6
Random Number Generation

6.1 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*.

6.2 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* (RNG), 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 RNG:

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 $r_{i-1}, r_{i-2}, \ldots$.

6.3 Congruential generators

Consider the set $\mathbb{N}^0$ of non–negative integers. That is, $\mathbb{N}^0 = 0, 1, 2, \ldots$. Let 'mod' represent the *modulo* operation, so that for $x, m \in \mathbb{N}^0, x \neq 0$, (x) mod m means that x is divided by m and the remainder is taken as the result. For example, in R:

```
> 15%%6
[1] 3
```

6.3.1 Example: modulo operations

1. What is 13 mod 4? Answer =
2. What is 19 mod 5? Answer =
3. What is 2008 mod 3? Answer =
4. What is 10,008 mod 11? Answer =

Now consider the relation

$$r_i = (ar_{i-1} + b) \bmod m, \quad i = 1, 2, \ldots, m \tag{6.1}$$

where r_0 is the initial number, known as the *seed*, and $a, b, m \in \mathbb{N}^0$ are the *multiplier*, *additive constant* and *modulo* respectively.
The modulo operation means that at most m different numbers can be generated before the sequence must repeat – namely, the integers $0, 1, 2, \ldots, m-1$. The actual number of generated numbers is $h \leq m$, called the *period* of the generator.

6.3.2 Example: Congruential generators

Selecting $a = 17$, $b = 0$, $m = 100$, $r_0 = 13$ in relation (6.1) generates the following sequence:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
r_i	13	21	57			41			33	61	37	29	93	81	77			1	17	

The next value, r_{20}, is found to be 13 so that this sequence then repeats. Thus, this sequence has period 20. We can use R to calculate this sequence:

```
> a=17
> b=0
> m=100
> r_0=13
> ran=vector("numeric",length=20)
> ran[1]=(a*r_0+b)%%m
> for(i in 2:20){
+   ran[i]=(a*ran[i-1]+b)%%m
+   }
> ran
 [1] 21 57 69 73 41 97 49 33 61 37 29 93 81 77  9 53  1 17
89 13
```

6.3.3 *Example: Bad random number generators*

In the 1970's, a popular random generator used was `RANDU`, where $m = 2^{31}, a = 65539$ and $b = 0$. Unfortunately, this is a spectacularly bad choice of parameters. On noting that $a = 65539 = 2^{16} + 3$, then we see that

$$r_{i+1} = ar_i = 65539 \times r_i = (2^{16} + 3)r_i \, .$$

So

$$r_{i+2} = a\, r_{i+1} = (2^{16} + 3) \times r_{i+1} = (2^{16} + 3)^2 r_i \, .$$

On expanding the square, we get

$$r_{i+2} = (2^{32} + 6 \times 2^{16} + 9)r_i = [6(2^{16} + 3) - 9]r_i = 6r_{i+1} - 9r_i \, .$$

Note: all these calculations should be to the mod 2^{31}. So, there is a large correlation between the three points! What does this mean in practice? Well, let's consider triplets from this random generator and compare them to R's standard random number generator.
The `RANDU` demo simulates 1000 random numbers from RANDU and from R's standard random number generator. It then plots the results.

```
> require(mas1343)
> demo(RANDU)
```

This gives Figure 6.1.

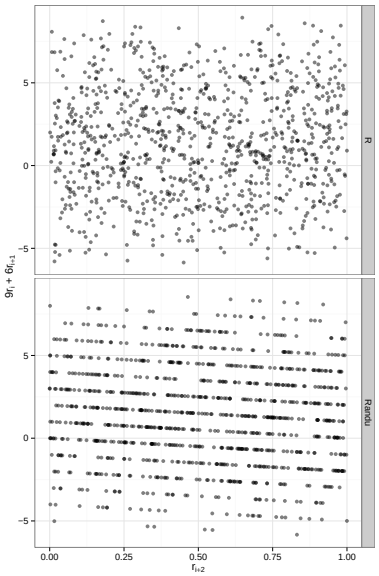

Figure 6.1: Comparison of the `Randu` algorithm and a standard R algorithm

Today's random number generators are a (bit) more sophisticated than what we come across in this course. Newer methods also have quite cool names: `Mersenne-Twister`[2] and `Super-Duper`.

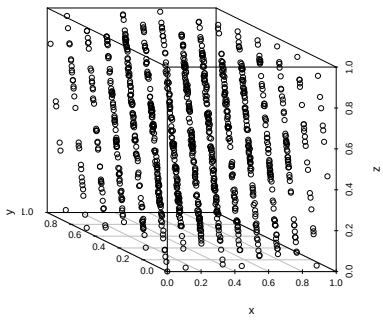

Figure 6.2: 3d scatterplot of `Randu` triples.

[2] http://en.wikipedia.org/wiki/Mersenne_twister

Computer generation of pseudo–random numbers - remarks

1. As computers essentially use numbers to base 2, generators generally use $m = 2^k$, where k is a very large number ($k \in \mathbb{N}$).
2. We want the period of the sequence to be as large as possible.

6.4 *Assessing random number generators*

Theorem

For the relation

(6.2)

the maximum period, m, is achieved for $b > 0$ if, and only if:

(i)

(ii)

(iii)

Proof: Beyond the scope of this course.

Remarks

1. If $m = 2^k$, then if $a = 4c + 1$ for some positive integer c, (ii) and (iii) will hold.

2. Similarly, for (i) to be true then b must be a positive odd integer if $m = 2^k$.

3. As a real example, the Numerical Algorithms Group (NAG) `Fortran` library uses $k = 59$, $b = 0$ and $a = 13^{13}$ in one of it's random number generator.

6.4.1 Example: Maximum periods of RNG

Check to see if the maximum period can be achieved if the congruential method with the following parameters is used to generate a sequence of pseudo–random numbers:

1. $\boldsymbol{a = 16, b = 5, m = 20}$

   - *All three* conditions must be satisfied for the maximum period to be obtained, so we check each in turn. Condition (i): False. $b = 5$ and $m = 20$ have a common factor, 5. Hence the maximum period is *not* achieved.

2. $\boldsymbol{a = 16, b = 3, m = 20}$

   - Condition (i): True. b and m have no common factors other than 1.
   - Condition (ii): False. $(a - 1) = 15$, and this is *not* divisible by 2. But 20 *is* divisible by 2.
   - Hence the maximum period is *not* achieved.

3. $\boldsymbol{a = 11, b = 3, m = 20}$

   - Condition (i):

   - Condition (ii):

   - Condition (iii):

   -

4. $a = 21, b = 3, m = 20$

- Condition (i):
- Condition (ii):
- Condition (iii):
-

6.5 Random numbers in R

6.5.1 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. For example,

```
> 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.

6.5.2 *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.

Example usage of `sample`

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.

6.5.3 *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 :-(

Summary of R commands

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 6.1: Summary of R commands in this chapter.

7
Simulating Discrete Random Numbers

7.1 Simulating a Bernoulli random variable

The Bernoulli random variable $I \sim \text{Bern}(p)$ has already been encountered in MAS1341. It is perhaps the simplest random variable and has the probability mass function

i	0	1
$\Pr[I = i]$		

To simulate such a quantity, we generate an observation u from a uniform $U(0,1)$ distribution and set

7.1.1 Example: Bernoulli random numbers

Suppose we generate a number from a uniform $U(0,1)$ distribution:

$$0.332,\ 0.739,\ 0.653,\ 0.110,\ 0.587,\ 0.144$$

and we wish to use these to simulate six independent observations from I, a Bern(0.8) random variable. Here $p = 0.8$, so we convert using:

to obtain the sequence 1, 1, 1, 0, 1, 0 as our Bernoulli random sample.

7.1.2 Using R to simulate a Bernoulli R.V.

To simulate a Bernoulli variable using R is straightforward. The easiest way is just to use the sample command:

```
> p = 0.5
> sample(c(0, 1), 1, prob=c(1-p, p), replace=TRUE)
[1] 1
> sample(c(0, 1), 10, prob=c(1-p, p), replace=TRUE)
 [1] 1 1 1 1 0 0 0 0 1 1
```

7.2 Simulation of discrete random variables

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

7.2.1 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 calcaluate the 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 $U \sim \text{Uniform}(0,1)$, so

- 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 $X = 4$.

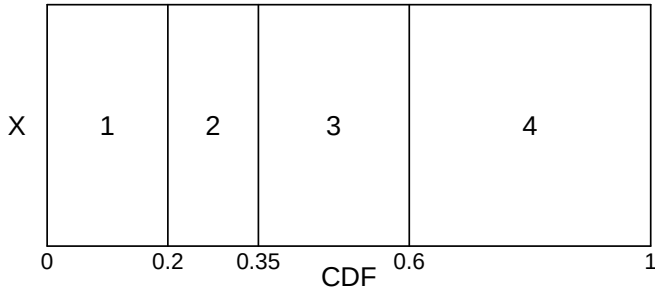

Figure 7.1: Simulating discrete random numbers.

Using R

To simulate the above distribution in R, there a two (obvious) methods that we can use: 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 use 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
+ }
```

7.2.2 Simulating a Poisson random variable

For the uniform random numbers,

$$0.253,\ 0.588,\ 0.789$$

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

Solution

The pdf of the poisson distribution is

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

where $\lambda > 0$. So we have

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

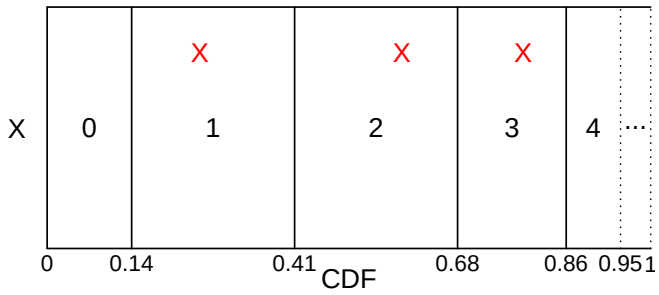

Figure 7.2: Simulating discrete random numbers from the Poisson distribution.

Hence our random numbers are 1, 2 and 3.

7.3 Simulating a Geometric random variable

Let $X \sim \text{Geom}(p)$, so that X takes the random values $1, 2, 3, \ldots$ with probabilities p, $(1-p)p$, $(1-p)^2 p$, ... Hence,

$$\Pr[X = k] = p(1-p)^{k-1}\ .$$

The technique descibed in this section only works for the geometric distribution!

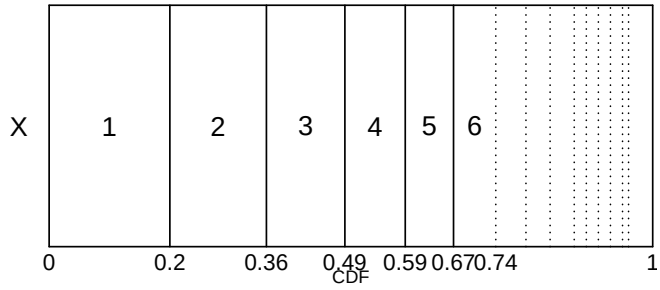

Figure 7.3: Diagram illustrating on simulating discrete random numbers from the Geometric (p=0.2) distribution.

The Geometric distribution is the distribution of the number of independent Bernoulli trials until the first success is encountered. For each individual trial, the probability of success is p. We know from §7.2 that we will obtain the value $X = k$ (say) when simulating an observation from X using a value of U if

$$\Pr[X=1]+\Pr[X=2]+\ldots+\Pr[X=k-1] \le U < \Pr[X=1]+\ldots+\Pr[X=k],$$

R's Geometric distribution: R defines the geometric distribution slightly different to how we do: $\Pr[X = k] = p(1-p)^x$. This means that when simulating geometric random numbers from R we have to add 1.

i.e.

$$\sum_{j=1}^{k-1}\Pr[X=j] \le U < \sum_{j=1}^{k}\Pr[X=j],$$

i.e.

$$\sum_{j=1}^{k-1}(1-p)^{j-1}p \le U < \sum_{j=1}^{k}(1-p)^{j-1}p\,. \tag{7.1}$$

We know (for $0 < r < 1$) that

$$r\sum_{k=0}^{n}(1-r)^k = 1-(1-r)^{n+1}$$

i.e. a Geometric progression. So

$$\begin{aligned}\sum_{j=1}^{k-1}(1-p)^{j-1}p &= p\sum_{j=1}^{k-1}(1-p)^{j-1}\\ &= p\sum_{j=0}^{k-2}(1-p)^{j}\\ &= 1-(1-p)^{k-1}\,.\end{aligned}$$

Hence for expression 7.1 we have

$$1-(1-p)^{k-1} \le U < 1-(1-p)^k,$$

i.e.

$$(1-p)^{k-1} \ge 1-U > (1-p)^k,$$

i.e.

$$(k-1)\ln(1-p) \ge \ln(1-U) > k\ln(1-p),$$

i.e.

$$k - 1 \leq \frac{\ln(1-U)}{\ln(1-p)} < k,$$

with the inequality sign reversing since $\ln(1-p) < 0$. Thus, we observe $X = k$ if

1. $X > \frac{\ln(1-U)}{\ln(1-p)}$

2. $X - 1 \leq \frac{\ln(1-U)}{\ln(1-p)}$, i.e. $X \leq 1 + \frac{\ln(1-U)}{\ln(1-p)}$.

Both (1) and (2) are satisfied by $X = 1 + \left\lfloor \frac{\ln(1-U)}{\ln(1-p)} \right\rfloor$.

The symbol $\lfloor \quad \rfloor$ denotes "the integer part" (e.g. $\lfloor 3.246 \rfloor = 3$, $\lfloor 5.685 = 5 \rfloor$). In R, we use the `floor` function.

7.3.1 Examples: geometric random numbers

1. Given $u = 0.2179$, simulate a Geom(0.2) random variable X.
   Solution

2. Given $u = 0.8923$, simulate a Geom(0.4) random variable X.
   Solution

8
Monte Carlo Methods

8.1 The continuous uniform U(0,1) distribution

We have already used this random variable a great deal: the Uniform distribution on the interval $(0,1)$, see Figure 8.1. This is denoted by $U(0,1)$ and denotes the random variable which has probability density function (PDF):

$$f_X(x) = \begin{cases} 1 & 0 < x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

In the Uniform $U(a,b)$ distribution, all values in the range $a \longrightarrow b$ are equally likely; thus, if $a = 0$ and $b = 1$, the PDF must be 1 for all values in the range $0 \longrightarrow 1$, since the area under the PDF must be 1.

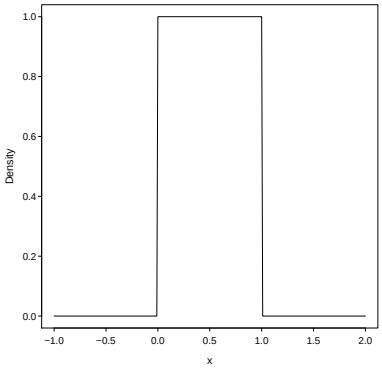

Figure 8.1: PDF of the uniform distribution.

Up to now, we have used $U(0,1)$ random variables to simulate from many discrete distributions, and taken the $U(0,1)$ random variables for granted - but how do we simulate them in the first place?

The key is in the accuracy to which the numbers are generated. Generating $U(0,1)$ random variables *precisely* would be very difficult! (Impossible?!?) However, we can get three decimal places by generating a random integer x from the set

$$\{0,1,2,\ldots,999\},$$

with all outcomes equally likely, and then putting $u = x/1000$. Then u is a simulation from a $U(0,1)$ random variable recorded to three decimals. The simulation of x is 'easy', e.g. using a congruential generator with $m = 1000$ and maximum period, as we saw in Chapter 5. So for a full period congruential generator with $m = 2^{32}$ we get set of integers:

$$\{0,1,2,\ldots,2^{32}-1\}\ .$$

Setting $u = x/2^{32}$ would give a pseudo–random number from the $U(0,1)$ distribution, recorded to about 8 decimal places.

8.2 Monte Carlo

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!

8.2.1 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. We have already seen simulation studies, for example in the monopoly practical. Often simulation studies involve estimating the probability of an event (e.g. in Practical 4 the probability of landing on a particular square). 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 $\Pr[A]$.

8.3 Approximation of integrals

Suppose we have a complicated function $f(x)$ defined on the interval $(0,1)$, and also suppose that $f(x) \in (0,1)$.

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. 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

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

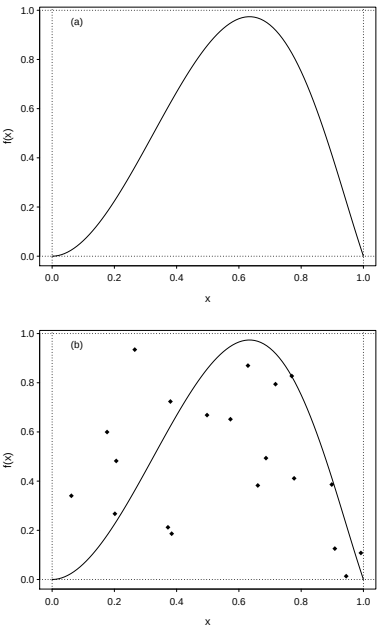

Figure 8.2: (a) An example function. (b) Twenty points randomly placed on the graph.

8.3.1 Example

Consider the function we saw earlier in Figure 8.2, defined on $(0,1)$, and with $f(x) \in (0,1)$. Estimate $\int_0^1 f(x)dx$ using Monte Carlo methods.

Solution:

1. We simulate n data points from the simulation grid; here this is the unit square $(0,1) \times (0,1)$.
2. Each of the coordinates x and y are generated using a $U(0,1)$ random variable. Note the fact that

   $$A = \{\text{data point } (x,y) \text{ lies below the curve}\} = \{y < f(x)\}.$$

3. If r points lie below the curve, then $\Pr[A] \simeq P_F(a) = r/n$, and thus

   $$\begin{aligned}\int_0^1 f(x)\,dx &= \Pr[A] \times [\text{area of simulation grid}] \\ &= \Pr[A] \times 1 \\ &= \Pr[A] \\ &\simeq P_F(A) \\ &= r/n.\end{aligned}$$

4. In our case, from the figure above we estimate:

   $$\int_0^1 f(x)\,dx \simeq \frac{r}{n} = \frac{14}{20} = 0.7\,.$$

8.3.2 *Example using R*

We can also 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 according to maple evaluates to approximately 0.71875. First plot the function to determine the necessary region:

```
> 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)
```

The plot generated in the code above is shown in figure 8.2. 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
```

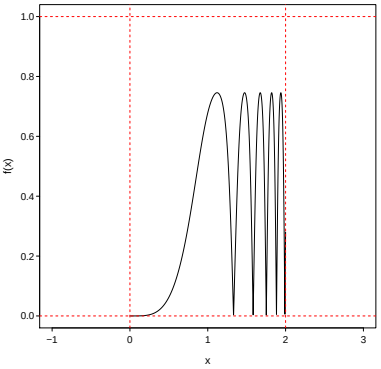

Figure 8.3: A plot of $|\sin\{\sin[\sin(x^4)]\}|$ with the sampling region.

8.4 Other examples of simulation studies

8.4.1 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 probability 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,

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

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

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

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!

8.4.2 Investigating the distribution of the sample mean

Recently in MAS1342 you have been thinking about *point estimates*. 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 after Easter in MAS1342

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

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

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

A plot of these values is shown in Figure 8.4. You can clearly see the Uniform distribution here, over the range $(100, 500)$.

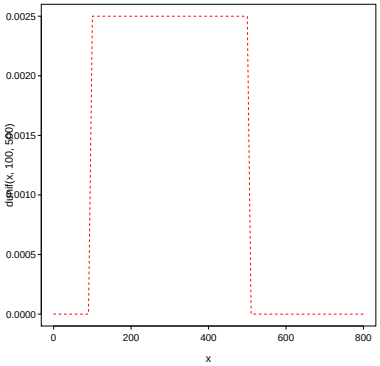

Figure 8.4: A plot of the $U(100, 500)$ population.

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 the the central limit theorem – we *should* have (approximately):

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

A histogram of these sample means is shown in Figure 8.5.

```
> 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
```

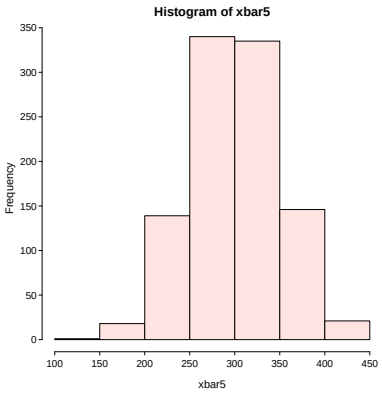

Figure 8.5: Sampling distribution for $\bar{x}$ with $n = 5$.

Sample size $n = 50$

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

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

See the output below and the graph in Figure 8.6, ploted over the same range as that in Figure 8.5.

```
> 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
```

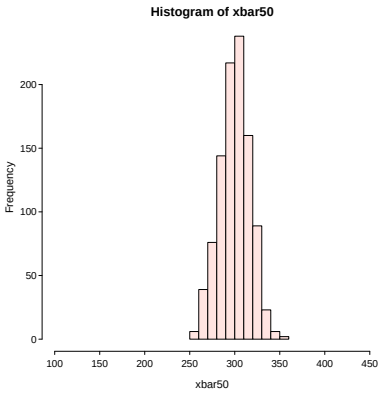

Figure 8.6: Sampling distribution for $\bar{x}$ with $n = 50$.

Sample size $n = 500$

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

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

See the output below and the graph in Figure 8.7, ploted over the same range as that in Figure 8.5.

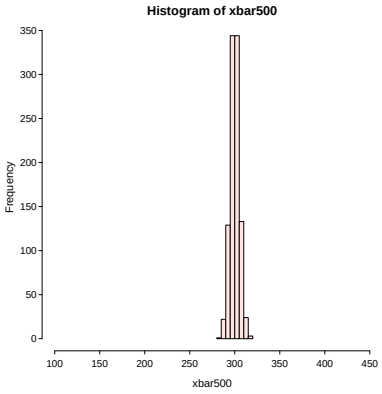

Figure 8.7: Sampling distribution for $\bar{x}$ with $n = 500$.

```
> 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