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(`min`,`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

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

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

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

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
$\Pr[X \le x]$	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 $0.2 \le u < (0.2 + 0.15) = 0.35$, set $X = 2$;
- if $0.35 \le u < (0.2 + 0.15 + 0.25) = 0.6$, set $X = 3$;
- if $u \ge 0.6$, 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.

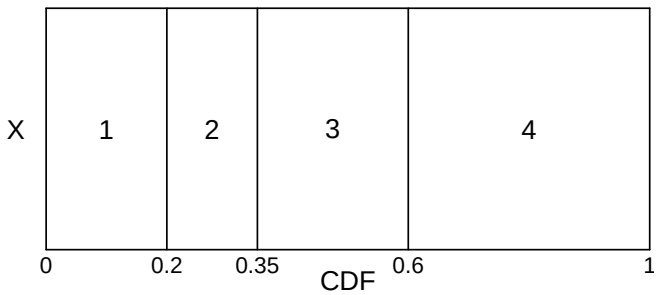

Figure 5.1: Simulating discrete random numbers.

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

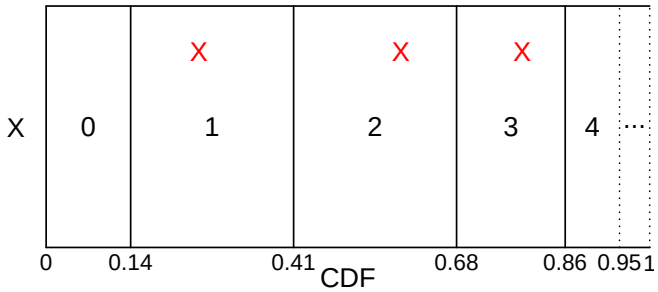

Figure 5.2: Simulating discrete random numbers from the Poisson distribution.

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!