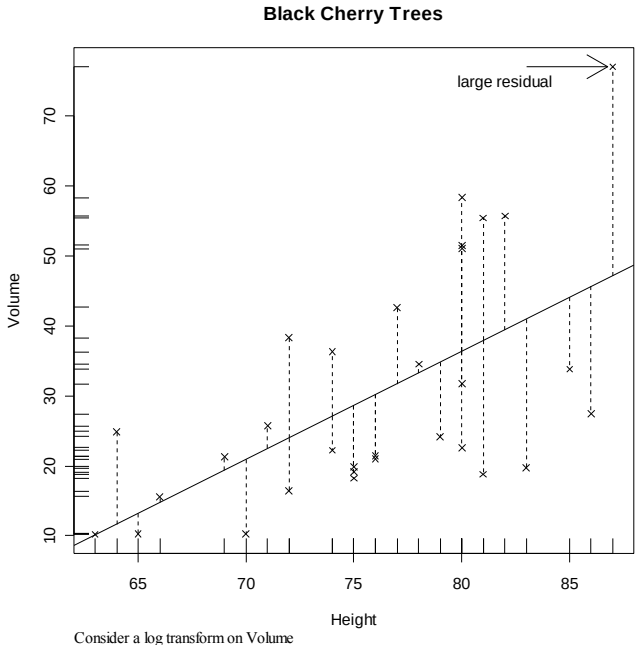
The R Guide

Version 2.5

W. J. Owen
Department of Mathematics and Computer Science
University of Richmond




"Humans are good, she knew, at discerning subtle patterns that are really there, but equally so at imagining them when they are altogether absent." -- Carl Sagan (1985) *Contact*

Preface

This document is an introduction to the program R. Although it could be a companion manuscript for an introductory statistics course, it is designed to be used in a course on mathematical statistics.

The purpose of this document is to introduce many of the basic concepts and nuances of the language so that users can avoid a slow learning curve. The best way to use this manual is to use a "learn by doing" approach. Try the examples and exercises in the Chapters and aim to understand what works and what doesn't. In addition, some topics and special features of R have been added for the interested reader and to facilitate the subject matter.

The most up-to-date version of this manuscript can be found at http://www.mathcs.richmond.edu/~wowen/TheRGuide.pdf.

Font Conventions

This document is typed using Times New Roman font. However, when R code is presented, referenced, or when R output is given, we use **`10 point Bold Courier New Font`**.

Acknowledgements

Much of this document started from class handouts. Those were augmented by including new material and prose to add continuity between the topics and to expand on various themes. We thank the large group of R contributors and other existing sources of documentation where some inspiration and ideas used in this document were borrowed. We also acknowledge the R Core team for their hard work with the software.

Contents

1. Overview and History

Functions and quantities introduced in this Chapter: **`apropos(), c(), FALSE or F, help(), log(), ls(), matrix(), q(), rm(), TRUE or T`**

1.1 What is R?

R is an integrated suite of software facilities for data manipulation, simulation, calculation and graphical display. It handles and analyzes data very effectively and it contains a suite of operators for calculations on arrays and matrices. In addition, it has the graphical capabilities for very sophisticated graphs and data displays. Finally, it is an elegant, object-oriented programming language.

R is an independent, open-source, and **free** implementation of the S programming language. Today, the commercial product is called S-PLUS and it is distributed by the Insightful Corporation. The S language, which was written in the mid-1970s, was a product of Bell Labs (of AT&T and now Lucent Technologies) and was originally a program for the Unix operating system. R is available in Windows and Macintosh versions, as well as in various flavors of Unix and Linux. Although there are some minor differences between R and S-PLUS (mostly in the graphical user interface), they are essentially identical.

The R project was started by Robert Gentleman and Ross Ihaka (that's where the name "R" is derived) from the Statistics Department in the University of Auckland in 1995. The software has quickly gained a widespread audience. It is currently maintained by the R Core development team – a hard-working, international group of *volunteer* developers. The R project web page is

```
http://www.r-project.org
```

This is the main site for information on R. Here, you can find information on obtaining the software, get documentation, read FAQs, etc. For downloading the software directly, you can visit the Comprehensive R Archive Network (CRAN) in the U.S. at

```
http://cran.us.r-project.org/
```

New versions of the software are released periodically.

1.2 Starting and Quitting R

The easiest way to use R is in an interactive manner via the command line. After the software is installed on a Windows or Macintosh machine, you simply double click the R icon (in Unix/Linux, type R from the command prompt). When R is started, the program's "*Gui*" (graphical user interface) window appears. Under the opening message in the R Console is the **>** ("greater than") prompt. For the most part, statements in R are typed directly into the R Console window.

Once R has started, you should be greeted with a command line similar to

```
R version 2.9.1 (2009-06-26)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

At the **>** prompt, you tell R what you want it to do. You give R a command and R does the work and gives the answer. If your command is too long to fit on a line or if you submit an incomplete command, a "**+**" is used for the continuation prompt.

To quit R, type **q()** or use the Exit option in the File menu.

1.3 A Simple Example: the c() Function and the Assignment Operator

A useful command in R for entering small data sets is the **c()** function. This function *combines* terms together. For example, suppose the following represents eight tosses of a fair die:

```
2 5 1 6 5 5 4 1
```

To enter this into an R session, we type

```
> dieroll <- c(2,5,1,6,5,5,4,1)
> dieroll
[1] 2 5 1 6 5 5 4 1
>
```

Notice a few things:

- We assigned the values to a variable called **dieroll.** R is case sensitive, so you could have another variable called **DiEroLL** and it would be distinct. The name of a variable can contain most combination of letters, numbers, and periods (.). (Obviously, a variable can't be named with all numbers, though.)

- The assignment operator is "**<-**"; to be specific, this is composed of a **<** ("less than") and a **-** ("minus" or "dash") typed together. It is usually read as "gets" – the variable **dieroll** *gets* the value **c(2,5,1,6,5,5,4,1)**. Alternatively, as of R version 1.4.0, you can use "**=**" as the assignment operator.
- The value of **dieroll** doesn't automatically print out. But, it does when we type just the name on the input line as seen above.
- The value of **dieroll** is prefaced with a **[1]**. This indicates that the value is a vector (more on this later).

When entering commands in R, you can save yourself a lot of typing when you learn to use the arrow keys effectively. Each command you submit is stored in the History and the up arrow (↑) will navigate backwards along this history and the down arrow (↓) forwards. The left (←) and right arrow (→) keys move backwards and forwards along the command line. These keys combined with the mouse for cutting/pasting can make it very easy to edit and execute previous commands.

1.4 The Workspace

All variables or "objects" created in R are stored in what's called the *workspace*. To see what variables are in the workspace, you can use the function **ls()** to list them (this function doesn't need any argument between the parentheses). Currently, we only have:

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

If we define a new variable – a simple function of the variable **dieroll** – it will be added to the workspace:

```
> newdieroll <- dieroll/2     # divide every element by two
> newdieroll
[1] 1.0 2.5 0.5 3.0 2.5 2.5 2.0 0.5
> ls()
[1] "dieroll"    "newdieroll"
```

Notice a few more things:

- The new variable **newdieroll** has been assigned the value of **dieroll** divided by 2 – more about algebraic expressions is given in the next session.
- You can add a comment to a command line by beginning it with the **#** character. R ignores everything on an input line after a **#**.

To remove objects from the workspace (you'll want to do this occasionally when your workspace gets too cluttered), use the **rm()** function:

```
> rm(newdieroll)     # this was a silly variable anyway
> ls()
[1] "dieroll"
```

In Windows, you can clear the entire workspace via the "Remove all objects" option under the "Misc" menu. However, this is dangerous – more likely than not you will want to keep some things and delete others.

When exiting R, the software asks if you would like to save your workspace image. If you click yes, all objects (both new ones created in the current session and others from earlier sessions) will be available during your next session. If you click no, all new objects will be lost and the workspace will be restored to the last time the image was saved. **Get in the habit of saving your work – it will probably help you in the future.**

1.5 Getting Help

There is text help available from within R using the function **`help()`** or the **`?`** character typed before a command. **If you have questions about any function in this manual, see the corresponding help file.** For example, suppose you would like to learn more about the function **`log()`** in R. The following two commands result in the same thing:

```
> help(log)
> ?log
```

In a Windows or Macintosh system, a *Help Window* opens with the following:

```
log                     package:base                     R Documentation

Logarithms and Exponentials

Description:

  `log' computes natural logarithms, `log10' computes common (i.e.,
  base 10) logarithms, and `log2' computes binary (i.e., base 2)
  logarithms. The general form `logb(x, base)' computes logarithms
  with base `base' (`log10' and `log2' are only special cases).
. . .          (skipped material)
Usage:

  log(x, base = exp(1))
  logb(x, base = exp(1))
  log10(x)
  log2(x)
. . .
Arguments:

       x: a numeric or complex vector.

    base: positive number.  The base with respect to which
          logarithms are computed.  Defaults to e=`exp(1)'.

Value:

  A vector of the same length as `x' containing the transformed
  values.  `log(0)' gives `-Inf' (when available).
. . .
```

So, we see that the **log()** function in R is the logarithm function from mathematics. This function takes two arguments: "**x**" is the variable or object that will be taken the logarithm of and "**base**" defines which logarithm is calculated. Note that base is defaulted to e = 2.718281828.., which is the natural logarithm. We also see that there are other associated functions, namely **log10()** and **log2()** for the calculation of base 10 and 2 (respectively) logarithms. Some examples:

```
> log(100)
[1] 4.60517
> log2(16)     # same as log(16,base=2) or just log(16,2)
[1] 4
> log(1000,base=10)     # same as log10(1000)
[1] 3
>
```

Due to the object oriented nature of R, we can also use the **log()** function to calculate the logarithm of numerical vectors and matrices:

```
> log2(c(1,2,3,4))      # log base 2 of the vector (1,2,3,4)
[1] 0.000000 1.000000 1.584963 2.000000
>
```

Help can also be accessed from the menu on the R Console. This includes both the text help and help that you can access via a web browser. You can also perform a keyword search with the function **apropos()**. As an example, to find all functions in R that contain the string **norm**, type:

```
> apropos("norm")
 [1] "dlnorm"    "dnorm"     "plnorm"          "pnorm"     "qlnorm"
 [6] "qnorm"     "qqnorm"    "qqnorm.default" "rlnorm"    "rnorm"
>
```

Note that we put the keyword in double quotes, but single quotes (' ') will also work.

1.6 More on Functions in R

We have already seen a few functions at this point, but R has an incredible number of functions that are built into the software, and you even have the ability to write your own (see Chapter 8). Most functions will return something, and functions usually require one or more input values. In order to understand how to generally use functions in R, let's consider the function **matrix()**. A call to the help file gives the following:

```
Matrices

Description:

  'matrix' creates a matrix from the given set of values.
Usage:

   matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE)
```

```
Arguments:

   data:  the data vector
   nrow:  the desired number of rows
   ncol:  the desired number of columns
   byrow: logical. If `FALSE' the matrix is filled by columns,
          otherwise the matrix is filled by rows.
   ...
```

So, we see that this is a function that takes vectors and turns them into matrix objects. There are 4 arguments for this function, and they specify the entries and the size of the matrix object to be created. The argument **byrow** is set to be either **TRUE** or **FALSE** (or **T** or **F** – either are allowed for logicals) to specify how the values are filled in the matrix.

Often arguments for functions will have *default* values, and we see that all of the arguments in the **matrix()** function do. So, the call

```
> matrix()
```

will return a matrix that has one row, one column, with the single entry **NA** (missing or "not available"). However, the following is more interesting:

```
> a <- c(1,2,3,4,5,6,7,8)
> A <- matrix(a,nrow=2,ncol=4, byrow=FALSE)  # a is different from A
> A
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
>
```

Note that we could have left off the **byrow=FALSE** argument, since this is the default value. In addition, since there is a specified ordering to the arguments in the function, we also could have typed

```
> A <- matrix(a,2,4)
```

to get the same result. For the most part, however, it is best to include the argument names in a function call (especially when you aren't using the default values) so that you don't confuse yourself. We will learn more about this function in the next chapter.

1.7 Printing and Saving Your Work

You can print directly from the **R Console** by selecting "Print…" in the File menu, but this will capture everything (including errors) from your session. Alternatively, you can copy what you need and paste it into a word processor or text editor (suggestion: use **Courier font** so that the formatting is identical to the **R Console**). In addition, you can save everything in the **R Console** by using the "Save to File…" command.

1.8 Other Sources of Reference

It would be impossible to describe all of R in a document of manageable size. But, there are a number of tutorials, manuals, and books that can help with learning to use R. Happily, like the program itself, much of what you can find is free. Here are some examples of documentation that are available:

- **The R program:** From the Help menu you can access the manuals that come with the software. These are written by the R core development team. Some are very lengthy and specific, but the manual "*An Introduction to R*" is a good source of useful information.

- **Free Documentation:** The CRAN website has several user contributed documents in several languages. These include:

  R for Beginners by Emmanuel Paradis (76 pages). A good overview of the software with some nice descriptions of the graphical capabilities of R. The author assumes that the reader knows some statistical methods.

  R reference card by Tom Short (4 pages). This is a great desk companion when working with R.

- **Books:** These you have to buy, but they are excellent! Some examples:

  Introductory Statistics with R by Peter Dalgaard, Springer-Verlag (2002). Peter is a member of the R Core team and this book is a fantastic reference that includes both elementary and some advanced statistical methods in R.

  Modern Applied Statistics with S, 4th Ed. by W.N. Venable and B.D. Ripley, Springer-Verlag (2002). The authoritative guide to the S programming language for advanced statistical methods.

1.9 Exercises

1. Use the help system to find information on the R functions **`mean`** and **`median`**.
2. Get a list of all the functions in R that contains the string **`test`**.
3. Create the vector **`info`** that contains your age, height (in inches/cm), and phone number.
4. Create the matrix **`Ident`** defined as a 3x3 identity matrix.
5. Save your work from this session in the file **`1stR.txt`**.

2. My Big Fat Greek[1] Calculator

Functions, operators, and constants introduced in this Chapter: **+, -, *, /, ^, %*%, abs(), as.matrix(), choose(), cos(), cumprod(), cumsum(), det(), diff(), dim(), eigen(), exp(), factorial(), gamma(), length(), pi, prod(), sin(), solve(), sort(), sqrt(), sum(), t(), tan()**.

2.1 Basic Math

One of the simplest (but very useful) ways to use R is as a powerful number cruncher. By that I mean using R to perform standard mathematical calculations. The R language includes the usual arithmetic operations: **+,-,*,/,^**. Some examples:

```
> 2+3
[1] 5
> 3/2
[1] 1.5
> 2^3             # this also can be written as 2**3
[1] 8
> 4^2-3*2         # this is simply 16 - 6
[1] 10
> (56-14)/6 – 4*7*10/(5^2-5)  # this is more complicated
[1] -7
```

Other standard functions that are found on most calculators are available in R:

Name	**Operation**
sqrt()	square root
abs()	absolute value
sin(), **cos()**, **tan()**	trig functions (radians) – type **?Trig** for others
pi	the number π = 3.1415926..
exp(), **log()**	exponential and logarithm
gamma()	Euler's gamma function
factorial()	factorial function
choose()	combination

```
> sqrt(2)
[1] 1.414214
> abs(2-4)
[1] 2
> cos(4*pi)
[1] 1
> log(0)          # not defined
[1] -Inf
> factorial(6)    # 6!
[1] 720
> choose(52,5)    # this is 52!/(47!*5!)
[1] 2598960
```

[1] Ahem. The Greek letters Σ and Π are used to denote sums and products, respectively. Signomi.

2.2 Vector Arithmetic

Vectors can be manipulated in a similar manner to scalars by using the same functions introduced in the last section. (However, one must be careful when adding or subtracting vectors of different lengths or some unexpected results may occur.) Some examples of such operations are:

```
> x <- c(1,2,3,4)
> y <- c(5,6,7,8)
> x*y
[1]  5 12 21 32
> y/x
[1] 5.000000 3.000000 2.333333 2.000000
> y-x
[1] 4 4 4 4
> x^y
[1]     1    64  2187 65536
> cos(x*pi) + cos(y*pi)
[1] -2  2 -2  2
>
```

Other useful functions that pertain to vectors include:

Name	Operation
`length()`	returns the number of entries in a vector
`sum()`	calculates the arithmetic sum of all values in the vector
`prod()`	calculates the product of all values in the vector
`cumsum(), cumprod()`	cumulative sums and products
`sort()`	sort a vector
`diff()`	computes suitably lagged (default is 1) differences

Some examples using these functions:

```
> s <- c(1,1,3,4,7,11)
> length(s)
[1] 6
> sum(s)       # 1+1+3+4+7+11
[1] 27
> prod(s)      # 1*1*3*4*7*11
[1] 924
> cumsum(s)
[1]  1  2  5  9 16 27
> diff(s)           # 1-1, 3-1, 4-3, 7-4, 11-7
[1] 0 2 1 3 4
> diff(s, lag = 2)   # 3-1, 4-1, 7-3, 11-4
[1] 2 3 4 7
```

2.3 Matrix Operations

Among the many powerful features of R is its ability to perform matrix operations. As we have seen in the last chapter, you can create matrix objects from vectors of numbers using the **`matrix()`** command:

```
> a <- c(1,2,3,4,5,6,7,8,9,10)
> A <- matrix(a, nrow = 5, ncol = 2)   # fill in by column
> A
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10

> B <- matrix(a, nrow = 5, ncol = 2, byrow = TRUE) # fill in by row
> B
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10

> C <- matrix(a, nrow = 2, ncol = 5, byrow = TRUE)
> C
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
>
```

Matrix operations (multiplication, transpose, etc.) can easily be performed in R using a few simple functions like:

Name	**Operation**
`dim()`	dimension of the matrix (number of rows and columns)
`as.matrix()`	used to coerce an argument into a matrix object
`%*%`	matrix multiplication
`t()`	matrix transpose
`det()`	determinant of a square matrix
`solve()`	matrix inverse; also solves a system of linear equations
`eigen()`	computes eigenvalues and eigenvectors

Using the matrices **`A`**, **`B`**, and **`C`** just created, we can have some linear algebra fun using the above functions:

```
> t(C)            # this is the same as A!!
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10

> B%*%C
     [,1] [,2] [,3] [,4] [,5]
[1,]   13   16   19   22   25
[2,]   27   34   41   48   55
[3,]   41   52   63   74   85
[4,]   55   70   85  100  115
[5,]   69   88  107  126  145

> D <- C%*%B
> D
     [,1] [,2]
[1,]   95  110
[2,]  220  260

> det(D)
[1] 500

> solve(D)        # this is D-1
      [,1]  [,2]
[1,]  0.52 -0.22
[2,] -0.44  0.19
>
```

2.4 Exercises

Use R to compute the following:

1. $\left|2^3 - 3^2\right|$.
2. e^e.
3. $(2.3)^8 + ln(7.5) - \cos(\pi/\sqrt{2})$.
4. Let $\boldsymbol{A} = \begin{pmatrix} 1 & 2 & 3 & 2 \\ 2 & 1 & 6 & 4 \\ 4 & 7 & 2 & 5 \end{pmatrix}$, $\boldsymbol{B} = \begin{pmatrix} 1 & 3 & 5 & 2 \\ 0 & 1 & 3 & 4 \\ 2 & 4 & 7 & 3 \\ 1 & 5 & 1 & 2 \end{pmatrix}$. Find $\boldsymbol{AB}^{-1}$ and $\boldsymbol{BA}^{T}$.
5. The *dot product* of [2, 5, 6, 7] and [-1, 3, -1, -1].

3. Getting Data into R

Functions and operators introduced in this section: **$, :, attach(), attributes(), data(), data.frame(), edit(), file.choose(), fix(), read.table(), rep(), scan(), search(), seq()**

We have already seen how the combine function **c()** in R can make a simple vector of numerical values. This function can also be used to construct a vector of text values:

```
> mykids <- c("Stephen", "Christopher")      # put text in quotes
> mykids
[1] "Stephen"     "Christopher"
```

As we will see, there are many other ways to create vectors and datasets in R.

3.1 Sequences

Sometimes we will need to create a string of numerical values that have a regular pattern. Instead of typing the sequence out, we can define the pattern using some special operators and functions.

- The colon operator **:**

  The colon operator creates a vector of numbers (between two specified numbers) that are one unit apart:

  ```
  > 1:9
  [1] 1 2 3 4 5 6 7 8 9

  > 1.5:10                    # you won't get to 10 here
  [1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5

  > c(1.5:10,10)              # we can attach it to the end this way
  [1]  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.0

  > prod(1:8)                 # same as factorial(8)
  [1] 40320
  ```

- The sequence function **seq()**

  The sequence function can create a string of values with any increment you wish. You can either specify the *incremental value* or the desired *length* of the sequence:

  ```
  > seq(1,5)           # same as 1:5
  [1] 1 2 3 4 5

  > seq(1,5,by=.5)     # increment by 0.5
  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
  ```

```
> seq(1,5,length=7)  # figure out the increment for this length
[1] 1.00000 1.66667 2.33333 3.00000 3.66667 4.33333 5.00000
```

- The replicate function **rep()**

  The replicate function can repeat a value or a sequence of values a specified number of times:

```
> rep(10,10)                  # repeat the value 10 ten times
 [1] 10 10 10 10 10 10 10 10 10 10

> rep(c("A","B","C","D"),2)   # repeat the string A,B,C,D twice
[1] "A" "B" "C" "D" "A" "B" "C" "D"

> matrix(rep(0,16),nrow=4)    # a 4x4 matrix of zeroes
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
>
```

3.2 Reading in Data: Single Vectors

When entering a larger array of values, the **c()** function can be unwieldy. Alternatively, data can be read directly from the keyboard by using the **scan()** function. This is useful since data values need only be separated by a blank space (although this can be changed in the arguments of the function). Also, by default the function expects *numerical* inputs, but you can specify others by using the "**what =**" option. The syntax is:

```
> x <- scan()        # read what is typed into the variable x
```

When the above is typed in an R session, you get a prompt signifying that R expects you to type in values to be added to the vector **x**. The prompt indicates the indexed value in the vector that it expects to receive. R will stop adding data when you enter a blank row. After entering a blank row, R will indicate the number of values it read in.

Suppose that we count the number of passengers (not including the driver) in the next 30 automobiles at an intersection:

```
> passengers <- scan()
1: 2 4 0 1 1 2 3 1 0 0 3 2 1 2 1 0 2 1 1 2 0 0     # I hit return
23: 1 3 2 2 3 1 0 3     # I hit return again
31:                     # I hit return one last time
Read 30 items

> passengers            # print out the values
[1] 2 4 0 1 1 2 3 1 0 0 3 2 1 2 1 0 2 1 1 2 0 0 1 3 2 2 3 1 0 3
```

In addition, the **scan()** function can be used to read data that is stored (as ASCII text) in an external file into a vector. To do this, you simply pass the filename (in quotes) to the **scan()** function. For example, suppose that the above passenger data was originally saved in a text file called **passengers.txt** that is located on a disk drive. To read in this data that is located on a C: drive, we would simply type

```
> passengers <- scan("C:/passengers.txt")
Read 30 items
>
```

Notes:

- ALWAYS view the text file first before you read it into R to make sure it is what you want and formatted appropriately.
- To represent directories or subdirectories, use the forward slash (/), not a backslash (\) in the path of the filename – even on a Windows system.
- If your computer is connected to the internet, data can also read (contained in a text file) from a URL using the **scan()** function. The basic syntax is given by:

  ```
  > dat <- scan("http://www...")
  ```

- If the directory name and/or file name contains spaces, we need to take special care denoting the space character. This is done by including a backslash (\) before the space is typed.

As an alternative, you can use the function **file.choose()** in place of the filename. In so doing, an explorer-type window will open and the file can be selected interactively.
More on reading in datasets from external sources is given in the next section.

3.3 Data Frames

3.3.1 Creating Data Frames

Often in statistics, a dataset will contain more than one variable recorded in an experiment. For example, in the automobile experiment from the last section, other variables might have been recorded like *automobile type* (sedan, SUV, minivan, etc.) and *driver seatbelt use* (Y, N). A dataset in R is best stored in an object called a *data frame*. Individual variables are designated as columns of the data frame and have unique names. However, all of the columns in a data frame must be of the same length.
You can enter data directly into a data frame by using the built-in data editor. This allows for an interactive means for data-entry that resembles a spreadsheet. You can access the editor by using either the **edit()** or **fix()** commands:

```
> new.data <- data.frame()    # creates an "empty" data frame
> new.data <- edit(new.data)  # request that changes made are
                              # written to data frame
```

OR

```
> new.data <- data.frame()    # creates an "empty" data frame
> fix(new.data)               # changes saved automatically
```

The data editor allows you to add as many variables (columns) to your data frame that you wish. The column names can be changed from the default **var1**, **var2**, etc. by clicking the column header. At this point, the variable type (either numeric or character) can also be specified.

When you close the data editor, the edited frame is saved.

You can also create data frames from preexisting variables in the workspace. Suppose that in the last experiment we also recorded the seatbelt use of the driver: Y = seatbelt worn, N = seatbelt not worn. This data is entered by (recall that since these data are text based, we need to put quotes around each data value):

```
> seatbelt <- c("Y","N","Y","Y","Y","Y","Y","Y","Y","Y",    # return
+ "N","Y","Y","Y","Y","Y","Y","Y","Y","Y","Y","Y","Y",      # return
+ "Y","Y","N","Y","Y","Y","Y")
>
```

We can combine these variables into a single data frame with the command

```
> car.dat <- data.frame(passengers,seatbelt)
```

A data frame looks like a matrix when you view it:

```
> car.dat
   passengers seatbelt
1           2        Y
2           4        N
3           0        Y
4           1        Y
5           1        Y
6           2        Y
. . .
```

The values along the left side are simply the row numbers.

3.3.2 Datasets Included with R

R contains many datasets that are built-in to the software. These datasets are stored as data frames. To see the list of datasets, type

```
> data()
```

A window will open and the available datasets are listed (many others are accessible from external user-written packages, however). To open the dataset called **trees**, simply type

```
> data(trees)
```

After doing so, the data frame **trees** is now in your workspace. To learn more about this (or any other included dataset), type **help(trees)**.

3.3.3 Accessing Portions of a Data Frame

You can access single variables in a data frame by using the **$** argument. For example, we see that the **trees** dataset has three variables:

```
> trees
   Girth Height Volume
1    8.3     70   10.3
2    8.6     65   10.3
3    8.8     63   10.2
4   10.5     72   16.4
5   10.7     81   18.8
. . .
```

To access single variables in a data frame, use a **$** between the data frame and column names:

```
> trees$Height
 [1] 70 65 63 72 81 83 66 75 80 75 79 76 76 69 75 74 85 86 71 64 78
[22] 80 74 72 77 81 82 80 80 80 87
> sum(trees$Height)                # sum of just these values
[1] 2356
>
```

You can also access a specific element or row of data by calling the specific position (in row, column format) in brackets after the name of the data frame:

```
> trees[4,3]            # entry at forth row, third column
[1] 16.4

> trees[4,]             # get the whole forth row
  Girth Height Volume
4  10.5     72   16.4
>
```

Often, we will want to access variables in a data frame, but using the **$** argument can get a little awkward. Fortunately, you can make R find variables in any data frame by adding the data frame to the *search path*. For example, to include the variables in the data frame **trees** in the search path, type

```
> attach(trees)
```

Now, the variables in trees are accessible without the **$** notation:

```
> Height
 [1] 70 65 63 72 81 83 66 75 80 75 79 76 76 69 75 74 85 86 71 64
[21] 78 80 74 72 77 81 82 80 80 80 87
```

To see exactly what is going on here, we can view the search path by using the **search()** command:

```
> search()
[1] ".GlobalEnv"        "trees"             "package:methods"
[4] "package:stats"     "package:graphics"  "package:utils"
[7] "Autoloads"         "package:base"
>
```

Note that the data frame **trees** is placed as the second item in the search path. This is the order in which R looks for things when you type in commands. FYI, **.GlobalEnv** is your workspace and the **package** quantities are libraries that contain (among other things) the functions and datasets that we are learning about in this manual.

To remove an object from the search path, use the **detach()** command in the same way that **attach()** is used. However, note that when you exit R, any objects added to the search path are removed anyway.

To list the features of any object in R, be it a vector, data frame, etc. use the **attributes()** function. For example:

```
> attributes(trees)
$names
[1] "Girth"  "Height" "Volume"

$row.names
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11"
[12] "12" "13" "14" "15" "16" "17" "18" "19" "20" "21" "22"
[23] "23" "24" "25" "26" "27" "28" "29" "30" "31"

$class
[1] "data.frame"

>
```

Here, we see that the **trees** object is a data frame with 31 rows and has variable names corresponding to the measurements taken on each tree.

Lastly, using the variable **Height**, this is a cool feature:

```
> Height[Height > 75]      # pick off all heights greater than 75
 [1] 81 83 80 79 76 76 85 86 78 80 77 81 82 80 80 80 87
>
```

3.3.4 Reading in Datasets

A dataset that has been created and stored externally (again, as ASCII text) can be read into a data frame. Here, we use the function **`read.table()`** to load the dataset into a data frame. If the first line of the text file contains the names of the variables in the dataset (which is often the case), R can take those as the names of the variables in the data frame. This is specified with the **`header = T`** option in the function call. If no header is included in a file, you can ignore this option and R will use the default variable names for a data frame. Filenames are specified in the same way as the **`scan()`** function, or the **`file.choose()`** function can be used to select the file interactively. For example, the function call

```
> smith <- read.table(file.choose(), header=T)
```

would read in data from a user-specified text file where the first line of the file designates the names of the variables in the data frame. For CSV files, the function **`read.csv()`** is used in a similar fashion.

3.3.5 Data files in formats other than ASCII text

If you have a file that is in a software-specific format (e.g. Excel, SPSS, etc.), there are R functions that can be used to import the data in R. The manual "**R Data Import/Export**" accessible on the R Project website addresses this. However, since most programs allow you to save data files as a tab delimited text files, this is usually preferred.

3.4 Exercises

1. Generate the following sequences in R:
   a. **`1 2 3 1 2 3 1 2 3`**
   b. **`10.00000 10.04545 10.09091 10.13636 10.18182 10.22727 10.27273 10.31818 10.36364 10.40909 10.45455 10.50000`**
   c. **`"1" "2" "3" "banana" "1" "2" "3" "banana"`**

2. Using the **`scan()`** function, enter 10 numbers (picked at random between 1 and 100) into a vector called **`blahblah`**.

3. Create a data frame called **`schedule`** with the following variables:
   `coursenumber`: the course numbers of your classes this semester (e.g. 329)
   `coursedays`: meeting days (either MWF, TR, etc.)
   `grade`: your anticipated grade (A, B, C, D, or F)

4. Load in the **`stackloss`** dataset from within R and save the variables **`Water.Temp`** and **`Acid.Conc.`** in a data frame called **`tempacid`**.

4. Introduction to Graphics

One of the greatest powers of R is its graphical capabilities (see the exercises section in this chapter for some amazing demonstrations). In this chapter, some of these features will be briefly explored.

4.1 The Graphics Window

When pictures are created in R, they are presented in the active graphical *device* or window (for the Mac, it's the Quartz device). If no such window is open when a graphical function is executed, R will open one. Some features of the graphics window:

- You can print directly from the graphics window, or choose to copy the graph to the clipboard and paste it into a word processor. There, you can also resize the graph to fit your needs. A graph can also be saved in many other formats, including pdf, bitmap, metafile, jpeg, or postscript.

- Each time a new plot is produced in the graphics window, the old one is lost. In MS Windows, you can save a "history" of your graphs by activating the *Recording* feature under the *History* menu (seen when the graphics window is active). You can access old graphs by using the "Page Up" and "Page Down" keys. Alternatively, you can simply open a new active graphics window (by using the function **`x11()`** in Windows/Unix and **`quartz()`** on a Mac).

4.2 Two Basic Graphing Functions

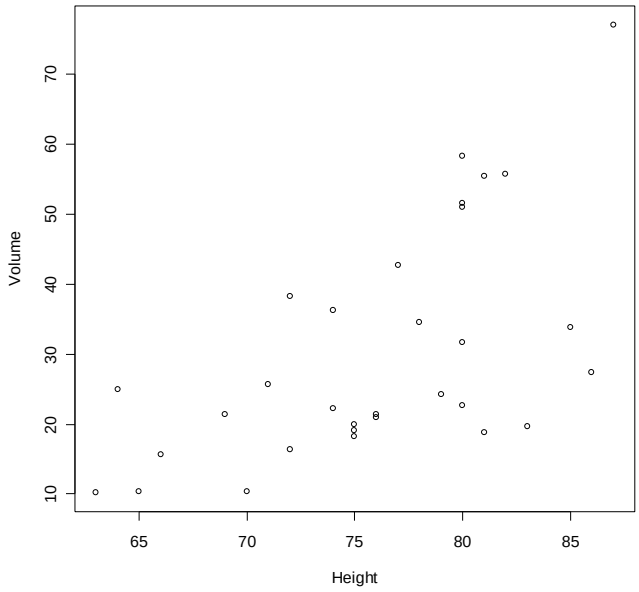
There are many functions in R that produce graphs, and they range from the very basic to the very advanced and intricate. In this section, two basic functions will be profiled, and information on ways to embellish plots will be given in the sections that follow. Other graphical functions will be described in Chapter 5.

*4.2.1 The **`plot()`** Function*

The most common function used to graph anything in R is the **`plot()`** function. This is a generic function that can be used for scatterplots, time-series plots, function graphs, etc. If a single vector object is given to **`plot()`**, the values are plotted on the y-axis against the row numbers or *index*. If two vector objects (of the same length) are given, a bivariate scatterplot is produced. For example, consider again the dataset **`trees`** in R. To visualize the relationship between **`Height`** and **`Volume`**, we can draw a scatterplot:

```
> plot(Height, Volume)     # object trees is in the search path
```

The plot appears in the graphics window:



Notice that the format here is the first variable is plotted along the horizontal axis and the second variable is plotted along the vertical axis. By default, the variable names are listed along each axis.

This graph is pretty basic, but the **`plot()`** function can allow for some pretty snazzy window dressing by changing the function arguments from the default values. These include adding titles/subtitles, changing the plotting character/color (over 600 colors are available!), etc. See **`?par`** for an overwhelming lists of these options.

This function will be used again in the succeeding chapters.

*4.2.2 The **`curve()`** Function*

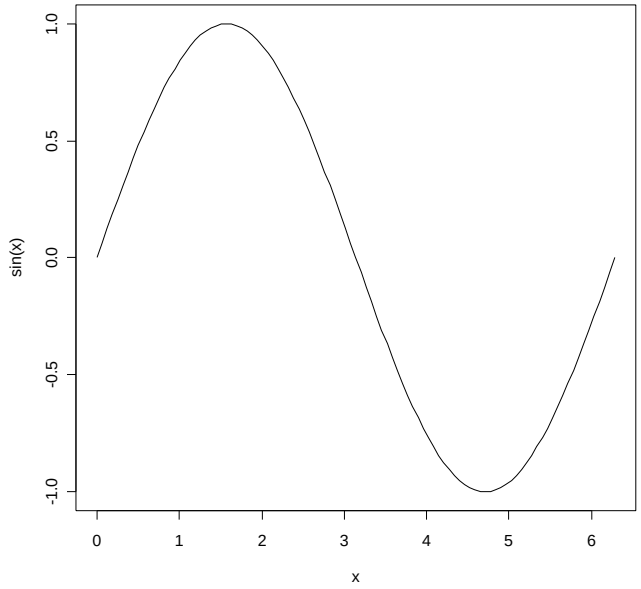
To graph a continuous function over a specified range of values, the **`curve()`** function can be used (although interestingly **`curve()`** actually calls the **`plot()`** function). The basic use of this function is:

```
curve(expr, from, to, add = FALSE, ...)

   Arguments:
   expr: an expression written as a function of 'x'

   from, to: the range over which the function will be plotted.

   add: logical; if 'TRUE' add to already existing plot.
```

Note that it is necessary that the **`expr`** argument is always written as a function of **`'x'`**. If the argument **`add`** is set to **`TRUE`**, the function graph will be overlaid on the current graph in the graphics window (this useful feature will be illustrated in Chapter 6).

For example, the **`curve()`** function can be used to plot the sine function from 0 to 2π:

```
> curve(sin(x), from = 0, to = 2*pi)
```



4.3 Graph Embellishments

In addition to standard graphics functions, there are a host of other functions that can be used to *add* features to a drawn graph in the graphics window. These include (see each function's help file for more information):

Function	**Operation**
`abline()`	adds a straight line with specified intercept and slope (or draw a vertical or horizontal line)
`arrows()`	adds an arrow at a specified coordinate
`lines()`	adds lines between coordinates
`points()`	adds points at specified coordinates (also for overlaying scatterplots)
`rug()`	adds a "rug" representation to one axis of the plot
`segments()`	similar to **`lines()`** above
`text()`	adds text (possibly inside the plotting region)
`title()`	adds main titles, subtitles, etc. with other options

The plot used on the cover page of this document includes some of these additional features applied to the graph in Section 4.2.1.

4.4 Changing Graphics Parameters

There is still more fine tuning available for altering the graphics settings. To make changes to how plots appear in the graphics window itself, or to have every graphic created in the graphics window follow a specified form, the default graphical parameters can be changed using the **par()** function. There are over 70 graphics parameters that can be adjusted, so only a few will be mentioned here. Some very useful ones are given below:

```
> par(mfrow = c(2, 2))  # gives a 2 x 2 layout of plots
> par(lend = 1)         # gives "butt" line end caps for line plots[2]
> par(bg = "cornsilk")  # plots drawn with this colored background
> par(xlog = TRUE)      # always plot x axis on a logarithmic scale
```

Any or all parameters can be changed in a **par()** command, and they remain in effect until they are changed again (or if the program is exited). You can save a copy of the original parameter settings in **par()**, and then after making changes recall the original parameter settings. To do this, type

```
> oldpar <- par(no.readonly = TRUE)

      ... then, make your changes in par() ...

> par(oldpar)     # default (original) parameter settings restored
```

4.5 Exercises

As mentioned previously, more on graphics will be seen in the next two chapters. For this section, enter the following commands to see some R's incredible graphical capabilities. Also, try pasting/inserting a graph into a word processor or document.

```
> demo(graphics)
> demo(persp)     # for 3-d plots
> demo(image)
```

[2] This is used for all line plots (e.g. page 31) presented herein.

5. Summarizing Data

One of the simplest ways to describe what is going on in a dataset is to use a graphical or numerical summary procedure. Numerical summaries are things like means, proportions, and variances, while graphical summaries include histograms and boxplots.

5.1 Numerical Summaries

R includes a host of built in functions for computing sample statistics for both numerical (both continuous and discrete) and categorical data. For *numerical data*, these include

Name	Operation
`mean()`	arithmetic mean
`median()`	sample median
`fivenum()`	five-number summary
`summary()`	generic summary function for data and model fits
`min(), max()`	smallest/largest values
`quantile()`	calculate sample quantiles (percentiles)
`var(), sd()`	sample variance, sample standard deviation
`cov(), cor()`	sample covariance/correlation

These functions will take one or more vectors as arguments for the calculation; in addition, they will (in general) work in the correct way when they are given a data frame as the argument.

If your data contains only discrete *counts* (like the number of pets owned by each family in a group of 20 families), or is *categorical* in nature (like the eye color recorded for a sample of 50 fruit flies), the above numerical measures may not be of much use. For categorical or discrete data, we can use the **`table()`** function to summarize a dataset.

For examples using these functions, let's consider the dataset **`mtcars`** in R contains measurements on 11 aspects of automobile design and performance for 32 automobiles (1973-74 models):

```
> data(mtcars)       # load in dataset
> attach(mtcars)     # add mtcars to search path
> mtcars
                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4          21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710         22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive     21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
. . .
```

The variables in this dataset are both continuous (e.g. **`mpg`**, **`disp`**, **`wt`**) and discrete (e.g. **`gear`**, **`carb`**, **`cyl`**) in nature. For the continuous variables, we can calculate:

```
> mean(hp)
[1] 146.6875
> var(mpg)
[1] 36.3241
> quantile(qsec, probs = c(.20, .80))     # 20th and 80th percentiles
   20%    80%
16.734 19.332
> cor(wt,mpg)           # not surprising that this is negative
[1] -0.8676594
```

For the discrete variables, we can get summary counts:

```
> table(cyl)
cyl
 4  6  8
11  7 14
```

So, it can be seen that eleven of the vehicles have 4 cylinders, seven vehicles have 6, and fourteen have 8 cylinders. We can turn the counts into percentages (or *relative frequencies*) by dividing by the total number of observations:

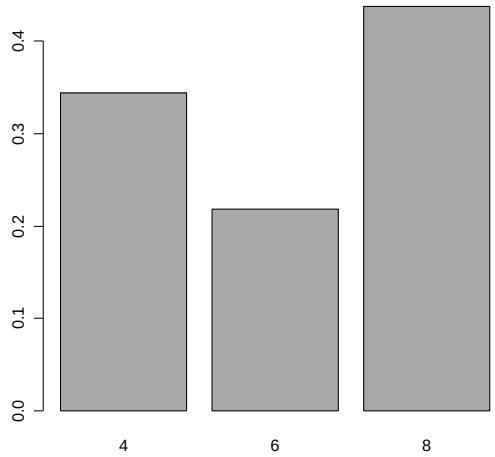
```
> table(cyl)/length(cyl)          # note: length(cyl) = 32
cyl
      4       6       8
0.34375 0.21875 0.43750
```

5.2 Graphical Summaries

- **`barplot()`:**

  For discrete or categorical data, we can display the information given in a table command in a picture using the **`barplot()`** function. This function takes as its argument a table object created using the **`table()`** command discussed above:

```
> barplot(table(cyl)/length(cyl))   # use relative frequencies on
                                    # the y-axis
```



See **?barplot** on how to change the fill color, add titles to your graph, etc.

- **hist():**

This function will plot a histogram that is typically used to display continuous-type data. Its format, with the most commonly used options, is:

```
hist(x,breaks="Sturges",prob=FALSE,main=paste("Histogram of" ,xname))

   Arguments:

   x: a vector of values for which the histogram is desired.

   breaks: one of:

      * a character string (in double quotes) naming an algorithm to
        compute the number of cells

         The default for 'breaks' is '"Sturges"': Other names for which
         algorithms are supplied are '"Scott"' and '"FD"'

      * a single number giving the number of cells for the histogram

   prob: logical; if FALSE, the histogram graphic is a representation
         of frequencies, the 'counts' component of the result; if
         TRUE, _relative_ frequencies ("probabilities"), component
         'density', are plotted.

   main: the title for the histogram
```

The **breaks** argument specifies the number of bins (or "classes") for the histogram. Too few or too many bins can result in a poor picture that won't characterize the data well. By default, R uses the Sturges formula for calculating the number of bins. This is given by

$$\lceil \log_2(n)+1 \rceil$$

where n is the sample size and $\lceil\ \rceil$ is the ceiling operator.

Other methods exist that consider finding the optimal *bin width* (the number of bins required would then be the sample range divided by the bin width). The Freedman-Diaconis formula (Freedman and Diaconis 1981) is based on the inter-quartile range (*iqr*)

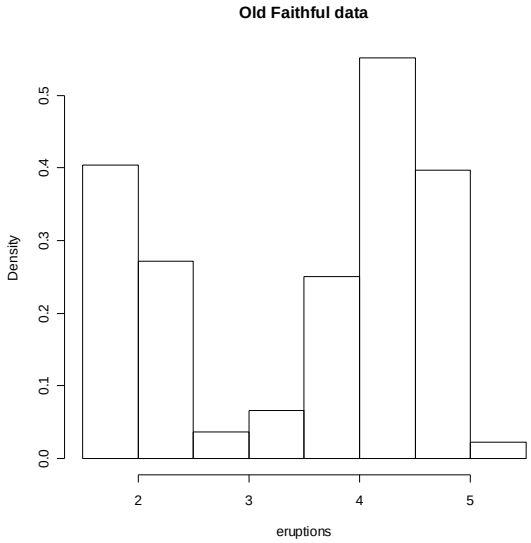
$$2 \cdot iqr \cdot n^{-1/3};$$

the formula proposed by Scott (1979) is based on the standard deviation (*s*)

$$3.5 \cdot s \cdot n^{-1/3}.$$

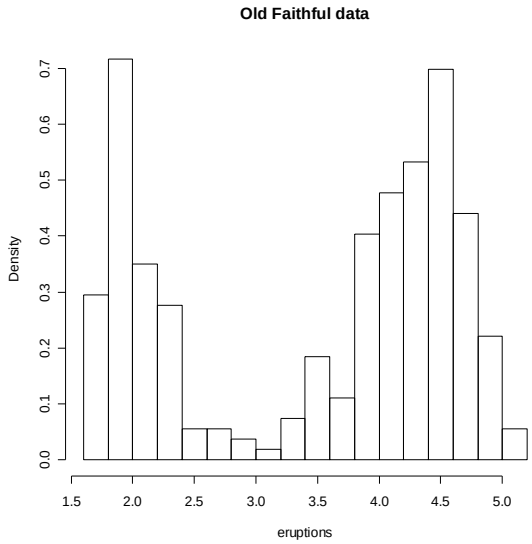
To see some differences, consider the **faithful** dataset in R, which is a famous dataset that exhibits natural bimodality. The variable **eruptions** gives the duration of the eruption (in minutes) and **waiting** is the time between eruptions for the Old Faithful geyser:

```
> data(faithful)
> attach(faithful)
> hist(eruptions, main = "Old Faithful data", prob = T)
```



We can give the picture a slightly different look by changing the number of bins:

```
> hist(eruptions, main = "Old Faithful data", prob = T, breaks=18)
```



- **stem()**

This function constructs a text-based stem-and-leaf display that is produced in the **R Console**. The optional argument **scale** can be used to control the length of the display.

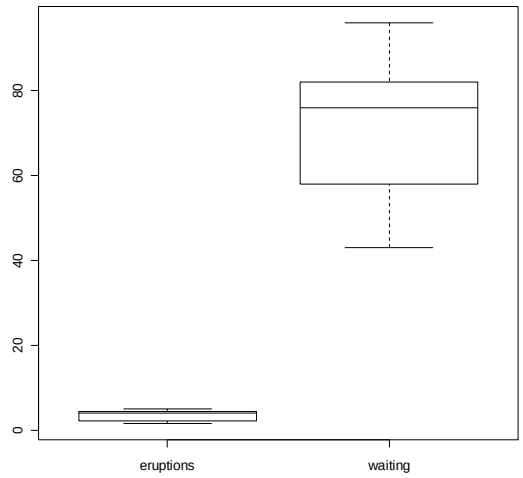
```
> stem(waiting)

  The decimal point is 1 digit(s) to the right of the |

  4 | 3
  4 | 55566666777788899999
  5 | 00000111111222223333333444444444
  5 | 555556666677788889999999
  6 | 00000022223334444
  6 | 555667899
  7 | 00001111123333333444444
  7 | 55555555666666666777777777778888888888888889999999999
  8 | 00000000111111111111222222222223333333333333334444444444
  8 | 55555666666677888888999
  9 | 00000012334
  9 | 6
```

- **boxplot()**

This function will construct a single boxplot if the argument passed is a single vector, but if many vectors are contained (or if a data frame is passed), a boxplot for each variable is produced *on the same graph*.

For the two data files in the Old Faithful dataset:

```
> boxplot(faithful)     # same as boxplot(eruptions, waiting)
```



Thus, the waiting time for an eruption is generally much larger and has higher variability than the actual eruption time. See **?boxplot** for ways to add titles/color, changing the orientation, etc.

- **`ecdf():`**

This function will create the values of the empirical distribution function (EDF) $F_n(x)$. It requires a single argument – the vector of numerical values in a sample. To plot the EDF for data contained in **`x`**, type

```
> plot(ecdf(x))
```

- **`qqnorm() and qqline()`**

These functions are used to check for normality in a sample of data by constructing a normal probability plot (NPP) or normal *q*-*q* plot. The syntax is:

```
> qqnorm(x)    # creates the NPP for values stored in x
> qqline(x)    # adds a reference line for the NPP
```

Only a few of the many functions used for graphics have been discussed so far. Other graphical functions include:

Name	**Operation**
`pairs()`	Draws all possible scatterplots for two columns in a matrix/dataframe
`persp()`	Three dimensional plots with colors and perspective
`pie()`	Constructs a pie chart for categorical/discrete data
`qqplot()`	quantile-quantile plot to compare two datasets
`ts.plot()`	Time series plot

5.3 Exercises

Using the **`stackloss`** dataset that is available from within R:

1. Compute the mean, variance, and 5 number summary of the variable **`stack.loss`**.
2. Create a histogram, boxplot, and normal probability plot for the variable **`stack.loss`**. Does an assumption of normality seem appropriate for this sample?

6. Probability, Distributions, and Simulation

6.1 Distribution Functions in R

R allows for the calculation of probabilities (including cumulative), the evaluation of probability density/mass functions, percentiles, and the generation of pseudo-random variables following a number of common distributions. The following table gives examples of various function names in R along with additional arguments.

Distribution	R name	Additional arguments	Argument defaults
beta	`beta`	`shape1, shape2`	
binomial	`binom`	`size, prob`	
Chi-square	`chisq`	`df (degrees of freedom)`	
continuous uniform	`unif`	`min, max`	`min = 0, max = 1`
exponential	`exp`	`rate`	`rate = 1`
F distribution	`f`	`df1, df2`	
gamma	`gamma`	`shape, rate (or scale = 1/rate)`	`scale = 1`
geometric	`geom`	`prob`	
hypergeometric	`hyper`	`m, n, k (sample size)`	
negative binomial	`nbinom`	`size, prob`	
normal	`norm`	`mean, sd`	`mean = 0, sd = 1`
Poisson	`pois`	`lambda`	
t distribution	`t`	`df`	
Weibull	`weibull`	`shape, scale`	`scale = 1`

Prefix each R name given above with '**d**' for the density or mass function, '**p**' for the CDF, '**q**' for the percentile function (also called the quantile), and '**r**' for the generation of pseudo-random variables. The syntax has the following form – we use the wildcard ***rname*** to denote a distribution above:

```
> drname(x, ...)  # the pdf/pmf at x (possibly a vector)
> prname(q, ...)  # the CDF at q (possibly a vector)
> qrname(p, ...)  # the p^th (possibly a vector) percentile/quantile
> rrname(n, ...)  # simulate n observations from this distribution
```

Note: since probability density/mass functions can be parameterized differently, R's definitions are given in the Appendix. The following are examples with these R functions:

```
> x <- rnorm(100)          # simulate 100 standard normal RVs, put in x
> w <- rexp(1000,rate=.1)     # simulate 1000 from Exp(θ = 10)
> dbinom(3,size=10,prob=.25)  # P(X=3) for X ~ Bin(n=10, p=.25)
> dpois(0:2, lambda=4)        # P(X=0), P(X=1), P(X=2) for X ~ Poisson
> pbinom(3,size=10,prob=.25)  # P(X ≤ 3) in the above distribution
> pnorm(12,mean=10,sd=2)      # P(X ≤ 12) for X~N(mu = 10, sigma = 2)
> qnorm(.75,mean=10,sd=2)     # 3rd quartile of N(mu = 10,sigma = 2)
> qchisq(.10,df=8)            # 10th percentile of χ²(8)
> qt(.95,df=20)               # 95th percentile of t(20)
```

6.2 A Simulation Application: Monte Carlo Integration

Suppose that we wish to calculate

$$I = \int_a^b g(x)dx,$$

but the antiderivative of $g(x)$ cannot be found in closed form. Standard techniques involve approximating the integral by a sum, and many computer packages can do this. Another approach to finding I is called Monte Carlo integration and it works for the following example. Suppose that we generate n independent Uniform random variables[3] (this we have already seen how to do) $X_1, X_2, \ldots, X_n$ on the interval $[a, b]$ and compute

$$\hat{I} = (b-a)\frac{1}{n}\sum_{i=1}^{n} g(X_i)$$

By the Law of Large Numbers, as n increases with out bound,

$$\hat{I}_n \to (b-a)\mathbf{E}\left[g(\mathbf{X})\right].$$

By mathematical expectation,

$$\mathrm{E}\left[g(\mathrm{X})\right] = \int_a^b g(x)\frac{1}{b-a}dx = \left(\frac{1}{b-a}\right)I$$

So, $\hat{I}_n$ can be used as an approximation for I that improves as n increases[4].

As it turns out, this method can be modified to use other distributions (besides the Uniform) defined over the same interval. Compared to other numerical methods for approximating integrals, the Monte Carlo method is not particularly efficient. However, the Monte Carlo method becomes increasingly efficient as the dimensionality of the integral (*e.g.* double integrals, triple integrals) rises.

Example: Consider the definite integral: $\int_0^{\pi/2} 4\sin(2x)\exp(-x^2)dx$.

A Monte Carlo estimate would be given by (using 1,000,000 observations):

```
> u <- runif(1000000, min=0, max=pi/2)
> pi/2*mean(4*sin(2*u)*exp(-u^2))   # a=0, b=pi/2, so b-a=pi/2
[1] 2.189178
```

Another call gets a slightly different answer (remember, it is a limiting value!):

```
> u <- runif(1000000, min=0, max=pi/2)  # generate a new uniform vector
> pi/2*mean(4*sin(2*u)*exp(-u^2))   # a=0, b=pi/2, so b-a=pi/2
[1] 2.191414
```

[3] The method can be easily modified to use another distribution defined on the interval [a, b]. See Robert and Casella (1999).

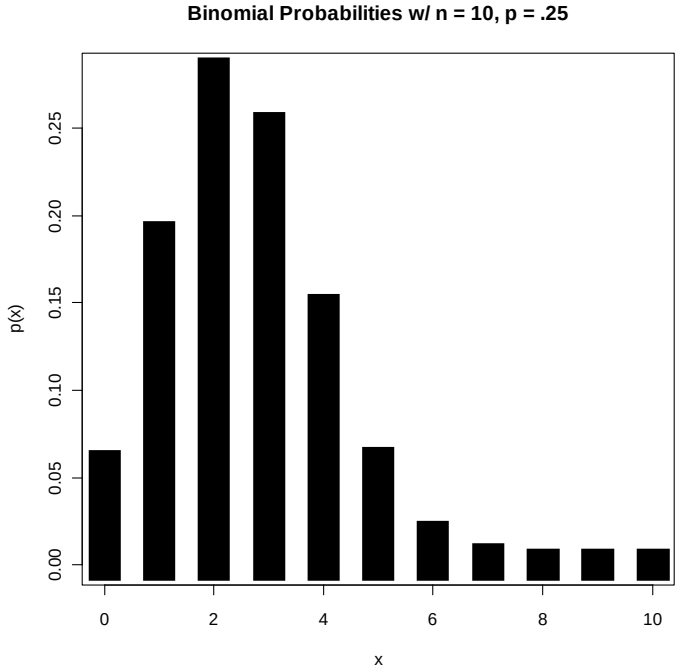
[4] The R function **integrate()** is another option for numerical quadrature. See **?integrate**.

6.3 Graphing Distributions

6.3.1 Discrete Distributions

Graphs of probability mass functions (pmfs) and CDFs can be drawn using the **`plot()`** function. Here, we give the function the support of the distribution and probabilities at these points. By default, R would simply produce a scatterplot, but we can specify the plot type by using the **`type`** and **`lwd`** (line width) arguments. For example, to graph the probability mass function for the binomial distribution with $n = 10$ and $p = .25$:

```
   > x <- 0:10
   > y <- dbinom(x, size=10, prob=.25)    # evaluate probabilities
   > plot(x, y, type = "h", lwd = 30, main = "Binomial Probabilities w/ n
= 10, p = .25", ylab = "p(x)", lend ="square")   # not a hard return here!
```



We have done a few things here. First, we created the vector **`x`** that contains the integers 0 through 10. Then we calculated the binomial probabilities at each of the points in **`x`** and stored them in the vector **`y`**. Then, we specified that the plot be of type **`"h"`** which is gives the histogram-like vertical lines and we "fattened" the lines with the **`lwd = 30`** option (the default width is 1, which is line thickness). Finally, we gave it some informative titles.

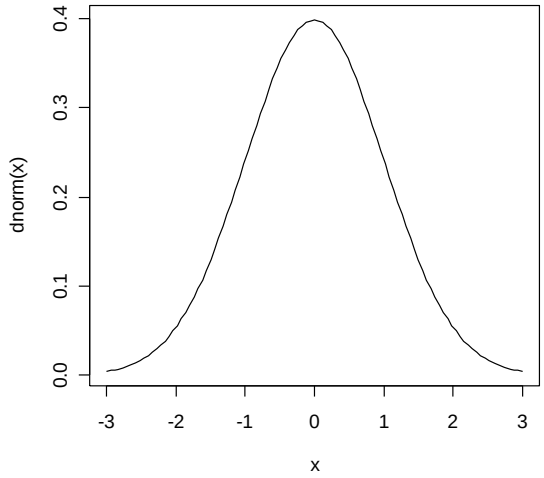
Lastly, we note that to conserve space in the workspace, we could have produced the same plot without actually creating the vectors **`x`** and **`y`** (embellishments removed):

```
   > plot(0:10, dbinom(x, size=10, prob=.25), type = "h", lwd = 30)
```

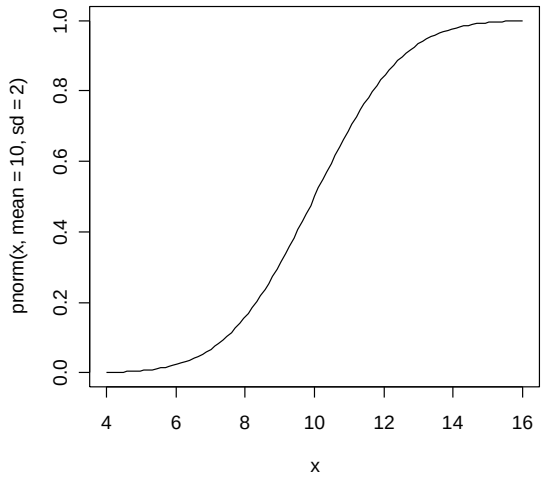
6.3.2 Continuous Distributions

To graph smooth functions like a probability density function or CDF for a continuous random variable, we can use the **`curve()`** function that was introduced in Chapter 4. To plot a density function, we can use the names of density functions in R as the expression argument. Some examples:

```
> curve(dnorm(x), from = -3, to = 3)  # the standard normal curve
```




```
> curve(pnorm(x, mean=10, sd=2), from = 4, to = 16)  # a normal CDF
```

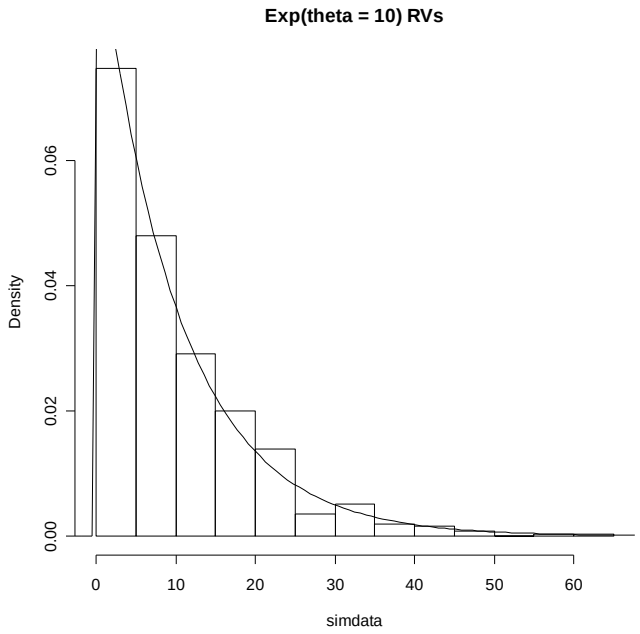


Note that we restricted the plotting to be between -3 and 3 in the first plot since this is where the standard normal has the majority of its area. Alternatively, we could have used upper and lower percentiles (say the .5% and 99.5%) and calculated them by using the **`qnorm()`** function.

Also note that the **`curve()`** function has as an option to be added to an existing plot. When you overlay a curve, you don't have to specify the **`from`** and **`to`** arguments because R defaults

them to the low and high values of the x-values on the original plot. Consider how the histogram a large simulation of random variables compares to the density function:

```
> simdata <- rexp(1000, rate=.1)
> hist(simdata, prob = T, breaks = "FD", main="Exp(theta = 10) RVs")
> curve(dexp(x, rate=.1), add = T)     # overlay the density curve
```



6.4 Random Sampling

Simple probability experiments like "choosing a number at random between 1 and 100" and "drawing three balls from an urn" can be simulated in R. The theory behind games like these forms the foundation of sampling theory (drawing random samples from fixed populations); in addition, *resampling methods* (repeated sampling within the same sample) like the bootstrap are important tools in statistics. The key function in R is the **`sample()`** function. Its usage is:

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

Arguments:

x:	**Either a (numeric, complex, character or logical) vector of more than one element from which to choose, or a positive integer.**
size:	**non-negative integer giving the number of items to choose.**
replace:	**Should sampling be with or without replacement?**
prob:	**An optional vector of probability weights for obtaining the elements of the vector being sampled.**

Some examples using this function are:

```
> sample(1:100, 1)              # choose a number between 1 and 100
[1] 34

> sample(1:6, 10, replace = T)   # toss a fair die 10 times
 [1] 1 3 6 4 5 2 2 5 4 5

> sample(1:6, 10, T, c(.6,.2,.1,.05,.03,.02))   # not a fair die!!
 [1] 1 1 2 1 4 1 3 1 2 1

> urn <- c(rep("red",8),rep("blue",4),rep("yellow",3))
> sample(urn, 6, replace = F)    # draw 6 balls from this urn
[1] "yellow" "red"    "blue"   "yellow" "red"    "red"
```

6.5 Exercises

1. Simulate 20 observations from the binomial distribution with $n = 15$ and $p = 0.2$.
2. Find the 20th percentile of the gamma distribution with $\alpha = 2$ and $\theta = 10$.
3. Find $P(T > 2)$ for $T \sim t_8$.
4. Plot the Poisson mass function with $\lambda = 4$ over the range $x = 0, 1, \ldots 15$.
5. Using Monte Carlo integration[5], approximate the integral
$$\int_0^2 \exp(x^3)dx$$
with $n = 1{,}000{,}000$.
6. Simulate 100 observations from the normal distribution with $\mu = 50$ and $\sigma = 4$. Plot the empirical cdf $F_n(x)$ for this sample and overlay the true CDF $F(x)$.
7. Simulate 25 flips of a fair coin where the results are "heads" and "tails" (hint: use the **`sample()`** function).

[5] Compare your answer with: **`integrate(f = function(x) exp(x^3), lower = 0, upper = 2)`**

7. Statistical Methods

R includes a host of statistical methods and tests of hypotheses. We will focus on the most common ones in this Chapter.

7.1 One and Two-sample t-tests

The main function that performs these sorts of tests is **`t.test()`**. It yields hypothesis tests and confidence intervals that are based on the *t*-distribution. Its syntax is:

```
Usage:

   t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"),
   mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95)

Arguments:

   x, y: numeric vectors of data values.  If y is not given, a one
         sample test is performed.

   alternative: a character string specifying the alternative
   hypothesis, must be one of `"two.sided"' (default), `"greater"'
   or `"less"'. You can specify just the initial letter.

   mu: a number indicating the true value of the mean (or difference in
   means if you are performing a two sample test).  Default is 0.

   paired: a logical indicating if you want the paired t-test (default
   is the independent samples test if both x and y are given).

   var.equal: (for the independent samples test) a logical variable
   indicating whether to treat the two variances as being equal. If
   `TRUE', then the pooled variance is used to estimate the variance.
   If 'FALSE' (default), then the Welsh suggestion for degrees of
   freedom is used.

   conf.level: confidence level (default is 95%) of the interval
   estimate for the mean appropriate to the specified alternative
   hypothesis.
```

Note that from the above, **`t.test()`** not only performs the hypothesis test but also calculates a confidence interval. However, if the alternative is either a "greater than" or "less than" hypothesis, a lower (in case of a greater than alternative) or upper (less than) *confidence bound* is given.

Example 1: Using the **`trees`** dataset, test the hypothesis that the mean black cherry tree *height* is 70 ft. versus a two-sided alternative:

```
> data(trees)
> t.test(trees$Height, mu = 70)
```

```
        One Sample t-test

data:  trees$Height
t = 5.2429, df = 30, p-value = 1.173e-05
alternative hypothesis: true mean is not equal to 70
95 percent confidence interval:
 73.6628 78.3372
sample estimates:
mean of x
       76
```

Thus, the null hypothesis would be rejected.

Example 2: the recovery time (in days) is measured for 10 patients taking a new drug and for 10 different patients taking a placebo[6]. We wish to test the hypothesis that the mean recovery time for patients taking the drug is less than fort those taking a placebo (under an assumption of normality and equal population variances). The data are:

With drug: 15, 10, 13, 7, 9, 8, 21, 9, 14, 8
Placebo: 15, 14, 12, 8, 14, 7, 16, 10, 15, 12

For our test, we will assume that the two population means are equal. In R, the analysis of a two–sample t–test would be performed by:

```
> drug <- c(15, 10, 13, 7, 9, 8, 21, 9, 14, 8)
> plac <- c(15, 14, 12, 8, 14, 7, 16, 10, 15, 12)
> t.test(drug, plac, alternative = "less", var.equal = T)

        Two Sample t-test

data:  drug and plac
t = -0.5331, df = 18, p-value = 0.3002
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
     -Inf 2.027436
sample estimates:
mean of x mean of y
     11.4      12.3
```

With a p–value of .3002, we would not reject the claim that the two mean recovery times are equal.

Example 3: an experiment was performed to determine if a new gasoline additive can increase the gas mileage of cars. In the experiment, six cars are selected and driven with and without the additive. The gas mileages (in miles per gallon, mpg) are given below.

Car	1	2	3	4	5	6
mpg w/ additive	24.6	18.9	27.3	25.2	22.0	30.9
mpg w/o additive	23.8	17.7	26.6	25.1	21.6	29.6

[6] This example is from SimpleR (see page 7) as documented on CRAN.

Since this is a paired design, we can test the claim using the paired t–test (under an assumption of normality for mpg measurements). This is performed by:

```
> add <- c(24.6, 18.9, 27.3, 25.2, 22.0, 30.9)
> noadd <- c(23.8, 17.7, 26.6, 25.1, 21.6, 29.6)
> t.test(add, noadd, paired=T, alt = "greater")

        Paired t-test

data:  add and noadd
t = 3.9994, df = 5, p-value = 0.005165
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 0.3721225       Inf
sample estimates:
mean of the differences
                  0.75
```

With a p–value of .005165, we can conclude that the mpg improves with the additive.

7.2 Analysis of Variance (ANOVA)

The simplest way to fit an ANOVA model is to call to the **aov()** function, and the type of ANOVA model is specified by a formula statement. Some examples:

```
> aov(x ~ a)              # one-way ANOVA model
> aov(x ~ a + b)          # two-way ANOVA with no interaction
> aov(x ~ a + b + a:b)    # two-way ANOVA with interaction
> aov(x ~ a*b)            # exactly the same as the above
```

In the above statements, the **x** variable is continuous and contains *all* of the responses in the ANOVA experiment. The variables **a** and **b** represent factor variables – they contain the levels of the experimental factors. The levels of a factor in R can be either *numerical* (e.g. 1, 2, 3,…) or *categorical* (e.g. low, medium, high, …), but the variables must be stored as *factor variables*. We will see how this is done next.

7.2.1 Factor Variables

As an example for the ANOVA experiment, consider the following example on the strength of three different rubber compounds; four specimens of each type were tested for their tensile strength (measured in pounds per square inch):

Type	*A*	*B*	*C*
Strength (lb/in^2)	3225, 3320, 3165, 3145	3220, 3410, 3320, 3370	3545, 3600, 3580, 3485

In R:

```
> Str <- c(3225,3320,3165,3145,3220,3410,3320,3370,3545,3600,3580,3485)
> Type <- c(rep("A",4), rep("B",4), rep("C",4))
```

Thus, the **Type** variable specifies the rubber type and the **Str** variable is the tensile strength. Currently, R thinks of **Type** as a *character* variable; we want to let R know that these letters actually represent factor levels in an experiment. To do this, use the **factor()** command:

```
> Type <- factor(Type)     # consider these alternative expressions[7]
> Type
 [1] A A A A B B B B C C C C
Levels: A B C
>
```

Note that after the values in **Type** are printed, a **Levels** list is given. To access the levels in a factor variable directly, you can type:

```
> levels(Type)
[1] "A" "B" "C"
```

With the data in this format, we can easily perform calculations on the subgroups contained in the variable **Str**. To calculate the sample means of the subgroups, type

```
> tapply(Str,Type,mean)
      A       B       C
3213.75 3330.00 3552.50
```

The function **tapply()** creates a *table* of the resulting values of a function *applied* to subgroups defined by the second (factor) argument. To calculate the variances:

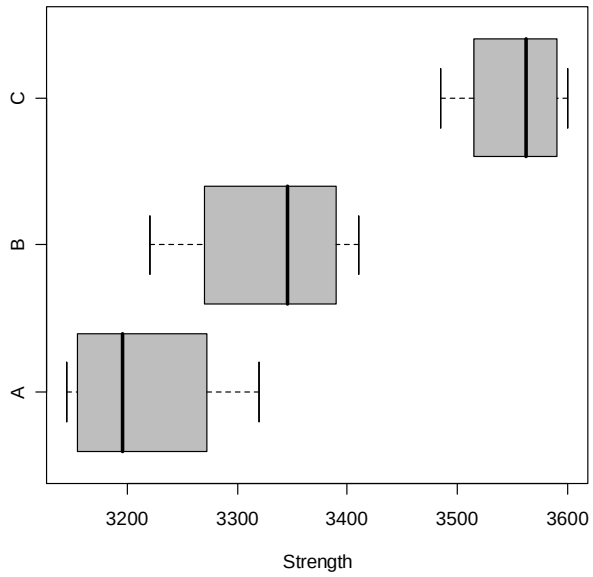
```
> tapply(Str,Type,var)
       A        B        C
6172.917 6733.333 2541.667
```

We can also get multiple boxplots by specifying the relationship in the **boxplot()** function:

```
> boxplot(Str ~ Type, horizontal = T, xlab="Strength", col = "gray")
```

[7] These accomplish the same thing:

```
> Type <- factor(rep(LETTERS[1:3], each = 4)
> Type <- gl(3, 4, lab = LETTERS[1:3])
```



7.2.2 The ANOVA Table

In order to fit the ANOVA model, we specify the single factor model in the **`aov()`** function.

```
> anova.fit <- aov(Str ~ Type)
```

The object **`anova.fit`** is a *linear model object*. To extract the ANOVA table, use the R function **`summary()`**:

```
> summary(anova.fit)
            Df Sum Sq Mean Sq F value    Pr(>F)
Type         2 237029  118515  23.016 0.0002893 ***
Residuals    9  46344    5149
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
```

With a p-value of 0.0002893, we can confidently reject the null hypothesis that all three rubber types have the same mean strength.

7.2.3 Multiple Comparisons

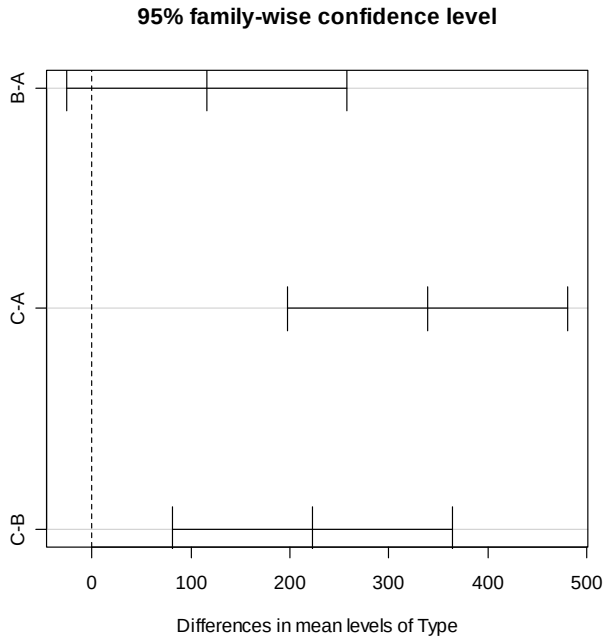
After an ANOVA analysis has determined a significant difference in means, a multiple comparison procedure can be used to determine which means are different. There are several procedures for doing this; here, we give the code to conduct Tukey's Honest Significant Difference method (or sometimes referred to as "Tukey's W Procedure") that uses the Studentized range distribution[8].

[8] Specific percentage points for this distribution can be found using **`qtukey()`**.

```
> TukeyHSD(anova.fit)  # the default is 95% CIs for differences
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = Str ~ Type)

$Type
      diff       lwr      upr     p adj
B-A 116.25 -25.41926 257.9193 0.1085202
C-A 338.75 197.08074 480.4193 0.0002399
C-B 222.50  80.83074 364.1693 0.0044914

>
```

The above output gives 95% confidence intervals for the pairwise differences of mean strengths for the three types of rubber. So, types "B" and "A" do not appear to have significantly different mean strengths since the confidence interval for their mean difference contains zero. A graphic can be used to support the analysis:

```
> plot(TukeyHSD(anova.fit))
```



7.3 Linear Regression

Fitting a linear regression model in R is very similar to the ANOVA model material, and this is intuitive since they are both linear models. To fit the linear regression ("least-squares") model to data, we use the **`lm()`** function[9]. This can be used to fit simple linear (single predictor), multiple linear, and polynomial regression models. With data loaded in, you only need to specify the linear model desired. Examples of these are:

[9] A nonlinear regression model can be fitted by using **`nls()`**.

```
> lm(y ~ x)                    # simple linear regression (SLR) model
> lm(y ~ x1 + x2)              # a regression plane
> lm(y ~ x1 + x2 + x3)         # linear model with three regressors
> lm(y ~ x – 1)                # SLR w/ an intercept of zero
> lm(y ~ x + I(x^2))             # quadratic regression model
> lm(y ~ x + I(x^2) + I(x^3))    # cubic model
```

In the first example, the model is specified by the formula **`y ~ x`** which implies the linear relationship between **`y`** (dependent/response variable) and **`x`** (independent/predictor variable). The vectors **`x1`**, **`x2`**, and **`x3`** denote possible independent variables used in a multiple linear regression model. For the polynomial regression model examples, the function **`I()`** is used to tell R to treat the variable "as is" (and not to actually compute the quantity).

The **`lm()`** function creates another linear model object from which a wealth of information can be extracted.

Example: consider the **`cars`** dataset. The data give the speed (**`speed`**) of cars and the distances (**`dist`**) taken to come to a complete stop. Here, we will fit a linear regression model using **`speed`** as the independent variable and **`dist`** as the dependent variable (these variables should be plotted first to check for evidence of a linear relation).

To compute the least-squares (assuming that the dataset **`cars`** is loaded and in the workspace), type:

```
> fit <- lm(dist ~ speed)
```

The object **`fit`** is a *linear model object*. To see what it contains, type:

```
> attributes(fit)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"
```

So, from the **`fit`** object, we can extract, for example, the residuals by assessing the variable **`fit$residuals`** (this is useful for a *residual plot* which will be seen shortly).

To get the least squares estimates of the slope and intercept, type

```
> fit

Call:
lm(formula = dist ~ speed)

Coefficients:
(Intercept)        speed
    -17.579        3.932
```

So, the fitted regression model has an intercept of –17.579 and a slope of 3.932. More information about the fitted regression can be obtained by using the **`summary()`** function:

```
> summary(fit)

Call:
lm(formula = dist ~ speed)

Residuals:
    Min      1Q  Median      3Q     Max
-29.069  -9.525  -2.272   9.215  43.201

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -17.5791     6.7584  -2.601   0.0123 *
speed         3.9324     0.4155   9.464 1.49e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 15.38 on 48 degrees of freedom
Multiple R-Squared: 0.6511,     Adjusted R-squared: 0.6438
F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.490e-12
```

In the output above, we get (among other things) residual statistics, standard errors of the least squares estimates and tests of hypotheses on model parameters. In addition, the value for **`Residual standard error`** is the estimate for σ, the standard deviation of the error term in the linear model. A call to **`anova(fit)`** will print the ANOVA table for the regression model:

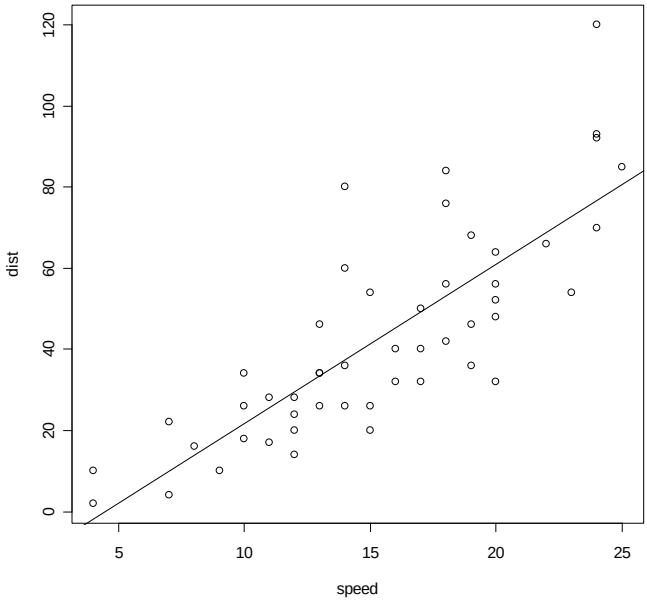
```
> anova(fit)
Analysis of Variance Table

Response: dist
          Df  Sum Sq Mean Sq F value    Pr(>F)
speed      1 21185.5 21185.5  89.567 1.490e-12 ***
Residuals 48 11353.5   236.5
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Confidence intervals for the slope and intercept parameters are a snap with the function **`confint()`**; the default confidence level is 95%:

```
> confint(fit)       # the default is a 95% confidence level
                2.5 %    97.5 %
(Intercept) -31.167850 -3.990340
speed         3.096964  4.767853
```

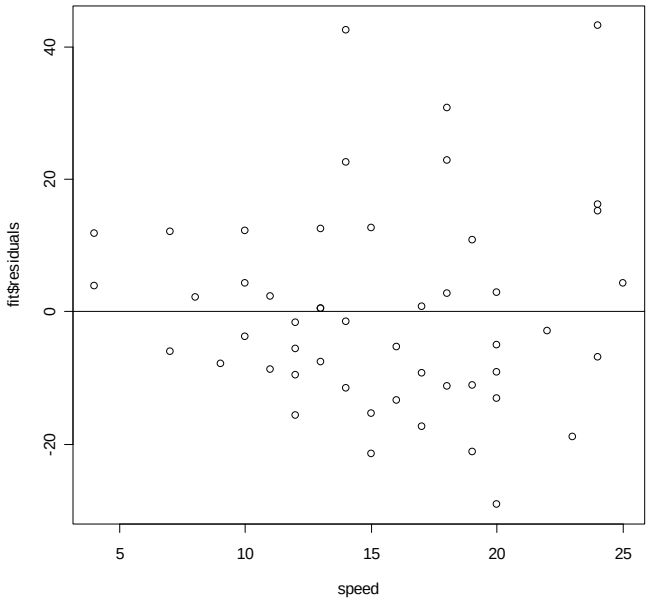
We can add the fitted regression line to a scatterplot:

```
> plot(speed, dist)        # first plot the data
> abline(fit)              # could also type abline(-17.579, 3.932)
```



and a residual plot:

```
> plot(speed, fit$residuals)
> abline(h=0)                  # add a horizontal reference line at 0
```



Lastly, confidence and prediction intervals for specified levels of the independent variable(s) can be calculated by using the **`predict.lm()`** function. This function accepts the linear model object as an argument along with several other optional arguments. As an example, to calculate (default) 95% confidence and prediction intervals for "distance" for cars traveling at speeds of 15 and 23 miles per hour:

```
> predict.lm(fit, newdata=data.frame(speed=c(15,22)),int="conf")
    fit      lwr      upr
1 41.40704 37.02115 45.79292
2 68.93390 61.89630 75.97149

> predict.lm(fit,newdata=data.frame(speed=c(15,22)),int="predict")
    fit      lwr       upr
1 41.40704 10.17482  72.63925
2 68.93390 37.22044 100.64735
```

If values for **newdata** are not specified in the function call, all of the levels of the independent variable are used for the calculation of the confidence/prediction intervals.

7.4 Chi-square Tests

Hypothesis test for count data that use the Pearson Chi-square statistic are available in R. These include the goodness-of-fit tests and those for contingency tables. Each of these are performed by using the **chisq.test()** function. The basic syntax for this function is (see **?chisq.test** for more information):

```
chisq.test(x, y = NULL, correct = TRUE, p = rep(1/length(x),length(x)))

   Arguments:

         x: a vector, table, or matrix.

         y: a vector; ignored if 'x' is a matrix.

   correct: a logical indicating whether to apply continuity correction
            when computing the test statistic.

         p: a vector of probabilities of the same length of 'x'.
```

We will see how to use this function in the next two subsections.

7.4.1 Goodness of Fit

In order to perform the Chi-square goodness of fit test to test the appropriateness of a particular probability model, the vector **x** above contains the tabular counts (if your data vector contains the raw observations that haven't been summarized, you'll need to use the **table()** command to tabulate the counts). The vector **p** contains the probabilities associated with the individual cells. In the default, the value of **p** assumes that all of the cell probabilities are equal.

Example: A die was cast 300 times and the following was observed:

Die face	1	2	3	4	5	6
Frequency	43	49	56	45	66	41

To test that the die is fair, we can use the goodness-of-fit statistic using 1/6 for each cell probability value:

```
> counts <- c(43, 49, 56, 45, 66, 41)
> probs <- rep(1/6, 6)
> chisq.test(counts, p = probs)

        Chi-squared test for given probabilities

data:  counts
X-squared = 8.96, df = 5, p-value = 0.1107

>
```

Note that the output gives the value of the test statistic, the degrees of freedom, and the p-value.

7.4.2 Contingency Tables

The easiest way to analyze a tabulated contingency table in R is to enter it as a matrix (again, if you have the raw counts, you can tabulate them using the **`table()`** function).

Example: A random sample of 1000 adults was classified according to sex and whether or not they were color-blind as summarized below:

	Male	*Female*
Normal	442	514
Color-blind	38	6

```
> color.blind <- matrix(c(442, 514, 38, 6), nrow=2, byrow=T)
> color.blind
     [,1] [,2]
[1,]  442  514
[2,]   38    6
>
```

In a contingency table, the row names and column names are meaningful, so we can change these from the defaults:

```
> dimnames(color.blind) <- list(c("normal","c-b"),c("Male","Female"))
> color.blind
       Male Female
normal  442    514
c-b      38      6
```

This was really not necessary, but it does make the **`color.blind`** object look more like a table.

To test if there is a relationship between gender and color-blind incidence, we obtain the values of the chi-square statistic:

```
> chisq.test(color.blind, correct=F)   # no correction for this one

        Pearson's Chi-squared test

data:  color.blind
X-squared = 27.1387, df = 1, p-value = 1.894e-07
```

As with the ANOVA and linear model functions, the **chisq.test()** function actually creates an output object so that other information can be extracted if desired:

```
> out <- chisq.test(color.blind, correct=F)
> attributes(out)
$names
[1] "statistic" "parameter" "p.value"   "method"    "data.name"
[6] "observed"  "expected"  "residuals"

$class
[1] "htest"
```

So, as an example we can extract the expected counts for the contingency table:

```
> out$expected             # these are the expected counts
         Male Female
normal 458.88 497.12
c-b     21.12  22.88
```

7.5 Other Tests

There are many, many more statistical procedures included in R, but most are used in a similar fashion to those discussed in this chapter. Below is a list and description of other common tests (see their corresponding help file for more information):

- **prop.test()**

  Large sample test for a single proportion or to compare two or more proportions that uses a chi-square test statistic. An exact test for the binomial parameter *p* is given by the function **binom.test()**

- **var.test()**

  Performs an F test to compare the variances of two independent samples from normal populations.

- **cor.test()**

  Test for significance of the computed sample correlation value for two vectors. Can perform both parametric and nonparametric tests.

- **`wilcox.test()`**

  One and two-sample (paired and independent samples) nonparametric Wilcoxon tests, using a similar format to **`t.test()`**

- **`kruskal.test()`**

  Performs the Kruskal-Wallis rank sum test (similar to **`lm()`** for the ANVOA model).

- **`friedman.test()`**

  The nonparametric Friedman rank sum test for unreplicated blocked data.

- **`ks.test()`**

  Test to determine if a sample comes from a specified distribution, or test to determine if two samples have the same distribution.

- **`shapiro.test()`**

  The Shapiro-Wilk test of normality for a single sample.

8. Advanced Topics

In this final chapter we will introduce and summarize some advanced features of R – namely, the ability to perform numerical methods (e.g. optimization) and to write scripts and functions in order to facilitate many steps in a procedure.

8.1 Scripts

If you have a long series of commands that you would like to save for future use, you can save all of the lines of code in a file and execute them together using the **`source()`** function. For example, we could type the following statements in a text editor (you *don't* precede a line with a "**`>`**" in the editor):

```
x1 <- rnorm(500)     # Simulate 500 standard normals
x2 <- rnorm(500)     #               ""
x3 <- rnorm(500)     #               ""
y1 <- x1 + x2
y2 <- x2 + x3
r <- cor(y1,y2)
```

If we save the file as **`corsim.R`** on the C: drive, we execute the script by typing

```
> source("C:/corsim.R")
> r
[1] 0.5085203
>
```

Note that not only was the object **`r`** was created, but so was **`x1`**, **`x2`**, **`x3`**, **`y1`**, and **`y2`**.

8.2 Control Flow

R includes the usual control-flow statements (like conditional execution and looping) found in most programming languages. These include (the syntax can be found in the help file accessed by **`?Control`**):

```
if
if else
for
while
repeat
break
next
```

Many of these statements require the evaluation of a logical statement, and these can be expressed using logical operators:

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<, <=`	Less than, less than or equal to
`>, >=`	Greater than, greater than or equal to
`&`	Logical AND
`|`	Logical OR

Some examples of these are given below. The first example checks if the number **x** is greater than 2, and if so the text contained in the quotes is printed on the screen:

```
> x <- rnorm(1)
> if(x > 2) print("This value is more than the 97.72 percentile")
```

The code below creates vectors **x** and **z** and defines 100 entries according to assignments contained within the curly brackets. Since more than one expression is evaluated in the **for** loop, the expressions are contained by **{}**.

```
> n <- 50
> for(i in 1:100)  {
+   x[i] <- mean(rexp(n, rate = .5))
+   z[i] <- (x[i] – 2)/sqrt(2/n)
+ }
>
```

This last bit of code considers a Monte Carlo (MC) estimate of π. The basis of it is as follows: if we generate a coordinate randomly over a square with vertices (1, 1), (1, –1), (–1, 1), and (–1, –1), then the probability that the coordinate lies within the circle with radius 1 centered at the origin (0,0) is π/4. In the code below, **n** is the number of coordinates generated and **s** counts the number of observations contained in the unit circle. Thus, **s/n** is the MC estimate of the probability and **4*s/n** is the MC estimate of π. The code stops when we are within .001 of the true value. Since the function uses MC estimation, the result will be different each time the code executes.

```
> eps <- 1; s <- 0; n <- 0    # initialize values
> while(eps > .001) {
+   n <- n + 1
+   x <- runif(1,-1,1)
+   y <- runif(1,-1,1)
+   if(x^2 + y^2 < 1) s <- s + 1
+   pihat <- 4*s/n
+   eps = abs(pihat - pi)
+ }
>

> pihat        # this is our estimate
[1] 3.141343
> n            # this is how many steps it took
[1] 1132
```

8.3 Writing Functions

Probably one of the most powerful aspects of the R language is the ability of a user to write functions. When doing so, many computations can be incorporated into a single function and (unlike with scripts) intermediate variables used during the computation are local to the function and are not saved to the workspace. In addition, functions allow for input values used during a computation that can be changed when the function is executed.

The general format for creating a function is

```
fname <- function(arg1, arg2, ...)  { R code }
```

In the above, **fname** is any allowable object name and **arg1, arg2, ...** are function arguments. As with any R function, they can be assigned default values. When you write a function, it is saved in your workspace as a function object.

Here is a simple example of a user-written function. We made the function a little more readable by adding some comments and spacing to single out the first **if** statement:

```
> f1 <- function(a, b) {
+ # This function returns the maximum of two scalars or the
+ # statement that they are equal.
+ if(is.numeric(c(a,b))) {
+   if(a < b) return(b)
+   if(a > b) return(a)
+   else print("The values are equal")    # could also use cat()
+ }
+ else print("Character inputs not allowed.")
+ }
>
```

The function **f1** takes two values and returns one of several possible things. Observe how the function works: before the values **a** and **b** are compared, it is first determined if they are both numeric. The function **is.numeric()** returns **TRUE** the argument is either real or integer, and **FALSE** otherwise. If this conditional is satisfied, the values are compared. Otherwise, the user gets the warning message. To use the function:

```
> f1(4,7)
[1] 7

> f1(pi,exp(1))
[1] 3.141593

> f1(0,exp(log(0)))
[1] "The values are equal"

> f1("Stephen","Christopher")
[1] "Character inputs not allowed."
```

The function object **f1** will remain in your workspace until you remove it.

To make changes to your function, you can use the **fix()** or **edit()** commands described in Section 3.3.

Here is another example[10]. Below is the formula for calculating the monthly mortgage payment for a home loan:

$$P = A \cdot \frac{r/1200}{1-(1+r/1200)^{-12y}},$$

where **A** is the loan amount, **r** is the nominal interest rate (assumed convertible monthly), and **y** is the number of years for the loan. The value of **P** represents the monthly payment. Below is an R function that computes the monthly payment based on these inputs:

```
> mortgage <- function(A = 100000, r = 6, y = 30)  {
+ P <- A*r/1200/(1-(1+r/1200)^(-12*y))
+ return(round(P, digits = 2))
+ }
```

This function takes three inputs, but we have given them default values. Thus, if we don't give the function any arguments, it will calculate the monthly payment for a $100,000 loan at 6% interest for 30 years. The **round()** function is used to make the result look like money and give the calculation only two decimal points:

```
> mortgage()
[1] 599.55
> mortgage(200000,5.5)     # use default 30 year loan value
[1] 1135.58
> mortage(y = 15)          # D'oh!  Bad spelling...
Error: couldn't find function "mortage"
> mortgage(y = 15)
[1] 843.86
```

8.4 Numerical Methods

It is often the plight of the statistician to encounter a model that requires an iterative technique to estimate unknown parameters using observed data. For likelihood-based approaches, a computer program is usually written to execute some optimization method. But, R includes functions for doing just that. Two important examples are:

- **optimize()**

  Returns the value that minimizes (or maximizes) a function over a specified interval.

- **uniroot()**

  Returns the root (i.e. zero) of a function over a specified interval.

[10] This is from *The New S Language*, by Becker/Chambers/Wilks, Chapman and Hall, London, 1988.

These are both for one-dimensional searches. To use either of these, you create the function (as described in Section 8.3) and the search is performed with respect to the function's first argument. The interval to be searched must be specified as well. The help file for each of these functions provides the details on additional arguments, etc.

As an example, suppose that a random sample of size *n* is to be observed from the Weibull distribution with unknown shape parameter θ and a scale parameter of 1. The pdf for this model is given by

$$f(x) = \theta x^{\theta-1} \exp(-x^{\theta}), \; x > 0, \; \theta > 0.$$

Using standard likelihood methods, the *log likelihood* function is given by

$$l(\theta) = n\log(\theta) + (\theta - 1)\sum_{i=1}^{n} \log(x_i) - \sum_{i=1}^{n} x_i^{\theta}$$

and it has a first derivative equal to

$$l'(\theta) = \frac{n}{\theta} + \sum_{i=1}^{n} \log(x_i) - \sum_{i=1}^{n} x_i^{\theta} \log(x_i).$$

To find $\hat{\theta}$, the maximum likelihood estimate for θ, we require the value of θ that *maximizes* $l(\theta)$, given the data $x_1, x_2, \ldots, x_n$. Equivalently, we could find the *root* of $l'(\theta)$. Clearly, the two functions introduced in this section can do the job. To see these at work, data will be simulated from this Weibull distribution with θ = 4.0:

```
> data <- rweibull(20, shape=4, scale=1)
> data
 [1] 0.6151766 0.9417053 0.8244651 1.3818235 0.9866513 0.6121007 1.1391467
 [8] 0.8711759 1.3590739 0.4826331 1.0755436 1.0318024 1.1728524 0.8062276
[15] 1.3630116 1.2094519 0.7547847 1.1178163 0.6184907 0.9310806
```

Next, we will define the log-likelihood function:

```
> l <- function(theta, x)
+ {
+ return(length(x)*log(theta) + (theta-1)*sum(log(x)) - sum(x^theta))
+ }
```

Thus, $\hat{\theta}$ is defined as the value that maximizes this function. To do this, enter:

```
> optimize(l, interval=c(0,20), x = data, max = T)
$maximum
[1] 3.874208            ← value that achieves the maximum

$objective
[1] -1.834197           ← value of the function at the maximum
```

So, $\hat{\theta} = 3.874208$, which is pretty close to the "true" value 4.0. The interval (0, 20) in the above function call can be considered a "guessed range" for the parameter.

Of course, this same maximum value could also be found by using the function $l'(\theta)$ and **`uniroot()`**.

8.5 Exercises

1. Write a function that:
   - simulates $n = 20$ observations from the normal distribution w/ $\mu = 10$, $\sigma = 2$
   - draws $m = 250$ random samples (each of size 20) w/ replacement from the above sample
   - calculates the mean of each of these samples
   - outputs a histogram of the 250 sample means

2. Rewrite the above function so that the values of n, m, μ, and σ can be chosen differently each time the function executes.

3. Repeat the example in Section 8.4, but define a function **`dl`** for $l'(\theta)$ and use **`uniroot()`** to find the maximum likelihood estimate using the same simulated data.

Appendix: Well-known probability density/mass functions in R

Discrete distributions $p(x)$ denotes the probability mass function

- **Binomial**: let `size` = n and `prob` = p

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x}, x = 0, 1, 2, ..., n$$

- **Geometric**: let `prob` = p

$$p(x) = p(1-p)^x, x = 0, 1, 2, ...$$

- **Hypergeometric**: let m = # of white balls, n = # of black balls, k = # of balls sampled

$$p(x) = \frac{\binom{m}{x}\binom{n}{k-x}}{\binom{m+n}{k}}, x = \text{\# of white balls drawn}$$

- **Negative binomial**: let `size` = n, `prob` = p

$$p(x) = \binom{x+n-1}{n-1} p^n (1-p)^x, x = 0, 1, 2, ...$$

- **Poisson**: let `lambda` = λ

$$p(x) = \frac{\lambda^x}{x!} e^{-\lambda}, x = 0, 1, 2, ...$$

Continuous distributions $f(x)$ denotes the probability density function

- **Beta**: let `shape1` = a, `shape2` = b

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}, 0 \le x \le 1$$

- **Continuous uniform**

$$f(x) = \frac{1}{\max - \min}, \min \le x \le \max$$

- **Exponential**: let `rate` = λ

$$f(x) = \lambda \exp(-\lambda x), x \geq 0$$

- **Gamma**: let `shape` = a, `scale` = s

$$f(x) = \frac{1}{\Gamma(a)s^a} x^{a-1} \exp(-x/s), x \geq 0$$

- **Normal**: let `mean` = μ, `sd` = σ

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right], -\infty < x < \infty$$

- **Weibull**: let `shape` = a, `scale` = b

$$f(x) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} \exp\left[-\left(\frac{x}{b}\right)^a\right], x \geq 0$$

References

Becker, R., Chambers, J., and Wilks, A. (1988). *The New S Language*, Chapman and Hall, London.

Freedman, D. and Diaconis, P. (1981). "On this histogram as a density estimator: L_2 theory," *Zeit. Wahr. ver. Geb*. **57**, p. 453 – 476.

Robert, C. and Casella, G. (1999). *Monte Carlo Statistical Methods*, Springer-Verlag, New York.

Scott, D.W. (1979). "On optimal and data-based histograms," *Biometrika* **66**, p. 605 – 610.

Sturges, H. (1926), "The choice of a class-interval," *J. Amer. Statist. Assoc*. **21**, p. 65 – 66.

Index