Numerical and Statistical Methods for Finance

Introduction to R

Pierpaolo De Blasi

University of Torino
email: `pierpaolo.deblasi@unito.it`
webpage: `sites.google.com/a/carloalberto.org/pdeblasi/`

Lectures no. 1–4
18–19–25–26 September 2013

R software

From R documentation:

R is a GNU (*recursive acronym for "GNU's Not Unix"*) project that provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, ...) and programming tools, and is **highly extensible**.

One of R 's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.

R software

From R documentation:

R is available as a Free Software under the terms of the Free Software Foundation's GNU General Public Licence in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.

Much statistical functionality is provided by the user community.

New methods are often implemented and distributed in `R`.

History

R is readily traced back to the S language -“ a language for programming with data ”- developed primarily by John Chambers at Bell Labs.

The `S` language was developed into a commercial product called Splus by a company now called “ Insightful ”. This development has been mainly in the user interface and graphics and inter-operability functionality.

R started out as an Open Source system similar to version 3 of the `S` language, developed by Ross Ihaka and Robert Gentleman. R is now developed and maintained by a core group.

Getting R

Can always download the most recent release either as a source or pre-compiled of R from a Comprehensive R Archive Network (CRAN) site (`http://cran.mirror.garr.it/mirrors/CRAN/` for Italy).

R consists of a *base system*, providing the language and the interface, and contributed *packages*, providing the enhanced statistical and graphical functionality.

In general it is better to upgrade the whole system on an occasional basis than to update individual packages on the release of new versions.

Install R

To obtain and install a pre-compiled windows binary:

1. Download `R-3.0.1-win.exe` (for Windows) or `R-3.0.1.pkg` (for Mac OS X) from CRAN
2. Execute and follow the installation program that guides the users through the process. If in doubts, accept defaults.
3. Start R .
4. From the `Packages` menu, select a CRAN mirror. Then choose `Install package(s)` and select the required packages from the list. *You need administrator rights to do so*!

Note that (i) installing packages can be slow, (ii) pre-requisites are automatically satisfied, (iii) can experience problems with version numbers.

Starting and Packages

We use R in an interactive mode. That is, we ask R a question, and it responds with an answer.

Whence we launch R (e.g. by clicking on the R icon), a new window pops up with a *command-line sub-window*. The command line, or console, is where we can interact with R . It looks something like this:

```
R version 3.0.1 (2013-05-16)
Copyright (C) 2013 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
.
.
.
[Previously saved workspace restored]

>
```

What is R ?

R should be regarded as an implementation of the S programming language - not a statistical package with limited programming tagged on (like SAS, SPSS, Minitab). As such, it provides

- Programming language constructs
- Data structures
- Functions for mathematics, statistics and graphics.

In particular, note that *everything* in R is an *object*... it will become clear what this means later!

Starting R

When R starts, it searches for any saved workspace in the current directory (from `File` menu choose `Change directory`).

The command prompt, >, is the final thing shown. This is where we type commands to be processed by R .

The use of an editor interface to R gives additional functionality. Examples are `Tinn-R`, `WinEdit` and `Emacs`.

The simplest option is to type commands on the console or to use a script file (extension `.R`).

Codes will be provided with the commands used in class as a script file.

R script file

The R console provides a convenient interface for simple commands.

For more complicated work, such as programming, R provides a script file, where a sequence of R commands can be entered, and processed later. To start a new script file, from the `File` menu, select `New script`.

Commands are entered here, and selected for execution by highlighting followed by `<Ctrl> r`. Scripting is a very useful feature, and for most tasks will be the default mode.

You can work with the code we are using in class by opening the text file as a script (first you need to set as working directory the folder were `script 1-2-3.R` is placed, then from the `File` menu, select `Open script`).

Packages

The recommended R distribution includes a number of packages in its library. This are collections of functions and data. We will make use of packages that are not included with the default distribution, like `UsingR` or `spuRs`.

The `base` package, `stats` package, `datasets` package and some other packages, are automatically attached at the beginning of the session. Other installed packages must be explicitly attached prior to use. Use `sessionInfo()` to see which packages are currently attached. (not installed)

they exist insidebut are not attached to be used.

Attach a previously installed package via the command

```
library(NAME.PACKAGE)
```

Type `data()` to get a list of data sets available in the current session.

curent library attached

Simple Data Analysis

To illustrate ideas, let us conduct some simple data analysis, involving a regression model, on the Scottish data set `hills` (package MASS). Data are distance, climb and record times for 35 Scottish hill races. Note the following features:

- Interaction via the command line interface
- The value of a function may be assigned a name
- Graphics occur as side effects to function calls
- Functions can operate on arguments of different types

Data

nome

funzione che calcola in mean delle colonne della package hills.

Consider

```
average<-colMeans(hills)
```

We have called the function `colMeans()`, with a single argument, and assigned, via the assignment operator `<-`, the value of the call to the object `average`.

All entities in R are *objects* and all objects have a *class*. The class dictates what the object can contain, and what many functions can do with it. Thus, `colMeans` is an object of class `function`, and `average` is an object of a type determined by the function, in this case, `numeric`. The object `hills` is of class `data.frame`.

Data

Everything we do will involve creating and manipulating objects.

Objects are accessed by name. Objects that we create should have names that are meaningful to us, and should follow the R syntax rules: letters and numbers (except first position) and periods (avoid underscore).

A problem is naming object that are already in use by R , reserved words like `if` or `for` or system objects like `mean`, `TRUE` and `pi`.

Remember that R syntax is case sensitive: `T` (a system representation of logical true) is different from `t` (function of matrix transpose).

Functions

Functions are called by name, followed by a bracketed list of arguments

```
range(x)   # range of values in the vector x
plot(y~x)  # same as plot(x,y)
plot(lm(time ~dist, data=hills), which=1)
```

y x

regression

Functions return a value. Graphics functions in addition have side effects, that create or modify plots. The argument list can take various formats, and not all arguments always need to be specified.

As with the examples above, functions behave differently according to the class of their arguments.

Look at the help file of a function by typing `help(range)`.

Note that # is the comment character: all text following this is treated as a comment (not processed by R).

Vectors

Consider the quadratic equation

$$x^2 - 3x + 2 = 0$$

In usual notation, we have coefficients $a = 1, b = -3, c = 2$. This equation has real roots if the *discriminant* is nonnegative

$$b^2 - 4ac \geq 0$$

THIS IS CALLED THE DISCRIMINANT

- Store coefficients as an object
- Compute discriminant
- Construct a plot of $f(x) = x^2 - 3x + 2$

Vectors

Consider

```
coeffs<-c(1,-3,2)
class(coeffs)
length(coeffs)
names(coeffs)
```

```
> x<-seq(1,3,length=21)
> x                              # [1] 1 1.1 1.2 ... 2.8 2.9 3
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6
2.7 2.8
[20] 2.9 3.0
> class(x)
[1] "numeric"
> j<-c(1,2,8,8)
> class(j)
[1] "numeric"
```

We create `coeffs` as a *vector* of type `numeric`. Vectors are a fundamental class in R - there are not scalars. All objects also have `length`, which is only informative for certain classes.

The `c()` (for combine) function is basic tool for creating vectors.

Vectors can have names, queried and modified by the `names` function.

Vectors

not sure how to
distinct them

The classes available for vectors are

- `numeric` – real numbers
- `character` – character strings, arbitrary length
- `logical` – vector of logical `TRUE` and `FALSE` values
- `integer` – (signed) integer values
- `complex` – complex numbers $a + ib$, $i = \sqrt{-1}$
- `list` – a "clothsline" object, each numbered element can contain any R object

Often functions return `list`, see `out` in the previous demo. We can refer to its element using the `$` operator as in `out$coeff`

this is the out reselut of out, so the list is a bunch of letters and numbers,you access to the elements by out$coeff like in the vectors b[2]

Vectors

Access the elements of a vector by number, or name

```
coeffs[2]
coeffs["b"]
```

We may wish to remove the names from a vector, `names(coeffs)<-NULL`.

The object `NULL` represents nothing and it has length zero.

Vectors can be created other ways

```
seq(0,3,length=200)          # regular spaced points
-2:2                         # sequence of integers
x<--2:2                      # create
c(x,x)                       # combine two vectors
```

Arithmetic

Simple arithmetic on constants follows the usual precedence rules:

```
ax<- 1; bx<- -3; cx<- 2
x<- ax-bx+2*cx                          # x =8
x<- bx/bx*cx/ax+bx   3/-3               # x=-1
x<- (bx/bx)*(cx/ax)+bx                  # better
bx^cx                                   # -3 raised to 3
```

Use brackets to simplify expressions. These must balance, otherwise R responds with either a syntax error, or a continuation request (a + prompt).

Recall that vectors are a fundamental class of objects in R . The examples above therefore involve arithmetic on vectors like `ax`.

Special Values

With computer arithmetic we require extra symbols to represent missing values and mathematical pathologies.

- Missing values are represented as `NA`. R also uses `Inf`, `-Inf` and `Nan`, the latter for indefinite results like `Inf/Inf`.
- There are functions for element-wise testing of vectors for the presence of special values, of the form `is.XX`, where `XX` can be `na`, `nan`, `finite`, `infinite`.
- Special values can cause problems in programming, so check for their presence!

THE RESPONSE IS TRUE FALSE

Things get more complicated. Consider

```
x<-seq(0,3,length=200)
y<-coeffs[1]*x^2+coeffs[2]*x+coeffs[3]
```

Here we are multiplying a vector of length 1 by a vector of length 200. This problem is dealt with by *recycling* the shorter vector until it matches the length of the longer vector.

SI APPLICA AI SINGOLI ELEMENTI DEL VETTORE

In `x^2` the square is performed *element-wise*. Product and ratio of vectors are also performed element-wise, according to the recycling rule if the lengths differ (matrix multiplication has different syntax).

A vector can have zero length and is represented as `numeric(0)`.

Simple Functions

There are a large collection of functions that operate on numeric vectors.

```
round(x,2); trunc(x); ceiling(x);
abs(x); log(x); sqrt(x); exp(x);
sin(x); acos(x); tanh(x)
```

In each case, the result of the function is a vector of the same length as the argument. Check the help files to see what these functions actually do.

Note that `log` can take a second argument, the base of the logarithm. The following are equivalent

```
log(cx); log(cx,exp(1)); log(x=cx,base=exp(1))
```

Simple Functions

Standard function for reducing numeric vectors

```
min(x); max(x); mean(x)
sum(x); prod(x)
```

and the *cumulative* equivalents

```
cumsum(x); cumprod(x)
```

Can already see how to usefully combine functions

```
sum(x)/length(x)                 # sample mean
prod(1:5)                        # factorial
sum(x-mean(x))^2/(length(x)-1) # sample variance
```

Logic

A full set of logical operators and functions are available in R .

Truth table (application of De Morgan law):

A	B	A and B $A \wedge B$	A or B $A \vee B$	non A $\neg A$	non B $\neg B$	non ((non A) or (non B)) $\neg(\neg A \vee \neg B)$
T	F	$\boldsymbol{F}$	T	F	T	$\boldsymbol{F}$
T	T	$\boldsymbol{T}$	T	F	F	$\boldsymbol{T}$
T	T	$\boldsymbol{T}$	T	F	F	$\boldsymbol{T}$
T	T	$\boldsymbol{T}$	T	F	F	$\boldsymbol{T}$
F	T	$\boldsymbol{F}$	T	T	F	$\boldsymbol{F}$

Logic

Logical conditions can be applied to numeric vectors

```
x <- 1:5;
A<- x > 1
```

Now `A` is a logical vector of length 5. The condition $>$ has been applied element-wise.

The other comparison operators are $>=$, $<$, $<=$, $==$ and `!=`. The final two are *exact* equality and inequality, respectively. Logical vectors are subject to the usual recycling rules.

Logical values can be combined and modified with

- `!` negation operator (logical NON or $\neg$)
- `&` intersection operator (logical AND or $\wedge$)
- `|` union operator (logical OR or $\vee$).

Logic

A nice feature of logical vectors is that they can be used in ordinary arithmetic.

```
A<- x < 5
A + 1
```

```
 A
[1]  TRUE  TRUE  TRUE  TRUE FALSE
> A + 1
[1] 2 2 2 2 1
```

The resulting vector is `c(1,2,2,2,2)`. R has noted the combination of logical and numeric vectors, and *coerced* the logical vector to numeric, by mapping `TRUE` to `1` and `FALSE` to zero.

The use of logical vectors in ordinary arithmetic means we can easily count numbers of `TRUE` or `FALSE` is a comparison

```
sum(A) > 1
```

```
> A
[1]  TRUE  TRUE  TRUE  TRUE FALSE
> sum(A)
[1] 4
>
```

Functions for Logical Vectors

The function `any` returns value `1` if at least one element of its logical arguments is `TRUE`. The function `all` returns value `1` if all elements are `TRUE`. Note we can implement similar functionality directly

```
sum(A) > 1             # the same as any(A)
sum(A) == length(A)    # the same as all(A)
```

The function `which` is used to know which element of a logical vector is `TRUE`. It returns the corresponding indexes.

```
B<- x < 5
which(B) which elements of B are true?1234.
[1] 1 2 3 4            # elements 1,2,3 and 4 are true
```

Character Vectors and Factors

With character vectors one can use the functions for set operations:

```
A<- c("beer","beer","wine","water")
B<-c("beer","wine")
union(A,B); intersect(A,B);  setdiff(A,B)
```

Instead, `factor` are like numeric vector with values $1, 2, \ldots, k$. The value k is the number of levels, the levels are the character strings. They are used to store sets of categorical data.

```
drinks<- factor(c("beer","beer","wine","water"))
```

It is informative to examine how the factor is stored, using

```
unclass(drinks)
```

Summarize Categorical Data

Categorical data is summarized by tables. The main R function for creating tables is `tables()`:

```
A<- c("beer","beer","wine","water")
table(A)                    # finds the unique values and
                            # tabulates their frequencies
```

It is useful to extract the (character) vector with the unique values. This can be done in two ways

```
drinks<- factor(A)          # coerce to a factor
levels(drinks)              # returns the unique values
names(table(A))             # the same
```

The function `table` is the main tool for creating contingency tables: needs two vectors of same size. Easy to extract proportions instead of frequencies:

```
round( table(beer,wine)/length(beer),2 )
```

Data Frames

The typical class used for data analysis in R is the *data frame*. This format combines many variables in a rectangular grid, where each column is a different variable, and usually each row corresponds to the same subject or experimental unit.

The advantage is that columns can be of different classes: numeric, logical, character and so on.

Look at the content of the `Cars93.summary` data set.

Data frames can have row names, common to all columns.

```
rownames(Cars93.summary)
```

For data frames, the column names are accessed with the function `names`. Strictly, a data frame is a *list* (see later), where all elements are vectors (of possibly different classes) of equal length.

Data Frames

Usually a data frame will be created by a suitable call to a data import function. It is also possible to combine vectors into a data frame. For example

```
data.frame( X1=1:10, X2=I(letters[1:10]),
            X3=factor(letters[1:10]) )
data.frame( 1:10, I(letters[1:10]),
            factor(letters[1:10]) )
```

By default, R will attempt to pick row names from the constituent vectors, and otherwise will use numeric row names, and guess at column names if they are not given. Row and column names can be provided as an extra argument.

Use `I()` to override the default behaviour of converting character vectors to factors.

Data Frames

If a data frame has names, we can refer to column using the `$` operator as follows

```
Cars93.summary$Min.passenger
```

This is now simply a vector, and can be manipulated as such.

We use the `attach` command to make the columns of `Cars93.summary` data available by name. To undo, use `detach`.

```
attach(Cars93.summary)
class(abbrev)          # vector of class factor
detach(Cars93.summary)
```

Manipulating data frames

We can use factor vectors inside a data frame for aggregating data with respect to a specified function for each combination of the factor levels.

`Cars93.summary` was created from the `Cars93` data set in the MASS package.

We reproduce the data set by using the command `aggregate` to form the vector that holds the min values of `Passengers` for each levels of `Type` (they are two variables of `Cars93`)

```
library(MASS)
attach(Cars93)
aggregate(Passengers,by=list(Type),min)
          # reproduce first variable of Cars93.summary
```

Manipulating data frames

For stacking columns of a data frame, that is placing successive columns one under the other, the function `stack()` is available.

```
stack(Cars93.summary, select=c(1,2))  # concatenates
```

We get an additional column of class factor named `ind` that has names of the concatenated columns as levels.

The `unstack(my.Cars)` command reverses the stacking operation.

Applying transformations to a single column in a data frame is straightforward

```
hills$time<-round(hills$time*60)
```

Operations on data frames

Data frames can participate in arithmetic like operations. The usual rules apply, vectors will be recycled.

Some function will operate element-wise on data frames, like `log`.

Other times, we need to operate on columns only, and functions `lapply` and `sapply` provide functionality for such procedure.

The difference between the two functions is that the former returns a list, while the latter attempts to simply the result into a vector or a matrix.

```
lapply(hills,max)    # max of each columns as a list
sapply(hills, max)   # the same, but as a vector
```

Identification of missing values

Many of the modeling functions will fail unless action is taken to handle missing values.

Two functions are useful when working with data frames. For example

```
!complete.cases(science)    # Which rows have missing values?
which(!complete.cases(science))
dim(science)                # 1385 observations
Science<-science[-ind,]
                            # Omit rows with missing values
dim(Science)                # 1383 observations
```

The function `complete.case` return a logical vector indicating which cases (i.e. rows) have no missing values. It takes as argument also a sequence of vectors and matrices.

Lists

We have mentioned lists a few times. Lists are the most general class in R . A list is simply a numbered collection of objects, of *any* class.

```
x.lis<-list( X1=1:10, X2=letters[1:3],
             X3=factor(rep(letters[1:4],2)) )
x.lis$X1
x.lis[[2]]   # 2nd element of the list
table(x.lis$X3)
```

We have already seen the use of `$` operator. Elements of a list can also be accessed by their index number, using the *double* square brackets operator `[[]]`.

The function `c()` can also be used with lists.

```
e<-list(e=10:1)
c(x.lis,e)
```

Matrices

While a data frame can collect together vectors of different class, and be displayed in a matrix-like manner, to explicitly operate on mathematical matrices we use the `matrix` class

It requires that the elements have the same type (numeric, character and so on).

To create a matrix

```
matrix(1:12,nrow=3,ncol=4)
```

By default, R fills the matrix by column. Note that it is sufficient to specify only the number of rows/columns.

A warning occurs if the vector being made into a matrix cannot recycle to `nrow`×`ncol`.

Matrices

A matrix has dimensions, accessed with the `dim` function.

Names can be associated with rows and columns using the function `dimnames`.

```
dimnames(x.mat)<-list( paste("row",1:3,sep=""),
                       paste("col",1:4,sep="") )
```

The names are a list, with a component for each dimension.

Here `paste(x,y,sep="")` create a character vector by concatenating element-wise `x` and `y` after converting them to characters. Terms are separated with the string specified by `sep`.

A similar trick can be used to create a matrix of characters

```
matrix( paste(paste("entry",rep(1:4, 2),sep=" "),
              rep(1:2, each=4),sep=","),
        nrow=4,ncol=2 )
```

Combining Data

We have seen that vectors can be combined with the function `c()`.

Matrices and data frames can be combined using the function `rbin` and `cbin`, for row and column-wise combination, respectively

```
xx<-cbind(x.mat,x.mat)
xxx<-rbind(x.mat,x.mat)
rbind(xx,xxx)
         # Error: dimensions do not coincide
```

Two vectors of equal length can be combined either to create a matrix or a data frame

```
data.frame(year=Year,carbon=Carbon)
matrix(c(Year,Carbon),nrow=length(Year))
cbind(Year,Carbon)
         # column names are automatically assigned
```

Indexing

We have seen different ways of selecting elements of vectors. Similar methods exist with matrices and data frames.

To refer to the *i*-th row, *j*-th column element of a matrix or data frames, use `x.mat[i,j]`.

Other ways of indexing objects, such that `i` and `j` can be:

- a vector of positive or negative integers;
- a logical vector of same size;
- a vector of character strings.

Matrix algebra

Element-wise operations (recycling rules applies whenever needed):

```
x.mat<- matrix(1:10, ncol=2)
x.mat + 1
x.mat + x.mat
```

Matrix multiplication:

```
x.mat %*% t(x.mat)
x.mat %*% x.mat        # non conformable matrices
```

where `t()` is the matrix transpose function. If the matrix and vector dimensions do not conform, an error message results.

Note the use of `sample()` in

```
matrix(sample(1:10,24,replace=T),ncol=6)
              # with replacement
matrix(sample(1:10, 24),ncol=3)
              # without replacement (error since 24>10)
```

Matrix algebra

To compute

$$X^T y$$

where X is a 4×6-matrix and y is a 4-dim vector, consider

```
t(X) %*% y
crossprod(X,y)
```

Note that the latter is more efficient. The function `crossprod` with a single matrix argument X computes $X^T X$.

Linear algebra functions are available: `eigen`, `det`, `solve`, ...

```
A<-crossprod(t(X))       # symmetric matrix
v<-eigen(A)              # list with eigenvalues
                         # and eigenvectors of A
det(A); prod(v$values)   # det(A)=prod_i(eigenvals(A)_i)
solve(A)                 # inverse of A
```

Operating on Matrices

We use the `apply` function to do an identical computation on rows or columns of a matrix. The function prototype is

```
apply(P, MARGIN, FUN, ...)
```

where `P` is a matrix, `MARGIN` refers to rows ($=1$) or columns ($=2$), `FUN` is the name of the function to be applied to each row or column, and `...` is a special symbol meaning extra optional arguments, in this case of the function `FUN`.

```
apply(P,1,sum)        # rows
apply(P,2,sum)        # columns
```

Where possible, we prefer using `apply` to explicit looping, for efficiency reasons (see later).

Built-in functions (1)

```
all()         # returns TRUE if all values are TRUE
any()         # returns TRUE if any values are TRUE
args()        # information on the arguments to a function
cat()         # prints multiple objects, one after the other
cumprod()     # cumulative product
cumsum()      # cumulative sum
diff()        # form vector of first differences
              # note that diff(x) has one element less than x
is.factor()   # returns TRUE if the argument is a factor
is.na()       # returns TRUE if the argument is an NA
              # see also is.logical(), is.matrix(), etc.
ls()          # list names of objects in the workspace
length()      # number of elements of a vector or of a list
mean()        # mean of the elements of a vector
median()      # median of the elements of a vector
```

Built-in functions (2)

```
order()       # x[order(x)] sorts x (by default, NAs are last)
print()       # print a single R object
range()       # minimum and maximum value elements of vector
sort()        # sort elements, by default omitting NAs
rev()         # reverse the order of vector elements
str()         # information on an R object
summary()     # statistical summaries of an R object
table()       # form a table of counts (for factor)
xtabs()       # form a table of totals (for factor)
unique()      # form the vector of distinct values
which()       # locate 'TRUE' indices of logical vectors
with()        # computation using col of a specified data frame
```

Functions Arguments

To examine the arguments of a function use `args()`

```
args(c)     # function (..., recursive = FALSE)
            # NULL
args(lm)    # function (formula, data, subset, weights,
            #  na.action, method = "qr", model = TRUE,
            #  x = FALSE, y = FALSE, qr = TRUE,
            #  singular.ok = TRUE, contrasts = NULL,
            #  offset, ...)
```

In the first case `NULL` is returned, meaning unspecified arguments. In the second case the arguments include `...`, referring to unspecified arguments.

Arguments lists include named values with specified defaults, in the format `name=value`.

Calling Functions

We have seen functions calls with both specified and unspecified arguments. In calling function, arguments can either be specified by

- Order. Provide arguments in the order given by the function prototype.
- Name. Provide arguments explicitly by name, as `name=value`. Only sufficient letters of the name to uniquely identify it are required.

This two approaches can be mixed. For example

```
plot(Year,Carbon,type="l")
```

Graphics in R

More on the use of the function `plot()`, adding text, points, lines, colors and labels.

Start with the data frame `primates`.

```
plot(Brainwt ~ Bodywt, data=primates)
```

Improve by putting labels on the axes and the points, as

```
attach(primates)
plot(Brainwt ~ Bodywt, xlim=c(0, 300),
     xlab="Body weight (kg)",
     ylab="Brain weight (g)", pch=16)
text(Brainwt ~ Bodywt,
     labels=row.names(primates), pos=4)
mtext("Primates data", 3, line=1, cex=1.5)
```

Graphics in R

- Specify `xlim` so that there is room for the labels
- Give names to the axes via `xlab` and `ylab`
- The setting `pch=16` gives a solid black dot
- The `text()` function requires the coordinates and the labels to be displayed, while `pos=4` places text to the right of the points
- The function `mtext(text,side,line,...)` adds text in the margin of the plot, `cex=1.5` controls the size of the text.

Use the `points()` function to add points to a plot. Use the `lines()` function to add lines to a plot. Can specify the color with `col=...`.

R Object Storage

R objects that we create occupy a *workspace*, the work environment, that we examine with

```
ls(all=T)          # see the contents of the workspace
objects()          # objects created by the user
```

Note that names that start with a period are hidden from the `ls()` command, and are useful system objects, like `.Random.seed`.

There is a collection of databases that R uses to store objects. This collection is maintained in a list called the *search path*, accessed with the `search()` function.

The default behaviour of `attach` put a list in the path, such that its elements can be accessed by name.

R Object Storage

All objects in the workspace are stored in memory.

When exit R we are prompted to save the workspace: the workspace is stored in its current state in a file (called `.RData` by default) and saved in the working directory.

It can be recovered for future work at a future date (select the working directory, then from the File menu select Load Workspace).

It is possible to have multiple `.RData` files, and switch between them. This provides a convenient mechanism for collecting together different projects.

It might be convenient to delete non necessary objects from the workspace with `rm(X)`, specially when we are doing memory intensive computation.

To remove all currently defined objects, use `rm(list = ls())`.

Importing Data in R

If data is represented in a file in a simple delimited tabular format, it is easiest to use `read.table`. Use `write.table` to write a data frame to a file.

```
my.hills<-read.table("hills.txt",header=TRUE)
```

Note the use of `header=TRUE` to ensure that R uses the first line to get names for the columns.
The `scan` function is sometimes useful for numeric vectors (use `write` to write a vector or a matrix to a file).

```
write(Carbon,file="Carbon.txt")
x<-scan("Carbon.txt")
```

Importing Data in R

We have assumed that the fields in `hills.txt` are separated by space and/or tab, as allowed by the default setting (`sep=""`) for `read.table()`. Other parameter setting are sometimes required; note in particular

```
fossifuel<-read.table("fuel.csv",header=TRUE,sep=",")
```

reads data from `fuel.csv` where fields are separated by commas.

Importing from other systems

The R Data Import/Export document says (chapter 3):

"*...reading a binary data file written by another statistical system. This is often best avoided...*"

This is an area where R has a limited functionality, for example in reading from Minitab, S-PLUS, SAS, SPSS.

Import is possible from spreadsheet style regular grids in text formats. Direct access to a `.xls` can be managed, but it is not recommended. Better to output the desired parts as a simple delimited tabular format.

Some additional functionality can be achieved by using appropriate packages.

R Getting Help

R has various interactive help facilities. The most useful, to access the manual pages for a specific command, is simply to use the `help()` function. For example

```
help(mean)
?mean        # the same
```

To conduct a more general search, akin to UNIX `apropos`, use `help.search`. For example

```
apropos("mean")
help.search("regression")
```

The function `help.start()` will open up the HTML help system. Lots of good stuff there!

Most help pages have interesting examples. Try `example(mean)`.

Portability

There are a variety of formats R graphics can be exported to.

One option is to create the figure, then from the File menu, select Copy to the Clipboard and select appropriately for the target application (for example a Word file).

Another option is to save the file in a specific format. Again, this can be done by the File menu, options being Metafile, Postscript, Pdf, Jpeg and so on.

This can also be achieved with commands, by embedding the graphics commands between a call to a driver and a driver termination call.

Portability

Example (pdf)

```
pdf("file.pdf")
... graphics commands
dev.off()

pdf("file.pdf",height=8,width=8)
... graphics commands
dev.off()          # figure region specified by height & width

pdf("file.pdf",paper="special",height=8,width=8)
... graphics commands
dev.off()          # paper size specified by height & width
```

Basic programming

A program is just a list of commands, which are executed one after the other. Typically a program has three parts: input, computation, output. A convenient way of writing a program is by collecting the commands in a script file and executing them by highlighting followed by `<Control> r`.

Suppose that we have a program saved as `prof.R` in the working directory: we can execute the whole program by using the command

```
source("prog.R")
```

The convenience of saving our programs in a file and execute them in this way is that it allows us to easily modify the program code and to re-run it with different output.

Note the use of `print()` to display the value of a variable inside a call of `source()`. The same applies for printing results inside a loop.

Conditional Evaluation

It is often that we want a program to take different actions according to a condition. The R language statement `if` provides this functionality. The general format is

```
if(logical_expression) {
  true.branch
} else {
    false.branch
}
```

When the `if` expression is evaluated, if `logical_expression` is `TRUE` then the first group of expressions is executed (and not the second). Conversely, if `logical_expression` is `FALSE` then only the second group of expressions is executed.

The `else` part of an `if` statement is optional. Remember you should not have a new line between `}` and `else` to avoid a syntax error in entering an `if ... else` construct.

Note the use of indenting to try to clarify structure. If there is only one expression in `true.branch` (or `false.branch`), then the braces are optional.

Conditional Evaluation

In R , `if` statement can occur within the branches of another `if` statements, that is, they can be nested. For example

```
if (x>3)
  if (x<6) {
    count<-1
  } else count<-2
```

It is often useful to have compound conditions. We can combine them with the logical operators `&` and `|`, for example

```
if (x>3 & x<6){
  count<-1
} else count<-2
```

check the difference with respect to the previous commands.

Iteration

There exist two different types of iteration construct: count controlled loops, provided by the `for` statement, and variable length loops, provided by the `while` statement (see also `repeat`statement).

WARNINGS: bad use of loops is the most common source of inefficient R code. This is particularly true in nested loops. Always think hard how to use functions like `apply` rather using a loop.

Of course, sometimes it is unavoidable.

Looping with `for`

The general format of the `for` statement is

```
for (x in vector) {
     expression_1
     ...
    }
```

When executed, the `for` command executes the group of expressions within braces `{ }`, once for each element of `vector`. The grouped expressions can use `x`, which takes on each values of the elements of `vector` as the loop is repeated. When there is a single expression, braces `{ }` can be omitted.

```
x_list <- seq(1,9,by=2)
sum_x<-0
for (x in x_list) {
     sum_x<-sum_x+x
     cat("The current loop element is",x,"\n")
     cat("The cumulative total is",sum_x,"\n")
    }
```

Note that to sum the elements of a vector, it is much easier (but less instructive) to use the built-in function `sum`.

Looping with `for`

The variable `x` is often used for indexing, as in

```
x<- (1:100)*(pi/40); y<-numeric(0,length(x))
for(i in 1:length(x))  y[i]<-sin(x[i])
```

In the example, you can see just how inefficient this is compared to using a vectorised function like `sin`.

Redimensioning of an array can slow down computation.

```
n<-1000000
x<-rep(0,n)
for (i in 1:n) {
     x[i]<-i
    }
```

```
n<-1000000
x<-1
for (i in 1:n) {
     x[i]<-i
    }
```

The first is faster, the reason being that changing the size of a vector takes just about as long as creating a new vector does.

Looping with `while`

Whenever we do not know beforehand how many times we need to go around a loop, we can use a `while` loop: each time we go around the loop, we check some condition to see if we are done yet.

```
while (logical_expr) {
   expression_1
   ...
}
```

When a `while` command is executed, `logical_expr` is vaulted first. If it is `TRUE` then the group of expressions in braces `{}` is executed. Control is then passed back to the start of the command: if `logical_expr` is still `TRUE`, then the grouped expressions are executed again, and so on.

Clearly, for the loop to stop eventually, `logical_expr` must eventually be `FALSE` To achieve this `logical_expr` usually depends on a variable that is modified within the grouped expressions.

Basic debugging

You will spend a lot of time correcting errors in your programs. To find an error or *bug*, you need to be able to see how your variables change as you move through the branches and loops of your code.

An effective and simple way of doing this is to include statements like `cat("var=", var, "\n")` throughout the program to display the values of variables such as `var` as the program executes.

```
## program threexplus1.R
x <- 3
for (i in 1:3) {
  show(x)
cat("i = ", i, "\n")
  if (x %% 2 == 0) {
    x <- x/2
  } else {
    x <- 3*x + 1
  }
 }
show(x)
```

Output

```
[1] 3
i = 1
[1] 10
i = 2
[1] 5
i = 3
[1] 16
```

Table 3.1 Charting the flow for program `threexplus1.R`

line	x	i	comments
1	3		i not defined yet
2	3	1	i is set to 1
3	3	1	3 written to screen
4	3	1	(`x %% 2 == 0`) is `FALSE` so go to line 7
7	10	1	x is set to 10
8	10	1	end of else part
9	10	1	end of for loop, not finished so back to line 2
2	10	2	i is set to 2
3	10	2	10 written to screen
4	10	2	(`x %% 2 == 0`) is `TRUE` so go to line 5
5	5	2	x is set to 5
6	5	2	end of if part, go to line 9
9	5	2	end of for loop, not finished so back to line 2
2	5	3	i is set to 3
3	5	3	5 written to screen
4	5	3	(`x %% 2 == 0`) is `FALSE` so go to line 7
7	16	3	x is set to 16
8	16	3	end of else part
9	16	3	end of for loop, finished so continue to line 10
10	16	3	16 written to screen

Programming tips

Good programming is clear rather than clever.

1. Solve the simplest possible version of the problem, then add complexity.
2. Make pilot runs of your code, using simple starting conditions for which you know what the answer should be.
3. Graphs and summary statistics of intermediate outcomes are helpful, the code to create them is easily commented out for production runs.
4. Careful use of indentation improves readability of the code, specially with an `if` statement or a `for` or `while` loop.
5. Use blank lines to separate sections of code.
6. Variables names should be descriptive, that is they should give a clue of what they represent.
7. Start each program with some comments giving name, author, date and what the program does. Document the program in details.
   You will find that even programs you write yourself can be very difficult to understand after only a few weeks have passed!

Exercises

- Chapter 2 Maindonald and Braun (2007)
  Ex 1, Ex 2, Ex 3, Ex 4, Ex 5, Ex 6
- Chapter 1 Verzani (2005)
  Ex 1.4–1.13, Ex 1.14–1.19, Ex 1.20–1.27

Ex 1 Chapter 1 M&B (2007)

The following table gives the size of the floor area (ha) and the price ($A000), for 15 houses sold in the Camberra (Australia) suburb of Aranda in 1999.

	area	sale.price		area	sale.price		area	sale.price
1	694	192.0	6	963	185.0	11	790	221.5
2	905	215.0	7	821	212.0	12	696	255.0
3	802	215.0	8	714	220.0	13	771	260.0
4	1366	274.0	9	1018	276.0	14	1006	293.0
5	716	112.7	10	887	260.0	15	1191	375.0

Type this data into a data frame with column names `area` and `sale.price`.

(a) Plot `sale.price` versus `area`.

(b) Use the `hist()` command to plot a histogram of the sale prices.

(c) Repeat (a) and (b) after taking logarithms of sale prices.

Ex 2 Chapter 1 M&B (2007)

The data frame `orings` gives data on the damage that had occurred in US space shuttle launches prior to the disastrous Challenger Launch of January 28, 1986. The observations in rows 1, 2, 4, 11, 13, and 18 were included in the pre-launch charts used in deciding whether to proceed with the launch, while remaining rows were omitted.

Create a new data frame by extracting these rows from `orings`, and plot `Total` incidents against `Temperature` for this new data frame. Obtain a similar plot for the full data set.

Ex 3 Chapter 1 M&B (2007)

For the data frame `possum`:

(a) Use function `str()` to get information on each of the columns

(b) Using the `complete.cases()`, determine the rows in which one or more values is missing. Print those rows. In which columns do the missing values appear?

Ex 4 Chapter 1 M&B (2007)

For the data frame `ais`:

(a) Use function `str()` to get information on each of the columns. Determine whether any of the columns hold missing values.

(b) Make a table, that shows the numbers of males and females for each different sport. In which sports is there a larger imbalance (e.g. by a factor of more than $2:1$) in the numbers of the two sexes?

Ex 5 Chapter 1 M&B (2007)

Create a table that gives, for each species represented in the data frame `rainforest`, the number of values of `branch` that are NAs, and the total number of cases.

[Hint: Use either `is.na()` or `complete.cases()` to identify NAs.]

Ex 6 Chapter 1 M&B (2007)

Create a data frame called `Manitoba.lakes` that contains the lake's `elevation` (in meters above sea level) and `area` (in square kilometers) as listed below. Assign the names of the lakes using the `row.names()` function.

```
               elevation area                 elevation area
Winnipeg             217 24387   Island             227 1223
Winnipegosis         254 5374    Gods               178 1151
Manitoba             248 4624    Cross              207 755
SouthernIndian       254 2247    Playgreen          217 657
Cedar                253 1353
```

Plot lake area against elevation, identifying each point by the name of the lake. Because of the outlying value of `area`, use of a logarithmic scale is advantageous.

(a) Add labeling information for area. Explain how distances on the scale relate to changes in area.
[xx Doubling the area increases `log2(area)` *by 1.0 xx]*

(b) Repeat the plot and associated labeling, now plotting `area` versus `elevation`, but specifying `log="y"` in order to obtain a logarithmic *y*-scale.

Ex 7

Plot the graph of the following functions for $x \in [0, 10]$

$$f(x) = \begin{cases} 2 + 3x & x \leq 5 \\ -0.5x^2 + 2x & x > 5 \end{cases}$$

$$h(x) = \begin{cases} 3x^3 & x \leq 3 \\ \log(x) & 3 < x \leq 5 \\ -0.5x^2 + 5 & x > 5 \end{cases}$$

Resources

- BOOKS
    - Owen J., Maillardet R. and Robinson A. (2009).
      Introduction to Scientific Programming and Simulation Using R.
      Chapman & Hall/CRC.
    - Verzani, J. (2005).
      Using R for Introductory Statistics. Chapman & Hall/CRC.
    - Venables, W.N. and Ripley, B.D. (2002).
      Modern Applied Statistics with S. 4th edn. Springer.
    - Maindonald, J. and Braun, J. (2007).
      Data Analysis And Graphics Using R. 2nd edn. Cambridge University Press.

Resources

- WEB
  - R software:
    `http://www.r-project.org/`
  - Owen, et al. (2009):
    `http://www.ms.unimelb.edu.au/spuRs/`
  - Verzani (2005):
    `http://wiener.math.csi.cuny.edu/UsingR/`
  - Venables and Ripley (2002):
    `http://www.stats.ox.ac.uk/pub/MASS4//`
  - Maindonald and Braun (2007):
    `http://wwwmaths.anu.edu.au/~johnm/r-book.html`
  - CRAN Task View in Empirical Finance:
    `http://cran.mirror.garr.it/mirrors/CRAN/`
    (go to *Task Views*)