

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 $0, 1, 2, \dots, 9$. 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 applications¹.

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

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

6.3 Congruential generators

Consider the set $\mathbb{N}^0$ of non-negative integers. That is, $\mathbb{N}^0 = 0, 1, 2, \dots$. Let ‘mod’ represent the *modulo* operation, so that for $x, m \in \mathbb{N}^0, x \neq 0$, $(x) \bmod m$ 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

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

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 $0, 1, 2, \dots, m - 1$. 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.

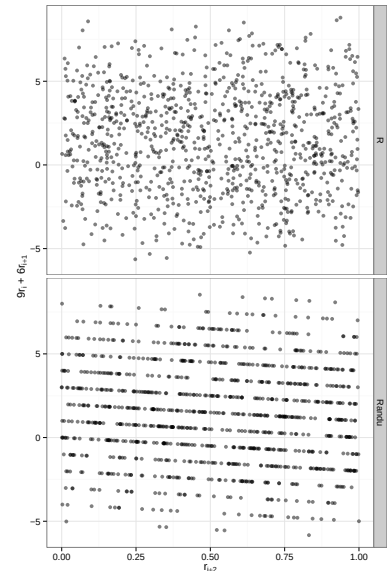


Figure 6.1: Comparison of the **Randu** algorithm and a standard R algorithm

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**.

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.

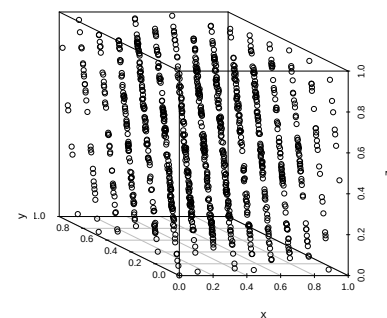


Figure 6.2: 3d scatterplot of **Randu** triples.

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

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 its 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. $a = 16, b = 5, m = 20$
 - 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. $a = 16, b = 3, m = 20$
 - 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. $a = 11, b = 3, m = 20$
 - 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.

```
> 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] 0.1475438
> runif(1)
[1] 0.5885058
> runif(5)
[1] 0.8960070 0.5196363 0.9398020 0.2590696 0.9195305
> runif(1, 6, 7)
[1] 6.096255
```

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.

```
> 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.

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

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.

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

```
> 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.

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

6.5.3 Simulating the Capital One Cup draw

We are in the semi-finals of the Capital One cup³, and need to organise the draw for the final stage. The remaining teams are:

Manchester Utd, Manchester City, Sunderland, West Ham.

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:

```
> 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

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



⁴ As actually happened

⁵ Perhaps. Actually, it didn't matter – Sunderland beat Man Utd and went on to bring the Cup home after beating Man City in the final!

Summary of R commands

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

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
$\text{Pr}[I = i]$		

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. The reason for the delay is for MAS1341 to cover sufficient material.:

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 $\text{Bern}(0.8)$ 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] 0
> sample(c(0, 1), 10, prob=c(1-p, p), replace=TRUE)
[1] 1 0 1 0 1 0 1 0 0 1
```

7.2 *Simulation of discrete random variables*

You would use 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 calculate 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 $X = 4$.

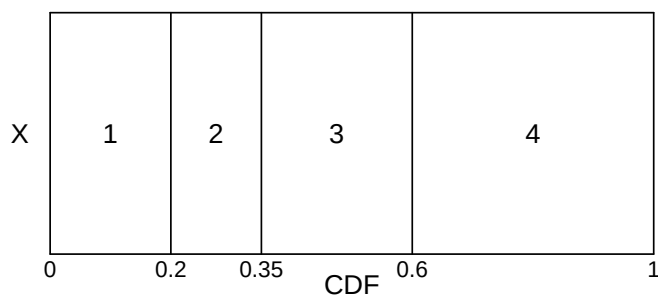


Figure 7.1: Simulating discrete random numbers.

Using R

To simulate the above distribution in R, there are 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] 3
```

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

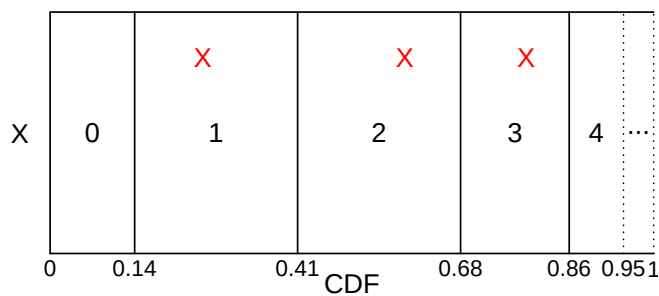


Figure 7.2: Simulating discrete random numbers from the Poisson distribution.

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 $1, 2, 3, \dots$ with probabilities $p, (1-p)p, (1-p)^2p, \dots$. Hence,

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

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

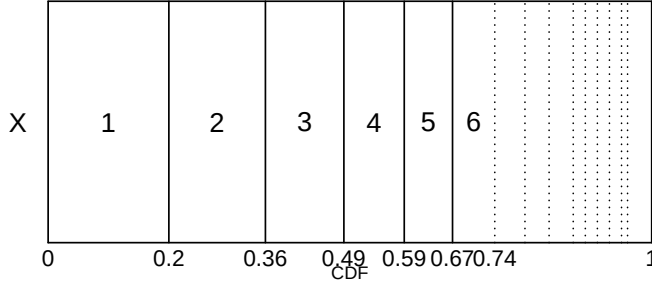


Figure 7.3: Diagram illustrating on simulating discrete random numbers from the Geometric ($p=0.2$) distribution.

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

$$\Pr[X = 1] + \Pr[X = 2] + \dots + \Pr[X = k - 1] \leq U < \Pr[X = 1] + \dots + \Pr[X = k],$$

i.e.

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

i.e.

$$\sum_{j=1}^{k-1} (1-p)^{j-1} p \leq U < \sum_{j=1}^k (1-p)^{j-1} p. \quad (7.1)$$

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

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

i.e.

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

i.e.

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

i.e.

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

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

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 $\lfloor \cdot \rfloor$ denotes “the integer part” (e.g. $\lfloor 3.246 \rfloor = 3$, $\lfloor 5.685 \rfloor = 5$). In R, we use the `floor` function.

7.3.1 Examples: geometric random numbers

1. Given $u = 0.2179$, simulate a $\text{Geom}(0.2)$ random variable X .

Solution

2. Given $u = 0.8923$, simulate a $\text{Geom}(0.4)$ random variable X .

Solution

$$\begin{aligned} X &= 1 + \left\lfloor \frac{\ln(1 - 0.8923)}{\ln(1 - 0.4)} \right\rfloor \\ &= 1 + \left\lfloor \frac{-2.228406}{-0.5108256} \right\rfloor \\ &= 1 + \lfloor 4.362361 \rfloor \\ &= 1 + 4 = 5. \end{aligned}$$