R Random numbers

Simple random number generator

Heres a simple proof of concept for a random number generator in R. From a randomness point of view, it's terrible, but it demonstrates a pseudo random function where the output from the function is more or less chaotic.

It also shows (hopefully), why the seed is important, and what actually happens when you set the seed to some value.

The function we'll define, simple_random() has the form:

$$f_{n+1} = (a * f_n + 1) modulo (resolution)$$

I.e. each value depends on the previous value. The function is completely deterministic, but due to the modulo operation (remainder after division), the value will vary chaotically. The seed is actually $f_0()$.

The resolution is the number of different values this method will generate, in our example below we have a resolution of 1000000, so we can generate one million different positive "real" values.

Getting started in R - global variable

First, we need to define a global variable to hold the seed. This is an implementation detail, and it's not important for the concept, but if we don't do this, we cannot modify the seed from within the function, which means that we can't make the function "remember" what its previous value was.

```
# Global variable to hold our "seed"
assign("seed", 1, envir = .GlobalEnv)
```

Defining the random function

This function generates "random" numbers between 0 and 1. I believe that the random functions in R (e.g. rnorm()) will use something like the below (but much more clever, hopefully) to generate a sequence of "random" numbers, one by one. The function is completely deterministic (it's dependent on the initial seed), but appears random, just like the rnorm() values that get generated.

```
simple_random = function() {
  new_value = (123456789*seed + 1) %% 1000000

  assign("seed", new_value, envir = .GlobalEnv)

  return(new_value / 1000000)
}
```

Slightly more complicated random function

This method tries to shift bits around to get a better distribution of numbers.

```
more_random = function() {

  tmp_new_value = bitwShiftL(seed, 5)
  new_value = ((123456789 + tmp_new_value)*seed + 1) %% 1000000
```

```
  assign("seed", new_value, envir = .GlobalEnv)

  return(new_value / 1000000)
}
```

Function to set seed

```
set_seed = function(x) {
  assign("seed", x, envir = .GlobalEnv)
}
```

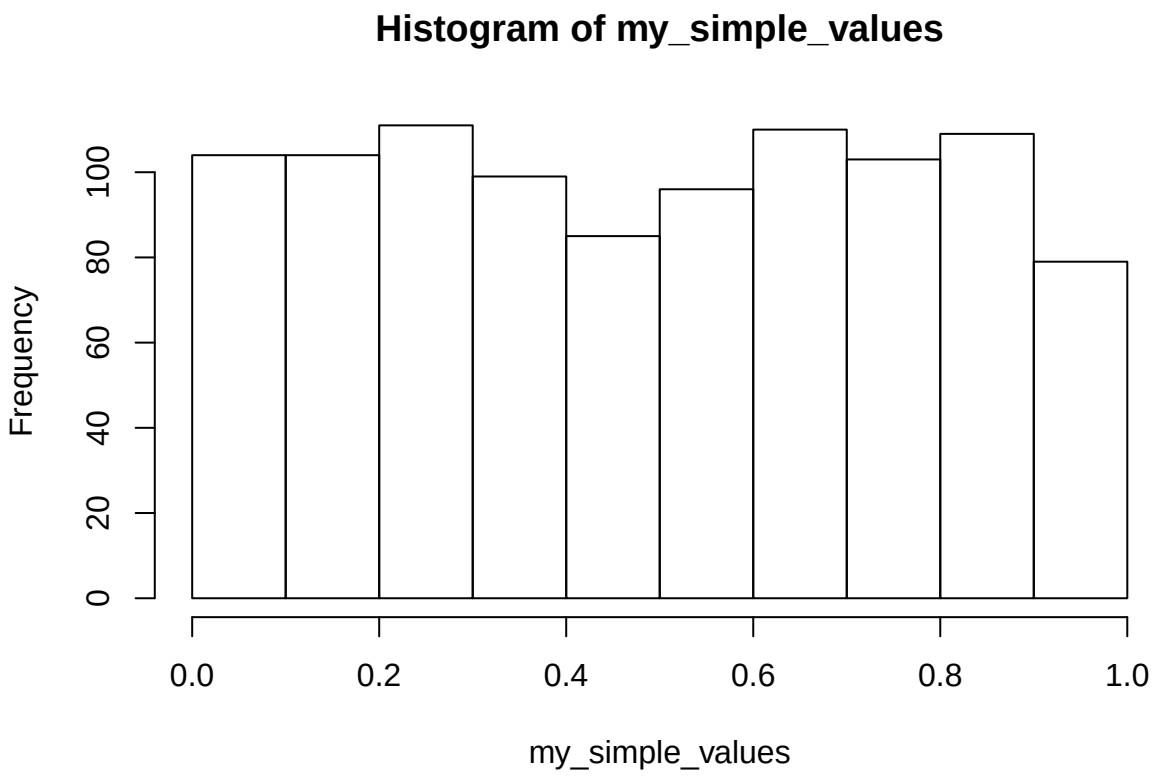
Testing the functions

Let's generate 1000 random numbers using both the functions, then draw a histogram of both. Note that I set the seed between the two for loops.

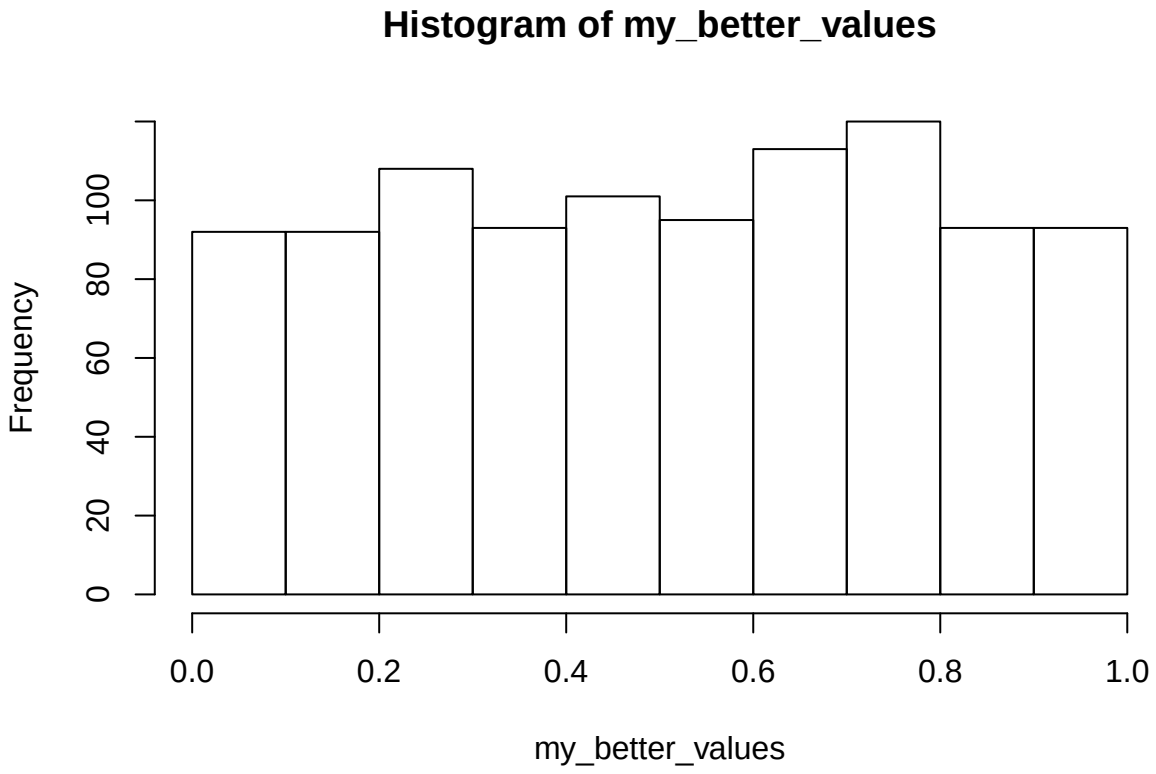
```
# Set seed to known value
set_seed(100)

my_simple_values = c()
for (i in 1:1000) {
  my_simple_values = c(my_simple_values, simple_random())
}


# reset seed to known value
set_seed(100)

my_better_values = c()
for (i in 1:1000) {
  my_better_values = c(my_better_values, more_random())
}
```

And now let's show the histograms

```
hist(my_simple_values)
```



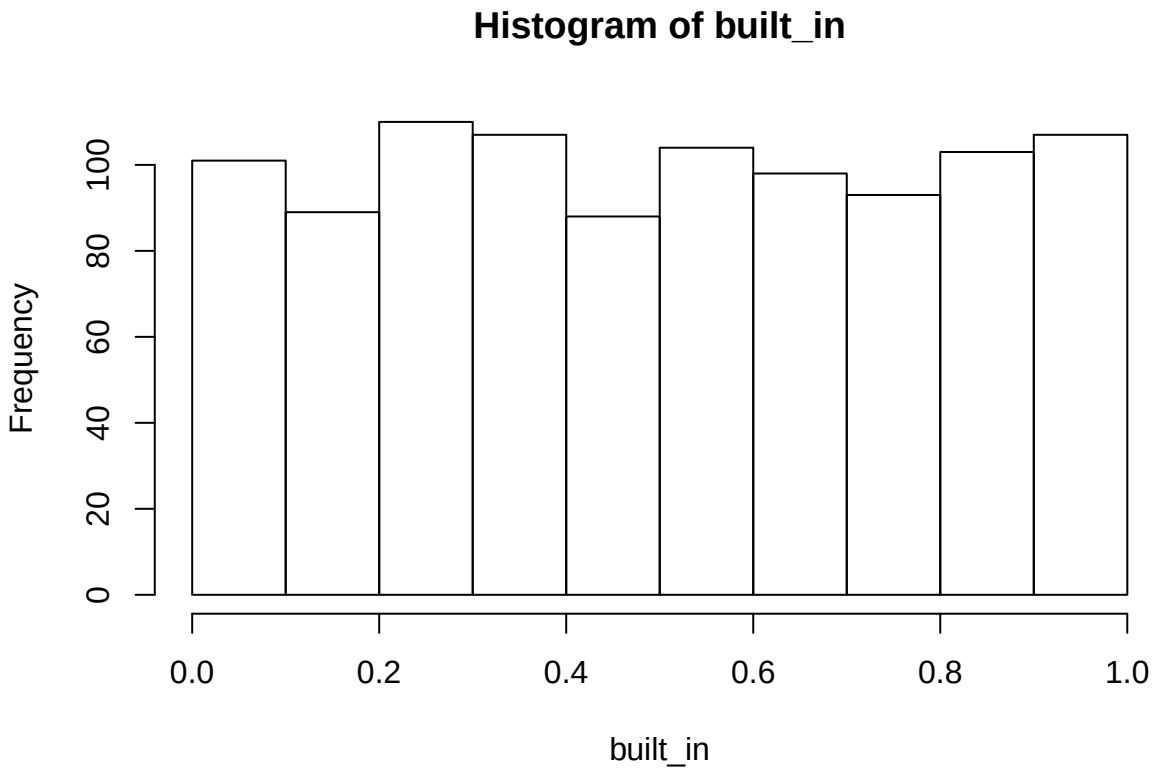

```r
hist(my_better_values)
```



Comparison to runif()

Let's use R's built in runif() to generate 1000 values as well, and see how well it behaves in terms of randomness:

```
built_in = runif(1000, 0, 1)

hist(built_in)
```



So it turns out the simple random generator isn't **too** bad, however there may be seeds that generate a very short "period" of numbers (i.e. repeating patterns), and the runif() function probably guards against that somehow.