

R for Statistics

(the fundamental stuff)

Monash University, Draft 6.2, © Ian Hunt
Not for distribution or citation. November 15, 2020

Contents

1	Introduction	10
1.1	Learning By Doing	11
1.2	Help and the Internet	12
1.3	Installation	13
2	The Basics	14
2.1	Goal	15
2.2	Environment	16
2.3	RStudio	17
2.4	R is like a Calculator	18
2.5	Calculation Order	19
2.6	Sequences	20
2.7	Functions	21
2.8	Random numbers	22
2.9	NA (missing values)	23
2.10	Comments	24
2.11	Strings and Concatenation	25
2.12	String Searches	26
2.13	String Substitutions	28
2.14	Dates	29
2.15	Dates to Strings	30
2.16	Logical Data Types	31
2.17	NULL	32

2.18	HELP	33
2.19	Creating R variables	34
2.20	Simple Numerical Variables	35
2.21	Deleting Variables	37
2.22	Naming and Overwriting Variables	38
2.23	Note: Naming Scripts, Functions and Folders	40
2.24	Entering multiple statements	41
2.25	Vectors and Matrices	42
2.26	Matrix References	43
2.27	Data Frames	44
2.28	Data Frame References	45
2.29	Data Types	46
2.30	The Best Built-in Functions	47
2.31	Applying Functions to Matrices	48
2.32	Applying Functions to Data Frames	49
2.33	Misc. Tricks and Tips	50

3 File Structures 51

3.1	Goal	52
3.2	Directories	53
3.3	Running Source Files	54
3.4	File Paths in R	55
3.5	Easy File Paths in R	56
3.6	Data Files	57
3.7	Links to Data	58

3.8	Writing Data	59
3.9	Reading Data	60
3.10	Reading Data - Tips and Tricks	61
3.11	Getting a Matrix	62
3.12	Row Names and Column Names	63
3.13	Packages	64
4	Data containers: matrices, vectors, data frames and lists	66
4.1	Goal	67
4.2	Creating Matrices and Vectors	68
4.3	Matrix Dimensions	69
4.4	Adding to R Vectors and Matrices	70
4.5	Reshaping a Matrix	71
4.6	Referencing the data that you want	72
4.7	Logical Referencing	74
4.8	Data Generation Tricks - R Vectors	75
4.9	Data Generation Tricks - Matrices	77
4.10	Data Generation Tricks - Replication	78
4.11	Operating on Matrices	79
4.12	Operating on Matrices with Apply	80
4.13	Matrix Row Names and Column Names	81
4.14	Losing Matrix Row Names and Column Names	82
4.15	Lists	83
4.16	Referencing Lists	85
4.17	Operating on Lists 1	86

4.18	Operating on Lists 2	87
4.19	Operating on Lists with Custom Functions	88
4.20	Names for vectors and lists	89
4.21	Data Frames	90
4.22	Operating on Data Frames	91
4.23	Turn a Data Frame into a List	92
4.24	What is a tibble?	93
5	Control Structures	94
5.1	Goal	95
5.2	FOR Loops	96
5.3	FOR Loops - over vectors	97
5.4	WHILE Loops	98
5.5	IF Conditionals	99
5.6	ELSE Conditionals	100
5.7	Nesting Loops and Conditionals	101
5.8	Breaking Loops Midway	102
5.9	Vectorisation vs. Loops	103
5.10	Other	104
6	“Random” Numbers	105
6.1	Goal	106
6.2	Generating Random Numbers	107
6.3	Random Normal Numbers	108
6.4	Other Useful Random Numbers	109

6.5	Random Samples	110
6.6	Setting the “Seed”	111
7	Linear Algebra	113
7.1	Goal	114
7.2	Element-by-Element Operations	115
7.3	Element-by-Element Operations - Two Matrices	116
7.4	Element-by-Element Functions	117
7.5	Advanced Element-by-Element Operations	118
7.6	Proper Linear Algebra	119
7.7	Matrix Multiplication	120
7.8	Matrix Addition and Subtraction	121
7.9	Inverses	122
7.10	Transpose	123
7.11	Trace	124
7.12	More Examples	125
8	Relations and Logic	126
8.1	Goal	127
8.2	Relations	128
8.3	Logic	129
8.4	Matrix Checks	130
8.5	TRUE or FALSE	131
8.6	Logical Referencing	133
8.7	Example: Missing Values	134

8.8	More Examples	136
9	Functions	137
9.1	Goal	138
9.2	R functions	139
9.3	Functions Must Be Loaded	140
9.4	A Test Function	141
9.5	A Test Function with Inputs	142
9.6	Functions with Multiple Inputs and Outputs	143
9.7	Entering Inputs	144
9.8	Default Inputs	145
9.9	Variable Scope and Passing Inputs Into Functions	146
9.10	Variable Scope and Outputs	147
9.11	Function Rules-of-Thumb	148
9.12	More Complex Examples	149
10	Optimisation	150
10.1	Goal	151
10.2	Optimising What?	152
10.3	Technical Tips	153
10.4	Typical Output	154
10.5	Routine Options	155
10.6	Useful R Optimisation Functions	156
10.7	Simple Optimisation Example	157
10.8	More Complex Optimisation 1	158

10.9 More Complex Optimisation 2	160
10.10 Quadratic Example	162
10.11 Quadratic Example Code	163
10.12 Root Finding	164
11 Charts	165
11.1 Goal	166
11.2 Basic Plots	167
11.3 Other Charts	168
11.4 Multiple Plots per Window	169
11.5 Titles and Labels	170
11.6 Multiple Points and Lines	171
11.7 Colours, Widths and Characters etc	172
11.8 Fitting All the Data In	173
11.9 Saving and Deleting Charts	174
11.10 Advanced	176
12 Scripts	177
12.1 Goal	178
12.2 Basics	179
12.3 Debugging	180
12.4 Variable Scope	181
12.5 Planning a Script	182
12.6 Miscellaneous Advice	183
13 Basic Probability	184

13.1 Goal	185
13.2 Probability Distributions	186
14 Self Study List	187
14.1 A List	188
15 Miscellaneous	190
15.1 Extensions	191
15.2 Tidyverse	192
15.3 Running Text as R Code	193
15.4 Setting-up and Cleaning a Script	194
15.5 Loading Functions from Files	195

1 Introduction

1.1 Learning By Doing

“People have now-a-days got a strange opinion that everything should be taught by lectures. Now, I cannot see that lectures can do so much as reading the books from which the lectures are taken. I know nothing that can be best taught by lectures, except where experiments are to be shewn. You may teach chemistry by lectures:– You might teach the making of shoes by lectures!”, attributed to Samuel Johnson by his biographer, Boswell.

I expect you to do as much “stitching” during our class time as you can, otherwise you might as well be reading these notes at home or at a café (which I encourage). This means *typing* in code examples and, most importantly, fiddling with code and running your own experiments.

Experiencing the limits of what you can get your computer to do is satisfying. And working in class will save you time on the when you work through more examples and do real research (much of which might be usefully done in proverbial café style).

1.2 Help and the Internet

“There is no limit to what people will believe if they see it on the internet”, attributed to Alan Turing, 1936, by <https://www.dpmms.cam.ac.uk/~twk/>.

There is no need to believe anything you read about programming — you can see for yourself what works. You will learn by doing. The best resource when you are stuck with an error, or planning something new, is the R help documentation. This is installed on your machine. We will use this a lot.

Excellent internet resources do exist, especially for open-source languages. The best place to search for answers to issues (your search engine will do it for you) is within `stackoverflow.com`.

1.3 Installation

In class you will need a computer with an installation of R (have your battery charged, but don't forget a power cord). R is free and should be downloaded from `cran.rstudio.com` for Macs, Linux and Windows operating systems.

After you install R, please also install R-Studio which helps organise and run R scripts. R-Studio is also free and can be found at `www.rstudio.com/products/rstudio/download/#download`.

The only piece of code that will not work on Macs is `windows` to open new chart windows. On a Mac use the function `quartz` in place of `windows`.

2 The Basics

2.1 Goal

Take your first, and most important, steps in R.

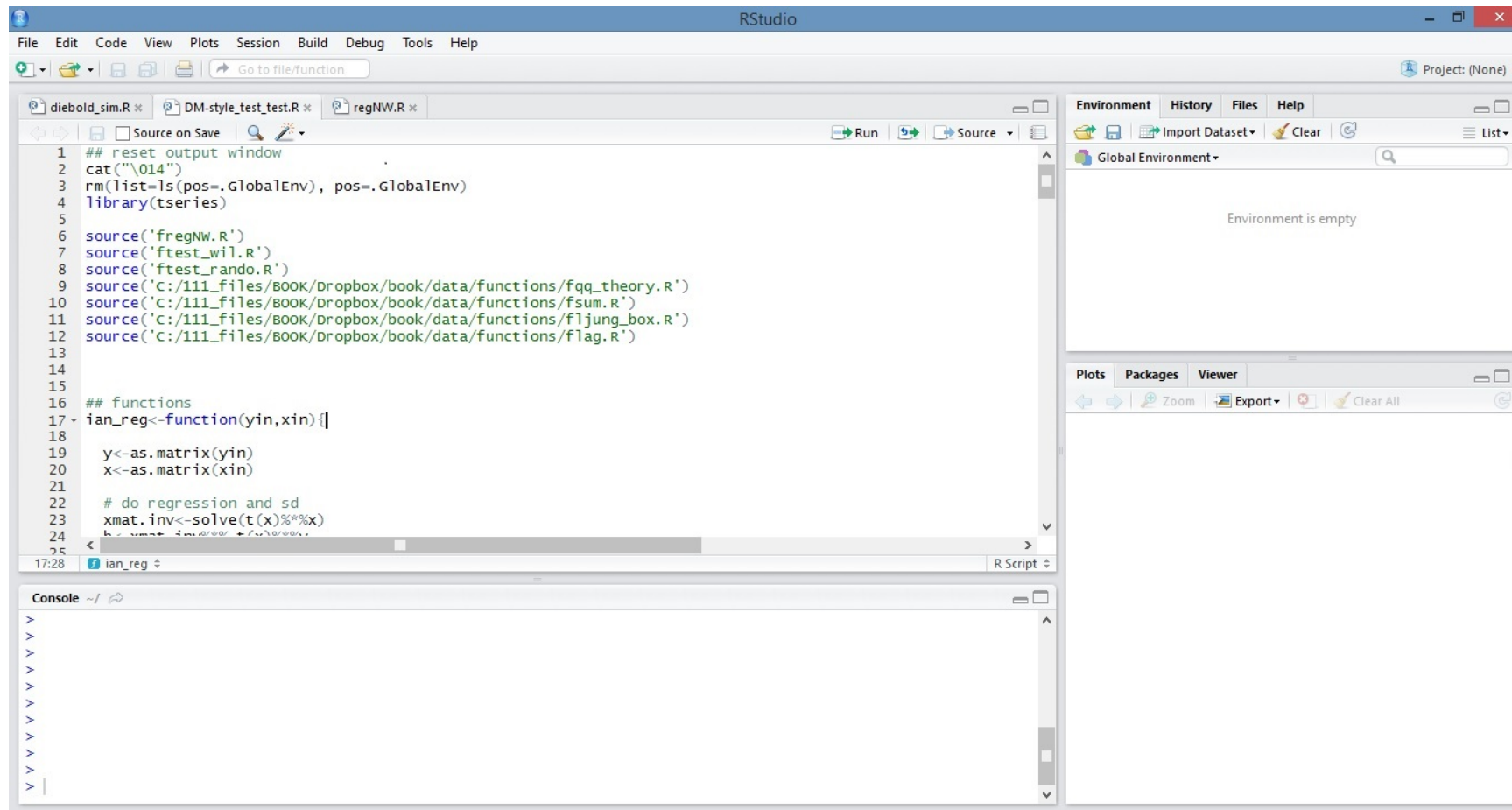
- How to start R.
- Do a few simple calculations.
- Understand the breadth and power of the R programming language.

2.2 Environment

I assume that you have RStudio installed on your machine. We will work with several different windows within its interface.


- Top left: the Source files, where you write and store your code.
- Bottom left: the Console window. This is where the action is.
- Top right: various windows, like a listing of variables, help and console history.
- Bottom right: various windows, notably: the default place for charts.

2.3 RStudio



2.4 R is like a Calculator

For example, you just need to type the expression that you want to evaluate. For example, suppose you want to calculate $1+2*3$. You simply type it in at the Console prompt (“>”) as follows,



```
1+2*3
```

and press the return key.

2.5 Calculation Order

Be sure to use round brackets to get the order of calculation as you require.

$(1+2)*3$

$1+(2*3)$

2.6 Sequences

The operator colon (“:”) is a convenient way to produce sequences. With just two numbers, say $x:y$, a sequence between x and y is returned.

For example

```
1:10  
10:-90
```

You will use this a lot in your R applications.

2.7 Functions

Using R effectively means running functions on data as well as typing in the maths directly. Functions simply carry out a set of operations on the *input* and deliver an *output*. Note that function inputs are often referred to as “*arguments*”.

You “call” or run a function by using its name and supplying the input it needs within round brackets. For example `c` concatenates (joins) things together into a vector and `sum` performs addition.

```
c(1,2,3)
c(1:10)
sum(1:3)
sum(c(1,2,3))
```

For functions with multiple arguments you can either get the order of the inputs precise or identify each argument by name (see examples later).

2.8 Random numbers ...

A particularly useful function is one which produces “random numbers”. These are useful to generate data for examples, among many other things. More than one input is usually needed — multiple inputs for any function can be specified by order or explicitly using the input variable name.

```
rmnorm(10,0,1) # 10 random normal numbers with mean=0 and sd=1
rmnorm(n=10,mean=0,sd=1) # same thing with inputs
rmnorm(sd=1,n=10,mean=0) # same thing with inputs shuffled (using names)
rmnorm(1,10,0) # but what is this?

hist(rmnorm(10,0,1)) # a function nested within a function!
```

2.9 NA (missing values)

R uses the place-holder `NA` for missing values, which often occur in financial time series. Any numerical function that includes an `NA` returns `NA`. This turns out to be very useful ...

```
NA
NA + 1
NA * 100
sum(c(1,2,3,NA))
```

Note that `NaN` (“not a number” - like zero divided by zero) is very similar to `NA` and works in similar fashion as a place-holder.

2.10 Comments

You can and should provide many comments (explanations and notes) in your code. This is so that you don't get lost or waste your time when developing and revisiting code. It also enables other people to understand and use your code.

Comments are made after typing “#”. For example

```
(1+2)*3 # this is a trivial calculation but important note
```

R ignores anything directly following the symbol #.

2.11 Strings and Concatenation

The two main types of data in R are numeric (numbers) and strings (words, if you like). Numbers are entered in an obvious fashion. Strings are entered by surrounding a word by a single quote mark, but in most contexts double quotes work as well.

```
'a string'  
"also a plain string"
```

There are many string manipulation functions in R, and handling them can be more tricky than you might think. However, we will stick to concatenating (i.e. joining) strings using `paste` or `paste0`. The first of these allows for different separations, and the second simply pastes everything together with no gaps. For example

```
paste('We','play','all','day','!')  
paste('We ','play ','all ','day!',sep="")  
paste('example',1,'.xls',sep="XXX")  
paste0('example',1,'.xls')
```

2.12 String Searches

R programming allows for extensive string operations. This may be of use to some scientists, such as biologists and geneticists working on DNA codes. In finance, strings only tend to be used to *organise* our data; this is still useful, for example when we work with data matrices in which each column is a different asset.

The main string search function we will use is `grep`, which searches for a specified pattern within a string or vector of strings. If it finds one, or more matches, then the indexed location of the match is returned. Otherwise nothing is returned.

```
grep('hel',c('hello'))  
grep('a',c('hello'))  
grep('LL',c('hello','bob the builder','HELLBOY is a good comic'))  
grep('LL',c('hello','bob the builder','HELLBOY is a good comic')  
,ignore.case=TRUE)
```

Note: in order to make a search case insensitive we can set the argument “`ignore.case`” to `TRUE`.

There are many possible qualifications to string searches: see the help file at [?regex](#) for a listing of so-called “regular expressions”.

We will only use “wild card” (anything), `.*`, “must begin with”, `^`, and “must end with”, `$`. For example

```
# hell anywhere
grep('hell',c('hello and goodbye','i don't know hellboy'))
# starts with hell
grep('^hell',c('hello and goodbye','i don't know hellboy'))
# ends in 'y'
grep('y$',c('hello and goodbye','i don't know hellboy'))
# bits (h, lo, g and y) in order, anything between
grep('^h.*lo.*g.*y',c('goodbye and hello','hello and goodbye'))
# begin with h, anything, end in e
grep('^h.*e$',c('hello and goodbye','hellboy'))
```

2.13 String Substitutions

The function `gsub` works in the same way as `grep` but changes the matched string according to an additional 'replacement' argument.

```
gsub('hello',replacement='goodbye',c('hello and goodbye',"hellboy hello"))

# same as above (arguments recognised by order of inputs)
gsub('hello','goodbye',c('hello and goodbye',"hellboy hello"))

gsub('^hell','heaven',c('hello and goodbye',"i don't know hellboy"))
```

Use such functions carefully and sparingly, otherwise you will make nonsense.

2.14 Dates

Dates are tricky on computers: precise syntax and formats are required (for any programming language). Dates in R can be stored as numbers, strings or special data variables. The easiest function to use is `as.Date`, which converts strings to dates. It is crucial that the format of `datetime` matches the format of the string exactly.

For example

```
'2001-01-01'# ISO date format as a string (most languages recognise this)
as.Date('2001-01-01')
as.Date('2001-01-01')+1
as.Date('01/12/2001')+1
as.Date('01/12/2001',format="%d/%m/%Y")+1
```

Advanced date manipulation may require `POSIXlt` and `POSIXct`.

2.15 Dates to Strings

We will usually stick to storing dates as strings, making conversions as required for outputs (many financial applications involve time series). A simple way to convert dates to strings is with `as.character`.

```
as.character(as.Date('2001-03-01'))  
as.character(as.Date('01/12/2001',format="%d/%m/%Y"))  
as.character(as.Date('2001-13-01')) # error!
```

2.16 Logical Data Types

We will use first-order logical variables in R. These are represented on the screen as **TRUE** and **FALSE**. If you try a mathematical operation on a logical variable, by default R automatically converts **TRUE** into 1 and **FALSE** into 0.

```
TRUE
```

```
FALSE
```

```
TRUE+1
```

```
FALSE*100
```

```
TRUE+TRUE+FALSE
```

2.17 NULL

Another special value in R is `NULL`. This is place-holder for something that does not exist (or better said, an empty set). Any mathematical operation on `NULL` results in `NULL`.

```
NULL
x<-NULL

x*10
is.null(x)    # function to check for NULL
```


2.18 HELP

Use the help functions as much as you can. This is how you will learn and use R properly; even advanced users look-up help frequently. There are two main ways to search the R code help files: prefixing by `?` (for precise entries) and prefixing by `??` (for every time it is mentioned). For example

```
?mean  
?grep  
??mean # this is a bit messy, depending on the word.
```

By default, in RStudio the help screen appears in the top-right display. You can right-click and 'Open Frame' to make it easier to read.

Also check the internet if you are stuck – <http://stackoverflow.com> articles are excellent.

2.19 Creating R variables

Variables are created by setting the name that you want equal to something (using “<-”). The “something” can be

- A typed-in entry (like some numbers, a formula or a matrix).
- The output from a R function.
- The output from a function that you have created.

In this section we concentrate on the first type. The others will follow later (quite quickly), but the all the same rules apply.

2.20 Simple Numerical Variables

```
x<-1  
y<-1:10  
z<-1+2*3  
  
x;  
y;  
z
```


Variable names can be used to refer to the results of the previous computations and inputs. For example computing $4x$ can be done with

```
4*x
```

You can also assign this result to a new variable, $x2$ say:

```
x2<-4*x
```

You can view all your variables by typing



```
ls()
```

All objects and variables can also be seen in Environment window on the top-right corner of RStudio.

Quitting R destroys all your objects (which is fine - you should save your code and be able to recreate them in a flash).

2.21 Deleting Variables

You can delete all your variables with the function `rm`.

```
x<-1
y<-2
z<-3
ls()

rm(x)
ls()

rm(y,z)  # more than one at a time
ls()

# remove everything and start again (just run all your code again!)
rm(list=ls(pos=.GlobalEnv), pos=.GlobalEnv)
```

Use this carefully, there is no going back. **But** you have your source data files and your code script — this is the power of a “scripting language”.

2.22 Naming and Overwriting Variables

You are the boss: R variables can be created at will. But they will be overwritten if you use the same name (you may desire this).

Be sure to use a sensible naming system. Some rules-of-thumb follow.

- You cannot have names with gaps in them.
- Don't use punctuation or special characters (e.g. \$ or !).
- Don't start a name with a number.
- Use capitals where appropriate.
- Join name parts with “_” or capital letters.

Finally, try not to use reserved R words or existing *functions* (you can check for existing names using the help command [?](#)).

```
# mean is an R function
mean
mean(1:10)

# but you are free to do the following
mean<-10
mean(1:10)

# the function call may work, BUT this is still not good
# so delete your object ... and find a better name
rm(mean)
MYmean<-10
mean(1:10)
```

2.23 Note: Naming Scripts, Functions and Folders

You need to find names for R scripts and functions, which you will store in folders. You should conform to the same principles as variable names, in particular for file and folder names:

- Try to avoid gaps e.g. use “E:\R\project1” instead of “E:\R\project 1”.
- Don’t use punctuation or special characters e.g. \$ or !.
- Don’t start a name with a number.

2.24 Entering multiple statements

Use a semicolon (“;”) to join multiple lines of code together. For example

```
x=1 ; y=2 ; z=3
```

2.25 Vectors and Matrices

A vector has one dimension and a matrix at least two. Each piece of data within a matrix or vector is called an “*element*”. We will only deal with two dimensional matrices — rows and columns. The function to make a vector is `c` and to make a matrix we typically use `matrix`, carefully specifying how many rows and columns we want if we need to.

A vector or matrix can only hold the same *type* of data i.e. one of numeric, dates, characters etc.

There is often useful default behaviour, but this can sometimes be confusing.

```
x<-c(1:16)
y<-matrix(1:16,nrow=4,ncol=4)

x; y

z<-c('hello',1:10) # defaults to converting everything to strings ...
z
```

2.26 Matrix References

Referencing the data from matrices and vectors is most easily done by using “*index numbers*” within square brackets. For a matrix you need two index numbers per element — the row and column. We can reference more than one element at a time. For example

```
x<-c(1:16)
y<-matrix(1:16,nrow=4,ncol=4)

x[1]
x[14:16]

y[1,1]*10
y[4,4]
y[c(1,2,3,4),4]/4
```

There are more complicated ways to reference the elements you want from matrices — we will introduce these methods in this course as required.

2.27 Data Frames

Data frame objects can contain different types of data in one structure. We can use the function `data.frame` to create data frames. Note that we will need to set the argument “`stringsAsFactors`” to `FALSE` or else strings (a collection of characters) get converted into what R calls a “factor” type of data.

For example

```
x<-c(1,2)
y<-paste0('duck',x)

z<-data.frame(x,y,stringsAsFactors=FALSE)
zFACTOR<-data.frame(x,y)

z
zFACTOR # looks the same but see referencing on next page
```

2.28 Data Frame References

Referring to parts of data frames can be done by typing in the data frame, then `$` followed by the column name you want. The column names are set when you create a data frame (there is default behaviour or you can set them yourself).

```
x<-c(1,2) ; y<-paste0('duck',x)
z<-data.frame(x,y,stringsAsFactors=FALSE)
zFACTOR<-data.frame(x,y)
z3<-data.frame(colONE=x,colTWO=y)

z$y
zFACTOR$y

z$x*10
z3$colONE*10
```

2.29 Data Types

The main species of data types you will use are `factor`, `numeric`, `character` and `logical`. You can check an entire data frame by using `str` or use “is.” followed by the data type name (`is.numeric` and so on).

Data types can be created or converted, where feasible, using “as.” (`as.factor` and so on).

```
df<-data.frame(x=c(1.01,2,8),y=rep(TRUE,3),z=c("duck","la","de"))
str(df) # indicates data types
df

is.numeric(df$x)
is.numeric(df$y)
as.character(df$x)
as.factor(df$z)
```

2.30 The Best Built-in Functions

There are some fantastic R functions that we will use often. These include `mean`, `var`, `max`, `min`, `abs`, `prod`, `cumprod`, `sum`, `cumsum`, `exp`, `log`, `sqrt`, `sign` ...

These functions typically all apply directly to individual elements of vectors or work across all elements of a vector.

```
log(1:10)    # applies to each element  
sum(1:10)    # summarises a vector  
cumsum(1:10) # applies across a vector
```

2.31 Applying Functions to Matrices

Most base functions that apply to vectors can also be applied directly to rows or columns of matrices if you specify the dimension (row=1, column=2). We will use the function `apply` to achieve this. The function `apply` requires three arguments: the matrix, the row or column indicator and the function to be applied.

```
x<-matrix(c(1:10),ncol=2)
x

log(x)  # automatically applies to each element
apply(x,1,sum)  # summarizes by rows
apply(x,2,sum)  # summarizes by columns
```

Always check the help file for details (e.g. `?sum`) if you are in doubt about how a function works.

2.32 Applying Functions to Data Frames

You can apply functions to rows and columns to data frames in the same fashion as matrices. The problem is that the function must work for all the data types used.

```
xdata<-data.frame(x=1:4,y=rnorm(4))
xdata

apply(xdata,1,sum)  # summarizes by rows
apply(xdata,2,sum)  # summarizes by columns

xdata_extra<-data.frame(group=c('A','A','B','B'),x=1:4,y=rnorm(4))
xdata_extra
str(xdata_extra)

apply(xdata_extra,2,sum) # error!!  check why (what is the function?)
apply(xdata_extra[,2:3],2,sum) # ok: only columns 2 and 3 go to function
```

2.33 Misc. Tricks and Tips

- Regularly use the function call `cat("\014")` to clear the Console window of RStudio (this will help code to run quickly). Pressing CTRL+L does the same trick.
- Comment blocks in scripts by highlighting and using the keyboard combination CTRL+SHIFT+C and uncomment blocks with the same command again.
- Run specific lines in scripts selected by the cursor (i.e. highlighted text) by pressing CTRL+RETURN on the keyboard.

3 File Structures

3.1 Goal

In this section you will learn how to create and save files with RStudio. Including the following formats.

- Source code in the form of *scripts* and *functions* that run many lines of R code — by convention with a `.R` suffix)
- Storage of R data, variables and functions — by convention with a `.RData` suffix.
- Data, usually numbers with row names and column names — usually with a `.csv` suffix, however formatted.

3.2 Directories

It is easiest to use an R “working directory” to arrange your work. Find a main directory and folder that you are happy with. Then save a RStudio “project” within it. You can then add folders with data and functions.

Each time you want to use your code you need to open *the project file* and go from there. When you open a session in this way the working directory in R is set to the folder in which the project resides.

You can see the working directory with the command `getwd()`.

```
getwd()
```

Note that `setwd` sets a new working directory, according to a file path.

3.3 Running Source Files

Source files can be scripts (any combination of R code) or simply saved functions you have written yourself. By default they should be saved as “.R” files.

You can run source files (as if you have typed them in and run them yourself during a session) by using the function `source`. For example, assume that you have saved a file called “test_script.R” in your working directory, the following code will run it.

```
source('test_script.R')
```

3.4 File Paths in R

Files from any valid file path can be run using `source`. However, typing file paths in R needs to be done with care, because the symbol “\” is reserved for special purposes. The easiest substitute is to use “//” in place of “\”.

For example, assume you have some code saved in “E:\Rproject1\test.R”. Then you simply need to run:

```
filetorun<-'E://Rproject1//test_script.R'  
source(filetorun)
```

This is a particular problem using Microsoft Windows ...

3.5 Easy File Paths in R

Accessing files from R only requires a *relative* reference to be made from the working directory. Arranging your work into different directories is the easiest and cleanest way to use R for multiple projects.

For example, assume you have set-up folders called “data” and “functions” in your working directory. Then all you need to do is use the function `file.path` to load and run files from these sub-folders (you can use “\” or “/” within this function if you want). For example:

```
# full path
myfunction<-file.path(getwd(),'functions/freg.R')
source(filetorun)

# quicker relative reference from working directory
source('functions/freg.R')

list.files(file.path(getwd(),'functions'))
```

Note the final example used `list.files` to display all files ...

3.6 Data Files

Objects (like variables, functions and data) from a session can be saved in separate `.RData` files. You can save multiple objects or just those you want. By default the files will be saved in your working directory (you can supply a full file path if you wish). For example

```
x<-1;y<-1:20
save(x,file='testx.RData')
rm(x)
x

# it was deleted so reload from file
load ('testx.RData')
x

# try with multiple variables and data folder
save(x,y,file=file.path(getwd(),'data/test.RData'))
```

3.7 Links to Data

A ubiquitous data container is Microsoft EXCEL. We will stick to using “.CSV” files that can be easily read and written from EXCEL. Use the EXCEL help file to learn how to create a CSV from within EXCEL.

Be aware of your system “locale”. In France, for example, a .CSV file has commas for decimal places and semicolons for column separators. This is no problem in R, we just need to tell R what we want and what to expect.

3.8 Writing Data

The best way to make data output files is with `write.table`. By default, this function makes a file in your working directory (but, as ever, a full file path can be specified).

The arguments `sep` and `dec` determine the column separator and decimal place sign respectively. These are crucial.

```
x<-matrix(rnorm(100),ncol=5)
write.table(x, file = "random.csv", sep = ",",dec = ".")
write.table(x, file = "randomFRENCH.csv", sep = ";",dec = ",")

# with a file path ...
write.table(x, file = file.path(getwd(),'data/randomFRENCH.csv')
, sep = ";",dec = ",")
```

This function can output both matrices and data frames.

3.9 Reading Data

The best way to upload data into R is with the function `read.table`. By default, it looks in your working directory (but, again, a full file path can be specified).

```
x<-matrix(rnorm(100),ncol=5)
write.table(x, file = "random.csv", sep = ",",dec = ".")
write.table(x, file = "randomFRENCH.csv", sep = ";",dec = ",")

# upload
read.table(file = "random.csv", sep = ",",dec = ".")
read.table(file = "randomFRENCH.csv", sep = ";",dec = ",")
```

This function creates data frames. Use the help file (`?read.table`) to determine its use for mixed data (numbers and characters, for instance).

3.10 Reading Data - Tips and Tricks

Reading data can be tricky in any language. Use the help file in case of any errors. Miscellaneous tips using `read.table`, include the following.

- Make sure you have one less header than columns in your data file if you want row names! To force column headers set `header=TRUE`.
- To avoid problems with quote characters set `quote = ""`.
- If you have rownames in column 1 then set `row.names=c(1)`.
- Special characters cannot be in rownames and colnames and will be converted to a ".". Examples include "'" (when `quote = ""`), "/", "\$" and "&".
- Stop conversion of rownames and colnames by setting `check.names=FALSE`.
- Force strings to stay as strings with `stringsAsFactors=FALSE`.
- Convert empty strings "" to NA using `na.strings=""`.

3.11 Getting a Matrix

You can easily convert a data frame into a matrix, assuming the data is all of the same type. This is most easily achieved with `as.matrix`. For example:

```
x<-matrix(rnorm(100),ncol=5) # a random N(0,1) matrix (see later in course)
write.table(x, file = "random.csv", sep = ",",dec = ".")

xdf<-read.table(file = "random.csv", sep = ",",dec = ".")
xmat<-as.matrix(xdf)

is.data.frame(xdf) # ask R to check the type of object
is.matrix(xdf)
```

3.12 Row Names and Column Names

Matrices often have row names and column names. Look carefully at the files in the `write.table` example — default row names and column names were added by R.

```
x<-matrix(rnorm(100),ncol=5)
colnames(x)<-c("A","B","C","DUCK","E")
rownames(x)<-paste0("this is row ",1:20)

write.table(x, file = "randomNONAMES.csv", sep = ",",dec = ".",
row.names = FALSE,col.names = FALSE)

write.table(x, file = "random.csv", sep = ",",dec = ".")

read.table(file = "randomNONAMES.csv", sep = ",",dec = ".")
as.matrix(read.table(file = "random.csv", sep = ",",dec = "."))
```

Notice that the functions `colnames` and `rownames` were used to set the initial names.

3.13 Packages

There are many so-called “packages” for R that you can load onto your computer. A package is basically a bunch of R code that is supposed to do something useful, over and above the base installation of R (which comes with a variety of packages as well). In this course we will almost exclusively be using the base installation. But loading packages is easy, if your computer has access to the internet.

The `library()` command displays all packages on your machine. Using `library()` with the name of a package in the brackets loads the package. Use `search()` to see which packages are loaded. Use `install.packages` to download a new package onto your machine (watch for inter-package dependencies ...). To unload a package you can try `detach`.

Note that there is also a user interface for packages and download sites in RStudio. Also, make sure you read background info about packages on cran.r-project.org.

See next page for examples.


```
library() # displays all active packages
search()
library(datasets) # this loads the datasets package (must already be installed)
ls("package:datasets") # lists the bits and pieces within the package

detach("package:datasets",unload=TRUE) # unload it without quitting

install.packages("glmnet") # lasso regression etc.
```

Also see [require](#).

4 Data containers: matrices, vectors, data frames and lists

4.1 Goal

In this section you will learn how to create and handle matrices in R.

4.2 Creating Matrices and Vectors

A matrix is a particular species of variable that can hold a lot of data. We will only work with matrices of two dimensions (say, T rows and K columns). Call them what you will (n, p, m, r, T, K or whatever), reference to the dimensions is by convention *row* first and *column* second e.g. a 2 by 3 matrix means one with two rows and three columns.

A “vector” in linear algebra is just a matrix where *either* T or K is one. But a “vector *object*” in R has no row or column dimensions.

The simplest way to create an R vector is to type it into R using `c`.

```
x<-c(1,2,3,4,5,6)
y<-as.matrix(x,ncol=1) # still a vector in terms of linear algebra
z<-matrix(c(1,2,3,4,5,6),nrow=2)
```

4.3 Matrix Dimensions

The dimensions of a matrix are found with the command `dim`. This returns `NULL` for a vector. The function `length` counts the number of elements in a vector or matrix. For example

```
x<-c(1:5)
dim(x)
length(x)

y<-as.matrix(x,ncol=1)
dim(y)
length(y)

z<-matrix(1:100,ncol=4)
dim(z)
length(z)
```

You can find particular dimensions of a matrix using `nrow` and `ncol`.

4.4 Adding to R Vectors and Matrices

Adding to R vectors is done with `c`. Adding rows to matrices is done with `rbind` and adding columns is done with `cbind`. Matrix variables with *matching* dimensions can be easily combined.

```
x<-c(1:4)
y<-matrix(1:12,ncol=6)
z<-matrix(1:18,ncol=6)

c(x,5:10)
rbind(y,z)
cbind(y,z) # error: dimensions not compatible
cbind(y,c(999,-999))
```

4.5 Reshaping a Matrix

A trick that is sometimes useful is to “reshape” a matrix into another. The function `matrix` is able to mould a matrix into a different form *if the dimensions are compatible*. For example

```
x<-matrix(1:12,nrow=2 , ncol=6 ) # a 2 by 6 matrix
y<-matrix(x,ncol=1) # 12 by 1
z<-matrix(x,nrow=3) # 3 by 4

matrix(x,nrow=5) # error: not feasible

as.vector(x) # make an R vector from columns
matrix(as.vector(x),nrow=3) # same as z
```

The data for reshaping is taken column-by-column, so be careful that a reshaped matrix has ordered the data in the way you want.

4.6 Referencing the data that you want

Referencing the elements of data in matrices looks complicated but is in fact simple. You use square brackets after the matrix name, inserting the parts of the matrix that you want to reference within the brackets. You need to specify which part of *each* dimension that you want (except for a vector, which needs just one reference, but can take both as well). For example take the following matrix

```
x<-matrix(1:18,nrow=6)
```

To reference the element in the second row and third column you type

```
x[2,3]
```

To reference all the numbers in a column or row use a blank space. For example

```
x[2,]          # all data in row 2  
x[,1]          # all data in column 1
```


To reference a subset of rows or columns you simply use an R vector with index numbers for the parts that you want. For example

```
x[c(1,2),c(2,3)]      # rows 1 and 2, cols 2 and 3
```

Notice that you can use vector variables to do the referencing for you

```
rowstoget<-c(1,2)  
colstoget<-c(2,3)  
x[rowstoget,colstoget] # rows 1 and 2, cols 2 and 3
```

Finally, we can use `nrow` and `ncol` to reference the end of a dimension. For example

```
x<-matrix(1:18,nrow=6)  
  
x[1:nrow(x) , ]  
x[1:nrow(x),1:ncol(x)]  
x[2:nrow(x),1:(ncol(x)-1)]
```

4.7 Logical Referencing

You can also get the data that you want by using *logical* referencing, rather than pointing to elements with index values. This very useful method requires *logical* variables, either **TRUE** or **FALSE**, to indicate which elements to return. Details about logical variables and how to use them for referencing is covered in sections 8.5 and 8.6.

4.8 Data Generation Tricks - R Vectors

There are a variety of tricks to create ready-made matrices and vectors. Useful vector techniques include using two numbers split by “:” and the function `seq`. Random numbers, which will cover in detail later, can easily be generated, too. For example, with `rnorm`.

```
nstart<-1;nend<-10  
  
nstart:nend  
seq(from=1,to=10)  
seq(from=1,to=10,by=.1)  
  
rnorm(100,0,1)  # 100 N(0,1) random numbers
```

Use `seq` carefully or consider setting the desired `length.out` argument

```
seq(0,10,by=1/10)    # check length  
seq.int(0,10,length.out=100)  # check length  
length(seq(0,10,by=1/10))
```

4.9 Data Generation Tricks - Matrices

Matrices can be easily generated with the command `matrix`, a number and the desired number of rows and columns.

```
nrows<-5;ncols<-10;  
matrix(1,nrows,ncols)  
matrix(0,nrows,ncols)
```

We will later use matrices of zeros to *pre-allocate* matrices that will subsequently be filled-up with calculation results.

4.10 Data Generation Tricks - Replication

Data can be replicated in R with the function `rep`. This makes multiple copies of an existing matrix and R vector, for example.

```
x<-1:10          # R vector length=10
rep(x,2)         # R vector length=20
matrix(rep(x,10),ncol=10)  # 10 by 10 matrix
y<-matrix(1:10,ncol=2)
rep(y,2)
matrix(rep(y,2),ncol=ncol(y)*2)
```

4.11 Operating on Matrices

The main thing to remember is that our matrices have two dimensions (more are possible, but are not usually needed in econometrics and finance). You may wish to operate on elements, just rows, just columns or the matrix itself. Many functions simply operate on elements directly. For row and column operations use the function `apply`, which requires three arguments: the matrix, the row or column indicator and the function to be applied. And for matrix operations we will use linear algebra.

```
x<-matrix(c(1:10),ncol=2) ; x

log(x)  # automatically applies to each element
apply(x,c(1,2),log)  # same result as above ...

apply(x,1,sum)  # summarizes by rows
apply(x,2,sum)  # summarizes by columns
apply(x,c(1,2),sum)  # summarizes element-by-element!

apply(x,2,function(i){max(i)})  # summarizes by columns
apply(x,2,max)  # summarizes by columns
```

4.12 Operating on Matrices with Apply

The function `apply` can take any function – even a custom-built function as its third argument. Beware that custom-built functions can run much slower than built-in functions.

```
x<-matrix(c(1:10),ncol=2)

apply(x,2,sum) # summarizes by columns
apply(x,2,function(i){sum(i)}) # custom function
apply(x,2,function(y){sum((y-mean(y))^2/(length(y)-1))}) # custom function
apply(x,2,var)

tempVAR<-function(x){sum((x-mean(x))^2/(length(x)-1))}
apply(x,2,tempVAR)
```


4.13 Matrix Row Names and Column Names

An important attribute of matrices is the ability to store names for rows and columns. IN econometrics these will typically be date strings and asset names, for example. The easiest way to operate on these names is with the functions `rownames` and `colnames`.

In addition, the function `dimnames` can be used to access the `list` object that stores names.

```
x<-matrix(c(1:12),ncol=3)
x

colnames(x)<-c("asset1","asset2","asset3")
rownames(x)<-as.character(as.Date(1:nrow(x),'2000-01-01'))
x

dimnames(x) # a list object
dimnames(x)[[2]]<-c("asset4","asset5","asset6")
x
```

4.14 Losing Matrix Row Names and Column Names

One wrinkle with names for rows and columns is that if a matrix reference results in only a vector (for example, you select one row or column), some of the name attributes are dropped. Otherwise, row and column names are retained when parts of a matrix are referenced.

```
x<-matrix(c(1:12),ncol=3)
colnames(x)<-c("asset1","asset2","asset3")
rownames(x)<-as.character(as.Date(1:nrow(x),'2000-01-01'))
x[,1:2] # retains names
x[,1]   # loses col names
x[2,]   # loses row names
x[1,1]
```

4.15 Lists

In R, a “list” is a container or object that can hold any number of different objects, of any format or type. This is often useful for storing and operating on matrices of different length. The cleanest way to use lists is to first set up an empty array object and then assign its sub-objects as required (a clean way is to use the command `vector`). Lists have a `length` attribute but no dimensions.

```
xa<-vector("list",3)
xa
length(xa)
```

Another convenient way to create a list is with the command `list`. This is convenient if you wish to *name* the objects within a list.

```
xa2<-list(ob1=NULL,ob2=NULL,ob3=NULL)
xa2 ; length(xa2)

# initialise an empty list and fill as required
x2<-list() ; x2$x<-1:10 ; x2$y<-"hello"
```

4.16 Referencing Lists

The main parts of a list can be accessed via double-square brackets `[[]]`. Depending on what the element of list is, you can then use single-square brackets to reference subsets of data.

```
xa<-vector("list",3)
xa[[1]]<-matrix(1:10,ncol=2)
xa[[2]]<-as.character(as.Date(1:1000,'2000-01-01'))
xa[[3]]<-c("asset1","asset2","asset3")

xa[[1]]
xa[[1]][c(1,3),]
xa[[2]][1:5]
```

4.17 Operating on Lists 1

You can apply functions to each element of a list using a variety of functions. We will use `lapply`. The results come back as list - you can use `unlist` to make them into a vector.

```
x1<-vector("list",3)
x1[[1]]<-matrix(rnorm(100),ncol=5)
x1[[2]]<-matrix(rnorm(100),ncol=5)
x1[[3]]<-matrix(rnorm(100),ncol=5)

lapply(x1,mean)
lapply(x1,sd)
```

4.18 Operating on Lists 2

The function `sapply` produces a formatted output from `lapply`. And the function `unlist` returns a vector from the output `lapply` (or simply from any list).

```
dum<-rnorm(90)
x1<-list(ob1=dum[1:30],ob2=dum[31:60],ob3=dum[61:90])
lapply(x1,mean)
sapply(x1,mean)
unlist(lapply(x1,mean))

unlist(x1) # beware strange things happening to names
```

Unfortunately, strange things can happen with name attributes of matrices and lists that are reshaped. And names can be dropped completely. Stay wary and adjust “by hand” when these things are a problem.

4.19 Operating on Lists with Custom Functions

You can apply custom-built functions with `lapply`, just like with `apply`.

```
x1<-vector("list",3)
x1[[1]]<-matrix(rnorm(100),ncol=5)
x1[[2]]<-matrix(rnorm(10),ncol=10)
x1[[3]]<-matrix(rnorm(1000),ncol=20)

# means of the columns
lapply(x1,function(i){c(apply(i,2,mean))})
```


4.20 Names for vectors and lists

Vectors and lists only have one dimension and a **length**. But they do have a **names** attribute that attaches to each element. You can refer to parts of a list by its name (if a name has a space in it then you can need to enclose it in apostrophes e.g. `` ``).

```
x1<-vector("list",3)
x1[[1]]<-matrix(rnorm(100),ncol=5)
x1[[2]]<-matrix(rnorm(10),ncol=10)
x1[[3]]<-matrix(rnorm(1000),ncol=20)

names(x1)<-c("sample 1","sample 2","sample 2")
x1$`sample 2`

y<-c(3,5,1)
names(y)
names(y)<-c("my score","your score","his score")
y
```

4.21 Data Frames

Data frames in R are very similar to matrices. There are two key differences. First, the rows and columns can have different data types (numeric, factors, characters, dates ...). Secondly, the column names can be used to access data directly. Use `$` to reference data frame columns by name. And use `attach` to enable columns to be referenced simply by their name (afterwards you can use `detach`).

```
x<-as.data.frame(matrix(c(1:12),ncol=3))    # auto row & col names
colnames(x)<-c("asset1","asset2","asset3")
rownames(x)<-as.character(as.Date(1:nrow(x),'2000-01-01'))
x<-cbind(x,rownames(x),as.Date(rownames(x)))

x[,1:2]
x$asset1  # direct name reference

attach(x) # ready for names directly
asset1
detach(x) # can't use just the name now ...
```

4.22 Operating on Data Frames

You can reference data frames in two dimensions, just like matrices. So you can use the `apply` function on the rows or columns of a data frame, as well. But you must take care use operations that suit the data types in each column specified with `apply`.

Since data frames are special types of lists, you can also use `lapply` and so on.

```
df<-data.frame(x=c(1:5),y=rep(2,5),z=c("duck","a1","b","1a","de"))
str(df) # indicates data types

nums<-unlist(lapply(df,is.numeric)) # checks numeric (or can be converted)
nums

apply(x,2,mean) # error
apply(x[,nums],2,mean) # ok (but apply needs a matrix)

apply(x,2,function(i){grep("^d",i)}) # strange things occur by default?
apply(as.matrix(x[,!nums]),2,function(i){grep("^d",i)})
```

4.23 Turn a Data Frame into a List

Data frames are already special types of lists. But a convenient way to subset data from a data frame into basic list is to use the function `split`. Also see `unsplit`

```
intelligence<-runif(5,40,75) # generate some fake scores
species<-c("duck","duck","rabbit","rabbit","rabbit")
weight<-runif(5,1000,2000) # generate some fake measurements
df<-data.frame(intelligence,species,weight)
str(df); df

data_by_species<-split(df,df$species)
str(data_by_species) ;data_by_species

lapply(data_by_species,function(i){apply(i[,c(1,3)],2,mean)})
unsplit(data_by_species, species, drop = FALSE) # go back to data.frame
```

Split data sets in the form of lists are fundamental for doing statistical analysis.

4.24 What is a tibble?

Tibbles is a good name for your cat. A `tibble` is also an “upgraded” data frame (i.e. a list) within the `tidyverse` suite of packages.

5 Control Structures

5.1 Goal

This section explains how to *control* the code execution of R scripts. The goal is to be able to do things like repeat calculations until some conditions are met, or to skip certain lines if a particular condition is not met.

It is convention to indent your code that is written within control structures (this does not affect the RB execution, but it is much easier to read and work on).

5.2 FOR Loops

Loops repeatedly execute the same block of code. First, use the command `for` to begin the loop and define the “iterator” inside round brackets. The iterator is usually in the form of variable that is an index counter, incrementing automatically after each loop. Secondly, enclose the code to repeat within curly brackets `{...}`. If you want to see output from the code then use `cat` or `print`.

```
for (i in 1:10) {i}
for (x in 1:10) {cat(x)}
for (i in 1:10) {print(i)}

for (i in 1:10) {
  out<-i^2
  print(c(i,out))
}

x<-rnorm(10)
for (i in x) {print(i)}
```


5.3 FOR Loops - over vectors

Loops can be over vectors that contain elements that are not simple counters. Though, my preference is to use counters to reference elements.

```
x<-rnorm(10)
# loop over elements directly
for (i in x) {print(i)}

# same thing using an iterator (tidier)
for (i in 1:length(x)) {print(x[i])}
```

5.4 WHILE Loops

Another type of loop is set by the command `while`. In this case the code inside the loop will execute until the condition specified is met. When this goes wrong you will get an infinite loop (!) - which you can try to break out of by typing **ESC** on your keyboard. For example

```
x<-rnorm(10)
counter<-1                                # initialise a counter
while (counter<=length(x)) {
  y<-(x[counter]*100)/100
  print(cbind(x[counter],y))
  counter=counter+1
}
```

We will focus on `for` loops. Even then, we will use them with extreme care: most of the things we want to do in R can be done *directly* on matrices, rather than looping through the elements of a matrix.

5.5 IF Conditionals

The most useful “conditional” is `if`. This allows you check a condition and then choose what happens - for example make a calculation, skip something or set-up a break in a loop. First, you open the `if` by checking some condition inside round brackets. Secondly, if the condition is met then the subsequent code inside curly brackets is run. If the condition is not met.

```
x<-rnorm(10);y<-rnorm(10)
if (sum(abs(x))>sum(abs(y))) {
  biggest<-x; print("x is 'biggest'")
}
if (sum(abs(x))<sum(abs(y))) {
  biggest<-y; print("y is 'biggest'")
}
if (sum(abs(x))==sum(abs(y))) { biggest<-NA; print("a tie") }

biggest
```

5.6 ELSE Conditionals

We can account for all other alternatives from an `if` with the command `else`. The command `else` goes after the closing curly bracket of the `if` and has its own curly brackets. Care must be taken with line breaks (usually try to keep the `if` close bracket on the same line as the `else` open bracket).

```
x<-rnorm(10);y<-rnorm(10)

if (sum(abs(x))>sum(abs(y))) {print("x is 'biggest'")} else {
print("either y is 'biggest' or its a tie")
}
```

Note that you can “nest” more `if` statements within the `else` brackets, but this can be very messy.

5.7 Nesting Loops and Conditionals

Loops and other conditionals can be usefully combined and nested. For example

```
nrows <-10; ncols <- 10; data <- matrix(1,nrows, ncols);

for (i in 1:nrows) {
  for (ii in 1:ncols) {
    if (i == ii) { data[i,ii]<- 2
    } else {
      if (abs(i - ii) == 1){ data[i,ii] <- -1
      } else {
        data[i,ii] <-0}
      } # end of first else
    } # end of ii
  } # end of i
}
data
```

The trick is to keep things as *simple* as possible. Too much nesting is unreadable, slow to process and prone to error.

5.8 Breaking Loops Midway

Loops can be stopped early based on a condition using `break`.

```
lowval<-NULL
for (i in 1:1000) {
  temp<-rnorm(1)
  if (temp<=-2) {
    lowval<-temp
    break
  }
}

i;lowval
```

5.9 Vectorisation vs. Loops

R is slow with loops relative to “lower-level” languages such as C.

The term “vectorisation” refers to writing your code so that matrices are created, altered, accessed and otherwise handled by directly referencing their elements. Typically R runs faster when code is vectorised rather than reliant on loops.

There are three issues here. First, vectorisation is not always faster than loops – you can always experiment to check. Second, it may not be feasible to write everything in vectorised form. Third, loops can sometimes be more “readable” or understandable.

A few rules of thumb are:

- Try to vectorise from the start when managing large matrices.
- Using loops for formatting outputs and results is usually fine.
- If a loop runs unreasonably slow then try to vectorise the procedure in order to speed-up performance.

5.10 Other

Other commands exist to help control your code. These include `switch`, `ifelse`, `try` and `catch` (ensure a script keeps going when trying to execute something that may cause an error) and `break` (to exit a loop after evaluating a condition).

Use the help system to explore these options.

6 “Random” Numbers

6.1 Goal

This section introduces how to generate and manage pseudo random numbers in R. In general, I will refer to this process as “simulation”.

Note that these numbers are not really *random* in a physical sense – R generates random numbers based on highly unpredictable (by us) sequences from deterministic formulas. But we will assume that random numbers from R are “random enough” for our purposes.

6.2 Generating Random Numbers

In general, generating random numbers from a particular distribution function (e.g. the Normal distribution) is simple. The simplest method is produce a `Uniform(0,1)` random number (a real number between 0 and 1) and work backwards from the desired distribution's cumulative density function.

We are not going to go into these details. For our purposes, R has built-in routines that will produce the random numbers that we need.

6.3 Random Normal Numbers

In finance, the ubiquitous and often unfairly maligned probability distribution is the Normal distribution. There are two parameters in the Normal: *mean* and *standard deviation*. Note that standard deviation is the square root of variance. The way we will generate Normal data is with the `rnorm` command, which generates $N(0,1)$ matrices by default, and can be used to generate any mean and standard deviation with its optional second and third inputs. Matrices can be made directly.

```
x<-rnorm(10) # vector of length 10 full of N(0,1) data

x<-matrix(rnorm(10000,.1,.15),ncol=10)
apply(x,2,mean)
sqrt(apply(x,2,var))
```

6.4 Other Useful Random Numbers

There are many other distributions that can be used directly in R — see [?Distributions](#).

```
runif(10,0,1)  # U(0,1) data  
rt(10,df=5)    # t-distribution with degrees of freedom specified (no default)
```

6.5 Random Samples

A convenient function for drawing random samples is `sample`. We can draw samples with or without (the default) replacement.

```
n<-100 # sample size required
x<-1:100 # set of data to sample from

# use sample to randomly select elements
x2<-sample(x,n)
x3<-sample(x,n,replace=TRUE)

cbind(x,sort(x2),sort(x3))
```

6.6 Setting the “Seed”

It is useful to be able to *reproduce* the same random numbers whenever you want. This is important when you want to compare the results of different calculations on the same data (so, for example, you never need to save your simulated data because you can just create everything again using the original code).

R uses the function `set.seed` to set a “seed” for all its random number generations. If you set the seed to a particular number and then generate a *given* set of random numbers, the results will be the same.

```
x<-rnorm(10); y<-rnorm(10)
cbind(x,y)

myseed<-101
set.seed(myseed) ; x<-rnorm(10)
set.seed(myseed) ; y<-rnorm(10)
cbind(x,y)
```

R generates its own seed if you do not set it. R increments all seeds after *each*

random generation task. But you can reset the seed to whatever you want at any time. Finally, a new R session starts with a “random” seed, based partly on the precise clock time that you open R (this is like rolling a die).

7 Linear Algebra

7.1 Goal

In this section you will learn how to do the most useful mathematical operations on matrices. This includes both linear algebra, for example multiplying two matrices, and working on the data within a matrix on an element-by-element basis.

7.2 Element-by-Element Operations

Element-by-element operations operate on the data *within* one matrix. For example Let $x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. The element-by-element square of x is $\begin{bmatrix} 1^2 & 2^2 \\ 3^2 & 4^2 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$.

In R this is:

```
x<-rbind(c(1,2),c(3,4))  
x^2
```

It is also simple to multiply or divide by a scalar (i.e. a number).

```
x<-rbind(c(1,2),c(3,4))  
x*2      # element-by-element  
x+2      # element-by-element
```

7.3 Element-by-Element Operations - Two Matrices

If you want to multiply, add, divide or subtract two matrices then they *must* be the same dimension. For example

```
x<-rbind(c(1,2),c(3,4))
```

```
y<-x
```

```
x*y
```

```
x/y
```

```
x+y
```

```
x-y
```

```
z<-matrix(1:6,ncol=2)
```

```
x+z      # error!
```

7.4 Element-by-Element Functions

Many built-in R functions apply by default on a element-by-element basis. Examples include `log` and `exp`. This is very useful. For example

```
x<-cbind(c(.1,.15),c(.05,.075))  
log(1+x)  
exp(log(1+x))-1
```

However: always check how a function works before using it.

7.5 Advanced Element-by-Element Operations

Sometimes you will want to do *complicated* element-by-element operations that are not automatic. The slowest-but-surest way to do this is to loop through each member of the matrix and do the operation one-by-one. This can be slow if the matrix is large.

The main short-cut method in R to speed this up is to use `apply`. This can still be slow for custom-built functions over millions of elements, however. Basic built-in functions like `sum`, however, work very quickly with `apply`.

7.6 Proper Linear Algebra

The linear algebra we will use in this course is straight-forward. We will multiply, add and subtract matrices. Division of matrices is ambiguous, but we will look at analogue to division with square matrices, which is straight-forward.

Any complex formula that we use can be broken down into a series of steps, with two matrices combining at each step.

The most important thing to know is that in each step the dimensions of the two matrices must be compatible with the operation.

Assume x is a T (rows) by K (columns) matrix and y is some other matrix.

Operation	R Code	Dimension Requirement
add	<code>x+y</code>	x and y must have same no. rows & cols
subtract	<code>x-y</code>	x and y must have same no. rows & cols
multiply	<code>x%*%y</code>	y must have as many rows as x has cols

7.7 Matrix Multiplication

Matrix multiplication is simple. The operation matches each row of the left-hand-side matrix with each column of the right-hand-side matrix. Each element of the result matrix is the sum of the product of corresponding left-hand-side row elements times the right-hand-side column elements. This is best explained by example:

$$\text{let } x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } y = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

$$\text{Then } x * y = \begin{bmatrix} (1 * 1 + 2 * 4) & (1 * 2 + 2 * 5) & (1 * 3 + 2 * 6) \\ (3 * 1 + 4 * 4) & (3 * 2 + 4 * 5) & (3 * 3 + 4 * 6) \end{bmatrix}$$

```
x<-rbind(c(1,2),c(3,4))  
y<-rbind(c(1,2,3),c(4,5,6))  
x%*%y
```

In general, multiplying a N by K matrix with a K by M matrix will result in a N by M matrix; where each element of the resulting matrix is the sum of the products of the corresponding elements of the first two matrices.

7.8 Matrix Addition and Subtraction

Matrix addition and subtraction just adds and subtracts corresponding elements.

7.9 Inverses

The “division” of matrices is more complex than regular algebra. In finance however we often deal with matrix inverses, which are analogous to division operations. In R the main way to take an inverse is to use `solve`. We will only do this for square matrices (a square matrix has the same number of rows as columns). Note that `solve` can be used on its own or by combining it with another matrix which subsequently multiplies the inverse of the first matrix.

```
x<-rbind(c(1,2),c(3,4))
y<-rbind(c(5,6),c(7,8))

solve(x)
solve(x)%*%x      # inverse of x * x
solve(x)%*%y      # inverse of x * y
solve(x,x)        # inverse of x * x
solve(x,y)        # inverse of x * y
```

In practice, `solve(x)%*%y` will usually work fine, but if you get computational errors then try `solve(x,y)`.

7.10 Transpose

A trick we will often use is the matrix *transpose*, which switches the dimensions of a matrix. This is useful to get the dimensions to match for various operations between matrices. In R the `t` transposes a matrix. For example let $x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. The transpose of x is $x' = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$. Note that sometimes this is written with “T” like x^T .

```
x<-matrix(1:4,ncol=2)
y<-matrix(1:10,ncol=2)
t(x)

t(y)
dim(y)
dim(t(y))
```

7.11 Trace

Sometimes you will need an easy way to access the diagonal elements of a square matrix. This is done with the command `diag`. The sum of these elements is called the “trace” of the matrix.

```
x<-t(matrix(1:9,nrow=3))  
diag(x)  
sum(diag(x))    # the trace
```

7.12 More Examples

More examples of valid and invalid matrix operations include the following.

```
x<-rbind(c(1,2,3),c(4,5,6))      # 2 by 3
y<-matrix(rep(1,ncol(x)*4),nrow=3) # 3 by 4
z<-matrix(rnorm(9),ncol=3)        # square matrix

y%*%x                             # error
solve(y)                          # error

x%*%y
t(y)%*%t(x)
solve(z)%*%z                      # identity matrix (like dividing by itself)
```

8 Relations and Logic

8.1 Goal

In this section you will learn how to produce and handle first order logical operations in R.

In practice, this simply means generating **TRUE** and **FALSE** values from matrices. And then using these values in subsequent calculations and to select sub-sets of data via referencing.

8.2 Relations

It is easy and useful to compare numerical values and matrices in R. A summary of comparative operators we will use is as follows.

R Code	Operation
<code>==</code>	Equal
<code>!=</code>	Not equal
<code><</code>	Less than
<code>></code>	Greater than
<code><=</code>	Less than or equal
<code>>=</code>	Greater than or equal

These operators result in `TRUE` or `FALSE` “logical” values.

8.3 Logic

It is also easy and useful to compare logical values or conditions in R. A summary of comparative operators we will use is as follows.

R Code	Operation
<code>&</code>	element-wise “AND” (use to compare matrices or scalars)
<code> </code>	element-wise “OR” (use to compare matrices or scalars)
<code>!</code>	“NOT”
<code>&&</code>	logical “AND” (use to compare scalars only)
<code> </code>	“OR” (use to compare scalars only)

We will normally only use the first three in the table above - they work for scalars and matrices.

8.4 Matrix Checks

There are variety of useful checks that can be carried out on R matrices. For example

R Code	Operation
<code>any</code>	True if any element of vector is <code>TRUE</code>
<code>all</code>	True if all elements of vector are <code>TRUE</code>
<code>is.na</code>	True if element is <code>NA</code>
<code>identical</code>	True if one matrix is equal to another
<code>is.numeric</code>	True if matrix is numeric (cf. character)
<code>is.logical</code>	True if the data is logical (<code>TRUE</code> or <code>FALSE</code>)

The `any` and `all` functions can be used on either rows or columns of matrices by using `apply` e.g. `apply(is.na(x), 1, all)` checks for NA values by rows.

8.5 TRUE or FALSE

The default data storage of the logical outcomes in R is “TRUE” and “FALSE”.

```
2.5==2.5  
5<0
```

These are *not* numeric values. But you can use them with R functions such as [sum](#) and normal operations like addition and so on (with R doing an automatic conversion).

```
x<-c(TRUE,FALSE,TRUE); y<-c(1,0,1)  
cbind(sum(x),sum(y))  
cbind(x*100 , y*100)  
is.logical(x)  
is.logical(y)  
is.numeric(x)  
  
y[x]           # logical reference element-by-element
```

You can change **TRUE** into **FALSE**, or vice-versa, using **!**.

```
x<-TRUE  
y<-!FALSE  
x  
y  
!x
```

8.6 Logical Referencing

Referencing data within matrices can be done by supplying logical values to identify the elements that you want.

```
x<-c(TRUE,FALSE,TRUE); y<-c(1,3)
data<-1:3; data2<-matrix(1:9,ncol=3)

data[x]
data[y]                # no need to specify pair for a vector

data2[x,3]
data2[y,3]

data2[,x]
data2[,y]
```

8.7 Example: Missing Values

So-called “missing values” are common in financial time series, for example when there are some securities that don’t have histories as long as others.

As already mentioned, R stores missing values as “NA”. This allows you to create and use matrices with missing value place-holders. But you need to take care that your calculations do not try to use the NA in functions.

Many functions discard missing values by default. Other functions need to be told to ignore them or have a different matching function that can be run (often functions have inputs that you can switch on, such as `na.rm=TRUE`, which removes missing values from the calculations). For example see the help documentation for calculating means in the presence of missing values:

```
?mean
```

A different and more sure-fire approach to missing values is to remove them before doing calculations. This is straightforward: you first identify which elements of a matrix are NA then you work with these elements removed.

```
# a vector with some NaNs at start and end
x<-c(rep(NA,5),1:6,NA )
x

# an index of logicals to identify missing values
xmissing<-is.na(x)
x[!xmissing]

# compare the 3 outputs
mean(x)
mean(x,na.rm=TRUE)
mean(x[!xmissing])
```

8.8 More Examples

More examples of comparisons and logical checks of both scalars and matrices include

```
x<-rnorm(10);y<-rnorm(10)

cbind( x,y, x>0&y>0 , x>0|y>0 )

x<-1+2 ; y<-rnorm(1);
x>0&&y>0
x>0&y>0
```


9 Functions

9.1 Goal

In this section you will learn how to create and use R functions.

9.2 R functions

Lots of useful functions already exist in R and have been used above. And the output of functions can be directly assigned to variables. For example

```
x<-rnorm(100)    # generate 100 random numbers  
y<-mean(x)       # calculate a mean
```

Many functions operate directly on elements. Others can be told to operate on either the rows *or* columns of a matrix. *Your own functions must explicitly account for element-by-element and row/column options.*

9.3 Functions Must Be Loaded

All R functions must be loaded into a session. Many are loaded by default. After loading, functions can usually be viewed by typing the name of the function, though details of the most powerful and basic functions (built-in and often pre-compiled C or Fortran) cannot be viewed directly.

```
# primitive functions that cannot be viewed  
mean  
is.na  
sum
```

All our own functions will be saved into a file and run before we can use them (running the function through the Console counts as “loading” for the session). By default our functions’ code will be able to be viewed by running them without arguments.

9.4 A Test Function

Open a new script file in RStudio. Then paste the following code into the file run the whole script. The key command is `function`. The round brackets after `function` contain the inputs (nothing in this example) and the function calculations are within the curly brackets.

```
# a function with no inputs
fctest <- function(){
  out <- 'my premier output'
  out
}

fctest      # view the code for function

x<-fctest() # x uses the function
y<-fctest   # y becomes a copy of the function!
```

The function output (designated by the `out` variable in the last line of the function) is assigned to `x`.

9.5 A Test Function with Inputs

Now we will add an input variable called `x` (you can call the input and out variables anything you like).

```
# a function with no inputs
ftest1 <- function(x){
  out <- 10*x
  out
}

# now use the function (inputs required)
ftest1(9.9)

y<-1:10
ftest1(y)

z<-matrix(1:4,ncol=2)
ftest1(z)
```

9.6 Functions with Multiple Inputs and Outputs

Functions typically have multiple inputs and can have many outputs. Inputs must be made explicit within the first set of round brackets following the function name. The easiest way to organise multiple outputs is with lists.

```
ftest2 <- function(in1,in2){  
  out1<-list()           # initialise an empty list  
  out1$premier = 10*in1;  
  out1$deuxieme = 100*in1;  
  out1$extra = in2*NA;  
  out1  
}  
  
# this function can now be called after it is run  
ftest2(1)                # error because 2nd input missing  
  
x<-ftest2(1,2)  
x$deuxieme
```

9.7 Entering Inputs

Inputs to any function can be done by name or simply by order.

```
# load ftest2 from previous page.
```

```
ftest2(1,2)
```

```
ftest2(in1=1,in2=2)
```

```
ftest2(in2=2,in1=1)
```


9.8 Default Inputs

Pre-specified or default inputs can be built into the function definition — just set the input definition equal to the desired default value. Note that the default value will be used within the function unless the function call explicitly enters an input for that variable.

```
fctest3 <- function(x,y=10){ x*y }  
  
# this function can now be called with or without y  
fctest3(10)  
fctest3(10,10)  
fctest3(10,3)
```

9.9 Variable Scope and Passing Inputs Into Functions

When we load a function by running its code we have the option to access already-defined variables. Often this causes a bug, but it can also be useful. The scope of a variable (see section 12.4) is essentially where it can be “seen” or used. Variables create *before* a function is loaded will be available to the code within functions. In fact, this occurs by default *unless* the variable name is defined as an input.

```
# variables with a scope that functions can see
x<-1:10 ; y<-3

# load these functions assuming x and y are defined above
ftest4 <- function(x){ x*x }
ftest4a <- function(y){ x*x }
ftest4b <- function(x){ x*y }

# look carefully at how these functions are evaluated
ftest4(10)
ftest4a() ; ftest4a(NA)
ftest4b(y)
```

9.10 Variable Scope and Outputs

The scope of variables defined *within* a function does not extend back to the level where the function is called. The main way to overwrite or create variables outside of a function is to assign a variable content with `<<-` rather than `<-`. **We will NOT do this.**

```
z<-100 ; zd<-100

# load these functions assuming variables above are defined
ftest5 <- function(x){ z<-x*x ; z }
ftest5a <- function(x){ zd<<-x*x ; znew<<-zd ; zd}

# look carefully at what these functions return and do
fctest5(3) ; z ; zd
fctest5a(3) ; z ; zd ; znew # overwrites zd and creates znew
```

The cleanest way to use functions is to define one output variable and return it at the end of function.

9.11 Function Rules-of-Thumb

Some good rules-of-thumb for your own code relating to functions are the following.

- Only use the input and output variables you will need when you build functions.
- Ensure something useful is outputted - directly or via a variable.
- Lists are best for multiple outputs.
- Keep things simple - so do not try to account for almost-redundant scenarios that may need optional inputs or outputs.
- Be very careful designing and running functions with many inputs.
- It does not hurt to use input names when calling functions!
- Beware the scope of variables: do not use variables in your function code that have already been defined outside the function unless you specifically want this!

9.12 More Complex Examples

More complex examples ...

10 Optimisation

10.1 Goal

This section introduces how to optimise functions in R.

10.2 Optimising What?

We will assume that we wish to optimise some mathematical *function*. In this context, the function will be some form of R *equation* with algebraic parameters. Some of the parameters will not be fixed, in the sense that we assume they are free to vary. R optimisation routines will seek to find the values for the free parameters at which the overall function has a maximal or minimal value.

Note that finding a maximum of a function value is the same as minimising the negative value of the same function. **By default R minimises functions.**

Optimisation is not always a precise science. For example there is no guarantee of finding a “global” solution for complex non-linear functions. And sometimes no solution is able to be found. In such cases, the set-up of the function being optimised and/or the optimisation routine may require careful attention.

There may be *constraints* on the values that the free parameters can have (e.g. that they must be greater than zero). These can be accounted for by particular R routines.

10.3 Technical Tips

- Always be clear, before you write any code, which variables are to be optimised and which are simple numerical inputs for which you have data.
- For complex functions with multiple inputs you can first define a function. Then you can make an secondary function that calls this function. In the secondary function you should add all the data inputs that you have (see section 9.9) and *leave* only the variable to be optimised as an input. Then you can use this second function in the optimisation routine.
- In most problems you need to provide R with an initial guess of the variables to be optimised.
- In cases where the optimisation routine produces an error you can: try to increase the number iterations it runs; adjust the stopping “tolerance”; try different initial guess values. See the R help files for more information about this.

10.4 Typical Output

Optimisation routines in R typically output several results.

- The optimal parameter values that have been found.
- The value of the overall objective function at these values.
- A convergence indicator (has an “optimal value” been found or was some other limit met).
- Some more details about the routine used, number of iterations completed etc.

10.5 Routine Options

Each routine comes with various options to set. There are defaults, so you can try running your code without changing anything.

10.6 Useful R Optimisation Functions

The main R optimisation routines are called `optim`, `constrOptim` and `uniroot`.

The first, `optim`, will efficiently minimise non-linear functions that have no constraints (linear functions are also handled, but there may be more efficient methods).

The second, `constrOptim`, is the most general and will minimise linear and non-linear functions, with or without, constraints. The main cost of `constrOptim` is usually speed. And the many options with respect to constraints can make using the routine tricky.

The last, `uniroot`, is for root finding.

We will also examine the package `quadprog` and its function `solve.QP` for specific quadratic programming applications.

10.7 Simple Optimisation Example

A simple example of a R equation to minimise is the following.

```
testfun1<-function(x) {x^2}  
plot(seq(-4,4,.1) , testfun1(seq(-4,4,.1)))  
  
x0<-1          # initial guess  
  
optim(x0 , testfun1)  
  
# after reading output ...  
optim(x0 , testfun1 , method="Brent")  
optim(x0 , testfun1 , method="Brent",upper=10000,lower=-10000)
```

10.8 More Complex Optimisation 1

A more complex example is maximising a “log likelihood” function. This has multiple parameters and a complex form. Also, we want to *input* some sample data and only optimise with respect to the parameters of the Normal distribution.

The code below sets up the problem and we solve the function on the next page.

```
# some sample data (where we know mu=0 and sigma=0)
xtest<-rnorm(100)          # random N(0,1)

# log likelihood of iid Normal data (want to maximise this)
testLL <-function(x,mu,sigma) {
  log( prod( 1/(sigma*sqrt(2*pi))*exp(-(x-mu)^2/(2*sigma^2) ) ) )
}

# now make another function to input data and only leave Normal parameters
testfun3<-function(x){ - testLL(xtest,mu=x[1],sigma=x[2])}
```

```
# initial guesses (assume the sample is real and you do not know ... )  
x0<-c(.5,.5)  
  
# now optimise  
optim(x0,testfun3)
```

10.9 More Complex Optimisation 2

The code below re-writes the likelihood in a more efficient form — the log of a product is now a sum. This ensures that for large samples the computer is able to accurately add up the likelihood.

```
xtest<-rnorm(10000)          # random N(0,1)

# make likelihood EFFICIENT (watch sigma though - can be neg or pos)
testLL <-function(x,mu,sigma) {
  sum(log( 1/(sqrt(sigma^2*2*pi))*exp(-(x-mu)^2/(2*sigma^2) ) ))
}
testfun3<-function(x){ - testLL(xtest,mu=x[1],sigma=x[2])}
x0<-matrix(c(.5,-1),ncol=1)

optim(x0,testfun3)          # silly starting value may result in neg variance
# optim(x0,testfun3, method="L-BFGS-B",lower=c(-100,.1))

## see next page for more ....
```



```
# try again with sensible starting values and constraints to make sure
x0<-matrix(c(1,1),ncol=1)
A<-rbind(c(0,1))
b<-matrix(c(0),ncol=1)
constrOptim(theta=x0,f=testfun3 , grad=NULL, ui=A, ci=b)

# constraints  $Ax \geq b$ 
```

10.10 Quadratic Example

The set-up for a quadratic optimisation is specific. In this case, a general function to be minimised can take the following form

$$\min_x \frac{1}{2} x' H x + f' x, \quad (1)$$

subject to the conditions that $Ax \geq b$, where the first m conditions are equality, rather than inequality.

This problem is most efficiently solved with a specific method — we will use the algorithm in `solve.QP` (though others exist). Using `constrOptim` would be problematic for two reasons. First, `constrOptim` is a general purpose function that isn't the most efficient for the quadratic programs. Secondly, there is no stable way to have *equality* constraints with `constrOptim`.

10.11 Quadratic Example Code

```
# install quadprog package (requires internet connection)
install.packages("quadprog") # this line not needed if already downloaded!
library("quadprog")

# set-up problem
H <- rbind(c(0.005, -0.010, 0.004), c(-0.010, 0.040, -0.002), c(0.004, -0.002, 0.023))
n <- ncol(H)

Aux <- matrix(0, ncol=n, nrow=n)
diag(Aux) <- 1

A <- rbind(rep(1, n), Aux)
b <- matrix(c(1, rep(0, n)), ncol=1)
m <- -1                      # first "m" rows of A are strict equality
f <- rep(0, n)               # not used here

# solve using a special algorithm (see documentation)
solve.QP(Dmat=H, dvec=f, Amat=t(A), bvec=b, meq=m)
```

10.12 Root Finding

Sometimes our goal is to find the “roots” of a function. “Root finding” simply means identifying values of the parameters which set the value of a function to zero. This is essentially the same as optimisation and uses the same sorts of numerical methods. For functions with one parameter, the main root finding command in R is `uniroot`. It needs an initial guess for a *range* to search over.

```
# find yield-to-maturity of a bond (cost is 90 with 5% coupon for 3 years)
payments<-c(-90,5,5,5,100)
timing<-c(0,1,2,3,3)
ytm<-function(i){sum(payments/(1+i)^timing)}
initial_range<-c(-1,1)  ## you guess the answer is within this range ...

# solve
uniroot(ytm,initial_range)
check<-as.matrix(seq(-.05,.15,.001))
plot(check,apply(check,1,ytm))
lines(c(min(check),max(check)),c(0,0),col=2,lty=3)
```

11 Charts

11.1 Goal

This section is a *introduction* to charts in R.

Extensive detail on plotting in R is contained in the help files. For example, start with `library(help = "graphics")` for an overview.

11.2 Basic Plots

The main chart interface in R is through a generic `plot` command. With the addition of special parameters and sub-commands, almost any type of chart can be generated with `plot`. The default is scatter plot, but other types, like lines, can be set with the `type` parameter.

The command `windows` opens new windows for plots to specified dimensions, otherwise the last plot window is overwritten.

```
x<-rnorm(100); y<-x+rnorm(100,0,.2)
plot(x,y)      # plots in current window

windows(5,5)   # opens a new window "quartz(5,5)" on a MAC
plot(x,x)

windows(5,5)
plot(sort(x),type="l") # line plot (sorting is a good idea)
```

11.3 Other Charts

There are specialist charting options in R. The most useful include [hist](#) and [boxplot](#).

```
x<-rnorm(100)
windows(5,5)
hist(x)

windows(5,5)
boxplot(x)
```


11.4 Multiple Plots per Window

The easiest way to add multiple plots per window is to invoke the `par` function and set the `mfrow` variable.

```
x<-rnorm(100)

windows(10,5)
par(mfrow=c(1,2)) # 1 by 2 window
hist(x)
boxplot(x)

windows(3,9)
par(mfrow=c(3,1)) # 3 by 1 window
hist(x)
boxplot(x)
plot(x)
```

11.5 Titles and Labels

The most typical things added to charts are titles, x-labels and y-labels. Many other options exist, however. The easiest way to add this information is in the initial plot, specifying parameter values for `main`, `xlab` and `ylab`.

```
x<-rnorm(100); y<-x+rnorm(100,0,.2)

windows(5,5)
plot(x,y, main="Linear Relationship",xlab="x-data",ylab="y-data")
```

11.6 Multiple Points and Lines

Often plots will display more than one dataset. The easiest way to do this is to use the `lines` and `points` commands *after* the initial `plot`.

```
x<-sort(rnorm(100))    # note that this is sorted
y<-x+rnorm(100,0,.2)
z<-x+rnorm(100,1,.2)

windows(5,10) ; par(mfrow=c(1,2))
plot(x,y)
points(x,z)

plot(x,y, type="l")
lines(x,z)
```

11.7 Colours, Widths and Characters etc

A variety of useful properties can be set directly within a `plot` call. These include colour (`col`), line type (dashes etc. `lty`), line width (`lwd`) and characters (`pch`). There are default index numbers for such things (see help).

```
x<-sort(rnorm(100))      # note that this is sorted
y<-x+rnorm(100,0,.2)
z<- x+rnorm(100,1,.2)

windows(10,5) ; par(mfrow=c(1,2))
plot(x,y, col=2)
points(x,z, pch=3, col=3)

plot(x,y, type="l",lty=2)
lines(x,z, col=2, lwd=3)
```

11.8 Fitting All the Data In

One common wrinkle is that `points` and `lines` do not expand the initial plot boundaries. If this is an issue then an easy work-around is to set-up an initial plot with the max and min extremes, and then add the actual data

```
x<-sort(rnorm(100))      # note that this is sorted
y<-x+rnorm(100,0,.2)
z<- x+rnorm(100,2,.2)

windows(10,5) ; par(mfrow=c(1,2))
plot(x,y, col=2)
points(x,z, pch=3, col=3)

plot(c(min(x),max(x)),c(min(c(y,z)),max(c(y,z))),col=0,xlab=NA,ylab=NA)
points(x,y, col=2)
points(x,z, pch=3, col=3)
```

11.9 Saving and Deleting Charts

Is is easy to print charts to files rather than a window. This is done by calling specific functions from R, depending on the file type — eg. `jpeg` or `pdf` — and then calling `dev.off()`. Specify a full file-path or simply find the chart in your working directory `getwd()`.

```
x<-sort(rnorm(100)); y<-x+rnorm(100,0,.2) ; z<- x+rnorm(100,2,.2)
```

```
jpeg("plot1.jpg") # will get dumped into getwd()
plot(c(min(x),max(x)),c(min(c(y,z)),max(c(y,z))),col=0,xlab=NA,ylab=NA)
points(x,y, col=2)
points(x,z, pch=3, col=3)
dev.off()
```

```
pdf("plot1.pdf") # note that width and height can be set here ...
plot(c(min(x),max(x)),c(min(c(y,z)),max(c(y,z))),col=0,xlab=NA,ylab=NA)
points(x,y, col=2)
points(x,z, pch=3, col=3)
dev.off()
```

The best non-pdf format is **tiff**. You should set the height and width of charts (check help), which helps control the resolution (remember the dimensions will be automatically re-scaled to fit A4 by Microsoft Word or Latex). Experiment to find the right output.

```
x<-sort(rnorm(100)); y<-x+rnorm(100,0,.2) ; z<-x+rnorm(100,2,.2)

# tiff uses pixels for dimensions
tiff("plot_300_300.tiff", height=300,width=300) # pixel units for dim
plot(x,y) ; dev.off()

tiff("plot_900_900.tiff", height=900,width=900) # pixel units for dim
plot(x,y) ; dev.off()

# pdf defaults to inches for dimensions
pdf("plot_3_3.pdf",height=3,width=3) ; plot(x,y) ; dev.off()
pdf("plot_9_9.pdf",height=9,width=9) ; plot(x,y) ; dev.off()
```

11.10 Advanced

There are many advanced options for charts, including: time series with dates on the x-axis, changing the format of the axis ticks, legends, blocks/shapes of colour, adding titles etc after the plot is generated, adding text boxes ...

Useful commands and functions include the following.

- `ts.plot` plots columns of a matrix separately in one go.
- `legend` adds a legend.
- `cex` setting (within functions) controls font size.
- `?par` for a vast variety of general settings.

12 Scripts

12.1 Goal

This section is a brief discussion of about R scripts.

12.2 Basics

A basic script is set of R commands, conventionally saved as a `.R` file. A variety of methods exist to run code from a script in RStudio. Perhaps the most intuitive is via selection of the lines to run and pressing the green arrow, or using the keyboard short-cuts on offer.

A more formal method is the save the script and call it from the console using `source`.

12.3 Debugging

You will often make syntax mistakes. R will trap each error for you and try to give useful feedback. Then you have to fix the problem and re-run the code. This is an iterative process and will take some getting used to.

You can also set so-called “break points” at which the code execution will pause, so that you can inspect results and variable assignments to-date.

12.4 Variable Scope

Loosely speaking the “scope” of a variable means where else it can be seen or used, which dictates how or if it can be accessed by other code.

- The scope of variables created inside functions is limited to inside of that function.
- Variables created from scripts may be used within other scripts and functions.
- Beware accidental overwriting of variables when running multiple scripts.

12.5 Planning a Script

When you are doing proper work (like the assignments) *always* write a plan for your scripts. An ordered list of what your program needs to do will suffice.

- Make frequent use of code comments `#`.
- Preallocate any output matrices i.e. matrices that will be filled with data within a loop. Starting with NA values or zeros is a good idea.
- Use vector indexing instead of loops when you can.
- Beware variables from scripts infecting functions.

12.6 Miscellaneous Advice

- When writing complex scripts start with simple cases, then add layers of complexity as you go.
- Don't wait until you are finished to run each part of your code - keep checking as you go!

13 Basic Probability

13.1 Goal

This section is an *introduction* to basic probability functions found in R.

The details of probability are for other courses – but you should be able to confidently use R as you learn about hypothesis testing, econometric models and probability theory.

13.2 Probability Distributions

The main probability distributions you will use are the Normal, Student-T, Chi-squared and F distributions. In R the following functions represent probability density functions for the first three: `dnorm`, `dt` and `dchisq`. Cumulative distributions are prefixed by “p” instead of “d”. Quantiles can be backed out with the prefix “q”. See [?Distributions](#) for more details.

```
# Normal with mean 0 and sd 1.
x<- seq(-5,5,.1)
windows(10,5); par(mfrow=c(1,2))
plot(x,dnorm(x),main="N(0,1) Density")
plot(x,pnorm(x),main="N(0,1) Cumulative Density")

# Chi-sqr with "df" degrees of freedom
windows(10,5); par(mfrow=c(1,2))
df<-5
y<-seq(0,qchisq(.99,df),.1) # note the use of quantile .99
plot(y,dchisq(y,df),main=paste0("Chi-sqr(",df,") Density") )
plot(y,pchisq(y,df),main=paste0("Chi-sqr(",df,") Cumulative Density") )
```

14 Self Study List

14.1 A List

The following list (over two pages) is a non-exhaustive list of functions, commands and settings in R that you may find useful. A few of these things will come up in the second part of this course.

You should be able to look up these sorts of things in the R Help file, run mini-examples and use them compentantly ... as required.

And don't be afraid to go further!

- Re-arranging vectors with `sort`, `order` and `rank`.
- Misc. vector summary functions `max`, `min`, `pmax`, `pmin` and `quantile`.
- Display a little bit of a matrix with `head` and `tail`.
- Use negative index numbers to drop elements e.g. `x[-c(1,3)]` returns `x` without elements 1 and 3.
- Display object properties e.g. `names` and `str`.

- Identify `TRUE` elements as index numbers with `which`.
- Check for matching values in vectors with `%in%` and `match`.
- Altering numbers with `round`, `trunc`, `ceiling` and `floor`.
- Operating on elements of a matrix quickly with `sweep` and `scale`.
- Using the super-flexible data type called `factor` and assign `levels` and `labels`.
And convert from strings to integers using `unclass` or `as.numeric(levels(x))[x]`
...
- Summarising by factor levels with `aggregate`.
- Reference elements of a matrix with a matrix of index references (rather than separate row and col indices).
- Join to data frames via unique keys with `merge` (like SQL).
- Use `View` to see a matrix or data frame in Rstudio.

15 Miscellaneous

This short section includes a few useful bits and pieces. And a short list of important things not covered.

15.1 Extensions

A variety of interesting and advanced topics were not covered in this document. These include the following.

- Advanced topics such as object-orientated programming.
- Index referencing different pieces of a matrix in one go.
- Advanced charting.
- Advanced statistical distribution theory and methods.
- Memory management and benchmarking code efficiency and speed.
- Linking your own C and FORTRAN scripts with R.

15.2 Tidyverse

You might be interested in the “Tidyverse” suit of packages for data science. See <https://www.tidyverse.org/>. Learning base R (from this document) is a great entry point to writing Tidyverse code.

15.3 Running Text as R Code

A block of text can be run as R code using the commands `eval` and `parse`.

```
todo<-"x<-1:10 ; x"  
eval(parse(text=todo))
```

15.4 Setting-up and Cleaning a Script

```
# reset output window
cat("\014")

# remove everything and start again (just run the code!)
rm(list=ls(pos=.GlobalEnv), pos=.GlobalEnv)

# set-up input and output folders
input.folder<- file.path(getwd() ,'input//')
output.folder<- file.path(getwd() ,'output//')
```

15.5 Loading Functions from Files

```
# assume there is separate folder
function.folder<-file.path(getwd(),'functions//')

# assume all function files begin with "f" and end with ".R"
function.files<-paste0("source('",function.folder,
  list.files(path=function.folder,pattern="^f.*.R$"),"', encoding='utf-8')")

# run the code in the function files to load everything
for(x in function.files){ eval(parse(text=x)) }
```

Index

" , 61
==, 128
.RData, 57
//, 55
:, 20
;, 41
<-, 34, 147
<<-, 147
?, 33, 39
??. 33
?Distributions, 109, 186
?par, 176
?regex, 27
[[]], 85
#, 24
\$, 45, 90
%in%, 189
&, 129
&&, 129
 , 132
{...}, 96
` ` , 89
|, 129
||, 129

abs, 47

aggregate, 189
all, 130
any, 130
apply, 48, 79, 80, 88, 91, 118
as.character, 30
as.Date, 29
as.factor, 46
as.matrix, 62
attach, 90

boxplot, 168
break, 102, 104

c, 21, 42, 68, 70
cat, 96
cat(\014), 50
catch, 104
cbind, 70
ceiling, 189
cex, 176
character, 46
check.names=FALSE, 61
colnames, 63, 81
constrOptim, 156
cumprod, 47
cumsum, 47

data.frame, 44

datenum, 29
dchisq, 186
dec, 59
detach, 64, 90
dev.off(), 174
diag, 124
dim, 69
dimnames, 81
dnorm, 186
dt, 186

else, 100
eval, 193
exp, 47, 117

factor, 46, 189
FALSE, 31, 74, 127, 128, 132
file.path, 56
floor, 189
for, 96, 98
function, 141

getwd(), 53
grep, 26, 28
gsub, 28

head, 188
header=TRUE, 61
hist, 168

identical, 130
if, 99, 100
ifelse, 104
install.packages, 64
is.logical, 130
is.na, 130
is.numeric, 46, 130

jpeg, 174

lapply, 86--88, 91
legend, 176
length, 69
library(), 64
lines, 171
list, 84
list.files, 56
log, 47, 117
logical, 46

match, 189
matrix, 42, 71, 77
max, 47, 188
mean, 47
merge, 189
mfrow, 169
min, 47, 188

NA, 23, 134
na.strings=", 61

names, 89, 188
NaN, 23
ncol, 69, 73
nrow, 69, 73
NULL, 32
numeric, 46

optim, 156
order, 188

par, 169
parse, 193
paste, 25
paste0, 25
pdf, 174
plot, 167
pmax, 188
pmin, 188
points, 171
POSIXct, 29
POSIXlt, 29
print, 96
prod, 47

quadprog, 156
quantile, 188
quartz, 13
quote = ", 61

rank, 188

rbind, 70
read.table, 60, 61
rep, 78
require, 65
rm, 37
rnorm, 75, 108
round, 189
row.names=c(1), 61
rownames, 63, 81

sample, 110
sapply, 87
scale, 189
search(), 64
sep, 59
seq, 75, 76
set.seed, 111
setwd, 53
sign, 47
solve, 122
solve.QP, 156
sort, 188
source, 54, 55, 179
split, 92
sqrt, 47
str, 46, 188
stringsAsFactors, 44
stringsAsFactors=FALSE, 61

sum, 21, 47, 131

sweep, 189

switch, 104

t, 123

tail, 188

tibble, 93

tidyverse, 93

tiff, 175

TRUE, 26, 31, 74, 127, 128, 132, 189

trunc, 189

try, 104

ts.plot, 176

type, 167

unclass, 189

uniroot, 156, 164

unlist, 86, 87

unsplit, 92

var, 47

vector, 83

View, 189

which, 189

while, 98

windows, 13, 167

write.table, 59, 63