DR LEE FAWCETT

MAS1802

PROBLEM SOLVING AND COMPUTING II (COMPUTING WITH R)

NEWCASTLE UNIVERSITY

Useful calendar:

Week beginning...	**Monday**	**Tuesday**	**Thursday**	**Friday**
2nd March	Intro lecture 11-12 *(Herschel LT1)*	Intro Practical 9-11 *(Herschel PC/ Barbara Strang Chart)*	✘	Lecture 1-2 *(Herschel LT1)* Start Practical 1
9th March	✘	Practical 1 9-11 *(Herschel PC/ Barbara Strang Chart)*	Practical 1 DI 2-3 *(Herschel PC)*	**Prac 1 due: 3:00pm** Lecture 1-2 *(Herschel LT1)* Start Practical 2
16th March	✘	Practical 2 9-11 *(Herschel PC/ Barbara Strang Chart)*	Practical 2 DI 2-3 *(Herschel PC)*	**Prac 2 due: 3:00pm** Lecture 1-2 *(Herschel LT1)* Start Practical 3
23rd March	✘	Practical 3 9-11 *(Herschel PC/ Barbara Strang Chart)*	Practical 3 DI 2-3 *(Herschel PC)*	**Prac 3 due: 3:00pm** Lecture 1-2 *(Herschel LT1)* Start Project

Week beginning...	**Monday**	**Tuesday**	**Thursday**	**Friday**
27th April	✘	Project 9-11 *(Herschel PC/ Barbara Strang Chart)*	Project DI 2-3 *(Herschel PC)*	Lecture 1-2 *(Herschel LT1)*
4th May	✘	Project 9-11 *(Herschel PC/ Barbara Strang Chart)*	Project DI 2-3 *(Herschel PC)*	Lecture 1-2 *(Herschel LT1)*
11th May	✘	**Project due: 15:00**	✘	✘

Contents

1 *Introduction and Housekeeping* 1

2 *Introduction to R* 7

3 *Data Summaries* 15

4 *Control Statements and Functions* 23

5 *Random Number Generation* 31

6 *Monte Carlo Methods* 39

1
Introduction and Housekeeping

1.1 Housekeeping

The computing part of MAS1802 will run during the second half of semester 2 (unfortunately broken up by the Easter vacation!). We will have weekly lectures, with printed notes, to support the computer practical sessions. Generally, a new topic will be introduced each week and will fit around the following schedule[1]:

Type of session	Day	Time	Place
Lecture	Friday	1–2	Herschel LT1
Practical	Tuesday	9–11	Herschel PC
Office hour	Wednesday	1–2	My office (Herschel 2.07)
Drop-in	Thursday	2–3	Herschel PC
Deadline:	*Friday*	*3pm*	*General office/online*

[1] To kick-start things, we will have two lectures in the first week, with an additional lecture being held on Monday 11am. The very first practical session will include an important introductory session in R.

1.1.1 Lecturer information

The module leader, and lecturer for the computing part of MAS1802, is Dr. Lee Fawcett. If you have any questions, comments or feedback on this course I can be contacted via email at `lee.fawcett@ncl.ac.uk`. More complicated questions about the course are best dealt with face-to-face.

1.1.2 Lectures

Given the nature of this course we will have fewer lectures than the other first year modules – as you might imagine, there will be a strong emphasis on practical skills in the computer cluster. I will hand out copies of the practical worksheets in the lecture before each practical session.

Slides will be used in lectures to accompany the notes. Lectures will also be recorded using the ReCap system. You can access all course materials, including the ReCap recordings, through Blackboard (BB) or via the dedicated course webpage:

http://www.mas.ncl.ac.uk/~nlf8/teaching/mas1802/

1.1.3 Computing practicals

During each computer practical, your general computer skills will be assessed. You are more likely to get high marks for the coursework if:

- You have attended the lecture;
- You don't panic, but read through the question, and the notes, carefully;
- You ask for help[2];
- You are organised! For example, if:
  - your files are organised in a sensible manner – i.e. if you have a directory for MAS1802 and sub-directories for each practical;
  - your files are given convenient and meaningful names;
  - Your code "looks nice".

We won't try to catch you out!

[2] As in MAS1801, there will be lots of help available in computer practicals: Dr. Fawcett and Dr. Matthews will both be on-hand, as will several Ph.D students:

In this course, **all** computer practical sessions are held in the Herschel PC cluster, and all machines in the cluster have the necessary software and packages installed. However, you can use computers in other parts of the University while doing your coursework. A list of free computers can be found at:

`http:m.ncl.ac.uk/itservice/#clusfree`

If you use a machine in a cluster and it is missing the necessary software, contact Christian Lawson-Perfect: `christian.perfect@ncl.ac.uk`.

1.1.4 Office hours

The scheduled office hour for this module is Wednesday 1-2pm. I will always be free at this time, in my office, to see students who need extra help with the work. I also have office hours for other courses which are usually quiet (see noticeboard outside my office), so feel free to drop in then too. In general, I am happy to see students anytime (if I'm free).

1.1.5 Assessment

- End of semester examination worth **40%** of your overall mark. This will be based on material from the pencil-and-paper part of the course with Dr. Duncan.
- Assessment of coursework worth **60%** of your overall mark:
  - Weekly computing exercises with Lee (10%)
  - Group computing project with Lee (30%)
  - Problem-solving (pencil-and-paper) exercises with Dr. Duncan (20%)

The following deadlines relate to the courswork for the computing part of this module:

- Practical 1: Friday 13th March, 3:00pm (online submission)

- Practical 2: Friday 20th March, 3:00pm (hard-copy submission to the General Office)
- Practical 3: Friday 27th March, 3:00pm (online submission)
- Group computing project: Tuesday 12th May, 3:00pm (hard-copy submission to the General Office)

For the group work you will be in the same groups as you were allocated in the problem solving part of the course[3].

[3] Although I might change my mind on this one...

1.1.6 Quiz questions

Last summer, we wrote lots of quiz questions in `Numbas` to help develop your computing skills for this part of the module. These quiz questions are written within the usual computer-based assessment (CBA) framework – if you're interested, go to:

`www.mas.ncl.ac.uk/~nlf8/basicR`

1.1.7 Late work policy

It is not possible to extend submission deadlines for coursework in this module and no late work can be accepted. For details of the policy (including procedures in the event of illness etc.) please look at the School web site

`http://www.ncl.ac.uk/maths-physics/students/resources`

1.2 What is R?

R is a computer package that is widely used for statistical software development and data analysis[4]. R uses a command line interface, though several graphical user interfaces are available. The system provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, and others) and graphical techniques.

[4] For example Google, IBM, Shell, Thomas Cook :-(, Facebook. The recent NY times article at `http://tinyurl.com/9m5qyh` is worth a read.

R is highly extensible through the use of user-submitted libraries for specific functions or specific areas of study. A particular strength of R is its graphical facilities, which produce quality graphs (better than Python!) that can include mathematical symbols. Although R is mostly used by statisticians and other practitioners requiring an environment for statistical computation and software development, it can also be used as a general matrix calculation toolbox with comparable benchmark results to many other software packages (e.g. Python!). Why is it called R? There are a few reasons, one of which is that it's named after the two people who developed the software: Ross Ihaka and Robert Gentleman, both from Auckland University in New Zealand.

You will use R throughout your degree at Newcastle and the best part is, it's **completely free to download to your own computer!**

				Voting statistics					Movie genre							
Title	Year	Length	Budget	Rating	Votes	r1	...	r10	mpaa	Action	Animation	Comedy	Drama	Documentary	Romance	Short
A.k.a. Cassius	1970	85	NA	5.7	43	4.5	...	14.5	PG	0	0	0	0	1	0	0
AKA	2002	123	NA	6.0	335	24.5	...	1.5	R	0	0	0	1	0	0	0
Alien Vs. Pred	2004	102	45000000	5.4	14651	4.5	...	4.5	PG-13	1	0	0	0	0	0	0
Abandon	2002	99	25000000	4.7	2364	4.5	...	4.5	PG-13	0	0	0	1	0	0	0
Abendland	1999	146	NA	5.0	46	14.5	...	24.5	R	0	0	0	0	0	0	0
Aberration	1997	93	NA	4.8	149	14.5	...	4.5	R	0	0	0	0	0	0	0
Abilene	1999	104	NA	4.9	42	0.0	...	24.5	PG	0	0	0	1	0	0	0
Ablaze	2001	97	NA	3.6	98	24.5	...	14.5	R	1	0	0	1	0	0	0
Abominable Dr	1971	94	NA	6.7	1547	4.5	...	14.5	PG-13	0	0	0	0	0	0	0
About Adam	2000	105	NA	6.4	1303	4.5	...	4.5	R	0	0	1	0	0	1	0

Table 1.1: The first ten rows of the movie data set. **Credit:** This data set was initially constructed by Hadley Wickham at http://had.co.nz/.

1.2.1 Accessing R

R is installed on all University machines. In this class we will be using Rstudio, which is an R IDE[5]. Both R and Rstudio are free, so you can install them on you own computer[6]. See

http://www.ncl.ac.uk/maths-physics/students/resources/

for more details.

[5] Integrated development environment (IDE).

[6] Versions of R and Rstudio are available for Windows, Apple Mac and Linux.

1.2.2 Previous computing knowledge

This course is intended to teach you the basics of programming. **No** previous programming knowledge is assumed, although the `Python` part of MAS1801 should have helped your general skills. It is crucial that you come to all the practicals for this course; it is also vital that you let me/other staff know, in the practical sessions, if you're not sure about anything. Burying your head in the sand will not help. Neither will crying, or getting angry[7].

[7] Please don't do this.

1.2.3 Recommended R textbooks

On the course webpage I have a list of suitable books on R programming. Since you will be using R throughout your degree, then it may be worthwhile buying a good R textbook. A really good, free-to-download book written by Venables and Smith, can be found here:

https://cran.r-project.org/doc/manuals/R-intro.pdf

1.3 Movies dataset

The internet movie database, (IMDB) http://imdb.com/, is a website devoted to collecting movie data supplied by studios and fans. It claims to be the biggest movie database on the web and is run by *Amazon*. More information `imdb.com` can be found online:

http://imdb.com/help/show_leaf?about

including information about the data collection process

http://imdb.com/help/show_leaf?infosource

IMDB makes their raw data available at http://uk.imdb.com/interfaces/.

Movies are selected for inclusion if they had a known length, had been rated by at least one IMDB user and had an MPAA (*Motion Picture Assocation of America*) rating. Amongst other things, the dataset contains the following fields:

- Title: Title of the movie.
- Year: Year of release.
- Budget: Total budget in US dollars. If the budget isn't known, then it is stored as 'NA'.
- Length: Length in minutes.
- Rating: Average IMDB user rating.
- Votes: Number of IMDB users who rated this movie.
- r1-10: Percentage(to nearest 5%) of users who rated this movie 1, ..., 10
- mpaa: MPAA rating.
- Genre: Action, Animation, Comedy, Drama, Documentary, Romance, Short. Binary variables representing if movie was classified as belonging to that genre. A movie can belong to more one genre. See for example the film *Ablaze* in Table 1.1.

This is **only a subset** of the data, the actual data set contains information on over 50,000 movies.

There are a total of 23 variables and 4988 films. The first few rows are given in Table 1.1. We will use this dataset to illustrate the concepts covered in this class.

1.4 Useful R package

We have written an add-on R package for the module which will allow easy access to many interesting datasets (including the movies dataset). Installing this package is straightforward, and will make loading datasets used in lectures really easy:

```
> install.packages("ncldata",
+                  repos="http://R-Forge.R-project.org",
+                  type="source")
```

When typing in code from the lecture notes, **never** include the + symbols (unless they're actually part of a calculation!). This just indicates, in the notes, that the code has 'wrapped' onto another line.

To load the package, use

```
> library(ncldata)
```

Then to load the movies dataset, for example, type:

```
> data(movies)
```

Alternatively, datasets can be downloaded directly from the course webpage/BB page (see practical work next week/week after). We will explore the movies dataset in Chapter 2.

2
Introduction to R

In this chapter we will play about with R and learn about the basics.

2.1 A simple R session

At its most basic R, can be used as a calculator, for example for multiplication and division:[1]

```
> 5*5
[1] 25
> 10.2/6
[1] 1.7
```

And more 'advanced' operations:

```
> 2^3
[1] 8
> exp(1.5)
[1] 4.481689
> log(10)
[1] 2.302585
> 4 %% 3
[1] 1
```

[1] The # symbol is used for commenting. We use comments to describe what a piece of code is doing. That way, when we look at the code in a few months/years we can figure out what is going on... For example:

```
> #Multiplication
> 5*5
[1] 25

> #Logarithms
> log(10)
[1] 2.302585
```

2.1.1 Assignment operations

In the practicals we will use assignment, i.e. `x = 5`. However, computer assignment is different from typical mathematical assignment. For example:

```
> x = 5
> x = x + 1
> x
[1] 6
```

In mathematics, $x = x + 1$ implies that $0 = 1$.

Notice that when we type `x = 5`, R doesn't display or print any output to the screen.[2] If we want to see what value has been assigned to the variable, we type `x`. An equivalent way is to surround the expression with brackets.[3] For example:

```
> (x = 2*x)
[1] 12
```

[2] Don't confuse this with R not "doing anything."

[3] In these notes, I will surround expressions with brackets so you can see what R has done.

2.1.2 Data types

R has a variety of data types:

```
> (v = TRUE)
[1] TRUE
> (w = "fred")
[1] "fred"
> (x = 5.0)
[1] 5
```

and also some "special" data types:

```
> (y = 5/0)
[1] Inf
> (z = y-y)
[1] NaN
```

Another important data type in R is `NA`. This is used to represent missing values. A list of data types is given in Table 2.1.

Type	Example 1	Example 2	Example 3	Example 4
Doubles	2	3.1242	-45.6	4e-10
Logicals/Boolean	TRUE	FALSE		
Characters	"FRED"	"x"	"Male"	"TRUE"
Infinity	Inf	5/0		

Table 2.1: Summary of some data types in R.

2.2 The R workspace

Once you create a variable, R stores that variable in memory for reuse. You can view available variables with the `ls()` command:

```
> rm(list = ls())
> library(ncldata)
> data(movies)
```

```
> ls()
[1] "movies"
```

To delete a variable in R, we use the `rm` function. For example:

```
> x = 0
> y = 1
> z = 2
> ls()
[1] "movies" "x"      "y"      "z"
> rm(x)
> ls()
[1] "movies" "y"      "z"
```

We can remove everything in the workspace using `rm(list=ls())`:

```
> rm(list=ls())
```

I would recommend running `rm(list=ls())` at the beginning of each new R session. This stops you relying on previously stored variables and makes your code more portable.

2.3 Vectors

Vectors are the most basic of all data structures, but are used in almost all R code. An R vector contains n values of the same type, where n can be zero. For example:

```
> c(0, 1, 2, 3, 4, 5)
[1] 0 1 2 3 4 5
> (my_first_vec = c(0, 1, 2, 3, 4, 5))
[1] 0 1 2 3 4 5
> (my_second_vec = c("Male", "Female", "Male"))
[1] "Male"   "Female" "Male"
```

In the above code, we create a vector of **doubles**; in line 2 we then assign the vector to the variable `my_first_vec`. We can create vectors of any data type. For example, `my_second_vec` is a vector of **characters**. In R, when we type:

```
> x = 5
> y = "Fred"
```

we have actually created a vector of doubles and characters (of length one). There are special functions in R to determine variable type:

```
> x = 5
> is.double(x)
[1] TRUE
> is.character(x)
[1] FALSE
> is.vector(x)
[1] TRUE
```

To determine the length of a vector in R, we use the `length` function:

```
> length(my_first_vec)
[1] 6
> length(my_second_vec)
[1] 3
```

To create sequences of numbers we use the `seq` command. For example:

```
> (x1 = seq(1, 6))
[1] 1 2 3 4 5 6
> (x2 = seq(-4, 4, by=2))
[1] -4 -2  0  2  4
```

Table 2.2 gives a few more useful R functions.

Command description	Example	Result
Length	`length(x)`	4
Reverse order	`rev(x)`	3,5,5,1
Sort	`sort(x)`	1,3,5,5
Sum	`sum(x)`	14
Extract unique elements	`unique(x)`	1,5,3
Indices of particular elements	`which(x==5)`	2,3

Table 2.2: Useful vector functions. In the above examples, `x = c(1,5,5,3)`. Check the associated R help for further information.

2.3.1 Vector operations

When our data is in a vector structure, we can apply standard operations to the entire vector. For example:

```
> (x = seq(-4, 4))
[1] -4 -3 -2 -1  0  1  2  3  4
> x*x
[1] 16  9  4  1  0  1  4  9 16
> x - 5
[1] -9 -8 -7 -6 -5 -4 -3 -2 -1
> x + x
[1] -8 -6 -4 -2  0  2  4  6  8
```

2.3.2 Extracting elements from vectors

R has a number of useful of methods that we can use to extract subsets of our data. For example, to pick out particular elements:

```
> my_first_vec[2]
[1] 1
> my_second_vec[2:3]
[1] "Female" "Male"
> my_first_vec[4:2]
[1] 3 2 1
```

We can also use other arguments. For example, to remove the last entry in the vector, we use the `length` function:

```
> l = length(my_first_vec)
> my_first_vec[1:(l-1)]
[1] 0 1 2 3 4
```

We determine the length of the vector using the `length` function and select particular elements using the `[·]` operator.

2.4 Logical vectors

R supports the logical elements: `TRUE` and `FALSE`. Boolean algebra tells us how to evaluate the truth of compound statements. Table 2.3 gives a summary of R operations and compares them to the notation used in MAS1604. So for example:[4]

```
> A = TRUE
> B = FALSE
> !A
[1] FALSE
> !B
[1] TRUE
> A & B
[1] FALSE
> A | B
[1] TRUE
```

[4] Read `!A` as **NOT** A.
Read `A & B` as A **AND** B.
Read `A | B` as A **OR** B.

Boolean	A	B	$\bar{A}$	$\bar{B}$	$A \cap B$	$A \cup B$
R	`A`	`B`	`!A`	`!B`	`A & B`	`A | B`
	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE

Table 2.3: Truth table for Boolean operations.

2.4.1 Using logicals for sub-setting vectors

We can construct vectors of logical operators and use them to take subsets of vectors. For example:

```
> (logic1 = c(TRUE, FALSE, TRUE, FALSE))
[1]  TRUE FALSE  TRUE FALSE
> (vec1 = seq(1, 4))
[1] 1 2 3 4
> vec1[logic1]
[1] 1 3
```

2.4.2 Relational operators

When programming it is often necessary to test relations for equality and inequality. To do this in R we use the relation operators. First let's define some variables:

```
> x = 5
> y = 7
```

To test for equality we use `==`:

```
> x == 5
[1] TRUE
> x == y
[1] FALSE
```

Similarly, to test for inequality we use `!=`:

```
> x != 5
[1] FALSE
> y != x
[1] TRUE
```

There are also commands for greater/less than:

```
> y > 6
[1] TRUE
> x >= 5
[1] TRUE
> x <= y
[1] TRUE
```

Table 2.4 gives a summary of the main commands.

WE CAN also apply these techniques to vectors. For example:

```
> (vec2 = seq(0, 10, by=2.5))
[1]  0.0  2.5  5.0  7.5 10.0
> vec2 > 3
[1] FALSE FALSE  TRUE  TRUE  TRUE
> vec2 < 9
[1]  TRUE  TRUE  TRUE  TRUE FALSE
> (vec2 > 3) & (vec2 < 9)
[1] FALSE FALSE  TRUE  TRUE FALSE
> !(vec2 > 3)
[1]  TRUE  TRUE FALSE FALSE FALSE
```

Table 2.4: Summary of R relational operators. The example is for `x = 5`.

Operator	Tests for	Example	Result
==	Equality	`x == 5`	`TRUE`
! =	Inequality	`x != 5`	`FALSE`
<	Less than	`x < 5`	`FALSE`
<=	Less or equal	`x <= 5`	`TRUE`
>	Greater	`x > 5`	`FALSE`
>=	Greater or equal	`x >= 5`	`TRUE`

Using relational operators allows us to extract subsets of data very easily. Consider the movie budgets:[5]

```
> length(Budget)
[1] 4988
```

To select movies where the budget is known, we use the following command:

```
> non_zero_b = Budget[!is.na(Budget)]
> length(non_zero_b)
[1] 3990
```

and to select movies where the movie length is greater than 60 minutes but shorter than 90 minutes:

```
> m_l = Length[Length > 60 & Length < 90]
```

[5] To extract certain variables from the movies dataframe, use the `attach` command: `library(ncldata); data(movies);attach(movies)`

2.5 Data frames

A data frame is a special kind of object. We use data frames for storing and managing data sets that have a rectangular structure. Typically the rows correspond to *cases* and the columns to *variables*. The crucial difference between a data frame and a `matrix` is that all values in a matrix must be of the same type. The next code segment constructs a simple data frame. First, we construct three vectors:

```
> age = c(24, 26, 25, 21)
> sex = c("Male", "Female", "Male", "Female")
> respond = c(TRUE, FALSE, FALSE, FALSE)
```

Then we combine them using the `data.frame` function:

```
> (df1 = data.frame(age=age, gender=sex, respond=respond))
  age gender respond
1  24   Male    TRUE
2  26 Female   FALSE
3  25   Male   FALSE
4  21 Female   FALSE
```

The data frame `df1` has three columns and four rows. Once we put our data into a data frame, then data manipulation is easier. To calculate the dimensions of a data frame we use `dim`:

```
> dim(df1)
[1] 4 3
```

To extract the first column we use square brackets:

```
> df1[ ,1]
[1] 24 26 25 21
```

Similarly, we can get the first row:

```
> df1[1, ]
  age gender respond
1  24   Male    TRUE
```

The column names are also easily manipulated:

```
> colnames(df1)
[1] "age"     "gender"  "respond"
> (colnames(df1) = c("Age", "Sex", "Respond"))
[1] "Age"     "Sex"     "Respond"
```

The `movies` dataset is actually a data frame[6]:

[6] Here, we're extracting the 3363rd row and columns 1 to 4

```
> dim(movies)
[1] 4988   23
> colnames(movies)[1:4]
[1] "Title"  "Year"   "Rating" "Votes"
> movies[3363, 1:4]
                                          Title Year Rating  Votes
3363 Star Wars: Episode I - The Phantom Menace 1999    6.5 607932
```

2.5.1 Subsets of data frames

We can also retrieve subsets from the data frame. For example, if we wanted only female responses, then:

```
> (female_only = df1$Sex=="Female")
[1] FALSE  TRUE FALSE  TRUE
> (df2 = df1[female_only, ])
  Age    Sex Respond
2  26 Female   FALSE
4  21 Female   FALSE
```

Or people 25 and over:

```
> over_25 = df1$Age>=25
> (df3 = df1[over_25, ])
  Age    Sex Respond
2  26 Female   FALSE
3  25   Male   FALSE
```

2.5.2 Example: movie data

We can select movies where the budget is greater than $100,000:

```
> m1 = movies[movies$Budget > 100000,]
> dim(m1)
[1] 4881   23
```

Or movies that cost more than $100,000 but are not R rated:

```
> m2 = movies[movies$Budget > 100000
+   & movies$mpaa != "R",]
> dim(m2)
[1] 3174   23
```

Or movies that are either PG or PG-13:

```
> m3 = movies[
+   movies$mpaa == "PG" | movies$mpaa == "PG-13",]
> dim(m3)
[1] 3229   23
```

3
Data Summaries

In this chapter we explore various options within R for summarising data. We focus primarily on some intrinsic functions within R to obtain numerical summaries of average and spread; we also consider some of the most commonly-used graphical procedures within R for summarising data.

3.1 Numerical summaries

3.1.1 Measures of average

One of the most important and widely used measures of average is the (arithmetic) sample mean:

$$\bar{x} = \frac{x_1 + x_2 + \ldots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i \ .$$

So if our data set was $\{0, 3, 2, 0\}$, then $n = 4$. Hence,

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{0+3+2+0}{4} = 1.25 \ .$$

As you will see in the following examples, the mean can be distorted if our sample contains unusually large or small values relative to most of the other values in our dataset.

The sample median can overcome the sensitivity of the mean to outliers, and so is often considered a more "robust" measure of average. It is the middle observation when the data are ranked in ascending order, and so the size of the largest or smallest values is not taken into account - only the *position* of these values is considered. Denote the ranked observations as $x_{(1)}, x_{(2)}, \ldots, x_{(n)}$. The sample median is defined as:

Remember that $x_{(n+1)/2}$ is the $(n+1)/2^{\text{th}}$ ordered observation.

$$\text{Sample median} = \begin{cases} x_{(n+1)/2}, & n \text{ odd;} \\ \frac{1}{2}x_{(n/2)} + \frac{1}{2}x_{(n/2+1)}, & n \text{ even} \ . \end{cases}$$

Although the median is more robust than the sample mean, you will see in other courses that it has less useful mathematical properties.

For our simple data set $\{0, 3, 2, 0\}$, to calculate the median we re-order

it to: $\{0,0,2,3\}$; then take the average of the middle two observations, to get 1.

The sample mode is the value which occurs with the greatest frequency. It only makes sense to calculate or use it with discrete data or qualitative data.

Warning: in R the function `mode` doesn't give you the sample mode. Use `table` instead.

3.1.2 Measures of spread

As well as knowing the location statistics of a data set, we also need to know how variable or "spread-out" our data are.

The range is easy to calculate. It is simply the largest minus the smallest[1]. In other words,

$$\text{Range } = x_{(n)} - x_{(1)} \ .$$

[1] When you get a new data set, calculating the range is useful when checking for obvious data-inputting errors.

So for our data set of $\{0,3,2,0\}$, the range is $3-0=3$. It is very useful for data checking purposes, but in general it's not very robust.

Obviously, the range can be distorted by outliers or extreme observations.

In statistics, the mean and variance are used most often. This is mainly because they have nice mathematical properties, unlike the median, say.

THE SAMPLE VARIANCE, s^2, is defined as

$$\begin{aligned} s^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2 \\ &= \frac{1}{(n-1)}\left\{\left(\sum_{i=1}^{n}x_i^2\right) - n\bar{x}^2\right\} \ . \end{aligned}$$

The second formula is often preferred for calculations. So for our data set $\{0,3,2,0\}$, we have

The divisor is $n-1$ rather than n in order to correct for the bias which occurs because we are measuring deviations from the sample mean rather than the "true" mean of the population we are sampling from - more on this in Stage 2 Statistics courses.

$$\sum_{i=1}^{4}x_i^2 = 0^2+3^2+2^2+0^2 = 13,$$

giving:

$$s^2 = \frac{1}{n-1}\left\{\left(\sum_{i=1}^{n}x_i^2\right) - n\bar{x}^2\right\} = \frac{1}{3}\left(13 - 4\times 1.25^2\right) = 2.25.$$

The sample standard deviation, s, is the square root of the sample variance, i.e. for our toy example $s=\sqrt{2.25}=1.5$.

The standard deviation is preferred as a summary measure as it is in the units of the original data. However, it is often easier from a theoretical perspective to work with variances.

The upper and lower quartiles are defined as follows:

- $Q1$: Lower quartile = $(n+1)/4^{\text{th}}$ smallest observation.
- $Q3$: Upper quartile = $3\ (n+1)/4^{\text{th}}$ smallest observation.

We can linearly interpolate between adjacent observations if necessary. The interquartile range is the difference between the third and first quartile, i.e. Then:

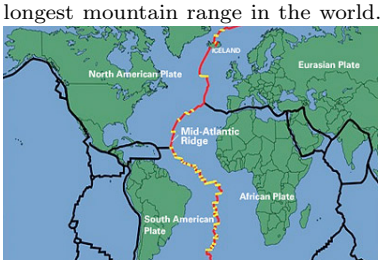
$$IQR = Q3 - Q1.$$

3.1.3 Example

The number of earth tremours recorded for five randomly chosen locations in Iceland, close to the *Mid-Atlantic ridge*[2], is recorded below:

Location	1	2	3	4	5
# tremours	7	1	14	13	20

[2] The Mid-Atlantic Ridge is a divergent tectonic plate along the floor of the Atlantic Ocean, and is part of the longest mountain range in the world.



- Calculate the mean and median number of earth tremours for this sample.

  Solution:

  The mean is:

  $$\bar{x} = \frac{1}{5}(1 + 7 + 13 + 14 + 20) = 11$$

  The median is the middle value, so we get 13.

- There was a recording error for location 5, which is closest to the Mid-Atlantic Ridge; this observation *should have* been 200, and not 20. Find new values for the mean and median, and comment.

  The mean can be distorted by unusually high or low values.

  Solution:

- Find the standard deviation and IQR for the earth tremours data, and comment.

  Solution:

3.2 Using R

R has many intrinsic (built-in) functions for basic statistical analyses. This includes the calculation of simple summary statistics. For example, for the movie data in section §1.3, we can easily use R to calculate averages and measures of spread. First, load the data:

```
> library(ncldata)
> data(movies)
> attach(movies)
```

To calculate averages, we use `mean` and `median`[3]:

```
> mean(Budget, na.rm = TRUE)
[1] 41245023
> median(Budget, na.rm = TRUE)
[1] 2.5e+07
```

[3] The argument `na.rm` can be invoked to remove missing values. Notice that the mean and median for movie budgets are substantially different... why?

We can also calculate measures of spread very easily[4]:

```
> range(Budget, na.rm = TRUE)
[1] 0.0e+00 1.9e+09
> var(Budget, na.rm = TRUE)
[1] 4.707623e+15
> sd(Budget, na.rm = TRUE)
[1] 68612118
```

[4] What do you think the `e` notation means? For example, `4.707623e+15` as in the output above?

To get the quartiles from R we use the `quantile` command[5], i.e.:

[5] Note the default in R gives you a slightly different quartile range, i.e. if you don't enter the `type=6` argument. As $n \to \infty$, the different quartile func-

```
> quantile(Budget, na.rm = TRUE, type = 6)
     0%     25%     50%     75%    100%
0.0e+00 9.0e+06 2.5e+07 5.0e+07 1.9e+09
> summary(Budget)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.      NA's
0.000e+00 9.000e+06 2.500e+07 4.125e+07 5.000e+07 1.900e+09       998
```

Table 3.1 summarises the most useful intrinsic R commands for producing numerical summary statistics.

Command	Comment	Example
`mean`	Calculates the mean of a vector	`mean(x)`
`sd`	Calculates the standard deviation of a vector	`sd(x)`
`var`	Calculates the variance of a vector	`var(x)`
`quantile`	The vector quantiles. Make sure you use `type=6`	`quantile(x, type=6)`
`range`	Calculates the vector range	`range(x)`
`summary`	Calculates various statistics, including the quartiles. **However,** it doesn't use `type=6` quartiles.	`summary(x)`

Table 3.1: Summary of R commands in this chapter.

3.3 Graphical summaries

3.3.1 Introduction

Graphical displays of data can be very useful for showing the main features of a data set. The appropriate form of graph depends on the nature of the variables being displayed and what aspects are to be shown. However it should always be borne in mind that the object is to provide a clear and truthful representation of the data, not to distort and not to impress with unnecessary "fancy" features.

3.3.2 Qualitative data: bar charts

The most useful way to display qualitative data is usually with a bar chart. The length of each bar is proportional to the frequency of the corresponding value of the variable in the sample of data. Note that the widths of the bars should be equal to avoid giving a false impression, as should the width *between* bars[6].

[6] Type `colours()` into R to get a list of colours.

Figure 3.1 shows the breakdown in MPAA ratings for the movies dataset. To create a bar chart in R we use the `table` command...

```
> table(movies$mpaa)
NC-17    PG PG-13     R
   16   526   989  3316
```

... inside the `barplot` function:

```
> barplot(table(movies$mpaa), xlab="MPAA Rating",
+         ylab="Frequency", border = "black",
+         col="mistyrose")
```

Remember to load the data first!

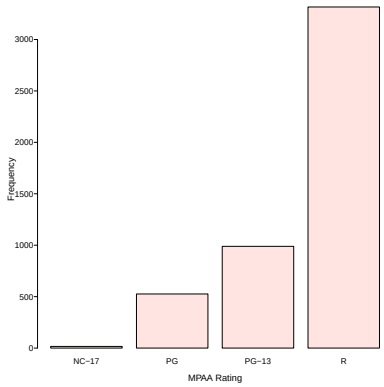


Figure 3.1: Barchart of the mpaa ratings for 4847 films.

3.3.3 Histograms

To represent the distribution of a sample of values from a continuous variable we can use a histogram. The range of values of the variable is divided into intervals, known as *classes* or *bins*, and the frequencies in bins are represented by columns. As the variable is continuous, there are no gaps between neighbouring columns, unlike a bar chart[7]. Note also that, strictly speaking, it is the *area* of the column which is proportional to the frequency, not the height. The reason for this is that columns need not be of the same width. Computer software tends to use bins of the same width - giving standard "frequency histograms" when the y-axis represents frequency[8]. However, this default can be overridden in R if you really want to do this.

[7] Unless, of course, a particular bin has zero frequency

[8] Rather than frequency *density* - important when the classes are *not* all the same width.

Rule	Formula	R command	Comment
Sturges'	$k_{ST} = \lceil \log_2 n + 1 \rceil$	Default	Tends not to be very good for $n > 30$.
Scott's	$h_{SC} = 3.49 \times s \times n^{-1/3}$	`breaks = 'Scott'`	
Freedman-Diaconis	$h_{FR} = 2 \times IQR(x) \times n^{-1/3}$	`breaks = 'FD'`	When the distribution is symmetric, this is very similar to Scott's rule.

Table 3.2: Standard bin width rules in R.

Figure 3.2 shows histograms of the film budgets. When displaying relative frequencies (or proportions; often termed "density histograms"), we can easily work out the height of each bin using:

$$\text{Height} = \frac{\text{frequency}}{n \times \text{ Bin-width}} .$$

When we display relative frequencies, the y-axis is labelled with "density" and the area under the histogram is one. Bin widths should be chosen so that you get a good idea of the distribution of the data, without being swamped by random variation.

To generate Figure 3.2 in R we use the following commands; notice the argument `freq=FALSE` to switch from frequencies to relative frequencies.

```
> hist(movies$Budget, col="grey",
+      main="Mean film budget",
+      freq=FALSE, xlab="Budget ($)")
```

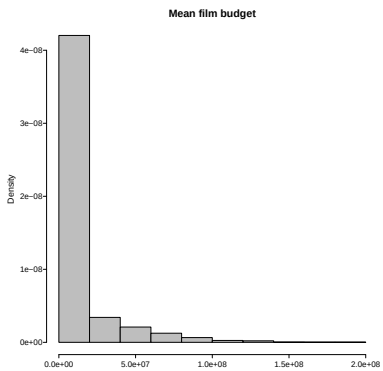


Figure 3.2: Histogram of movie budgets.

How many bins should we have?

Let:

- n: the sample size;
- k: the number of bins in the histogram;
- h: the bin-width.

Then the number of bins we use to construct a histogram is:

$$k = \left\lceil \frac{\max(x) - \min(x)}{h} \right\rceil \tag{3.1}$$

where $\lceil \cdot \rceil$ is the ceiling function.

Table 3.2 gives a summary of the different rules. Briefly, R uses Sturges' rule by default, which isn't always that good. Notice that Sturges' rule gives you k, but the other rules give you the bin width. Typically, it is best to go with Scotts' rule or the Freedman-Diaconis rule. You should experiment with the different rules in R to find the histogram which you think best displays your data!

To use the different rules, we use the breaks argument. For example, the following piece of code:

```
> hist(movies$Budget, col="wheat",
+      main="The FD rule",
+      freq=FALSE, xlab="Budget ($)", breaks="FD")
```

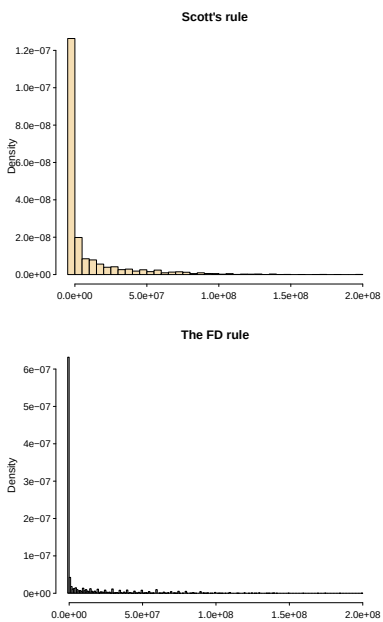


Figure 3.3: Histogram of (a) movie budgets using Scott's rule and (b) the Freedman-Diaconis rule. Compare these histograms to Figure 3.2 which uses Sturges' rule (default).

uses the Freedman-Diaconis rule - see Figure 3.3b. When we compare Figure 3.3a with Figure 3.2, we have used many more bins, which we might argue has resulted in a *better* histogram; however, it is clear that Figure 3.3b uses too *many* bins! In practical 2 you will get a chance to experiment with the different rules yourself.

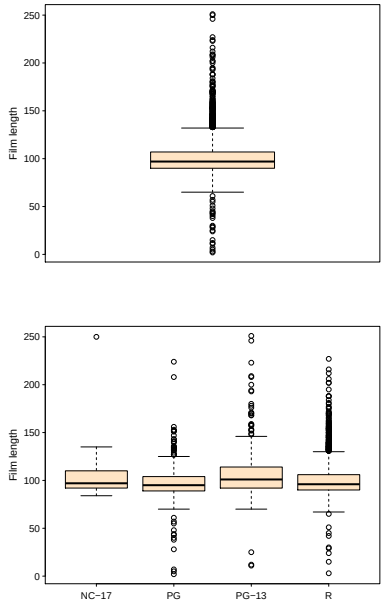


Figure 3.4: Box and whisker plots of (a) film length (b) film length split according to the mpaa rating.

3.3.4 Box and whisker plots

A box and whisker plot, often referred to simply as a boxplot, is another way to represent continuous data. This kind of plot is particularly useful for comparing two or more groups, by placing the boxplots side-by-side. Figures 3.4 and 3.5 shows boxplots of film length for different categories of film.

The central bar in the "box" is the sample *median*. The top and bottom of the box represent the upper and lower sample *quartiles*, respectively. Just as the median represents the 50% point of the data, the lower and upper quartiles represent the 25% and 75% points respectively.

The lower whisker is drawn from the lower end of the box to the smallest value that is no smaller than $1.5IQR$ below the lower quartile. Similarly, the upper whisker is drawn from the middle of the upper end of the box to the largest value that is no larger than $1.5IQR$ above the upper quantile. Points outside the whiskers are classified as outliers.

To do this in R we use the following commands:

```
> par(mfrow=c(2, 1))
> boxplot(movies$Length, ylab="Film length",
+         col="bisque")
> boxplot(movies$Length ~ movies$mpaa, ylab="Film length",
+         col="bisque")
> boxplot(movies$Length ~ movies$mpaa + movies$Romance,
+         ylab="Film length")
```

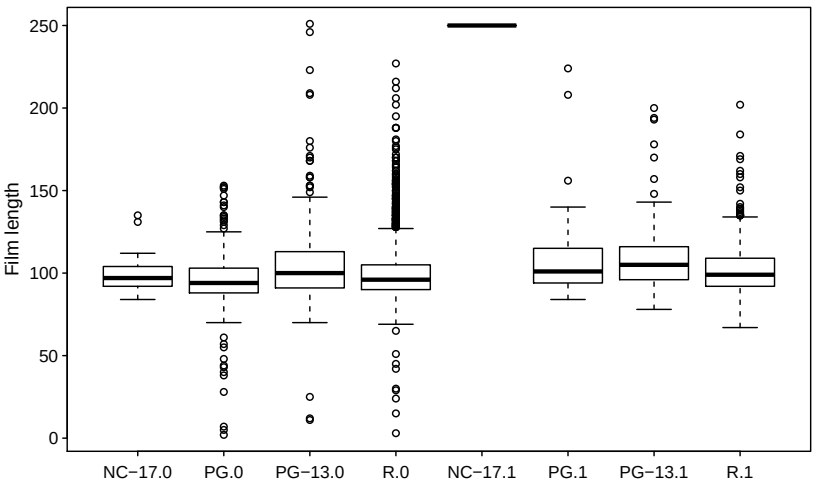


Figure 3.5: Box and whisker plots for movie length split according to MPAA rating and whether the film was a romance.

Command	Comment	Example
`table`	Contingency table	`table(x)`
`barplot`	Generate a bar chart	`barplot(table(x))`
`hist`	Histogram	`hist`
`plot`	Scatter plot. See Practical 2.	`plot(x, y)`
`points`	Add points to a plot. See Practical 2.	`points(x,y)`
`lines`	Add lines to a plot. See Practical 2.	`lines(x,y)`
`boxplot`	Box and whiskers plot	`boxplot(x)`

Table 3.3: Summary of R commands in this chapter.

4
Control Statements and Functions

4.1 Functions

A very powerful aspect of R is that it is relatively easy to write your own functions. Functions can take inputs (or *arguments*) and return a *single* value. Let's look at some simple functions.

4.1.1 Basic functions

This function takes in a single argument x and returns x^2:

```
> Fun1 = function(x) {
+   return (x*x)
+ }
```

The key elements in the function call are:

- The word `function`;
- The brackets () which enclose the **argument** list.
- A sequence of statements in curly braces { }.
- A `return` statement.

We *call* `Fun1` in the following manner:

```
> Fun1(5)
[1] 25
> y = Fun1(10)
> y
[1] 100
> z = c(1, 2, 3, 4)
> Fun1(z)
[1]  1  4  9 16
```

Of course, the old saying 'garbage in, garbage out' is true:

```
> Fun1()
Error in Fun1() :  argument "x" is missing, no default
```

```
> Fun1("5")
Error in x * x :  non-numeric argument to binary operator
```

The error messages give you an idea of what went wrong.

Other variations to this simple function are:

```
> Fun2 = function(x=1) {
+   return (x*x)
+ }
> Fun2()
[1] 1
> Fun2(4)
[1] 16
> Fun3 = function(x, y) {
+   return (x*y)
+ }
> Fun3(3, 4)
[1] 12
```

4.1.2 A more useful function

Here the function below takes in a vector, plots a histogram and returns a vector containing the mean and standard deviation:

```
> Investigate = function(values) {
+   hist(values)
+   m_std = c(mean(values), sd(values))
+   return(m_std)
+ }
```

Once we have created our function, we can put it to good use[1]:

```
> Investigate(movies$Rating)
[1] 5.522715 1.451864
```

[1] Obviously, a histogram would also be created – it's just not shown here. See Section 3.3.3 for examples of histograms.

4.1.3 Variable scope

When we call a function, R first looks for *local* variables, then *global* variables. For example, `Fun4` uses a global variable:

```
> blob = 5
> Fun4 = function() {
+   return(blob)
+ }
> Fun4()
[1] 5
```

R scoping rules are actually a bit more complicated than described below. R uses something called *lexical* scope, but this doesn't affect us.

However, in `Fun5`, we use a local variable:

```
> Fun5 = function() {
+   blob = 6
+   return(blob)
+ }
> Fun5()
[1] 6
> blob
[1] 5
```

As a general rule, functions should only use **local** variables[2].

[2] This makes your code more portable and less likely to have bugs.

4.2 Conditionals

Conditional statements are features of a programming language which perform different computations or actions depending on whether a condition evaluates to `TRUE` or `FALSE`. They are used in almost all computer programs.

4.2.1 *If* statements

The basic structure of an `if` statement is:

```
> if(expr) {
+   ##do something
+ }
```

where `expr` is evaluated to be either `TRUE` or `FALSE`. The following example illustrates `if` statements in R:

```
> x = 5
> y = 5
> if(x<5) {
+   y = 0
+ }
> y
[1] 5
```

In this code chunk, `x < 5` evaluates to be `FALSE` so the following brackets are not evaluated. Similarly, we can test for greater than:

```
> x = 5
> y = 5
> if(x > 0) {
+   y = 0
+ }
> y
[1] 0
```

Here `x > 0` evaluates to be `TRUE` so, `y` is set equal to 0. If we wanted to test for equality with zero, then we would use `==`.

4.2.2 *If else* statements

We can link together a number of `if` statements[3]:

```
> x = 0
> if(x > 0) {
+   cat("x is greater than zero")
+ } else if(x < 0) {
+   cat("x is less than zero")
+ } else {
+   cat("x must be zero!")
+   cat("\n")
+ }
x must be zero!
```

The final `else` is optional.

[3] What do you think the `cat` command does?

We can also use `if` statements in functions, for example to check that our data is negative we can construct the following function:

```
> IsNegative = function(value) {
+   I = FALSE
+   if(value < 0) {
+     I = TRUE
+   }
+   return(I)
+ }
> IsNegative(1)
[1] FALSE
> IsNegative(-5.6)
[1] TRUE
```

A more sophisticated function could be:

```
> IsGreaterThan = function(value1, value2) {
+   is_greater_than = FALSE
+   if(value1 > value2) {
+     is_greater_than = TRUE
+   }
+   return(is_greater_than)
+ }
```

Which we can then call:

```
> IsGreaterThan(-5, -6)
[1] TRUE
> IsGreaterThan(10, 10)
[1] FALSE
```

4.3 Control statements

At times we would like to perform some operation on a vector or a data frame. Often R has built-in functions that will do this for you, e.g. `mean`, `sd`,... Other times we have to write our own functions. For example, suppose we want to calculate $\sum_{i=1}^{10} i^2$.
In R we can use a `for` loop:

```
> x = 0
> for(i in 1:10) {
+   x = x + i^2
+ }
> x
[1] 385
```

Or perhaps $\sum_{j=-5}^{-1} e^j/j^2$, then:

```
> total = 0
> for(j in -5:-1) {
+   total = total + exp(j)/j^2
+ }
> total
[1] 0.4086594
```

A more tricky example: Calculate $\sum e^k/k^2$, for $k = 3, 6, 9, \ldots, 21$:

```
> total = 0
> for(i in 1:7) {
+   k = i*3
+   total = total + exp(k)/k^2
+ }
> total
[1] 3208939
```

Exercise: Using the inbuilt R function `sum`, calculate the above summations without using `for` loops.

4.4 *Putting it all together*

Rather than have to constantly write R code to solve the summations in §4.3 we can create a function to solve the general form:

$$\sum_{i=i_s}^{i_e} \frac{e^i}{i^2} \quad \text{for } i = i_s,\ i_s + j,\ i_s + 2j,\ \ldots, i_e\ .$$

Can you see what's wrong with the function `Summation1`? There's a prize for anyone who can spot this – and write a function which works all the time!

So in R we have:

```
> Summation1 = function(i_s, i_e, j) {
+   total = 0
+   for(i in 1:(i_e/j)) {
+     k = i*j
+     total = total + exp(k)/k^2
+   }
+   return(total)
+ }
> Summation1(3, 21, 3)
[1] 3208939
```

4.5 *The apply family*

R has been designed with manipulating data in mind. Due to this, there are two important functions that are unique to R.

Probably not unique, but not common in other programming languages.

4.5.1 *The `apply` function*

We use the `apply` function when we want to *apply* the same function to every row or column of a data frame. For example, suppose we have a data frame with three columns:

```
> (df4 = data.frame(c1 = 1:4, c2 = 4:7, c3 = 2:5))
  c1 c2 c3
1  1  4  2
2  2  5  3
3  3  6  4
4  4  7  5
```

The `apply` function takes (at least) three arguments. The first argument is the data frame, the second the number `1` or `2` indicating row or column

(respectively) and the third a function to apply to each row or column. So:

```
> apply(df4, 1, mean)
[1] 2.333333 3.333333 4.333333 5.333333
```

calculates the mean value of every row, while:

```
> apply(df4, 2, sd)
      c1       c2       c3
1.290994 1.290994 1.290994
```

calculates the standard deviation of every column.

Suppose one of the columns was non-numeric:

```
> (df5 = data.frame(c1 = 1:3, c2 = 4:6,
+    c3 = LETTERS[1:3]))
  c1 c2 c3
1  1  4  A
2  2  5  B
3  3  6  C
```

then taking the mean doesn't really make sense:

```
> apply(df5, 1, mean)
[1] NA NA NA
```

Instead, we remove the column, then calculate the mean:

```
> apply(df5[ ,1:2], 1, mean)
[1] 2.5 3.5 4.5
```

4.5.2 The `tapply` function

The function `tapply` is very useful, but at first glance can be tricky to understand. It's best described using an example:

```
> tapply(movies$Length, movies$mpaa, mean)
    NC-17        PG     PG-13         R
110.18750  97.23384 104.97877 100.18818
```

In the above code, we have calculated the average movie length conditional on its MPAA rating. So the average length of a PG movie is 97 minutes and the average NC-17 movie length is 110mins. With `tapply` we can do we very interesting things. For example, in the next piece of code, we plot the average movie length conditional on it's rating:

```
> tapply(movies$Length, movies$Rating, mean)[1:6]
   1  1.2  1.3  1.4  1.5  1.6
85.5 93.0 87.5 85.0 67.0 86.0
> rating_by_len = tapply(movies$Length, movies$Rating,
+   mean)
> plot(names(rating_by_len), rating_by_len)
```

Imagine trying to produce Figure 4.1 in Excel!

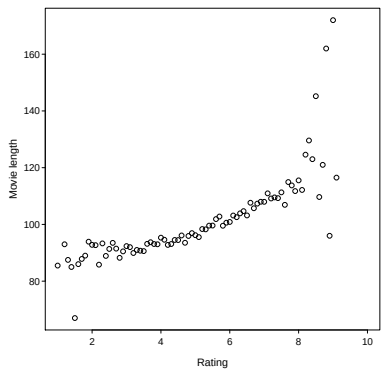


Figure 4.1: Plot of mean movie length conditional on it's rating.

4.6 Help

R has a very good help system. If you need information about a particular function – say `plot` – then typing `?plot` in an R terminal will bring up the associated help page.

The internet is another very good source of R help. Unfortunately, using Google isn't particularly useful since the letter "R" appears on most web pages! However, you can use

http://www.rseek.org/

Using this search engine limits searches to R web-pages.

Summary of R commands

Command	Comment
`for`	A for loop. See §4.3
`if` or `else`	A conditional statement. See §4.2.
`function`	An R function constructor. See §4.1
`cat`	Print command.
`apply` or `tapply`	See §4.5

Table 4.1: Summary of R commands in this chapter.

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 'West Ham vs Man City'[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 use the techniques described in this section to simulate from the Poisson and binomial distributions. We will see how to simulate from continuous distributions in MAS2602 next year.

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.

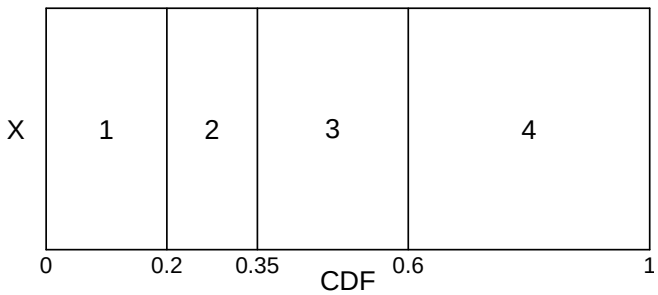


Figure 5.1: Simulating discrete random numbers.

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

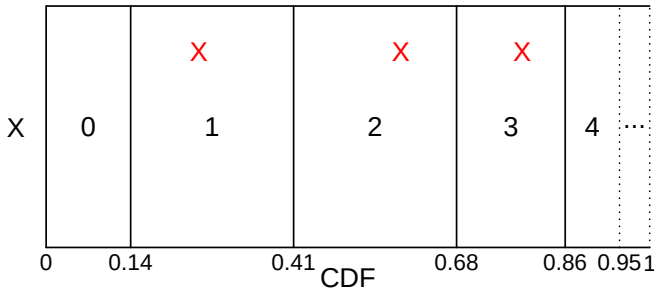


Figure 5.2: Simulating discrete random numbers from the Poisson distribution.

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!

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

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

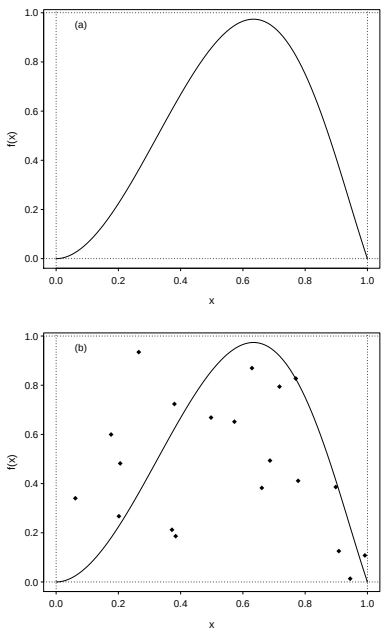


Figure 6.1: (a) An example function. (b) Twenty points randomly placed on a *simulation grid*.

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)
```

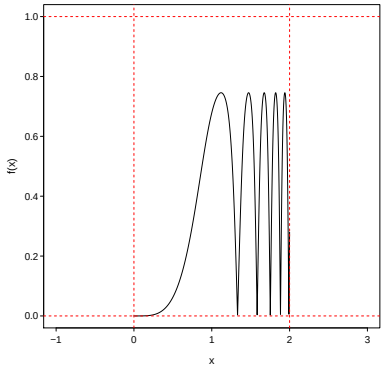


Figure 6.2: A plot of $|\sin\{\sin[\sin(x^4)]\}|$ with the sampling region.

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 $(100, 500)$.

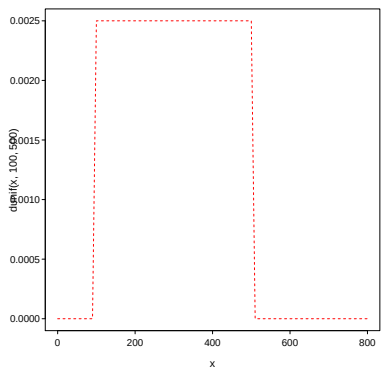


Figure 6.3: A plot of the $U(100, 500)$ population.

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
```

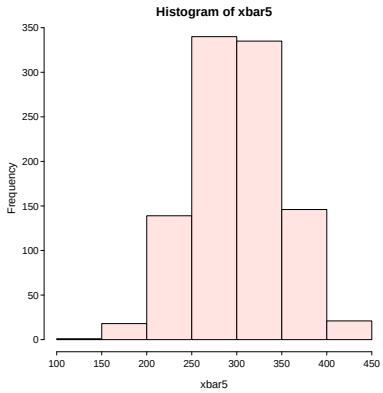


Figure 6.4: Sampling distribution for $\bar{x}$ with $n = 5$.

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
```

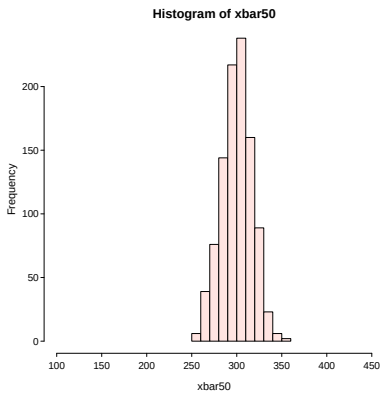


Figure 6.5: Sampling distribution for $\bar{x}$ with $n = 50$.

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.

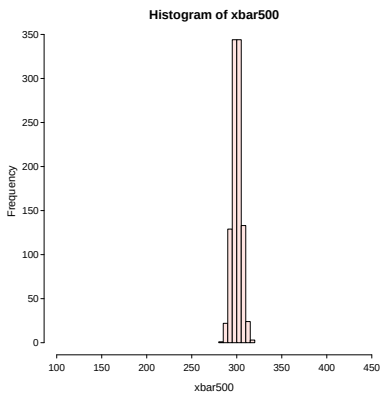


Figure 6.6: Sampling distribution for $\bar{x}$ with $n = 500$.

```
> 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