Simulation/Monte Carlo

Back to the basics

Probability

- Probability allows us to quantify statements about the chance of an event taking place.

For example - Flip a fair coin

1. What's the chance it lands heads?
2. Flip it 4 times, what proportion of heads do you expect?
3. Will you get exactly that proportion?
4. What happens when you flip the coin 1000 times?

4 Flips

- In 4 flips, we can get 0, 1, 2, 3, or 4 Heads and so the proportion of Heads can be: 0, 0.25, 0.5, 0.75, or 1
- We expect the proportion to be 0.5
- But, a proportion of 0.25 is quite likely:

There are 16 possible ways for 4 tosses to land, e.g. HHHH, HHHT, HHTH, …

Each is equally likely, so the chance of any particular sequence of Hs and Ts is 1/16
So chance of 0.25 proportion is 4/16
HTTT, THTT, TTHT, TTTH

4 Flips

- We can think of the proportion of Heads in 4 flips as a statistic because it summarizes data

- Notice that it is a random quantity – it takes on 5 possible values, each with some probability

value	0.00	0.25	0.50	0.75	1.00
chance	1/16	4/16	6/16	4/16	1/16

1,000 Flips

- When we flip the coin 1,000 times, we can get a many different possible proportions of Heads, i.e. 0, 0.001, 0.002, 0.003, …, 0.998, 0.999, 1.000
- It's highly unlikely that we would get 0 for the proportion – how unlikely?
- What does the distribution of the proportion of heads in 1000 flips look like?

1,000 Flips

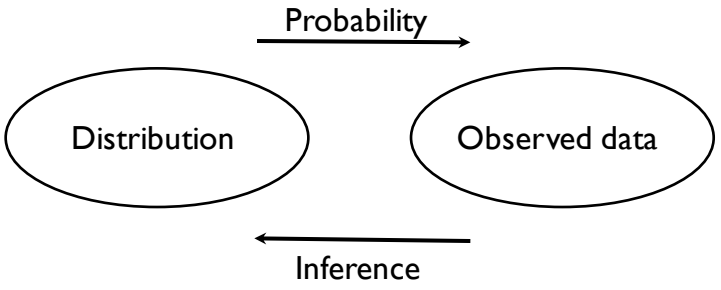
- With some advanced math tools, we can figure this out.
- But we can also get a good idea using a simulation.
- In our simulation we will assume that the chance of Heads is 0.5 and find out what the possible values for the proportion of heads in 1,000 flips looks like
- If we were to carry out an experiment with a coin and get a particular proportion, say 0.37, then we could use this simulation study to help us understand the results of our experiment.

Distribution of the Sample Median

Example: Carry out a simulation study of the median when sampling from the normal distribution. How does it vary with the sample size and with the standard deviation of the normal distribution?

To understand the role that simulation can play in helping us understand statistics, let's take a step back and think about the big picture.

We can think of probability theory as complimentary to statistical inference.



A *statistic* is often just a function of a random sample, for example the sample mean, the 95th percentile, or the sample proportion.

Statistics are often used as *estimators* of quantities of interest about the distribution, called *parameters*. Statistics are random variables (since they depend on the sample); parameters are not.

In simple cases, we can study the *sampling distribution* of the statistic analytically. For example, we can prove that under mild conditions the distribution of the sample proportion is close to normal for large sample sizes.

In more complicated cases, we turn to simulation.

Monte Carlo Methods

We use the sample to *estimate* features of the distribution, such as the behavior of various statistics under repeated sampling from the distribution.

This set of techniques, sometimes called Monte Carlo methods, is very powerful. Statisticians/scientists routinely use it to evaluate complicated methods for which exact mathematical results are difficult or impossible to obtain.

The main idea in a simulation study is to replace the mathematical expression for the distribution with a *sample* from that distribution.

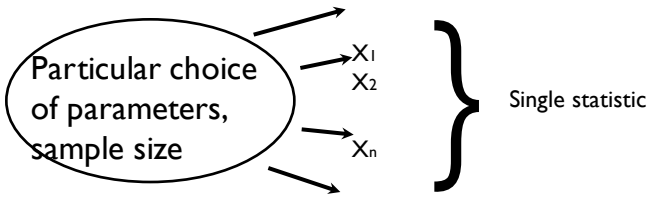
In our example: $X_1, X_2, \ldots, X_n$ are independent observations from the same distribution.
The distribution has center (mean/expected value) μ and spread (standard deviation) σ

We are interested in the distribution of $median(X_1, X_2, \ldots, X_n)$

So we take many samples of size n, and study the behavior of the sample medians

The downside: whereas mathematical results are symbolic, in terms of arbitrary parameters and sample size, in a simulation we must specify particular values.

A *single experiment within a simulation* looks like this:



To approximate the *sampling distribution* of the statistic, we repeat the whole experiment B times. The larger B is, the better our approximation will tend to be.

Steps in carrying out a simulation study:

1. Specify what makes up an individual experiment: sample size, distributions, parameters, statistic of interest.
2. Write an expression or function to carry out an individual experiment and return the statistic.
3. Determine what inputs, if any, to vary (e.g. different sample sizes or parameters).
4. For each combination of inputs, repeat the experiment B times, providing B samples of the statistic.
5. For each combination of inputs, summarize the *empirical distribution* of the statistic of interest.
6. State and/or plot the results. (Sometimes go back to 3.)

Useful Random Number Generators

```
sample(x, size, replace = FALSE,
       prob = NULL)
```

Think of an urn with tickets, each ticket marked with a value. Mix up the tickets and draw one at a time from the urn

- x = vector with one element for each ticket, values correspond to what is written on the ticket.
- size = number of draws to take from the urn
- replace = replace the ticket between draws or not.
- prob = set of weights for the elements in x (an element might represent more than one ticket)

Useful Random Number Generators

Standard Probability Distributions:

runif(n, min = 0, max = 1) – sample from the uniform distribution on the interval (0, 1).

So the chance the value drawn is:
between 0 and 1/3 has chance 1/3;
between 1/3 and 1/2 has chance 1/6;
between 9/10 and 1 has chance 1/10

The min and max allow you to change the interval from which to sample, e.g. min = 100, max = 150 will produce random values between 100 and 150

Useful Random Number Generators

Standard Probability Distributions:

rnorm(n, mean = 0, sd = 1) – sample from the normal distribution with center = mean and spread = sd

rbinom(n, size, prob), - sample from the binomial distribution with number of trials = size and chance of success = prob

Other distributions: rexp(), rpois(), rt(), rf() – each has arguments for parameter values relevant to the distribution … See ?Distributions for more information

Example: Carry out a simulation study of the median when sampling from the normal distribution. How does it vary with the sample size and with the standard deviation of the normal distribution?

Median of sample from Normal distribution with mean 0
Vary the SD of the distribution:, 1, 5, 10, 20
Vary the sample size: 100, 200, 1000

```
median(rnorm(100, mean = 0, sd = 1))

replicate(1000, myExpt(n = 100, sd = 1))
```

Experiment:

- Generate `n` random normal values from a Normal(0, `s^2`)
- Take the median of these `n` values

```
n = 27
s = 3
median(rnorm(n = n, sd = s))
```

Repeat Experiment:

- Repeat the experiment `B` times
- Examine the distribution of the `B` medians

```
B = 1000
sampleMs = replicate(1000,
       median(rnorm(n = n, sd = s))
mean(sampleMs)
sd(sampleMs)
hist(sampleMs)
```

Repeat Simulation

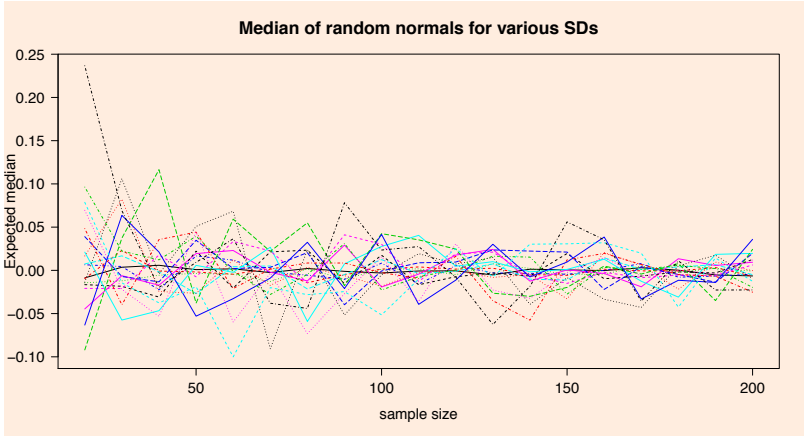
- Repeat the simulation for different values of `n` and `s`
- Compare/Examine the behavior for these different values

```
ns = seq(20, 200, by = 10)
ss = seq(1, 10, by = 0.5)
```

Repeat Simulation

```
ns = seq(20, 200, by = 10)
ss = seq(1, 10, by = 0.5)
samples = matrix(nrow = length(ns),
                 ncol = length(ss))

for (i in 1:length(ns)) {
  for (j in 1:length(ss)) {
   samples[i , j] =
    mean(replicate(1000,
           median(rnorm(n = ns[i],
                        sd = ss[j]))))
  }
}
```



Repeat Simulation

```
ns = seq(20, 200, by = 10)
ss = seq(1, 10, by = 0.5)
samples = matrix(nrow = length(ns),
                 ncol = length(ss))

parValues = expand.grid(ns, ss)

mapply(function (num, sd){
  mean(replicate(1000,
                 median(rnorm(num,sd))))
 },
 parValues[ , 1], parValues[, 2])
```

How does R generate random numbers?

Actually, it doesn't

R uses a **pseudo random number generator**:

- It starts with a **seed** and an **algorithm** (i.e. a function)
- The seed is plugged into the algorithm and a number is returned
- That number is then plugged into the algorithm and the next number is created

The algorithms are such that the numbers produced behave/look like random values

Simple Congruential Generator

The congruential method uses modular arithmetic to generate "random" numbers.

From inputs a and b and an initial value, x_0 , the first "random number" is generated as follows:

```
x_1 = a * x_0 mod b
```

and subsequent numbers are generated recursively,

```
x_(n+1) = a * x_n mod b
```

We call x_0 the **seed**

Congruential a = 3,b = 64

Seed = 17

3 * 17 mod 64 = 51 mod 64 = 51

3 * 51 mod 64 = 153 mod 64 = 25

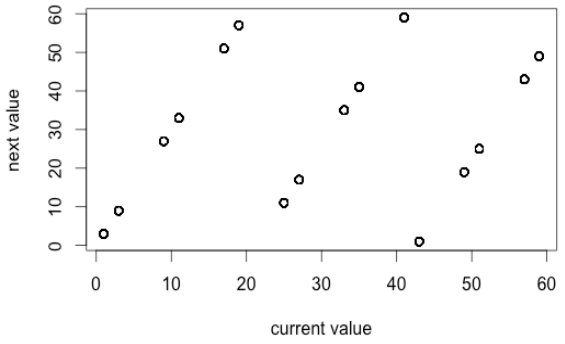
3 * 25 mode 64 = 75 mod 64 = 11

And so on. The first 20 "random" numbers are

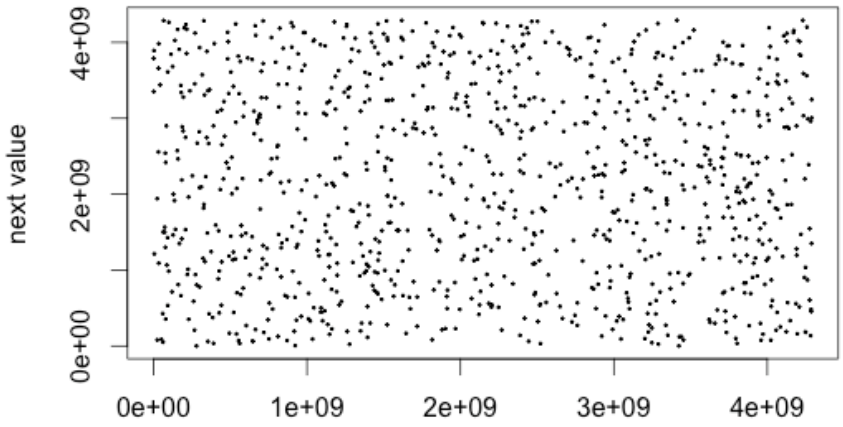
```
51 25 11 33 35 41 59 49 19 57 43  1  3  9 27
17 51 25 11 33
```

Generate 1000 values

```
plot(x3b64[1:(n-1)], x3b64[2:n],
 xlab = "current value",ylab = "next value")
```



```
cong(n, a = 69069, b = 2^32)
```



Uniform Random Numbers

- We can take the sequence we just generated and simply divide it by by b. Then all values will lie between 0 and 1.
- This gives us a sequence of pseudo-random uniform(0,1) random variables.
- All random number generation algorithms are built upon uniform(0,1) random number sequences! E.g. (simple) how do we generate coin flips with U(0,1) random variables?
- Much of the research in Monte Carlo centers around clever algorithms to turn uniform(0,1) variables into random variables from other distributions.

The Seed

There is one big advantage to pseudo-random number generators:

You can reproduce your simulation results by controlling the seed:

`set.seed()` allows you to do this:

When researchers publish results from simulation studies, they typically include the random number generator and the seed that was used so that others can verify/replicate their results

The Seed

```
> set.seed(69069)
```
Set the seed for the RNG

```
> runif(3)
[1] 0.1648855 0.9564664 0.3345479
```
Call the uniform RNG

```
> runif(3)
[1] 0.01109596 0.18654873 0.94657805
```
Call the uniform RNG again

```
> set.seed(69069)
> runif(3)
[1] 0.1648855 0.9564664 0.3345479
```
Set the seed back to 69069