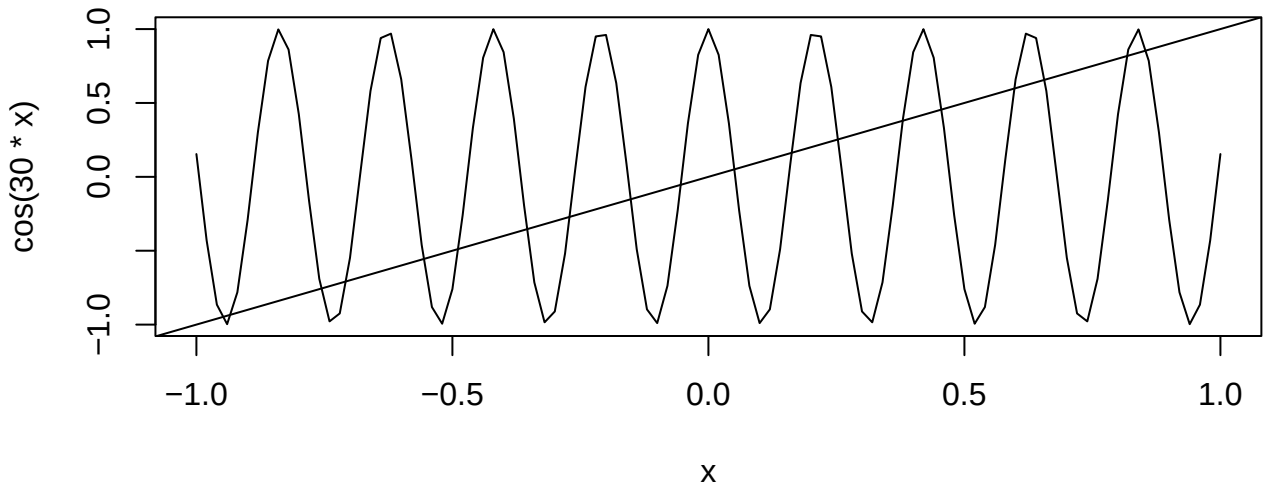
A mapping for a pseudorandom number generator should not have a stable fixed point.

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.**

Introduction: Nonlinearity - Example

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



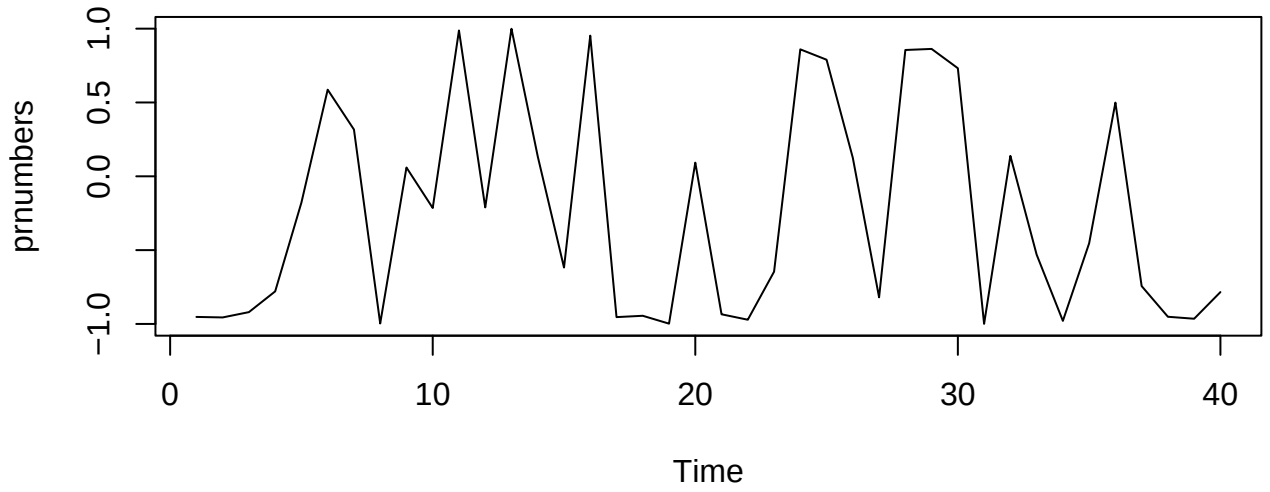
Note that the slopes near fixed points (points of intersection between the overlaid line and the curve) are also relatively large, inducing instability.

Introduction: Nonlinearity - Example

Illustration:

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

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

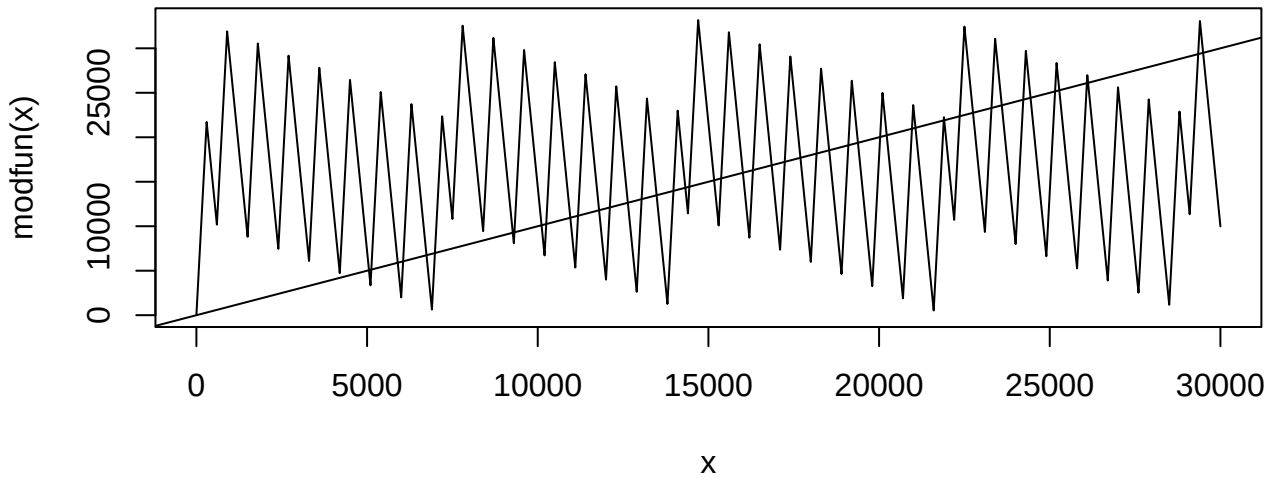


The numbers produced by this function are certainly less predictable than before, as can be seen in the trace plot above.

Introduction: Nonlinearity

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

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

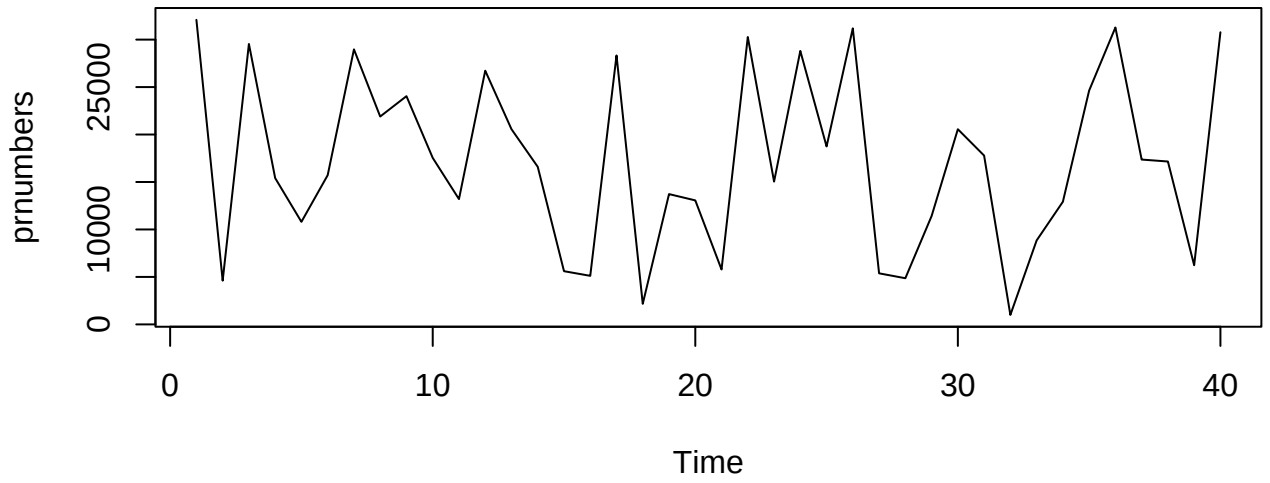


Introduction: Nonlinearity - Example

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**

Types of 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: $x_{n+1} = (ax_n + c) \bmod m$ for a positive integer c.

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.

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

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** $(0, 1)$.

Seed issues

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 is critical (Savage et al., 1994)

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
## [16]  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"
## [12] "T" "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
## [16]  1
```

```
##  [1] "H T" "T H" "H H" "H T" "T T" "T T" "T T"
##  [8] "T T" "T H" "H T" "T T" "T H" "H H" "H H"
## [15] "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"
##  [6] "T T T" "T T T" "T T H" "T H T" "H T T"
## [11] "T T H" "T H H" "H H H" "H H H" "H H H"
## [16] "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
```

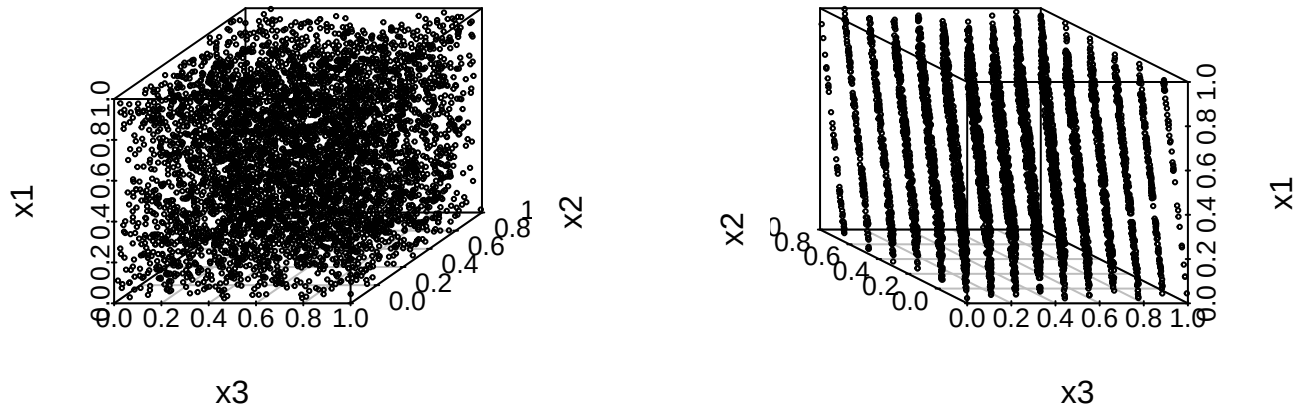
Equally likely outcomes are assured if only one occurrence of each outcome is allowed. Restricting the possible seeds to the set $1, 3, 6, 7, 9, 12, 15, 16\}$ **will perfectly produce a set of three independent coin tosses.**

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

"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 and Dieharder**
- **BigCrush**
- **TestU01**

These are all collections of statistical tests. Gevorkyan et al. (2020) provides an up-to-date review.

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 R *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.

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 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.

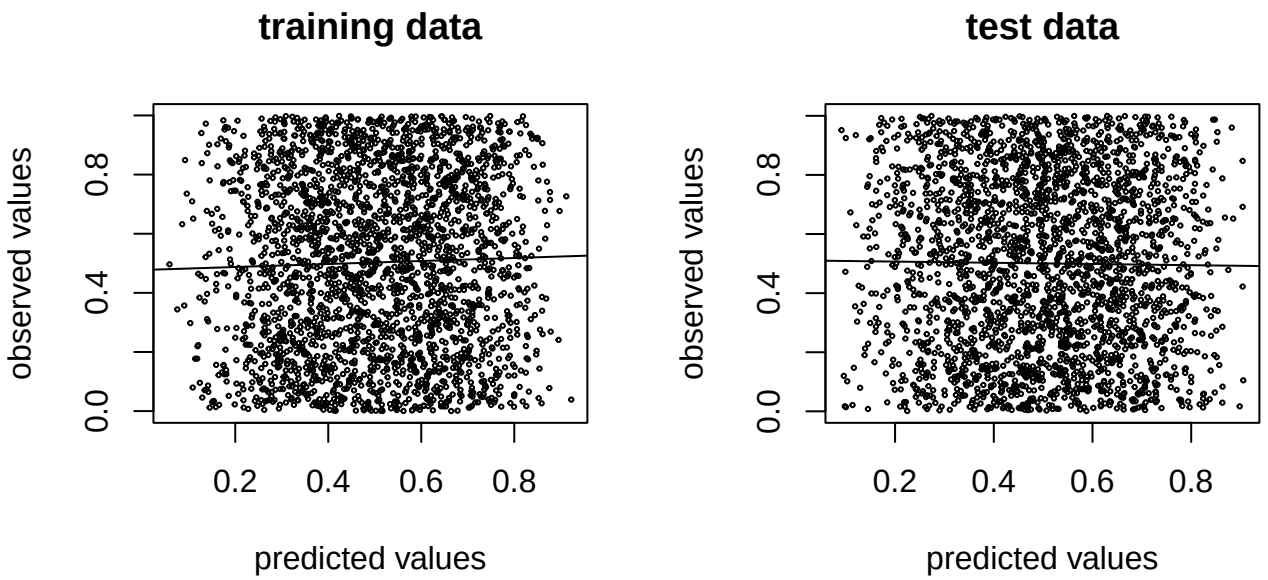
Example - testing the default R generator

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)
```

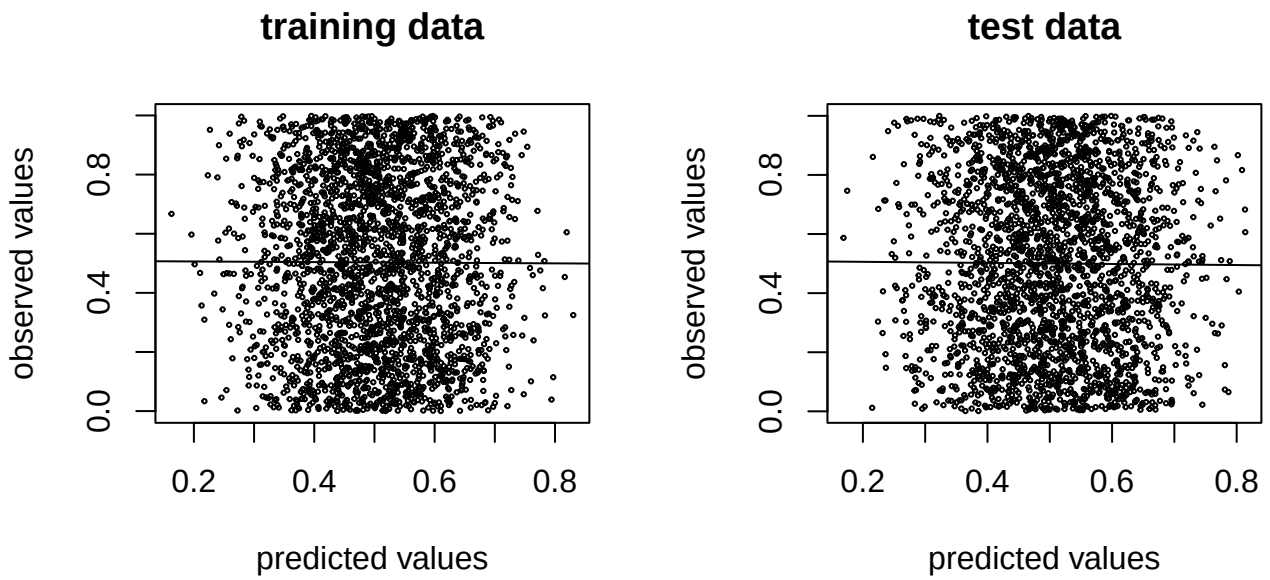
Random forest test for the default generator in R, using $m = 1$

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



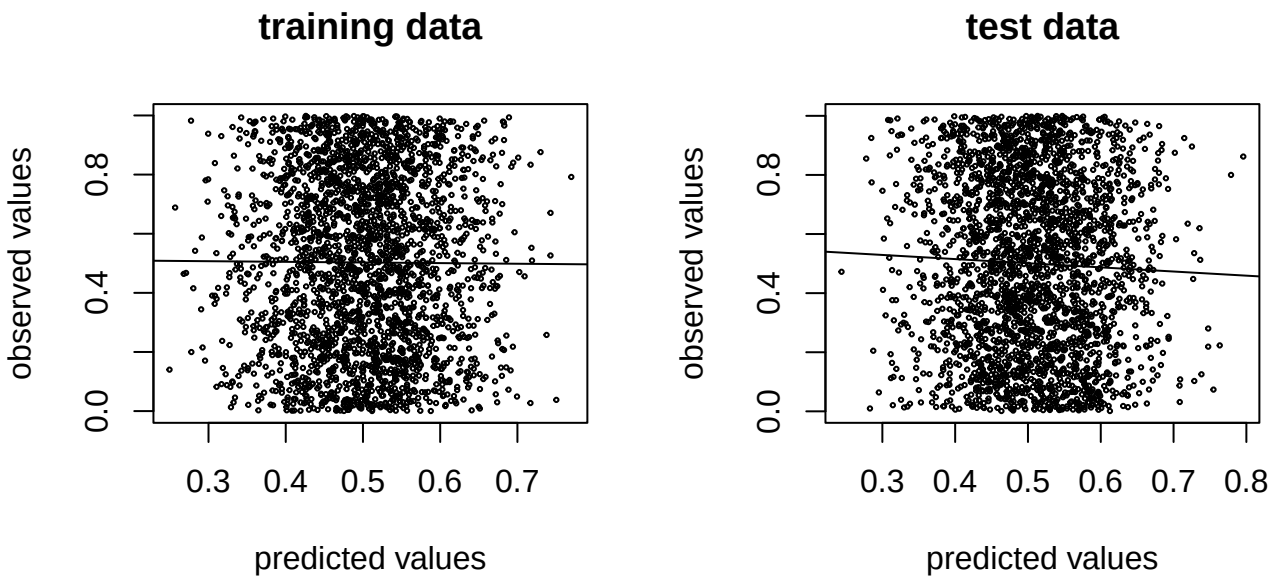
Random forest test for the default generator in R, using $m = 2$

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



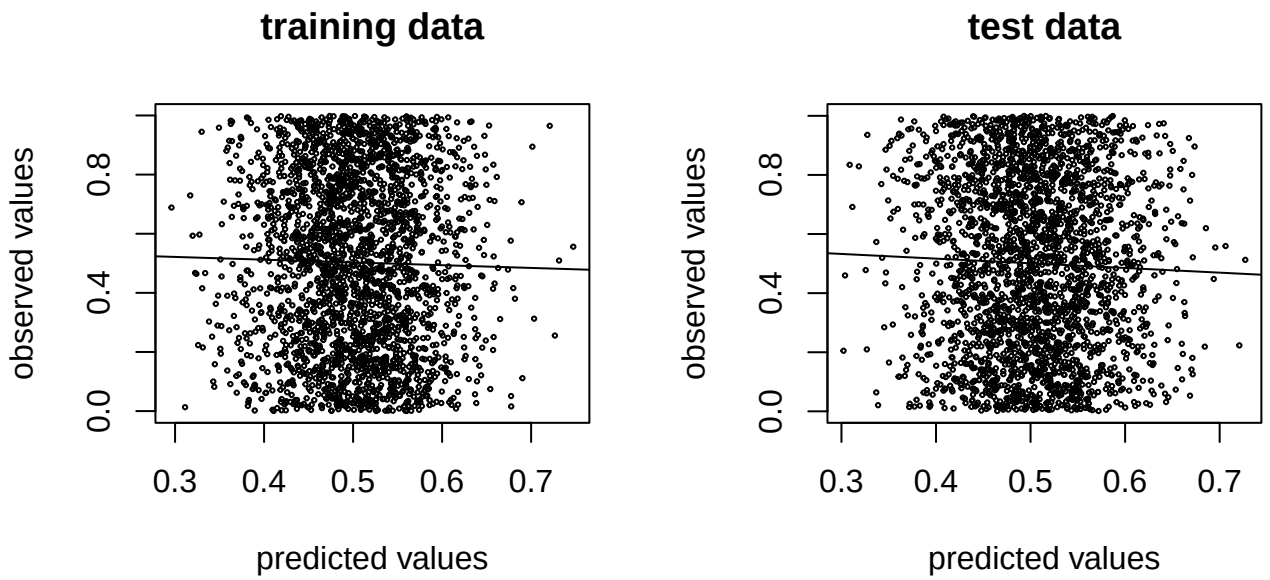
Random forest test for the default generator in R, using $m = 3$

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



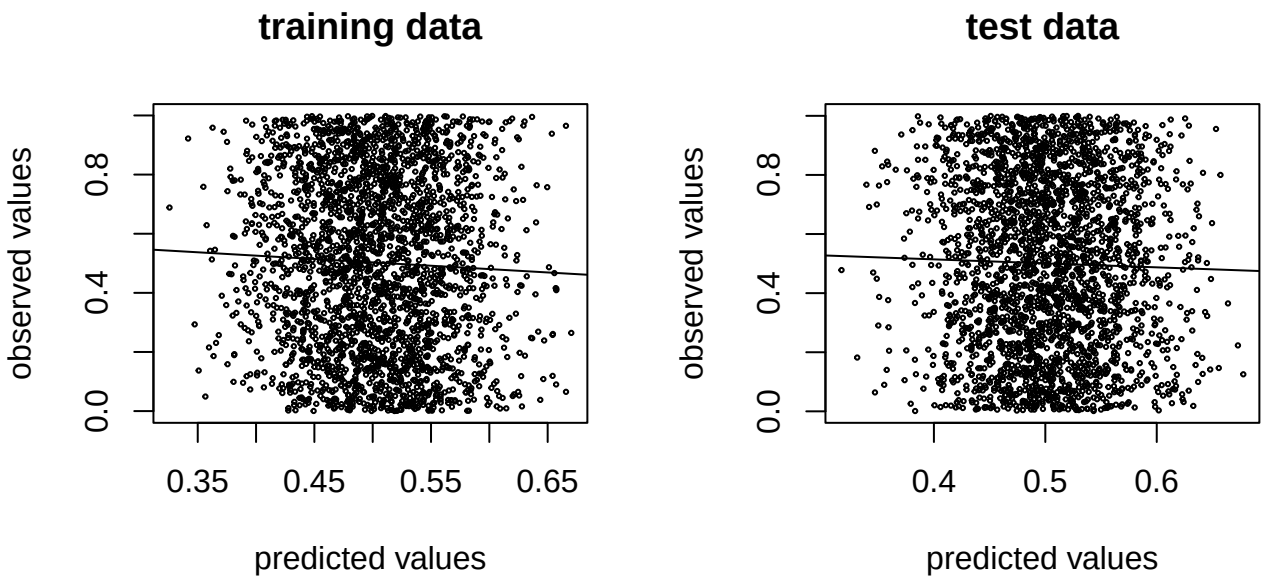
Random forest test for the default generator in R, using $m = 4$

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



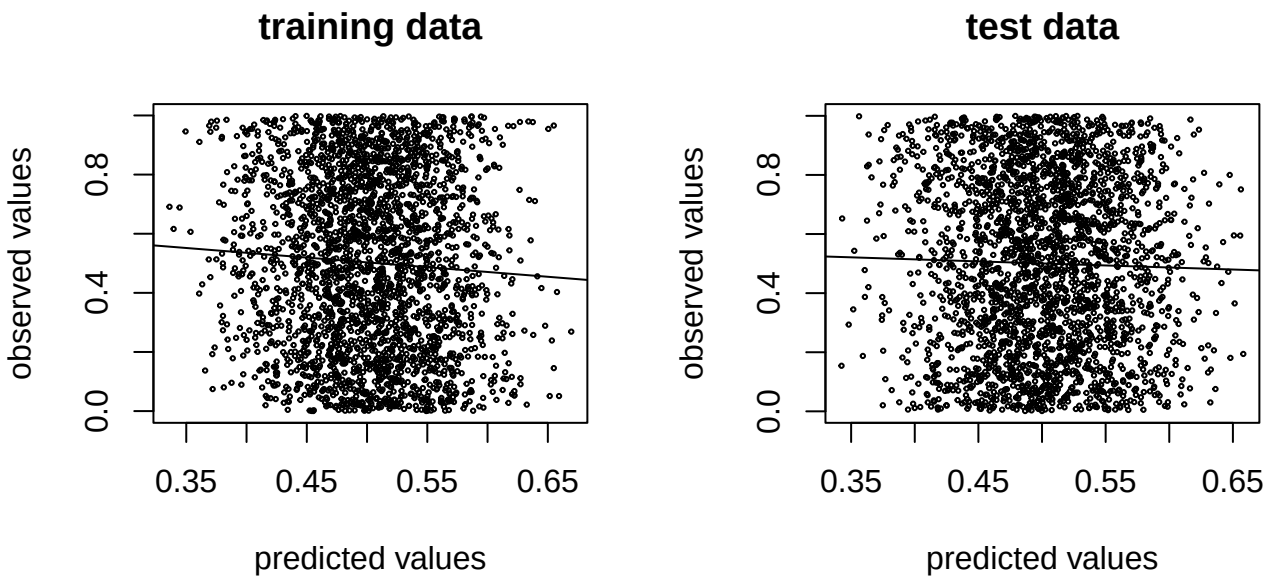
Random forest test for the default generator in R, using $m = 5$

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



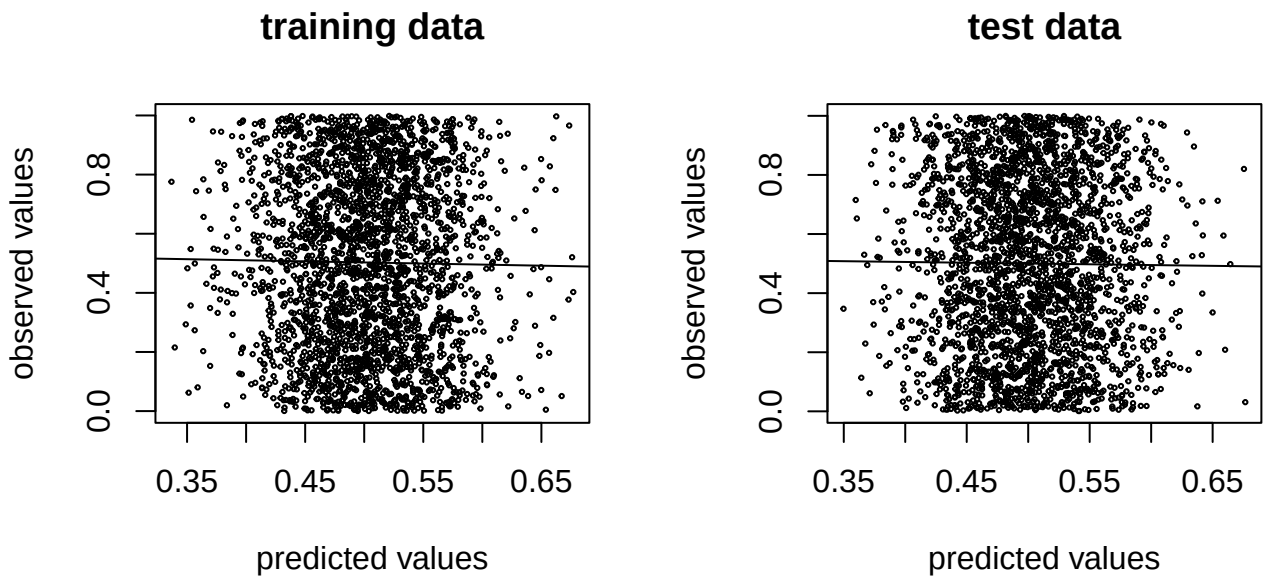
Random forest test for the default generator in R, using $m = 6$

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



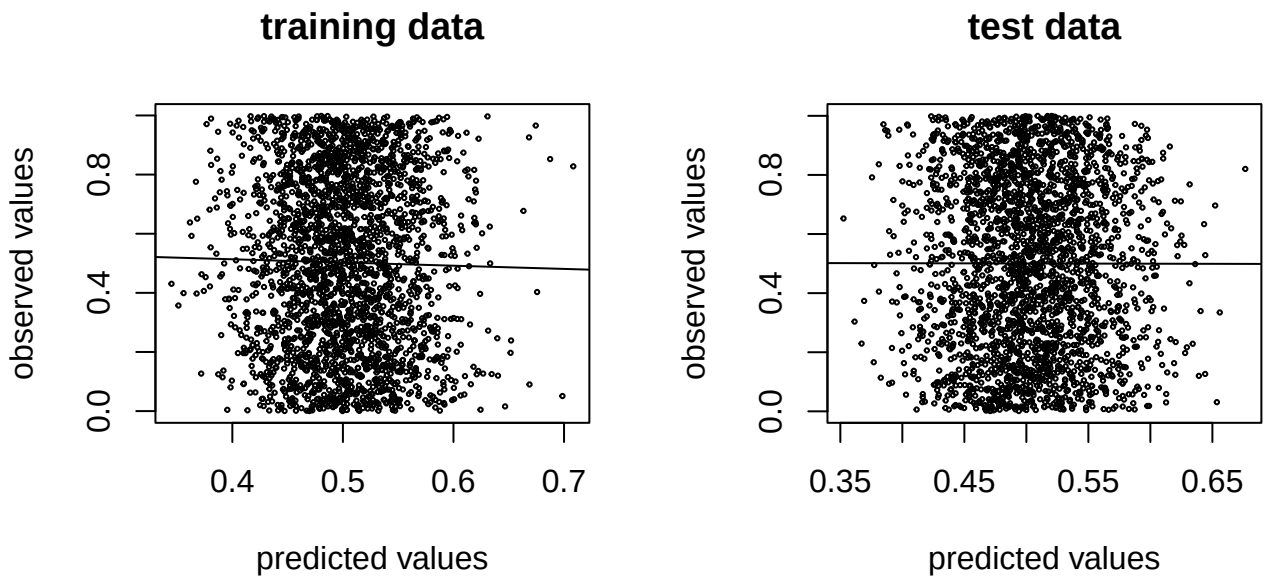
Random forest test for the default generator in R, using $m = 7$

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



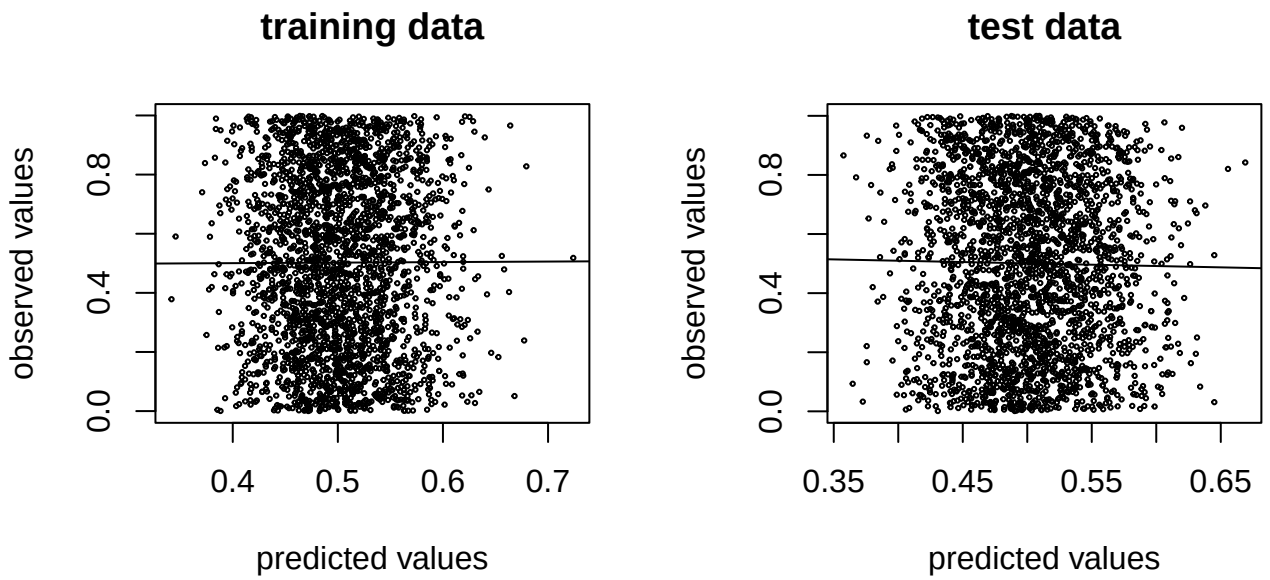
Random forest test for the default generator in R, using $m = 8$

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



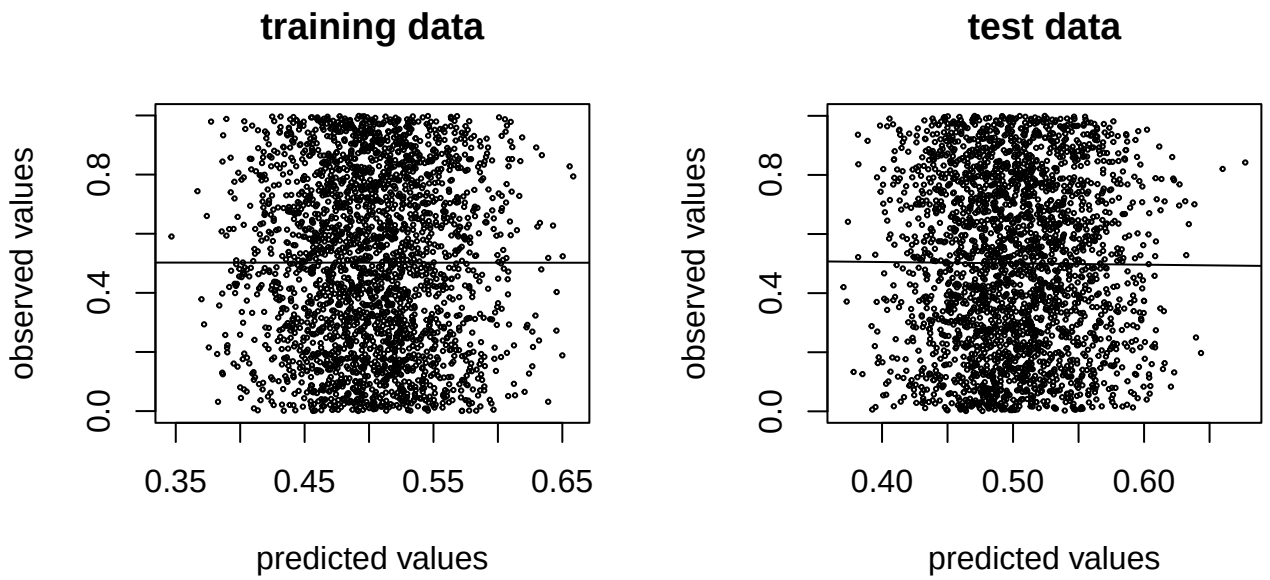
Random forest test for the default generator in R, using $m = 9$

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



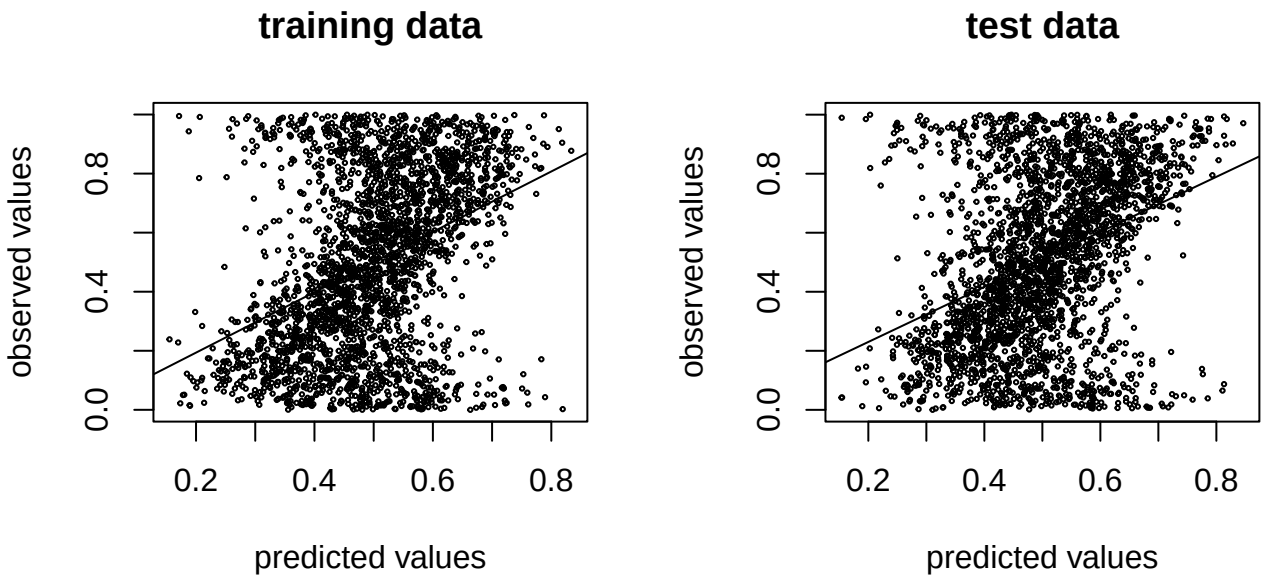
Random forest test for the default generator in R, using $m = 10$

```
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.

“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 $X = g(U)$.

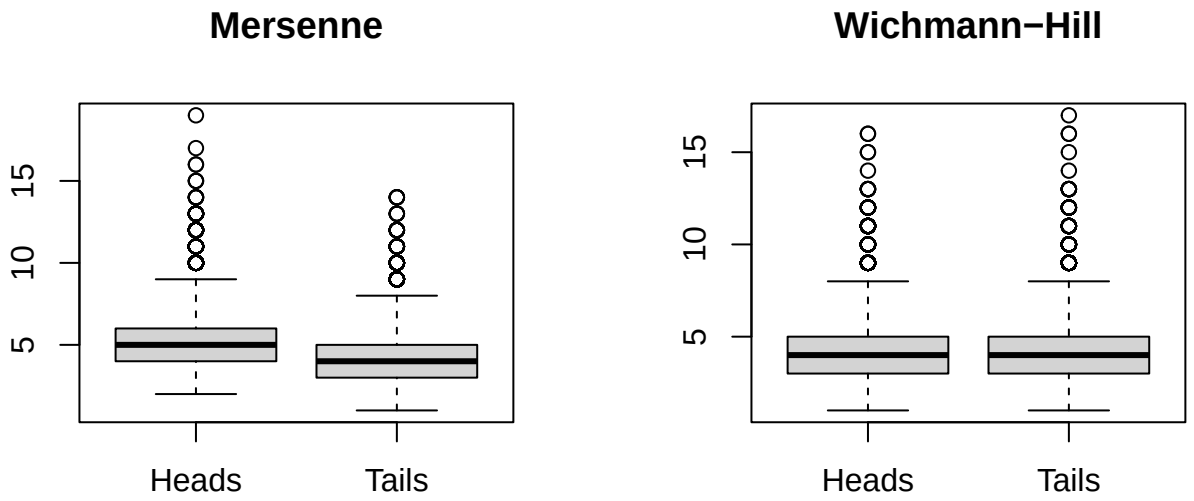
- **Convert U to its binary representation and retain the leading 32 binary digits.**

- **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] 0.7365 0.3546 0.7940 0.6209 0.1207
```

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) 53.884 57.93 58.50  58.12 58.82 73.06   100
##    rcmrg(2e+06) 45.638 49.71 50.55  49.92 50.81 61.63   100
##    rlxs0(2e+06) 46.485 49.73 50.62  50.02 50.71 61.14   100
##  dqrunif(2e+06)  8.293 12.29 12.48  12.31 12.39 22.16   100

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

## Unit: milliseconds
##            expr   min    lq mean median    uq   max neval
##  dqrunif(2e+06) 7.888 11.87 12.1  11.92 12.05 22.94   100

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

## Unit: milliseconds
##            expr   min    lq  mean median    uq   max neval
##  dqrunif(2e+06) 6.757 10.66 10.91  10.73 10.92 21.17   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.

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.

Gevorkyan, M. N., Kulyabov, D. S., Demidova, A. V., and Korolkova, A. V. (2020). A practical approach to testing random number generators in

computer algebra systems. *Computational Mathematics and Mathematical Physics*, 60:65–73.

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*****.**

Savage, S., Schrage, L., Lewis, P., and Empey, D. (1994). The bad seeds—a parallel random number generation problem weeded out long ago, crops up again. In ***Unpublished manuscript dated 14th January 1994 (presented at the 35th TIMS/ORSA Joint National Meeting in Chicago in 1993)*****.**

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.**