Fire

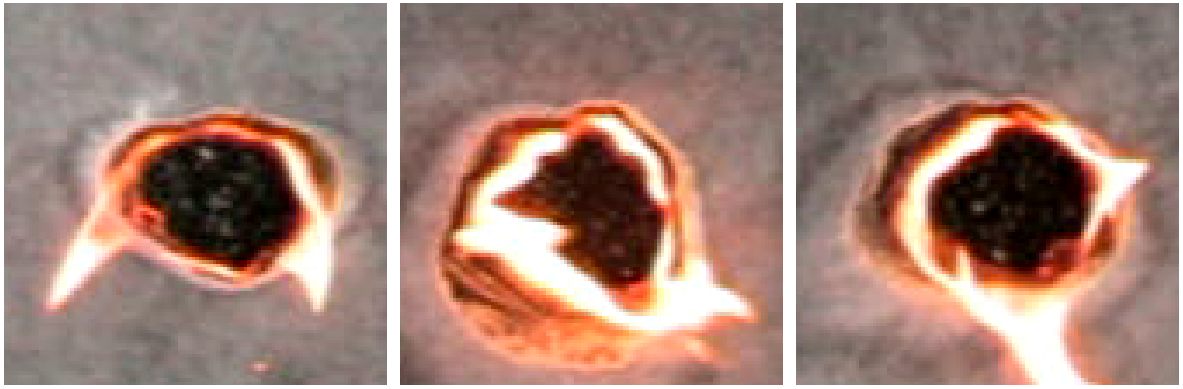
Many of us are fascinated by watching a fire burn. Ideas of uncertainty and randomness can be introduced by watching the patterns produced by an evolving fire.

Rectangular sheets of waxed paper have been burned in a lab.

In each case, the paper has been ignited at a point near the center, and the fire burns for a few seconds.

The next slide displays screen shots from these three fires a few seconds after ignition.

Fire



Photos of three tiny fires, burning under identical conditions for the same length of time.

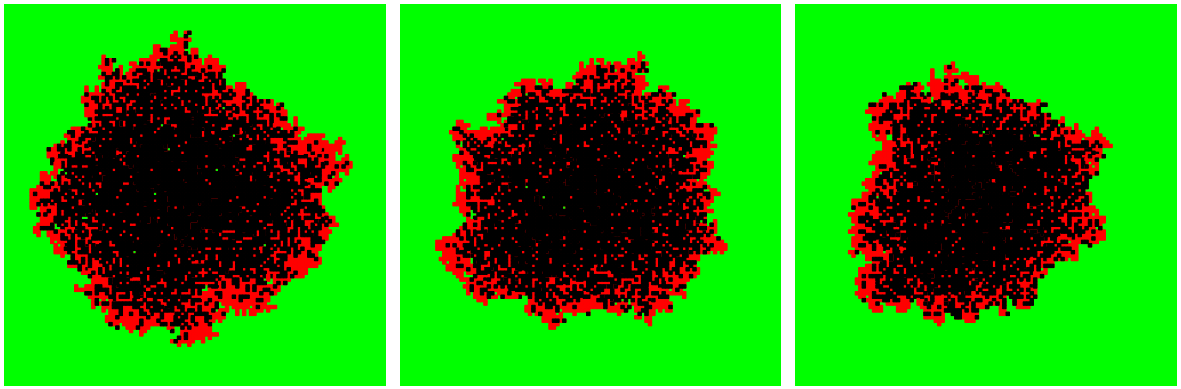
Fire

The demonstration shows that different burn patterns result, even under fairly controlled conditions: in each case, we have used the same type of burning material, as well as the same slope, aspect, temperature, moisture and wind conditions.

Very slight and unmeasured differences in some or all of these conditions are leading to unpredictable differences in shape and rate of spread.

Simulated Fire

The basic characteristics of fire spread and burning can be captured with a simple Markovian fire simulation model (see Boychuk et al, 2007).



Plots of simulated fires, burning under identical conditions for the same length of time.

The burning areas are represented by red pixels, and the burned-out areas are black.

Simulated Fire

The three fires pictured in the previous slide have evolved differently and have not been growing at exactly the same speed.

How do we generate different pictures when input conditions are the same?

More generally, how do we generate unpredictable sequences of numbers on a computer, when the computer is designed to correctly give the same output whenever given the same input?

Generating Pseudorandom Sequences

Although the multiplicative congruential generator does not necessarily generate the best quality pseudorandom sequences, it still serves an instructive purpose because of its inherent simplicity.

You only to know how to divide numbers and calculate remainders. e.g.

1. **What is the remainder after dividing 7 by 3?**

2. **What is the remainder after dividing 19 by 7?**

Generating Pseudorandom Sequences

The following simple sequence of remainders, result from a multiplicative congruential algorithm:

For example, start with 10

$$\textbf{Remainder}((13 \times 10)/31) = 6$$

$$\textbf{Remainder}((13 \times 6)/31) = 16$$

$$\textbf{Remainder}((13 \times 16)/31) = 22$$

$$\textbf{Remainder}((13 \times 22)/31) = 7$$

$$\textbf{Remainder}((13 \times 7)/31) = 29$$

$$\textbf{Remainder}((13 \times 29)/31) = 5$$

Is there a pattern to the output?

Of course, there is some kind of pattern, but it is not a simple pattern.

The operation of taking remainders is giving unpredictable answers, though the small size of the numbers allows us to predict easily, if we know the rule.

Generating Pseudorandom Sequences

With the use of R, we can see what happens with larger values:

```
ix <- 20003
       ix <- (171*ix) %% 30269
ix

## [1] 116
```

```
       ix <- (171*ix) %% 30269
ix

## [1] 19836
```

```
       ix <- (171*ix) %% 30269
ix

## [1] 1828
```

```
      ix <-  (171*ix) %% 30269
ix

## [1] 9898
```

```
      ix <- (171*ix) %% 30269
ix

## [1] 27763
```

Now, we have some difficulty seeing a pattern.

Simulating Dice

Having been motivated by the fire example and having seen how to create unpredictable sequences of numbers, let's consider a simpler problem.

Rolling pairs of dice does not always yield the same outcome. The following code allows us to simulate pairs of dice:

```
sample(1:6, replace=TRUE, size=2)

## [1] 5 5
```

Roll again:

```
sample(1:6, replace=TRUE, size=2)

## [1] 4 5
```

Simulating Dice

We can sum the values of the dice to get a single value:

```
sum(sample(1:6, replace=TRUE, size=2))


## [1] 3
```

The following code shows how the sums of 10 pairs of dice can be simulated.

```
dice <- numeric(10)
for (i in 1:10) {
    dice[i] <- sum(sample(1:6, replace=TRUE, size=2))
}
dice


##  [1]  6  9  7 12 10  7  6 11  9  6
```

Simulating Dice

We can tabulate these values easily:

```
table(dice)

## dice
##  6  7  9 10 11 12
##  3  2  2  1  1  1
```

The following code will simulate a large sample:

```
dice <- numeric(10000)
for (i in 1:10000) {
    dice[i] <- sum(sample(1:6, replace=TRUE, size=2))
}
table(dice)

## dice
##    2    3    4    5    6    7    8    9   10   11   12
##  271  521  862 1079 1405 1614 1458 1084  861  563  282
```

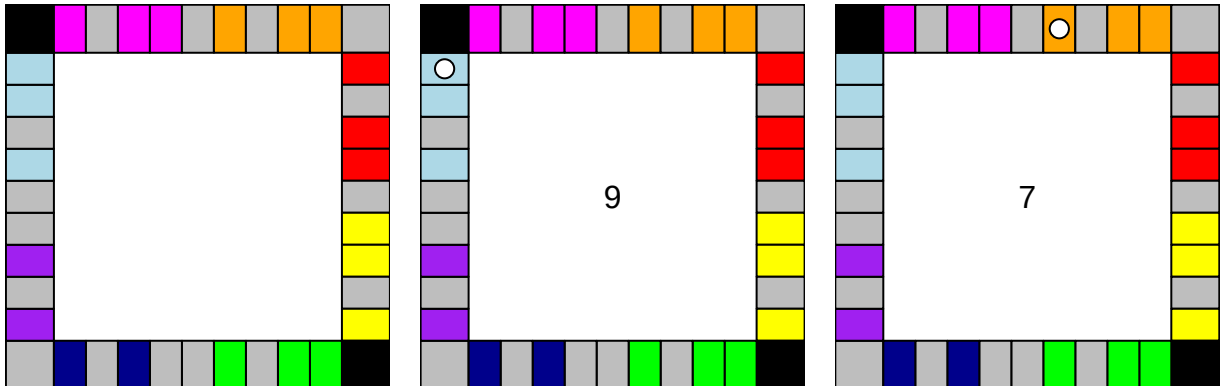
The pattern of frequencies can now be seen clearly. 7's are lucky, and 2's and 12's are not.

See also

http://rtricks4kids.ok.ubc.ca/test/RTrix/RT4K/dice/

Simulating a Monopoly Game - A Markov Chain

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.



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 represents 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.

Simulating a Monopoly Game

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. The basic 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.

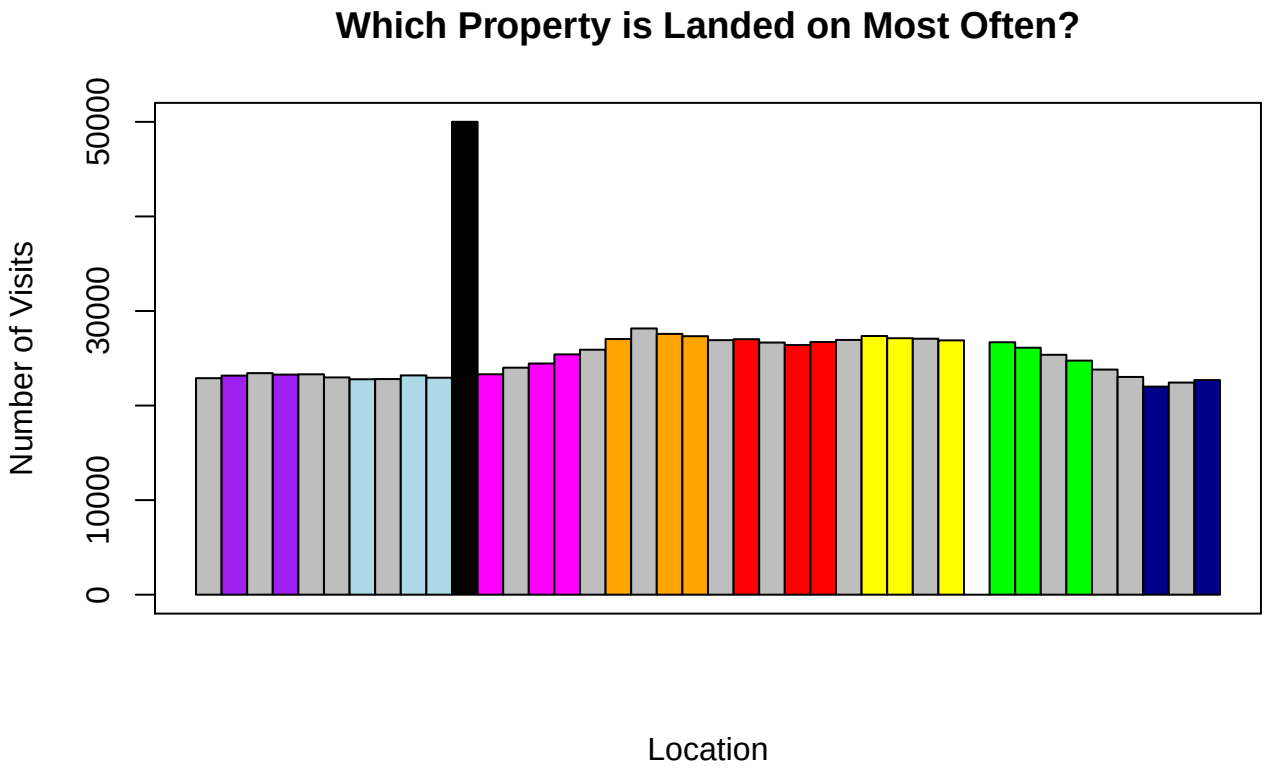
Simulating a Monopoly Game

The following code produces a histogram whose bars are color-coded according to the property colors.

```
propertycolors <- c("grey", "purple", "grey", "purple", "grey",
  "grey", "lightblue", "grey", "lightblue", "lightblue", "black",
  "magenta", "grey", "magenta", "magenta", "grey", "orange",
  "grey", "orange", "orange", "grey", "red", "grey", "red", "red",
  "grey", "yellow","yellow", "grey", "yellow", "black", "green",
   "green", "grey", "green", "grey", "grey", "darkblue", "grey",
 "darkblue")
```

```
hist(lands, breaks=seq(.5, 40.5, 1), col=propertycolors,
xlab="Location", main="",  ylab="Number of Visits", axes=FALSE)
box()
axis(2)
title("Which Property is Landed on Most Often?")
```

Simulating a Monopoly Game



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?

Simulating a Monopoly Game

See

`http://rtricks4kids.ok.ubc.ca/test/RTrix/RT4K/Monopoly/`

for a simple Monopoly simulator.

Let's Recap

Fire, rolling dice, and monopoly games all involve uncertainty (modelled by randomness).

Pseudorandom number generators are the computers way of taking certain input and producing unpredictable output. More precisely, it is hard to predict output.

R has built-in capabilities to generate pseudorandom numbers, including the `sample()` function. The kinds of functions can be used to create relatively sophisticated processes which can be used as simulators for natural phenomena like fire, rolling dice and monopoly.

The hardest part of simulating these kinds of processes is often putting together the visualization tool or nice user interface.

... But the more you know about probability and statistics, the easier it is to build the "engine".

Welcome to Modelling and Simulation I

In this course, you will learn about statistical models and how to simulate from them.

Along the way, we will learn about the most commonly encountered probability distributions and some of their properties.

We will learn about independence and something about why it is important (but unfortunately taken for granted in practice)

We will also learn about some of the basic statistical modelling techniques using simulation.

We begin with the uniform distribution ....

Generation of Pseudorandom Numbers

One of the simplest methods for *s*imulating independent uniform random variables on the interval [0,1] is the multiplicative congruential random number generator.

It produces a sequence of pseudorandom numbers, $u_0, u_1, u_2, \ldots,$ **which can appear like independent uniform random variables on the interval [0,1].**

Multiplicative Congruential Random Number Generators

Let m be a large integer, and let b be another integer which is smaller than m.

The value of b is often chosen to be near the square root of m.

Different values of b and m give rise to pseudorandom number generators of varying quality.

To begin, an integer x_0 is chosen between 1 and m. x_0 is called the seed. We discuss strategies for choosing x_0 later.

Multiplicative Congruential Random Number Generators

Once the seed has been chosen, the generator proceeds as follows:

$$\begin{aligned} x_1 &= b\, x_0 (\ \textbf{mod}\ m) \\ u_1 &= x_1/m. \end{aligned}$$

u_1 is the first pseudorandom number, taking some value between 0 and 1. The second pseudorandom number is then obtained in the same manner:

$$\begin{aligned} x_2 &= b\, x_1\ \textbf{mod}\ m \\ u_2 &= x_2/m. \end{aligned}$$

Multiplicative Congruential Random Number Generators

If m and b are chosen properly and are not disclosed to the user, it is difficult to predict the value of u_2, given the value of u_1 only.
u_2 is another pseudorandom number.

In other words, for most practical purposes u_2 is approximately independent of u_1.

The method continues according to the following formulas:

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

Multiplicative Congruential Random Number Generators

This method produces numbers which are entirely deterministic, but to an observer who doesn't know the formula above, the numbers *a*ppear to be random and unpredictable, at least in the short term.

Example

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

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

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.

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 b and m to ensure that the cycle length is actually m.

Note, for example, what happens when $b = 171$ 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 b and m

To avoid this kind of problem, we should choose m so that it is not divisible by b; 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] 0.76385080 0.61848756 0.76137302 0.19478675 0.30853348
##  [6] 0.75922561 0.82757937 0.51607255 0.24840596 0.47741914
## [11] 0.63867323 0.21312234 0.44391952 0.91023820 0.65073177
## [16] 0.27513297 0.04773861 0.16330239 0.92470845 0.12514454
## [21] 0.39971588 0.35141564 0.09207440 0.74472232 0.34751726
## [26] 0.42545178 0.75225478 0.63556774 0.68208398 0.63636063
## [31] 0.81766824 0.82126929 0.43704780 0.73517460 0.71485678
## [36] 0.24051009 0.12722587 0.75562457 0.21180085 0.21794575
## [41] 0.26872378 0.95176583 0.75195745 0.58472364 0.98774324
## [46] 0.90409330 0.59995375 0.59209092 0.24754700 0.33053619
```

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] 0.005649344 0.966037861 0.192474148 0.913079388 0.136575374
```

Pseudorandom Number Generator Testing

We saw earlier that the multiplicative congruential generator was designed to produce numbers that look random, but what does that mean?

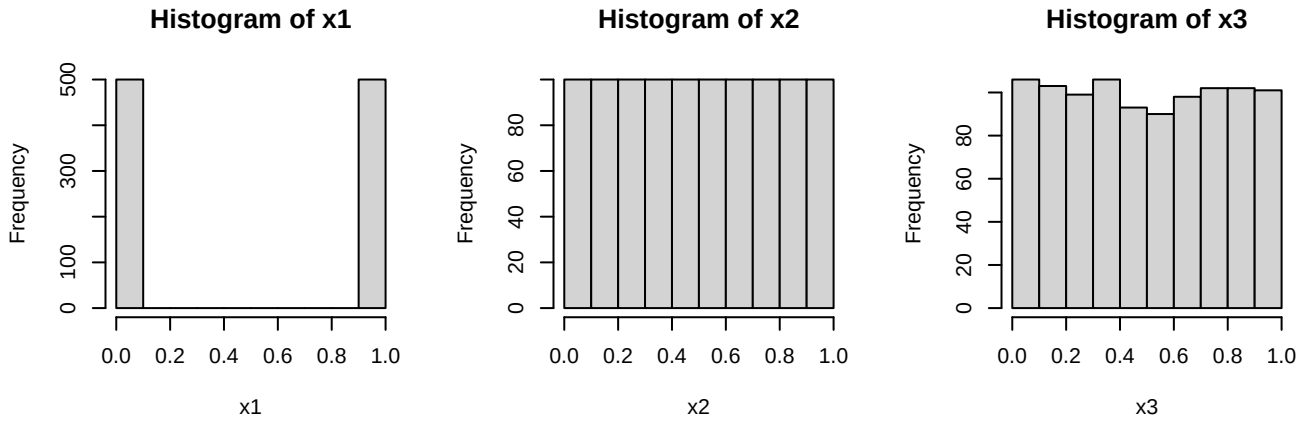
We consider 2 basic checks, noting that there are many other tests that should be performed to ensure high quality performance (see Knuth's Art of Computer Programming).

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)
```

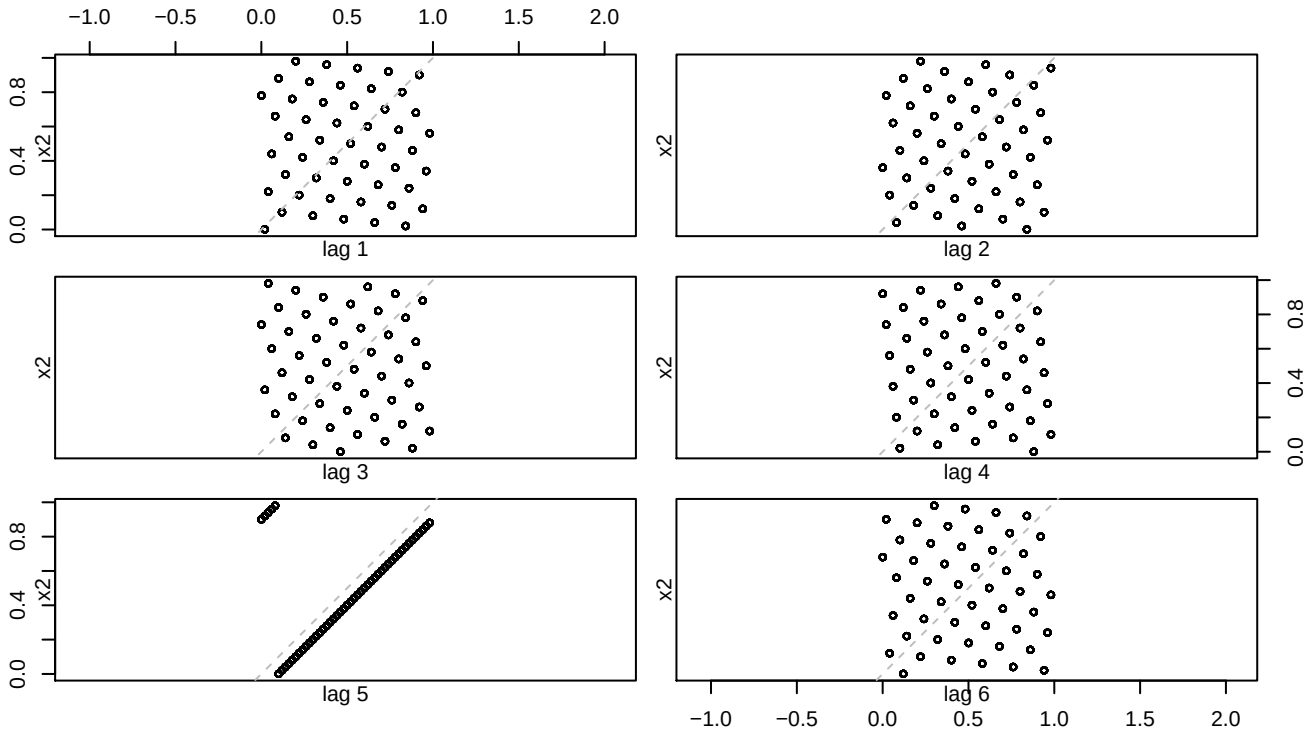


x1 fails; x2 and x3 both appear to be uniform.

Basic tests - autocorrelation

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

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

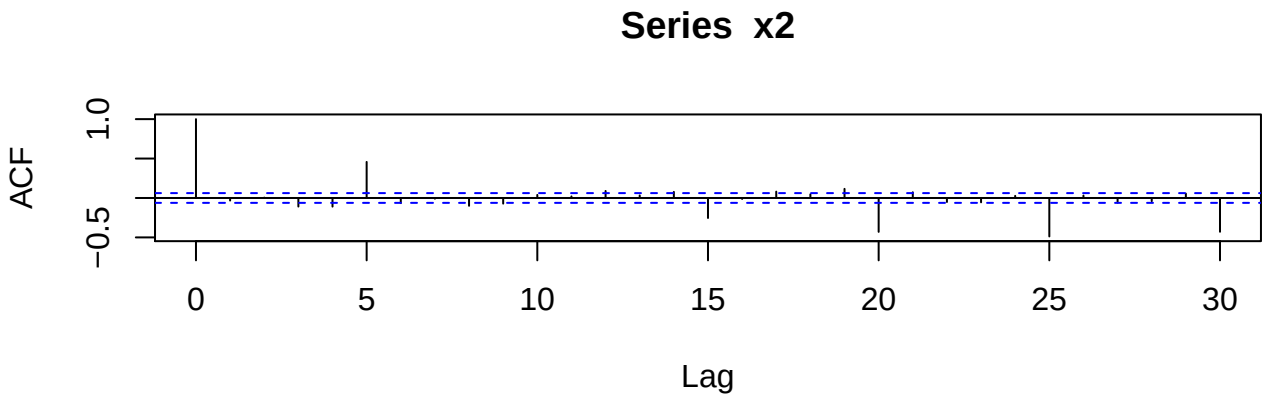


At lags 1, 2, 3, 4 and 6, it would be hard to predict the current value of x2, but the lag 5 plot shows that the current value of x2 depends a lot on the value 5 time units earlier.

Basic tests - autocorrelation

The autocorrelations for the first 5 lags are:

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

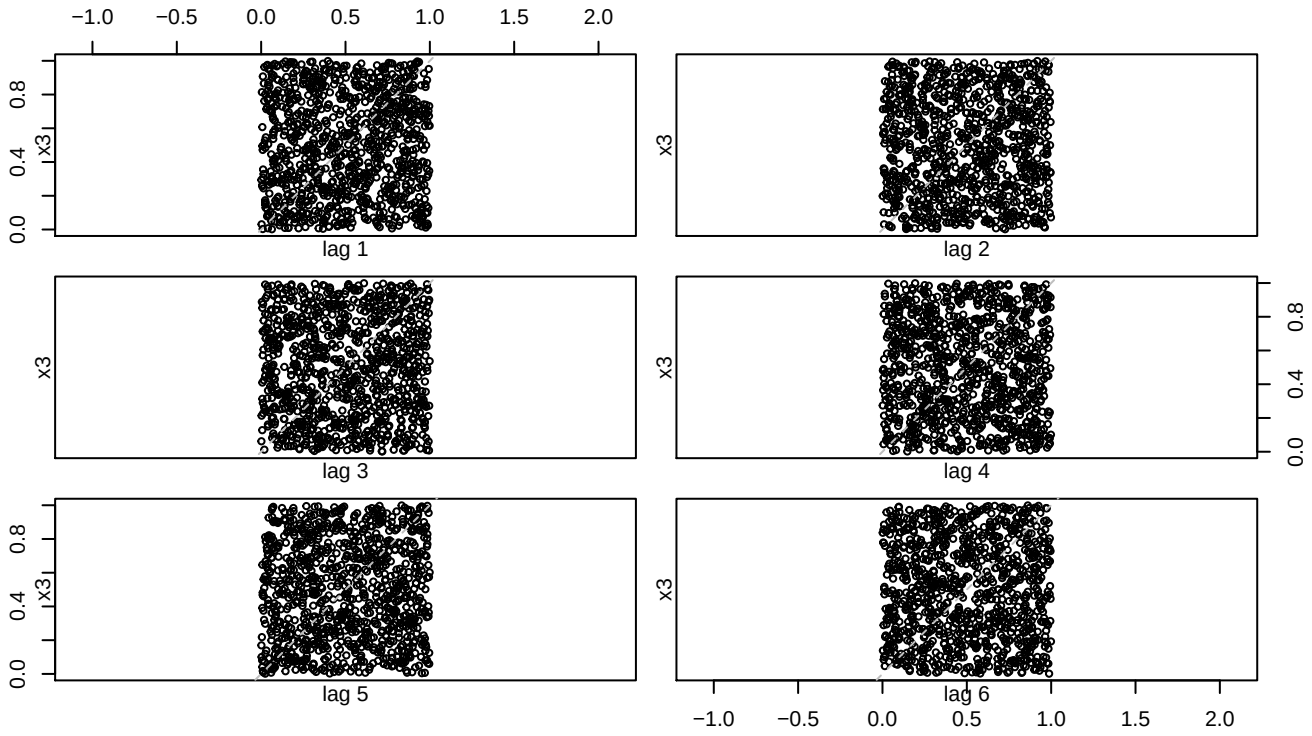



```
## [1] -0.0328  0.0049 -0.1090 -0.1097  0.4575
```

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

Basic tests - autocorrelation

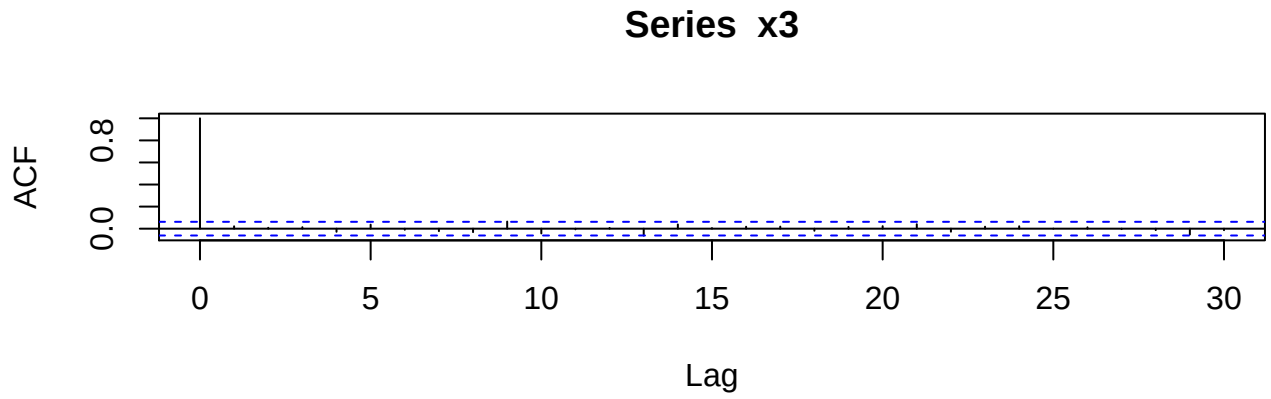
```
lag.plot(x3, lag=6)
```



Basic tests - autocorrelation

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

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




```
## [1]  0.02165  0.00835  0.01357 -0.02832  0.03662
```

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

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

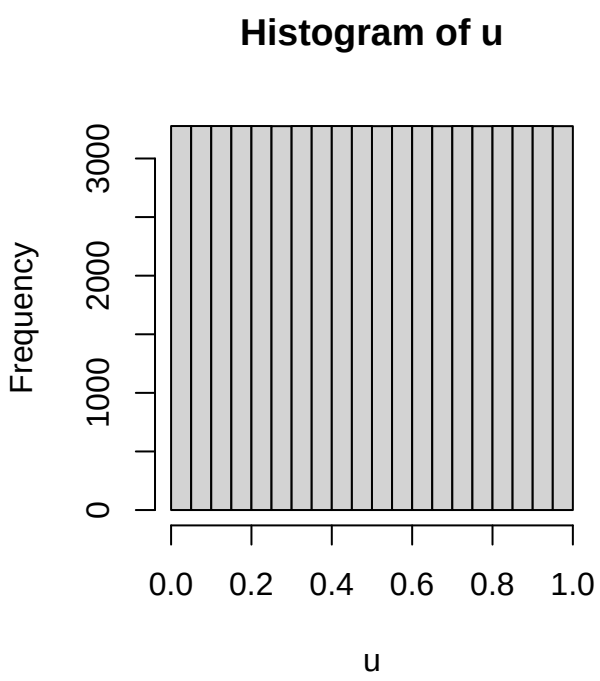
Run example with default settings and seed 372737:

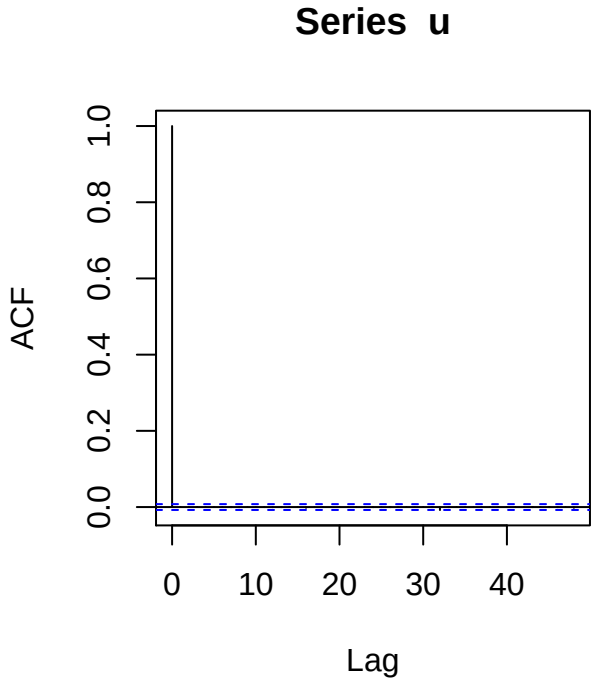
```
u <- rlincong(2^16-1, seed = 372737)
u[1:4]

## [1] 0.691 0.707 0.735 0.774

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

## [1] 0
```

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

Basic tests

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



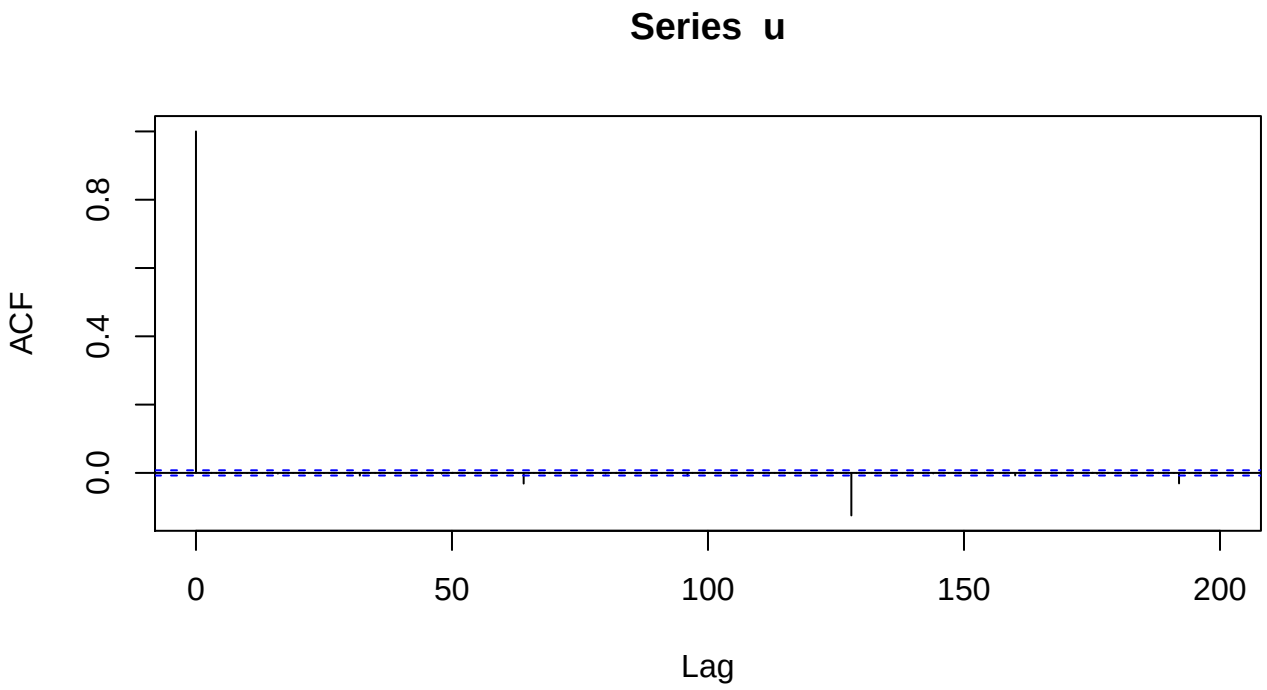


These tests indicate that the resulting numbers are appropriately uniformly distributed and that there is no short range linear dependence, but they are not rigorous enough for us to say this is a good generator; for example, see what happens when we check out larger lag autocorrelations.

Implementation

Check autocorrelations at larger lags:

```
acf(u, lag.max =200)
```



This generator is obviously far from perfect! Somehow, the value 129 lags earlier contains some information about the current value.

Other Types of Random Number Generators

A quadratic method: Let x_0 mod 4 = 2, and

$$x_{n+1} = x_n(x_n + 1) \textbf{ mod } 2^\omega.$$

Implementation:

```
rquad <- function(n, seed, omega = 29) {
    if ((seed%%4)!=2) seed <- seed*4 + 2;  x0 <- seed
    numbers <- numeric(n)
    for (j in 1:n) {
        numbers[j] <- (x0*(x0+1))%%(2^omega)
        x0 <- numbers[j]
    }
    numbers/(2^omega)
}
```

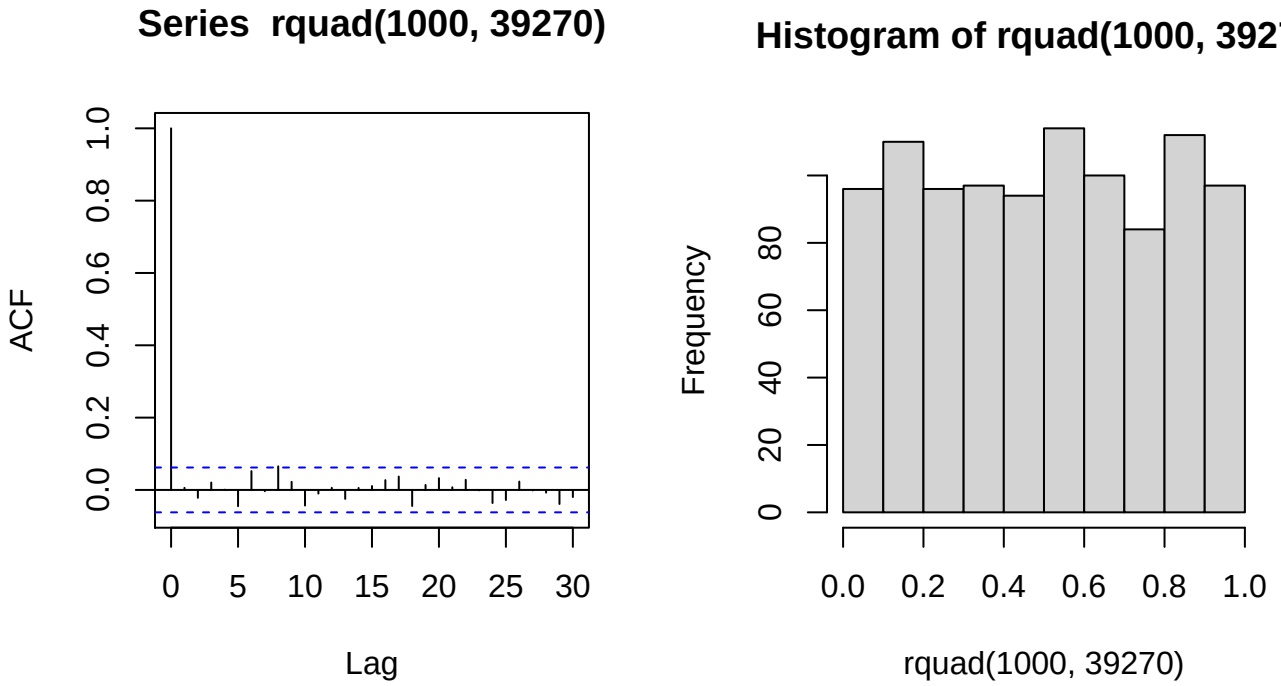
Quadratic Generator

Example: Generate 8 numbers

```
rquad(8, 39270)


## [1] 0.873 0.644 0.209 0.688 0.655 0.183 0.848 0.723
```

Checking for Autocorrelation and Uniformity

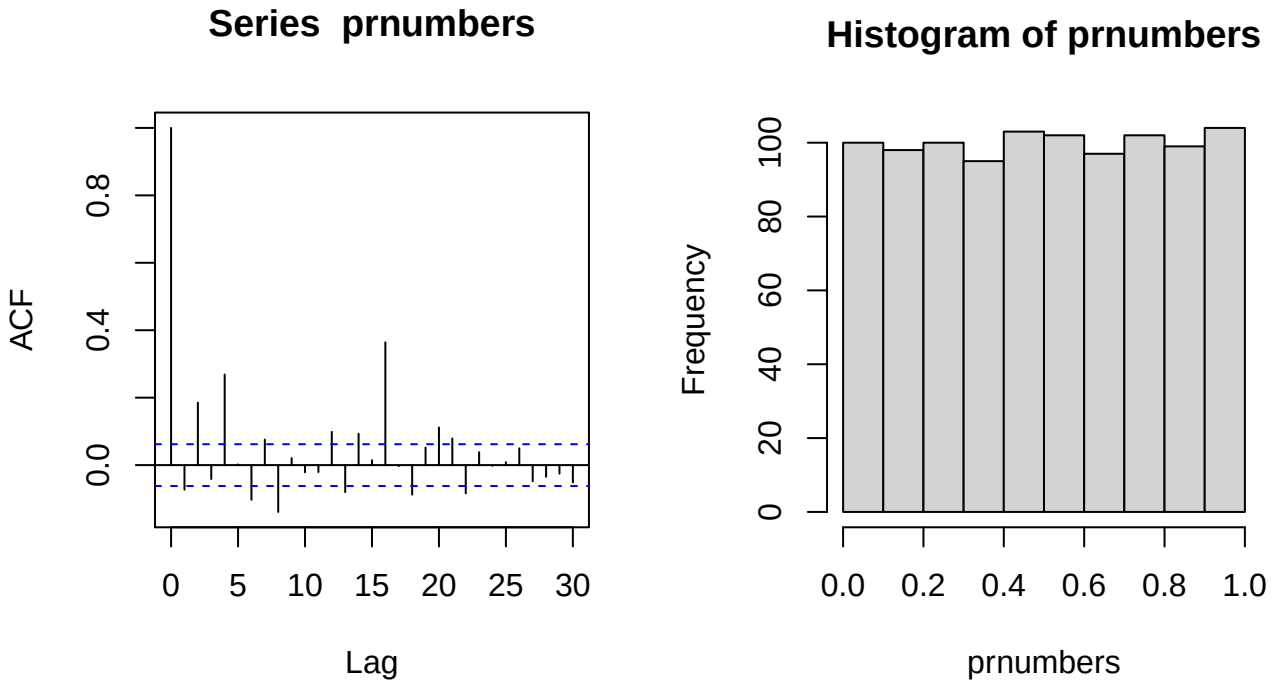
```
par(mfrow=c(1,2))
acf(rquad(1000, 39270)); hist(rquad(1000, 39270))
```



... Looks okay; Autocorrelations are near 0 and histogram is flat.

Checking for Autocorrelation and Uniformity

```
par(mfrow=c(1,2))
prnumbers <- rquad(1000, 39270, 10) # different m
acf(prnumbers); hist(prnumbers)
```



... looks very bad; some autocorrelations are very large.

Other Approaches

The Fibonacci sequence:

$$x_{n+1} = (x_n + x_{n-1}) \textbf{ mod } m.$$

Additive number generator; e.g. The Mitchell and Moore sequence:

$$x_n = (x_{n-24} + x_{n-55}) \textbf{ mod } m, n \geq 55$$

(55 integer seeds are needed).

An R Generator: Wichman and Hill

```
rWH <- function(n, seed) {
       if (missing(seed)) {
           seed <- 1000*as.numeric(paste(strsplit(
                   format(Sys.time(), "%H:%M:%OS3"), ":")[[1]],
                   collapse=""))
       }
       ix <- (171*seed)%%30269
       iy <- (172*seed)%%30307
       iz <- (170*seed)%%30323
       numbers <- numeric(n)
       for (i in 1:n) {
           ix <- (171*ix)%%30269
           iy <- (172*iy)%%30307
           iz <- (170*iz)%%30323
           numbers[i] <- (ix/30269.0+ iy/30307.0+
                 iz/30323.0)%%1.0
       }
       numbers
}
```

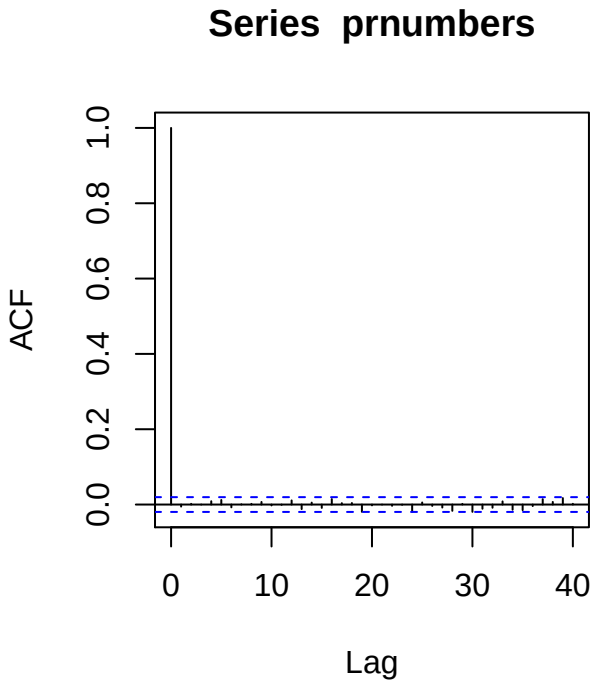
Wichman and Hill Generator

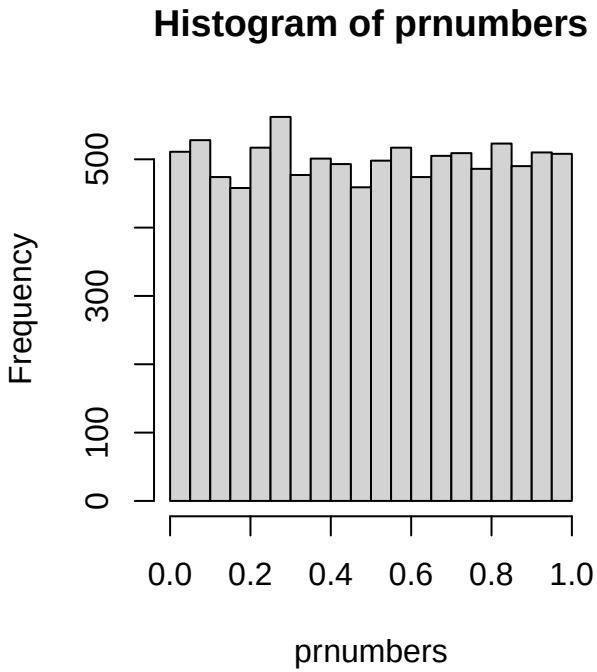
Example of use:

```
rWH(4) # automatically generated seed

## [1] 0.288 0.494 0.470 0.400

rWH(4)

## [1] 0.0783 0.7170 0.3490 0.6561

rWH(4, 3329913) # our own seed

## [1] 0.637 0.401 0.493 0.261
```

Checking for Autocorrelation and Uniformity

```
par(mfrow=c(1,2))
prnumbers <- rWH(10000)
acf(prnumbers); hist(prnumbers)
```

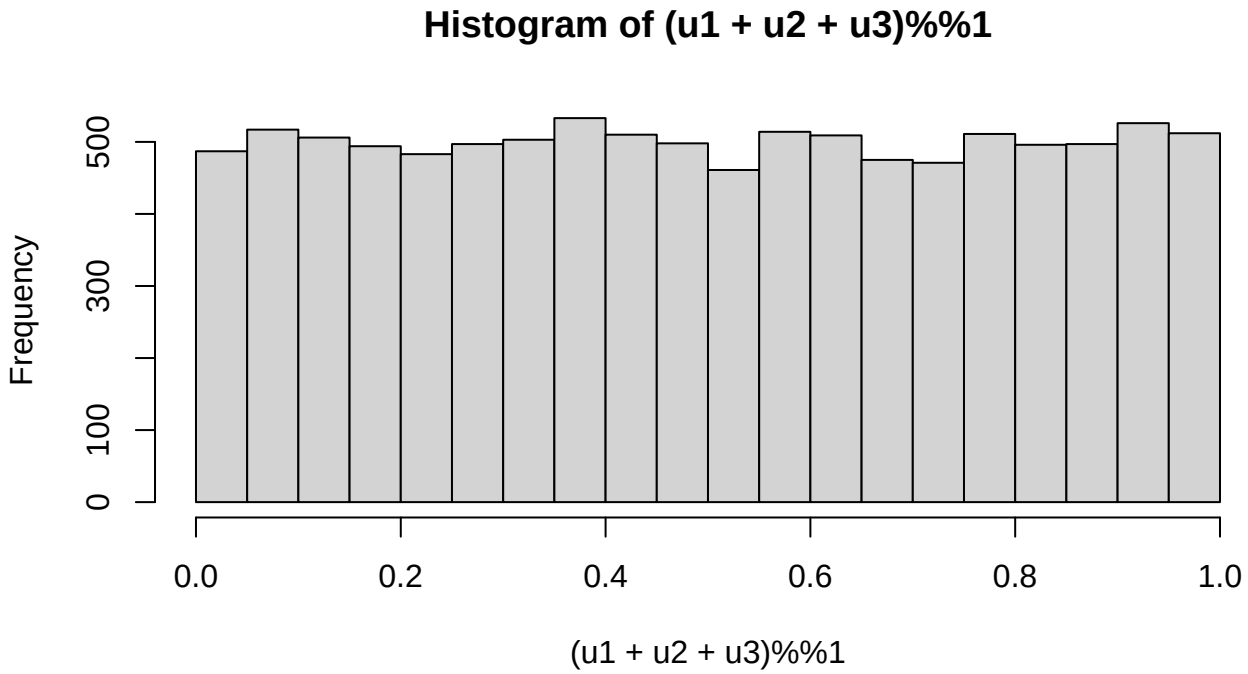




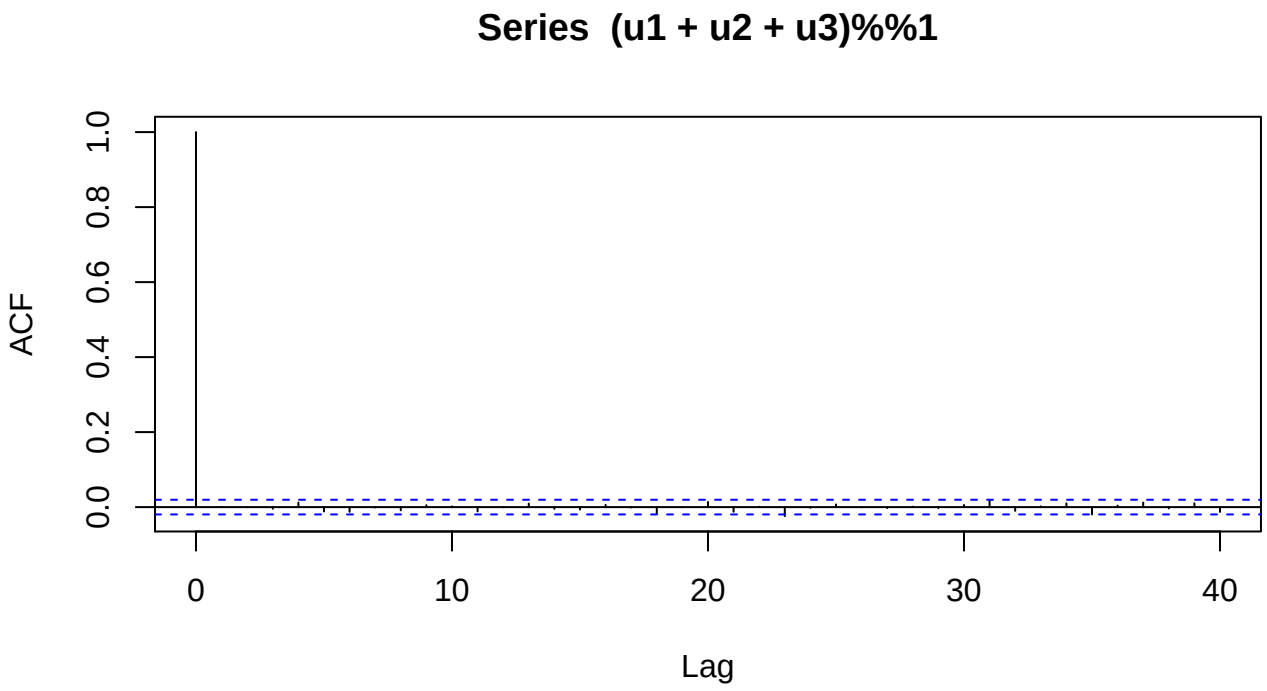
... ACF is small; histogram is flat

Checking for Uniformity of Sum of Three Uniforms mod 1

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

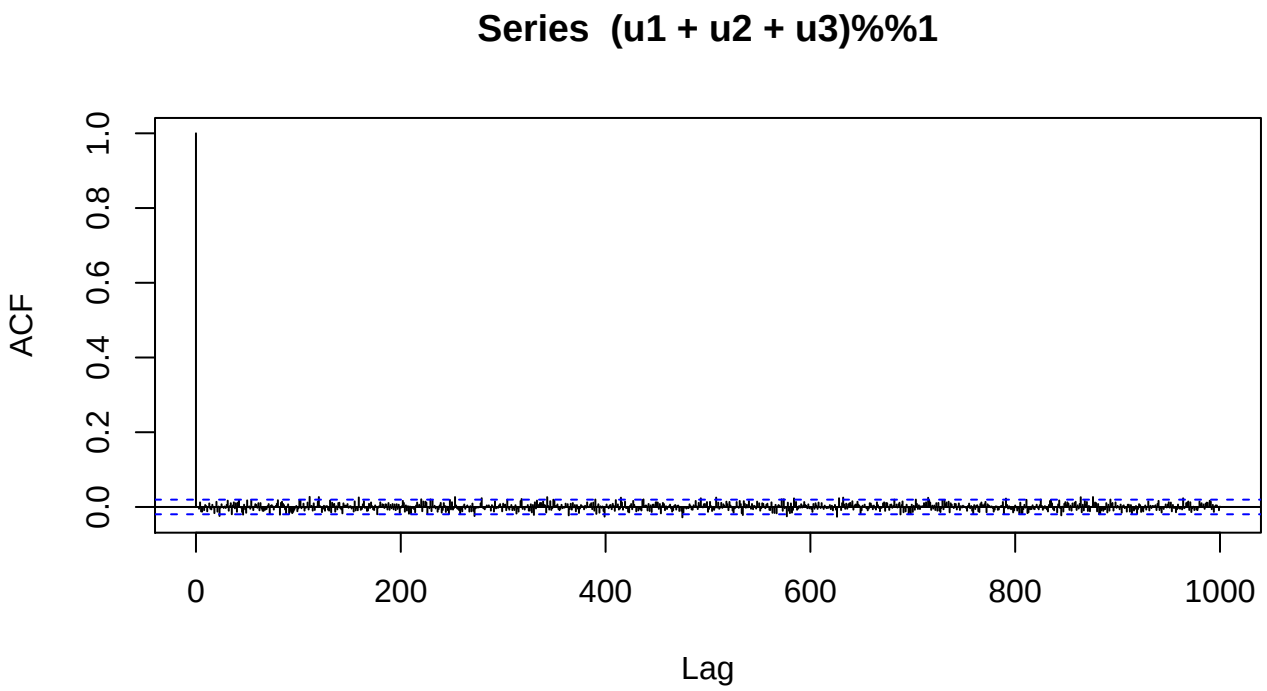


```
acf((u1+u2+u3)%%1)
```



Checking for Uniformity of Sum of Three Uniforms mod 1

```
acf((u1+u2+u3)%%1, lag.max=1000)
```



Default Generator in R: Mersenne Twister

Maksumoto and Nishimura (1997)

It is named for the fact that its cycle length is exactly a Mersenne prime (a prime number of the form $2^p - 1$, e.g. $2^3 - 1 = 7$).

Cycle length in 32-bit implementation: $2^{19937} - 1$.

Theoretical basis is more complicated than congruential approaches.

It passes most statistical tests for randomness.

It is slow by modern standards.

Default Generator in R: Mersenne Twister

```
runif(n, min = a, max = b)
```

Execution of this command produces n pseudorandom uniform numbers on the interval $[a, b]$. The default values are $a = 0$ and $b = 1$. The seed is selected internally.

Example

Generate 5 uniform pseudorandom numbers on the interval $[0, 1]$, and 10 uniform such numbers on the interval $[-3, -1]$.

```
runif(5)
```

```
## [1] 0.7783 0.4251 0.4348 0.2250 0.0502
```

```
runif(10, min = -3, max = -1)
```

```
##  [1] -2.74 -1.73 -1.69 -1.95 -2.02 -2.25 -1.31 -2.61 -2.55
## [10] -1.06
```

Comment: Starting Seeds

If you execute the above code yourself, you will almost certainly obtain different results than those displayed in our output.

This is because the starting seed that you will use will be different from the one that was selected when we ran our code.

There are two different strategies for choosing the starting seed x_0.

Starting Seeds

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] 0.358 0.354 0.267 0.997 0.132
```

What to Take Away from this Lecture

Uncertainty can be modelled with random processes.

Randomness can be approximated (with varying degrees of success) using computer generated sequences of numbers, such as those from multiplicative congruential generators.

The goal of the random number generator is to simulate sequences of independent uniform random variables.

A uniform random variable is defined as a variable that is equally likely to take any value on a given interval. The interval is $[0, 1]$ in the case of random number generation.

The histogram can be used to check that a sequence of variables follows a uniform distribution. It should be relatively flat over the required interval.

The autocorrelation function (acf) is a crude way to check whether variables in a sequence can be predicted from earlier values in the sequence. E.g. a spike at lag 5 indicates that u_6 can be predicted from u_1, u_7 can be predicted from u_2, and so on.

R is a useful language for doing statistics and for *prototyping* simulations. For large-scale simulations, use Julia or python (perhaps called from R).

What to Take Away from this Lecture

What to not worry about: most of the R code; we will get to this over the course of the entire program, not in this module.

R functions to start to remember:

```
set.seed() # e.g. set.seed(9131963)
runif()  # e.g. runif(20, 3, 18)
%%  # e.g.  29%%6   # = 5
<-  # e.g. x <- c(2, 3, 7)  # 2, 3, 7 assigned to x (a vector)
sum()   # e.g. sum(x) # adds up all values in the vector x # = 12
hist   # e.g. hist(x) # plots a histogram of values in x
acf()   # e.g acf(x) # plots autocorrelations in x
lag.plot()   # e.g. lag.plot(x)
```