5
Random Number Generation

5.1 Randomness: quantifying uncertainty

The concepts of uncertainty and randomness have intrigued humanity for a long time. The world around us is not *deterministic* and we are faced continually with chance occurrences. Uncertainty is inherent in nature: for example, the behaviour of fundamental physical particles, genes and chromosomes in biology, and individuals in society under stress or strain. The methodology for exploring uncertainty involves the use of *random numbers*.

5.2 Pseudo–random numbers

Suppose we need to obtain a list of random digits $0, 1, 2, \ldots, 9$. How might we go about this? There are several options:

- **Fair ten–sided die:**

- **Tosses of a fair coin:**

- **The decimal expansion of π:**

- **Other physical devices:**

A disadvantage of the mechanical and electronic devices is that the sequence of random numbers generated is not reproducible. So we often use *Pseudo–random numbers generators* (PRNG), that is, algorithms for generating sequences of numbers that approximate the properties of true random numbers.

THE GERMAN Federal Office for Information Security (The *Bundesamt für Sicherheit in der Informationstechnik*, or BSI) has established criteria for quality PRNG:

1. A sequence of random numbers has a high probability of containing no identical consecutive elements;
2. A sequence of numbers which is indistinguishable from 'true random' numbers (tested using statistical tests);
3. It should be impossible to calculate, or guess, from any given subsequence, any previous or future values in the sequence;
4. It should be impossible, for all practical purposes, for an attacker to calculate, or guess, the values used in the random number algorithm.

http://en.wikipedia.org/wiki/Statistical_randomness

Points 3 and 4 are crucial for many applications[1].

[1] Everytime you make a phone call, make contact with a wireless point, or pay using your credit card, random numbers are used.

As we will see in this course, it is important to be able to reproduce a sequence of random numbers. The advent of digital computers gave rise to a number of techniques that use a *recursive relation*, that is, the number r_i in a sequence is a function of the preceding numbers $r_{i-1}, r_{i-2}, \ldots$. Some of these PRNG's have quite cool names: for example, the *Super-Duper* PRNG; the *Blum-Blum-Shub* PRNG (after Lenore Blum, Manuel Blum and Michael Shub); and the *Mersenne-Twister* PRNG, as used by R.

5.3 Random numbers in R

5.3.1 The *runif* function

R has a number of commands that generate and manipulate random numbers. One function that we use throughout this course is:

The default random number generator used by R is the Mersenne-Twister.

```
> runif(n, min=0, max=1)
```

This function will generate `n` random numbers between the values of `min` and `max`. If the arguments `min` or `max` are omitted, then the default values are 0 and 1 respectively[2]. For example:

[2] What R actually does is generate `n` values randomly from a Uniform $U(\texttt{min},\texttt{max})$ distribution; see MAS1604 for more details. You should read '`runif`' as 'R-Unif' - **R**andom values from a **Unif**orm distribution.

```
> runif(1)
[1] 0.8698216
> runif(1)
[1] 0.5790736
> runif(5)
[1] 0.08108483 0.58695501 0.50411941 0.97594053 0.19490056
> runif(1, 6, 7)
[1] 6.687428
```

Notice that calling `runif` returns different values. If we wish to rerun a computer experiment, then we need repeatability. In this case we use the command `set.seed`, e.g.:

The function `set.seed` is a user friendly version that sets the seeds of all possible underlying random number generators. This function is actually an interface to `.Random.seed`. Don't worry about this.

```
> set.seed(12345)
> runif(1)
[1] 0.7209039
> runif(1)
[1] 0.8757732
> set.seed(12345)
> runif(1)
[1] 0.7209039
```

If we want to generate integers, say 0, 1, ..., 9, then we could simply take the first value after the decimal place.

5.3.2 The *sample* function

Another important R function that we will use is the `sample` function:

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

This takes the following arguments:

- `x`: a list of values
- `size`: non-negative integer giving the number of items to choose.
- `replace`: Should sampling be with replacement? Default: `FALSE`.
- `prob`: A vector of probability weights. Default: All values equally likely.

See `?sample` for help.

Suppose we wish to sample five numbers from $\{1, 2, 3, 4, 5, 6\}$, then:

In `sample` the default is without replacement, i.e. no value more than once.

```
> set.seed(1)
> x = c(1, 2, 3, 4, 5, 6)
> sample(x, 5)
[1] 2 6 3 4 1
```

We can also sample with replacement:

```
> sample(x, 5, replace=TRUE)
[1] 6 6 4 4 1
```

This means that values **may** appear more than once.

5.3.3 Simulating the Capital One Cup draw

We are in the semi-finals of the Capital One cup [3], and need to organise the draw for the final stage. The remaining teams are:

Manchester Utd, Manchester City, Sunderland, West Ham.

[3] Unfortunately, Newcastle didn't make it through. What a shame. Not.

Here's how we do this in R.

```
> set.seed(3)
> teams = c("Man Utd", "Man City", "Sunderland", "West
Ham")
> sample(teams, 4)
[1] "Man Utd"    "Sunderland" "West Ham"   "Man City"
```

So, we have 'Man Utd vs Sunderland' and 'Man City vs West Ham' [4]. However, if we think Sunderland are likely to get beaten by Manchester United, we can rig the voting:

[4] As actually happened in 2014.

```
> prob_weights = c(0.4, 0.4, 0.05, 0.2)
> sample(teams, 4, prob=prob_weights)
[1] "Man Utd"    "Man City"   "West Ham"   "Sunderland"
```

That's better! [5]

[5] Perhaps. Actually, it didn't matter – Sunderland beat Man Utd anyway! But lost in the final to Man City :-(

Command	Comment	Example
`sample`	Sample discrete numbers	`sample(c(1,2,3))`
`runif`	Generate a random number between 0 & 1	`runif(1)`
`set.seed`	Set the seed of the random number generator	`set.seed(10)`

Table 5.1: Summary of R commands in this chapter.

5.4 Simulating from discrete probability distributions in R

You would this technique described in this section to simulate from the Poisson and binomial distributions.

Suppose we want to simulate a discrete random variable X which has probability mass function

$$\Pr[X = x_j] = p_j \quad \text{for } j = 1, 2, \ldots,$$

and where $\sum_{\text{all}} p_j = 1$. To accomplish this we first simulate a value u from a $U(0,1)$ distribution and set

$$X = \begin{cases} x_1 & \text{if } u < p_1 \\ x_2 & \text{if } p_1 \leq u < p_1 + p_2 \\ x_3 & \text{if } p_1 + p_2 \leq u < p_1 + p_2 + p_3 \\ \vdots & \vdots \\ x_j & \text{if } \sum_{i=1}^{j-1} p_i \leq u < \sum_{i=1}^{j} p_i \\ \vdots & \vdots \end{cases}$$

Now, for a Uniform random variable u, and for $0 < a < b < 1$, it is the case that $\Pr[a \leq U < b] = b - a$. Thus,

$$\begin{aligned} \Pr[X = x_j] &= \Pr\left[\sum_{i=1}^{j-1} p_i \leq U < \sum_{i=1}^{j} p_i\right] \\ &= \sum_{i=1}^{j} p_i - \sum_{i=1}^{j-1} p_i \\ &= p_j\,. \end{aligned}$$

as required.

5.4.1 Example: discrete random numbers

Simulate a random variable with the following probability mass function:

x	1	2	3	4
$\Pr[X = x]$	0.2	0.15	0.25	0.4

Solution:

We calculate the cumulative distribution function (CDF) of the distribution:

x	1	2	3	4
$\Pr[X = x]$	0.2	0.15	0.25	0.4
$\Pr[X \leq x]$	0.2	0.35	0.60	1.0

Then we generate an observation u from $U \sim \text{Uniform}(0,1)$, and:

- if $u < 0.2$, set $X = 1$;
- if $0.2 \leq u < (0.2 + 0.15) = 0.35$, set $X = 2$;
- if $0.35 \leq u < (0.2 + 0.15 + 0.25) = 0.6$, set $X = 3$;
- if $u \geq 0.6$, set $X = 4$.

See Figure 5.1 overleaf.

Suppose $u = 0.821$ (perhaps obtained in R using `runif(1)`). Then our simulated value on X would be 4.

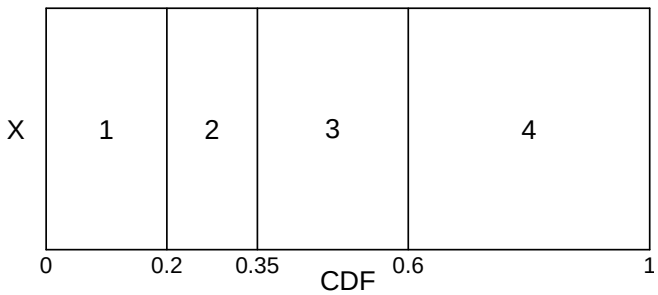

Figure 5.1: Simulating discrete random numbers.

Using R

To simulate from the above distribution in R, there are two (obvious) methods that we can use, based on the material we have covered so far: using the `sample` command...

```
> x = c(1, 2, 3, 4)
> prob = c(0.2, 0.15, 0.25, 0.4)
> sum(prob)
[1] 1
> sample(x, 1, prob, replace=TRUE)
[1] 1
```

...or using a bunch of `if` statements:

```
> u = runif(1)
> if(u <= 0.2) {
+   X = 1
+ } else if(u <= (0.2+0.15)) {
+   X = 2
+ } else if(u <= (0.2+0.15+0.25)) {
+   X = 3
+ } else {
+   X = 4
+ }
```

5.4.2 *Example: Simulating Poisson random variates*

For the uniform random numbers:

$$u_1 = 0.253,\ u_2 = 0.588,\ u_3 = 0.789,$$

simulate three random numbers from a Poisson distribution with mean $\lambda = 2$.

Solution:

The probability mass function of the poisson distribution is

$$Pr[X = x] = \frac{e^{-\lambda}\lambda^x}{x!}$$

where $\lambda > 0$. So we have:

For example,

$$Pr[X = 3] = \frac{e^{-2}2^3}{3!} = 0.180;$$

see MAS1604 for more examples.

x	0	1	2	3	4
$\Pr[X = x]$	0.135	0.271	0.271	0.180	0.090
$\Pr[X \leq x]$	0.135	0.406	0.677	0.857	0.947

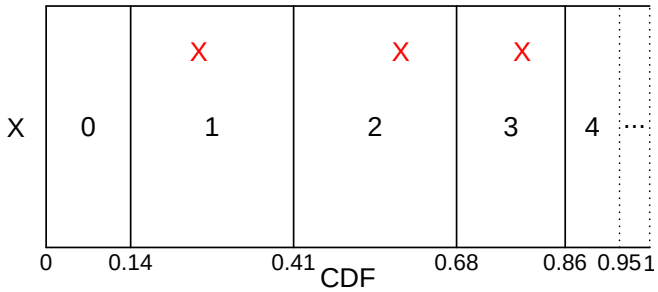

Figure 5.2: Simulating discrete random numbers from the Poisson distribution.

Hence our random numbers are 1, 2 and 3.

Using R

We could simulate values from the Poisson distribution in R in exactly the same way we did for the probability distribution in Section 5.4.1 – that is, use the `sample` command or use a bunch of `if` statements. However, much easier is to use the built-in R function `rpois`[6]:

[6] As with `runif`, read this as 'R-Pois', for **R**andom **Pois**son.

```
> rpois(1,2)
[1] 3
```

```
> rpois(10,2)
[1] 0 1 2 0 0 0 2 1 3 1
```

5.4.3 Example: Simulating from other distributions

Simulating random values from other standard probability distributions is easy in R. Just as we have seen the intrinsic functions `runif` and `rpois`, there are other functions such as `rgeom` and `rbinom`. For example, suppose $X \sim \text{Bin}(6, 0.2)$ and $Y \sim \text{Geom}(0.75)$, and we want to simulate 5 random values from X and 3 random values from Y:

```
> (X=rbinom(5,6,0.2))
[1] 0 1 1 3 0
```

```
> (Y=rgeom(3,0.75))
[1] 2 0 1
```

At this point it is worthwhile mentioning some variations on `runif`, `rpois`, `rbinom` and `rgeom`. Replacing the `r` prefix with:

- `d`, calculates the density of the distribution at a particular value. For example, `dpois(1,2)` would find $Pr[X = 1]$ when $X \sim Po(2)$;
- `p`, calculates the distribution function at a particular value. For example, `pbinom(2,6,0.2)` would find $Pr[X \leq 2] = Pr[X = 0] + Pr[X = 1] + Pr[X = 2]$ when $X \sim \text{Bin}(6, 0.2)$;

- `q`, finds quantiles of a distribution for a particular probability. For example, `qnorm(0.7,0,1)` finds x such that $Pr[X \leq x] = 0.75$, when $X \sim N(0,1)$.

Lots of other well-known probability distributions are catered for within R's built-in functions - explore at your leisure!

6
Monte Carlo Methods

6.1 Introduction

The term "Monte Carlo" is used to describe any simulation study which involves random numbers. The name is a reference to the famous Monte Carlo Casino in Monaco, where repetition of random events is the order of the day! The work in this chapter gives some insight into the practical use of random number generation in modern Mathematics and Statistics.

6.2 What is a simulation study?

For our purposes, a simulation study is any study where we examine the properties of a system using random numbers. The *Monopoly* exercise in your group project is therefore an example of a simulation study. Often simulation studies involve estimating the probability of an event (e.g. the probability of landing on a particular square in the *Monopoly* board game). In general, suppose we do an experiment which has the event A as one possible outcome. We would like to estimate the probability of A, denoted by $\Pr[A]$. Then, by repeatedly simulating the experiment, it is simple to estimate $\Pr[A]$, the probability of the event A, using $P_F(A)$, where:

$$P_F(A) = \frac{\text{No. of times } A \text{ occurs}}{\text{Number of times experiment simulated}}.$$

Here $P_F(A)$ is the *frequency estimate* of $\Pr[A]$. Why does this work? Well, suppose we denote the number of times we simulate the experiment by n; then $P_F(A)$ has the following important property:

$$\text{as } n \to \infty, \text{ then } P_F(A) \to \Pr[A].$$

This means that the more simulations we do, the more *accurate* our estimate of $\Pr[A]$.

6.3 Approximation of integrals

Suppose we have a complicated function $f(x)$ defined on the interval $(0,1)$, and also suppose that $f(x) \in (0,1)$ (see Figure 6.1 overleaf). We would like to evaluate $\int_0^1 f(x)\,dx$, but the function may be too

complicated to integrate. We can find an approximate answer by noting that the integral is equal to the area under the curve, and using Monte Carlo methods to approximate this area. We design an experiment which would work in general, even if the function was defined on a general range (a, b), and if $f(x) \in (0, c)$, for any positive value c. We generate a random data point from a *simulation grid*. Let

$$A = \{\text{the data point lies below the curve}\}.$$

Then

$$\Pr[A] = \frac{\text{area under curve}}{\text{area of simulation grid}},$$

so that

$$\begin{aligned}\int_a^b f(x)\,dx &= [\text{area under curve}] \\ &= \Pr[A] \times [\text{area of simulation grid}].\end{aligned}$$

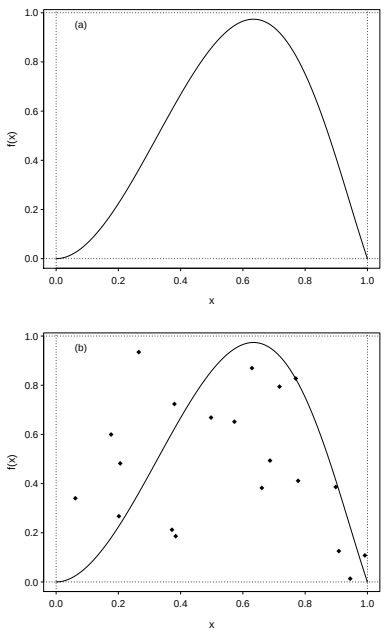

Figure 6.1: (a) An example function. (b) Twenty points randomly placed on a *simulation grid*.

6.3.1 Example

Consider the function we saw earlier in Figure 6.1, defined on $(0, 1)$, and with $f(x) \in (0, 1)$. Estimate $\int_0^1 f(x)dx$ using Monte Carlo methods.

Solution:

6.3.2 *Example using R*

We can easily implement the above algorithm in R. As an extreme example, consider the function

$$f(x) = |\sin\{\sin[\sin(x^4)]\}| \,.$$

We wish to calculate

$$\int_0^2 f(x)\,dx = \int_0^2 |\sin\{\sin[\sin(x^4)]\}|\,dx$$

(which evaluates to approximately 0.71875). First plot the function to determine the necessary simulation grid:

```
> x = seq(0, 2, by=0.001)
> plot(x, abs(sin(sin(sin(x^4)))),
+      xlab="x", ylab="f(x)", type="l",
+      xlim=c(-1,3), ylim=c(0,1))
> abline(h=0, col=2, lty=2)
> abline(h=1, col=2, lty=2)
> abline(v=0, col=2, lty=2)
> abline(v=2, col=2, lty=2)
```

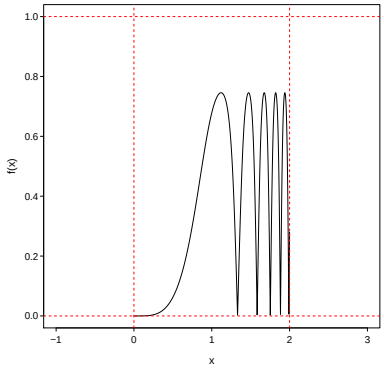

Figure 6.2: A plot of $|\sin\{\sin[\sin(x^4)]\}|$ with the sampling region.

The plot generated in the code above is shown in figure 6.2. Then we use a `for` loop to simulate lots of random numbers:

```
> set.seed(1)
> N = 100000
> no_of_hits = 0
> for(i in 1:N) {
+   x = runif(1, 0, 2); y = runif(1)
+   if(abs(sin(sin(sin(x^4)))) > y) {
+     no_of_hits = no_of_hits + 1
+   }
+ }
> area_under_curve = no_of_hits/N*2
> area_under_curve
[1] 0.71524
```

6.4 Other examples of simulation studies

6.4.1 Simulating the number of sixes on three rolls of a die

Suppose you roll a fair, six–sided die three times. What is the distribution of the number of sixes that can be rolled?

- Each roll of the dice is an experiment or trial which gives a "six" (success, or s) or "not a six" (failure, or f)
- The probability of a success is

$$p = \Pr(\text{six}) = 1/6.$$

- We have $n = 3$ independent experiments or trials (rolls of the dice)
- Let X be the number of sixes obtained

So, for example, the probabibility of getting 2 sixes

$$\Pr(ssf) = \left(\frac{1}{6}\right)^2 \left(\frac{5}{6}\right);$$

or perhaps

$$\Pr(sfs) = \left(\frac{1}{6}\right)\left(\frac{5}{6}\right)\left(\frac{1}{6}\right) = \left(\frac{1}{6}\right)^2 \left(\frac{5}{6}\right);$$

or even

$$\Pr(fss) = \left(\frac{5}{6}\right)\left(\frac{1}{6}\right)^2,$$

giving

$$\Pr(X = 2) = 3 \times \left(\frac{1}{6}\right)^2 \times \left(\frac{5}{6}\right) = 0.069.$$

Clearly $X \sim \text{Bin}(3, 1/6)$, and so, more generally,

$$\begin{aligned} \Pr(X = r) &= {}^n\text{C}_r \times p^r \times (1-p)^{n-r} \\ &= {}^3\text{C}_r \times \left(\frac{1}{6}\right)^r \times \left(\frac{5}{6}\right)^{3-r}, \end{aligned}$$

giving

$$\Pr(X = 2) = {}^3\text{C}_2 \times \left(\frac{1}{6}\right)^2 \times \left(\frac{5}{6}\right) = 0.069.$$

Similarly,

$$\Pr(X = 0) \quad =$$

$$\Pr(X = 1) \quad =$$

$$\Pr(X = 3) \quad =$$

Giving

x	0	1	2	3
$\Pr(X = x)$	0.579	0.347	0.069	0.005

Now let's use R to simulate this die-rolling experiment to see what we observe!

```
> set.seed(1)
> N = 100000
> sixes = vector("numeric",N)
> for(i in 1:N) {
+   game = sample(1:6, 3, replace=TRUE)
+   sixes[i] = length(game[game==6])
+   }
> table(sixes)
sixes
    0     1     2     3
57846 34767  6952   435
```

We can then estimate $\Pr(X = r)$, $r = 0, 1, 2, 3$ in the following way:

$$P_F(X = r) = \frac{\text{No. of times } r \text{ sixes occurs}}{\text{No. of times experiment simulated}},$$

giving, for example,

$$P_F(X = 2) = 6952/10000 = 0.06952.$$

The full estimated probability distribution is then:

x	0	1	2	3
$P_F(X = x)$	0.57846	0.34767	0.06952	0.00435

Look how close these are to the true values!

6.4.2 Investigating the distribution of the sample mean

In Stage 2 Statistics courses you will think about *point estimates* for population parameters. For example, the sample mean $\bar{x}$ is a point estimate of the population mean μ: $\bar{x}$ is an estimate of μ and, when you calculate it, it gives you a single 'point' on the real line.

Is the sample mean $\bar{x}$ a "good" estimator of μ? In other words, will it lie close to the value of the population mean? Since $\bar{x}$ is just a point on the real line, it is unlikely that $\bar{x} = \mu$. However, it would be good if we could figure out just how close our estimate is to the "true" value, and it would be good if we could get a handle on the variability of our estimate. For example, the sample mean is bound to vary from sample to sample. Can we capture this variability by using just our single sample of size n?

More details of what makes a "good" estimator will be provided in later Statistics courses

The answer is "Yes"! We can do this through the *Central Limit Theorem* (CLT) . Informally, the CLT tells us that, *no matter what the distribution of the parent population*, as $n \to \infty$ the sample mean $\bar{x}$ follows a Normal distribution with mean μ and variance σ^2/n, where μ and σ^2 are the population mean and variance, respectively, and n is the sample size. In other words,

You will be introduced to this idea formally in second year

$$\bar{x} \sim N\left(\mu, \frac{\sigma^2}{n}\right) \qquad \text{for large } n.$$

We will now use Monte Carlo methods in R to test out this theorem. Suppose we have a population of 10,000 values from a $U(100, 500)$ distribution:

```
> set.seed(1)
> population = runif(10000,100,500)
```

A plot of these values is shown in Figure 6.3. You can clearly see the Uniform distribution here, over the range $(100, 500)$.

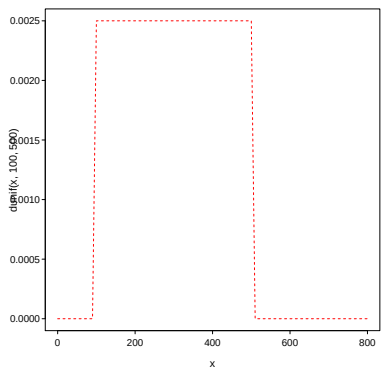

Figure 6.3: A plot of the $U(100, 500)$ population.

We can work out the 'true' values of the mean and variance in the population by typing:

```
> (mu = mean(population))
[1] 300.0672
> (sigma = var(population))
[1] 13553.12
```

Thus, according to the central limit theorem, we have

$$\bar{x} \sim N\left(300, \frac{13553}{n}\right).$$

Let's test this out for different n.

Sample size $n = 5$

Let's now take 1000 samples of size $n = 5$ from the population; calculate the sample mean in each; and then look at some summaries of our sample means. Note that – according to the central limit theorem – we *should* have (approximately):

$$\bar{x} \sim N\left(300, 2710.6\right).$$

A histogram of these sample means is shown in Figure 6.4.

```
> xbar5 = vector("numeric",1000)
> for(i in 1:1000){
+   S = sample(population, size=5, replace=FALSE)
+   xbar5[i] = mean(S)
+ }
> (mean(xbar5))
[1] 300.1127
> (var(xbar5))
[1] 2479.201
```

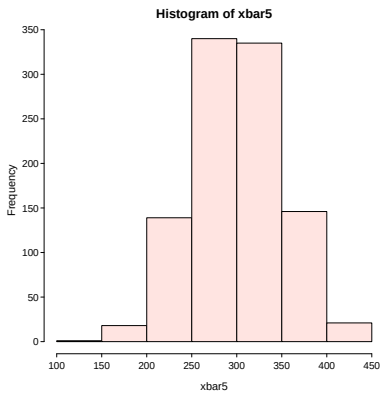

Figure 6.4: Sampling distribution for $\bar{x}$ with $n = 5$.

Sample size $n = 50$

Now let's take 1000 samples of size $n = 50$ and repeat the whole procedure. As before, according to the central limit theorem, we should have

$$\bar{x} \sim N(300, 271.06).$$

See the output below and the graph in Figure 6.5, plotted over the same range as that in Figure 6.4.

```
> xbar50 = vector("numeric",1000)
> for(i in 1:1000){
+   S = sample(population, size=50, replace=FALSE)
+   xbar50[i] = mean(S)
+ }
> (mean(xbar50))
[1] 300.2136
> (var(xbar50))
[1] 278.2053
```

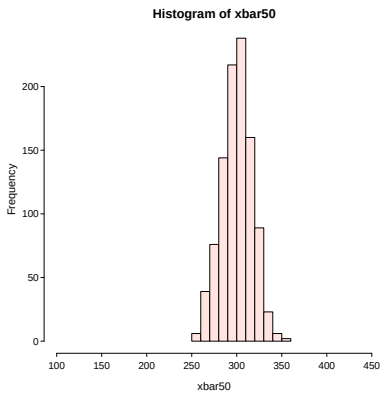

Figure 6.5: Sampling distribution for $\bar{x}$ with $n = 50$.

Sample size $n = 500$

Now let's take 1000 samples of size $n = 500$ and repeat the whole procedure *again*. According to the central limit theorem, we should have

$$\bar{x} \sim N(300, 27.106).$$

See the output below and the graph in Figure 6.6, ploted over the same range as that in Figures 6.4 and 6.5.

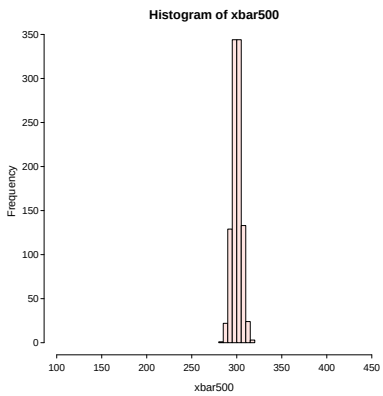

Figure 6.6: Sampling distribution for $\bar{x}$ with $n = 500$.

```
> xbar500 = vector("numeric",1000)
> for(i in 1:1000){
+   S = sample(population, size=500, replace=FALSE)
+   xbar500[i] = mean(S)
+ }
> (mean(xbar500))
[1] 300.0444
> (var(xbar500))
[1] 25.36247
```

Comments

- The Central Limit Theorem seems to hold! In each case, the mean of $\bar{x}$, and the variance of $\bar{x}$, are close to what we'd expect
- In each of the plots, we see that normalilty seems to hold – even though the population distribution was non-normal
- As the size of the samples increases, the sampling distribution of the mean becomes less variable
- In all cases, the sampling distributions are centred around the true mean μ