Introduction to R

David Stephens and Niall Adams

[d.stephens,n.adams]@imperial.ac.uk

Department of Mathematics, Imperial College London

9th November 2005

http://stats.ma.ic.ac.uk/das01/public_html/RCourse/

Timetable

Part 1	Introduction
Part 2	Demos, Essentials
Part 3	Basic analysis techniques
Part 4	Programming Constructs
Part 5	Writing Functions
Part 6	Automation and Efficiency Issues

All parts include demos and tutorials.

We will be using R for Windows, v2.0.1.

Some familiarity with probability and statistics is assumed.

R

From the R documentation:
R is a GNU project that provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, 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

From the R documentation

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License 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 (`http://www.insightful.com/`). This development has been mainly in the user interface and graphics and interoperability functionality. Splus uses version 4 of the S language.

R started out as an Open Source system “not unlike” version 3 of the S language, developed by Ross Ihaka and Robert Gentleman, Based on a different implementation. R is now developed and maintained by a core group.

Getting R

Can always download the most recent release (currently 2.1.1) either as source or pre-compiled of R from a Comprehensive R Archive Network (CRAN) site (`http://cran.uk.r-project.org/` for UK).

R consists of a *base system*, providing the language and 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. Different releases are deliberately installed in separate locations, so it is possible to have concurrent versions.

Install R

To obtain and install a pre-compiled windows binary:

1. `rwXXXX.exe` from CRAN (currently `rw2011.exe`).
2. Execute, and follow instructions. Defaults are sensible.
3. Start `R` .
4. Select `packages|install`
5. Select a CRAN mirror
6. Select the required packages

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

Demo: Starting and Packages

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!

Demo: Simple Data Analysis

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

- Interaction via a command line interface (scripts later)
- The value of a function may be assigned a name.
- Graphics occurs as side effects to function calls.
- Functions can operate on arguments of different types.
- Prompt to save workspace with `q()`

Data

Consider

```
hills <- read.table("hills.txt")
```

We have called the function `read.table`, with a single, unnamed argument, and assigned, via the assignment operator `<-`, the value of the call to the object `hills`.

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, `read.table` is an object of class `function`, and `hills` is an object of a type determined by the function, in this case, a `data.frame`. Everything we do will involve creating and manipulating objects.

Data

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

Another common irritation for new users is that `R` is cases sensitive, so `T` (a system representation of logical true) is different to `t` (function for matrix transpose).

Functions

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

```
plot(x,y)
plot(lm(time~dist))
```

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

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

Demo: Vectors

Consider the quadratic equation

$$x^2 + x + 1 = 0$$

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

$$b^2 - 4ac > 0$$

- Store coefficients as an object
- Compute discriminant
- Construct a plot

Vectors

Consider

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

Here `coeffs` is a *vector* of type `numeric`. Vectors are a fundamental class in `R` – there are no scalars. All objects also have a `length`, which is only informative for certain classes.

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

Vectors can have names, queried and modified by the `names` function. Note the function call occurring on the right hand side of the assignment operator.

Vectors

The classes available for vectors are

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

Vectors

Vectors can be created other ways

```
seq(-3,3,length=200) # regular spaced points

-2:2 # sequence of integers

x <- vector("numeric",length=20) # create

c(x,x) # combine two vectors
```

Care must be taken in combining vectors of different types. R will deploy a set of internal rules to resolve the class of the combined vector.

Note that # is the comment operator.

Vectors

The classes available for vectors are

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

Vectors

Access the elements of a vector by number, or name

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

We may wish to remove the names from a vector

```
names(coeffs) <- NULL
```

The object `NULL` represents nothing, the empty set. `NULL` has length zero, and `R` will deploy special rules when `NULL` occurs.

Arithmetic

Simple arithmetic on constants follows the usual precedence rules

```
ax <- 1 ; bx <- 2 ; cx <- 3
x <- ax+bx*cx # x =7
x <- bx/bx*cx/ax+bx # x=5
x <- (bx/bx)*(cx/ax)+bx # better
axˆbx # 2 raised to 3
10 %% 9 # 10 mod 9
```

Use parentheses to simply expressions. Note that these must balance. If they do not R responds with either a syntax error, or a continuation request (a + prompt).
Recall that vectors are a fundamental class of object in R .
The examples above therefore involve arithmetic on vectors, like ax.

Arithmetic

Things get more complicated. Consider

```
x <- seq(-3,3,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. Fractional recycling will result in a warning message.

A vector can have zero length. This is represented as

`numeric(0)` As a function call, this creates a zero length

vector. This can be useful if we need to construct a vector of unknown length.

Special Values

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

- Missing values are represented as `NA`. The IEEE special values for floating point arithmetic are also used: `Inf`, `-Inf` and `NaN`. `NaN` is used for indefinite results like `Inf/Inf`.
- There are functions for elementwise 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, and if we are being careful, we should check for their presence.

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); log10(x) ; sqrt(x) ; exp(x);
sin(x) ; acos(x); tanh(x) ; # radians
```

In each case, the result of the function is a vector of the same length as the argument. Note that `log` can take a second argument, the base of the logarithm. The default base is e. Note that the following are equivalent

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

Simple Functions

Standard functions for reducing numeric vectors

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

R Demo: Logic

A full set of logical operators and functions are available.

Logic

Logical conditions can be applied to numeric vectors

```
x <- seq(-3,3,length=200) > 0
```

Now x is a logical vector of length 200. The condition $>$ has been applied elementwise.

The other comparison operators are >=, < , <=, ==,and !=. The final two are *exact* equality and inequality, respectively. As such, they should only be applied to entities that are represented exactly, like integers.

Logical vectors are subject to the usual recycling rules.

Logic

Logical values can be combined and modified with `!`, the negation operator, `&` the intersection operator (logical AND) and `|`, the union operator (logical OR).

Truth tables

A	B	A AND B	A OR B
T	T	T	T
T	F	F	T
F	F	F	T
F	T	F	F

```
c(T,T,F,F) & c(T,F,F,T)
```

Be careful with negation, the symbol `!` is overloaded,and may be interpreted as a shell escape. Again, brackets are used to resolve such problems.

Logic

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

```
A <- c(T,F,T)
A + 1
```

The resulting vector is `c(2,1,1)`. `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. There are a range of *coercion* functions, pre-fixed with "as", like `as.logical`.

The use of logical vectors in ordinary arithmetic means we can easily count numbers of `TRUE` or `FALSE` in a comparison `sum(x > 0.5)`

Functions for Logical Vectors

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

```
sum(x)> 1 #x logical
sum(x) == length(x)
```

Sometimes useful are the functions for set operations : `union` - $A \cup B$,`intersect` - $A \cap B$ and `setdiff` - $A \cup \bar{B}$.
I have found `setdiff` very useful in classification experiments.

Factors

`factors` are vector like objects (class?) used to store sets of categorical data. For example

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

Here, we have constructed a vector of class character, and converted it to a factor. The factor is displayed without quotes.

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

```
unclass(drinks)
```

Refer to individual elements in the same way as a vector.

Data Frames

The typical class used for data analysis in R is the *data frame*. These are suitable for storing the usual observations in rows, variables in columns data format. The advantage of this class is that the columns can be of different classes: numeric, logical, character and so on.

We have already seen an example of a data frame, the object `hills`.

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

```
row.names(hills)
```

For a data frame, the column names are accessed with the function `names`. Strictly, a data frame is a *list*, where all elements are required to have the same 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 names can be provided as an extra argument.

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

Data Frames

Earlier, we used the attach command to make the columns of the hills data frame available by name. This is sometimes useful, although there is a risk of masking other objects. To undo, use `detach(hills)`.
If a data frame has names, we can refer to the columns using the `$` operator as follows

```
hills$time
```

This is now simply a vector,and can be manipulated as such. The real power of factors arises when they are constituents of data frames. In a statistical model, the factor will be treated in an appropriate manner.

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, which requires that all elements have the same type.

Create a matrix with something like

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

Matrices

Note the use of named arguments in the function call. A warning occurs if the vector being made into a matrix cannot recycle to `nrow` $\times$ `ncol`. A matrix has dimensions, accessed with the `dim` function, that returns the number of rows and columns.

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

Here, the names are a list, with a component for each dimension.

Combining Data

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

Matrices and data frames be combined using he function `rbind` and `cbind`, for row and columnwise combination respectively. For example

```
xx <- cbind(x,x)
xxx <- rbind(x,x)
rbind(xx,xxx) # Error
```

Note that the dimensions of the objects being combined must be compatible.

Indexing

We have seen that we can refer to individual components of vectors. More general facilities are available for selecting components from vectors, matrices and data frames.

To refer to the ith row, jth column element of a matrix or data frame, use `x[i,j]`.

There are more general ways of indexing objects, such that `i` and `j` can be: a vector of positive or negative integers, a logical vector, a vector of character strings, or empty.

Indexing

Examples

```
x <- 1:10
names(x) <- letters[x]
x[1:3], # elements 1,2,3
x[c(1,10)] # elements 1 and 10
x[c(-1,-2)] # remove 1 and 2
x[ x > 5] # elements > 5
x[c("a","d")] # elements 1 and 4
x[] # all elements
jj1 <- matrix(1:100,ncol=10)
jj1[1:5,] # first 5 rows, all cols
jj1[1:4,x[x <3]]
```

Manipulating data frames

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

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

Groups of columns can be handled similarly, by an appropriate indexing operation.

```
x.df[,1:3] <- x.df[,1:3]/2
```

Note that the division operator here is applied element wise to each element.

Operations on data frames

As we saw above, data frames can participate in arithmetic like operations. The usual rules apply, vectors will be recycled - sometimes giving strange results.

Some functions will operate elementwise on data frames, like `log`.

Other times we need to be operate on columns only, and the functions `lapply` and `sapply` provide functionality for such a procedure. The difference between the two function is that the former returns a list, while the latter attempts to simply the result into a vector or matrix.

```
x <- matrix(1:10,ncol=2)
lapply(x,max)
sapply(x,max)
```

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(a=1:10,b=letters[1:3],b=matrix(1:10,ncol=2))
x.lis$1
x.lis[[2]]
```

We have already seen the use of the $ operator. Elements of a list can also be accessed by their index number, using the *double* square brackets operator. The usual indexing operations can also be applied to a list.

The c function can also be used with lists.

Operating on Matrices

We distinguish computation with data frames from computation with matrices. We have elementwise computations

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

As usual we need to be careful about how recycling rules (which are complex for such situations) will apply. We also have matrix multiplication from linear algebra

```
x.mat %*% t(x.mat)
```

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

Operating on Matrices

To compute

$$X^\top y$$

where X is a vector and y is matrix, could consider

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

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

Typical linear algebra functions are available: `eigen`, `svd`, `qr`, `solve`, and so on.

Operating on Matrices

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

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

where `X` 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, for the function `FUN`.

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

Where possible we prefer using `apply` (and the related function `sweep`) to explicit looping, for efficiency reasons.

Object Attributes

An attribute is an R object attached to another R object by name. Objects can have any number of attributes, which are represented as a list.

For example, the dimensions (and dimnames) of a matrix are attributes.

```
attributes(x.mat)
```

Attributes are often used for storing ancillary information, derivative information in optimisation problems, for example.

Function Arguments

To examine the arguments for a function use `args`.

```
args(c)
args(pmax)
args(lm)
```

In the first case `NULL` is returned, meaning unspecified arguments, where an arbitrary number of arguments can be given.

In the second and thirds cases, the arguments include `...`, referring to unspecified arguments.

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

Calling Functions

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

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

These two approaches can be mixed. For example

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

Transformations

In R , transformations are straightforward using the algebraic operators outlined above:

```
# Square
y<-x^2

# Exponential
y<-exp(-x)
```

Transformations

```
#log (to various bases)
#Base e (natural log)
y<-log(x)

# Base 10
y<-log10(x)

# Base 2
y<-log2(x)

# Base 7.2
y<-log(x,base=7.2)


# Square root
y<-sqrt(x)
```

R Object Storage

R objects that we create occupy a *workspace* that we can examine with

```
ls(all=T)
objects()
```

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

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

R Object Storage

The default behavior of `attach` puts a list in the second position of the path, such that its elements can be accessed by name. The database in position 1 is the working database.

All objects in the workspace are stored in memory. When we exit `R` , we are prompted to save a workspace image. Doing this means that the workspace is stored in its current state, in a file (called `.RData` by default), and can be recovered for further work at a future date.

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

R Object Storage

Storing everything in memory has some implications for how we work, especially if we are doing memory intensive computations. In such cases keep the number of duplicate and intermediate results should be kept to a minimum. A handy trick is to use names like `jj1`, `jj2` for such results, then routine delete them with

```
rm(list=objects(pattern="jj*"))
```

The function `object.size` provides an estimate, in bytes of the memory allocation for an object.

Scripts

The R console provides a convenient interface for simple commands. For more complicated work, such as programming, R provides scripts, where a sequence of R command can be entered, and processed later. To start a new script choose

```
File -> 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 work mode.

R Simple Data IO

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.

The `scan` function is sometimes useful for numeric vectors. This can be used both interactively, for entering numbers, or for reading a stream of numbers from a file.

To manually enter data, create a dummy data frame, and invoke the data editor, as follows

```
a.df <- data.frame()
fix(a.df)
```

Importing from other systems

The R Data Import/Export document says ". . . reading a binary data file written by another statistical system. This is often best avoided. . . ". This is an area where R is not as evolved as Splus. R has limited functionality for reading binary objects from EpiInfo, Minitab, S-PLUS, SAS, SPSS.

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

Some limited access to RDBMS systems is possible, 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 `?` function. For example

```
?mean
```

Like UNIX manual pages, the R manual pages include a "See Also" and "Examples" section, which can be very useful. To conduct a more general search, akin to unix `apropos`, use `help.search`. For example

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

The function `help.start` will fire up the HTML help system. Lots of good stuff here!

Demo: Simple Statistics

Numeric Summaries

Finally, we can start to look at some statistical functions. A full range of summary statistics are available, including

```
mean(x) ; mean(x,trim=0.95)
median(x)
sqrt(var(x)); var(x,y); var(x.mat)
range(x)
cor(x); cor(x,y); cor(x.mat); cor(x.mat,y.mat)
```

Note different behaviour for `cor` and `cov` depending on the argument list. Some functions have functionality for dealing with missing values (`Na`). The `mean` function, for example, has argument prototype `na.rm=FALSE`.

Summary Function

The summary function is particularly useful with many of the statistical functions. When applied to a numeric vector

```
summary(1:20)
```

the function produces a 6 point summary, comprising the minimum, maximum, lower and upper quartiles, and the median and mean. Applied to a matrix, the summary function generates this data for each column.

If the arguments include missing values, `summary` additionally counts the number.

Probability Distributions

R includes functions for computing quantities associated with a variety of distributions. The generic prototype for these function is

`*dist(args)` where `*` is one of

- `p` (probability),
- `d` (density),
- `q` (quantile),
- `r` (random number)

and `dist` is the nickname of a distribution.

For example, to generate 20 (pseudo) random variates for a specific Normal distribution

```
rnorm(20,mean=2,var=3)
```

whereas for a Chi-squared distribution

```
rchisq(20,df=5)
```

Extra distributions are available in the package `SuppDists`.

Random number generation

R provides a collection of sophisticated random number generators. The default is the "Mersenne Twister". This requires a start value, called the *seed*. The random number sequence is completely specified by this. It is sometimes useful to fix the seed so as to achieve a repeatable sequence. Do this with

```
set.seed(1)
```

To sample from a finite population, use the function `sample`. For example, to sample from a biased coin experiment

```
sample(c("Head","Tail"), 10,probs=c(0.3,0.7),replace=T)
```

By default, all elements of the population are sampled with equal probability.

Graphical Summaries

R has impressive graphics functionality - except for dynamic graphics. A potential downside for new users is that there is no GUI for graph construction.

A common data analysis task is comparing a sample with a distribution. This is often done with a QQ plot. Typically we plot theoretical quantiles on the horizontal axis, and empirical quantiles on the vertical axis. For example to compare a sample with an Exp(1) distribution

```
plot(qexp(ppoints(x),1),sort(x))
```

The function `ppoints` generates a sequence of probability points at which to evaluate the theoretical distribution.

Graphical Summaries

If our data is consistent with the theoretical distribution, the points should fall on a straight line through the origin

```
abline(0,1)
title("QQ plot") # add a title
```

Such plots are most frequently used in residual analysis.
Note that R provides function `qqnorm` for comparing against a standard normal distribution, and `qqplot` for comparing two samples.
Another simple way of comparing two samples is as follows

```
plot(c(x,y),rep(0:1,c(length(x),length(y))),xlab="",ylab="")
```

Note the use of the extra arguments to override the default axis labelling.

Graphical Summaries

Other useful graphical tools include the histogram, and the box and whisker plot.

```
hist(x,prob=T)
boxplot(x)
```

Histograms are generated using a default binning strategy. To specify the number of bins, modify the argument `nbreaks`. Specify the breakpoints by providing a vector for the argument `breaks`.

A box and whisker plot displays the median, upper and lower quartiles, and whiskers extending to the normal distribution based 5% and 95% points. All observations beyond these points are flagged as stars.

Graphical Summaries

Box and whisker plots, and to a lesser extent histograms, are useful for comparing multiple samples. Box plots can take a model specification (of which more later), or a list. For example, suppose vector `x` contains observations of three groups, and vector `ind` is a code, with a value 1,2, or 3 representing the group, then

```
boxplot(split(x,ind))
```

will produce a display with one box plot for each class. The function `split` divides an object according to values in an indicator vector.

Displaying overlaid histograms is more problematic, since we must match the breakpoints for each display. An alternative is generate three histograms in the same figure.

Graphical Summaries

The `R` function for general control of graphical parameters is `par`. This has many (!) arguments. The argument `mfrow` determines how many plots will be placed on the figure. For example, to display two separate histograms, one above the other

```
par(mfrow=c(2,1))
hist(x)
hist(y)
```

The vector valued argument `mfrow` is the number of rows and columns of plots to be placed in the figure. By default, the plots are displayed in row major order.

Graphical summaries: `plot` example

Consider the Pima Indians data: a collection of variables observed on a particular group of native American Indians who are healthy or have diabetes.

These data includes measurements of tricep, skinfold and blood glucose level. After attaching the data frame

```
plot(triceps,glucose,type="n",xlab="Tricep",ylab="glucose")
plot(triceps[diabetes=="neg"],glucose[diabetes=="pos"])
points(triceps[diabetes=="pos"],glucose[diabetes=="pos"],col=2,pch=2)
```

We use the `type="n"` argument to set up a plot to accommodate all the data, without displaying anything, then put down the pieces separately.

R Graphical Summaries

The `col` argument specifies a colour, and the `pch` argument specifies the plotting symbol. It may be useful to add a legend, with the `legend` function. This function has (simplified) prototype

```
function (x, y = NULL, legend, ...
```

So we need to specify coordinates for the legend. For this example, a reasonable choice is

```
legend(40,50,c("Diabetes","Healthy"),pch=1:2)
```

If we had been plotting data connecting with lines we would use the argument `lty` which refers to line style.

Add text to a plot with the `text` function.

Portability

There are a variety of formats `R` graphics can be exported to. One option is to create the figure, then use `File -> Copy to the Clipboard`, and select appropriately for the target application.

Portability

Another option is to save the file in a specific format. Again, this can be done via the `File` menu.

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

```
jpeg("file.jpg")
...graphics commands
dev.off()
```

creates a JPG file containing the result of the graphics commands.

Simple Statistical Examples

First, simulate some data:
Set the random number generator seed:

set.seed(290905)

Simulate some Normal data:
Three groups, means 20, 40 and 50, variance 25.
Sample sizes 250, 150, 200.

```
x1 <- rnorm(250,mean=20,sd=sqrt(25))
x2 <- rnorm(150,mean=40,sd=sqrt(25))
x3 <- rnorm(200,mean=50,sd=sqrt(25))
```

Examples

```
# Summary statistics:
mean(x1)
var(x1)
sd(x1)
median(x1)
max(x1)
min(x1)
range(x1)
quantile(x1)
quantile(x1,prob=c(0.025,0.975))
summary(x1)
```

Graphical summaries

```
# Histogram
hist(x1)
hist(x1,nclass=20)
hist(x1,breaks=c(0:50))
hist(x1,breaks=c(0:50),main="My Histogram")

# Boxplot
group<-rep(c(1:3),c(250,150,200))
boxplot(split(c(x1,x2,x3),group))
```

Hypothesis Testing

Hypothesis Testing

Do a t.test for x1 against x2

```
t.test(x1,x2)
t.test(x1,x2,var.equal=T)
test1<-t.test(x1,x2,var.equal=T)
names(test1)
test1$statistic
test1$p.value
```

Hypothesis Testing

```
# Hypothesis Testing

# Do an ANOVA between the three groups
xvec<-c(x1,x2,x3)
aov(xvec~group)
summary(aov(xvec~group))
```

Co-variability

```
# Co-variability

# Scatterplot
a<-2
b<-1.5
x<-c(1:20)
y<-a+b*x+rnorm(10)
plot(x,y)
```

Regression

```
# Regression
lm(y~x)

y.xreg<-lm(y~x)
summary(y.xreg)

plot(x,y)
abline(y.xreg$coeff)

summary(y.xreg)$coeff
```

Plots

```
# Plots
# Univariate Functions
x<-c(1:100)/10
y<-(x^3)*exp(-2*x)
plot(x,y)
plot(x,y,type="l")
plot(x,y,type="l",col="red")

plot(x,y,type="n")
lines(x,y,col="red")

plot(x,y,type="n")
lines(x,y,col="red",lty=2)
```

Plots of Functions

Probability distributions

Plot the Chi-squared density function
for 1,2,3,4,5 degrees of freedom

```
y1<-dchisq(x,df=1)
plot(x,y1,type="l")
y2<-dchisq(x,df=2)
lines(x,y2,col="red")
y3<-dchisq(x,df=3)
lines(x,y3,col="blue")
y4<-dchisq(x,df=4)
lines(x,y4,col="green")
y5<-dchisq(x,df=5)
lines(x,y5,col="orange")
```

2D Plots

image, contour, persp

Plot out a function on a 50 $\times$ 50 grid

```
x<-c(0:50)
y<-c(0:50)
z<-matrix(0,nrow=50,ncol=50)
for(i in 1:50){
    for(j in 1:50){
        z[i,j]<-(x[i]-25)*(y[j]-25)
    }
}
image(x,y,z)

contour(x[1:50],y[1:50],z)
```

2D Plots

```
image(x,y,z,col=heat.colors(20))
contour(x[1:50],y[1:50],z,add=T)

image(x,y,z,col=terrain.colors(20))

persp(x[1:50],y[1:50],z)
```

2D Plots

```
# Another plot
x<-c(0:50)/10-2.5
y<-c(0:50)/10-2.5

z<-outer(x[1:50]^2,y[1:50]^2,"+")
z<-exp(-z/2)

image(x,y,z)
contour(x[1:50],y[1:50],z,add=T)
```

2D Plots

```
persp(x[1:50],y[1:50],z)
persp(x[1:50],y[1:50],z,xlab="x",ylab="y")

persp(x[1:50],y[1:50],z,phi=45)
```

R Functions

User-defined functions can be used in R. The main function definition syntax is

```
functionname <- function (args) {
   computation
   return(result)
}
```

where *args* is a set of arguments.

A function is called as follows

```
functionname(args)
```

Example

Here is a small function to evaluate the function

$$f(x) = \alpha_1 \exp\{-\lambda_1 x\} + \alpha_2 \exp\{-(\lambda_1 + \lambda_2)x\}$$

at any value of $x > 0$, for user-supplied parameters $(\alpha_1, \alpha_2, \lambda_1, \lambda_2)$.

The function needs to have the arguments

$$x, \alpha_1, \alpha_2, \lambda_1, \lambda_2$$

supplied (in some form) and return the function value

my.function

```
my.function<-function(x,al1,al2,lam1,lam2) {
  y<-al1*exp(-lam1*x)+al2*exp(-(lam1+lam2)*x)
  return(y)
}
x<-seq(from=0,to=10,length=101)
fx<-my.function(x,1.0,2.0,0.1,0.2)
plot(x,fx,type="l",ylim=range(0,4))
```

Alternative `my.function`

Supply the parameters as a vector $\theta = (\alpha_1, \alpha_2, \lambda_1, \lambda_2)$.

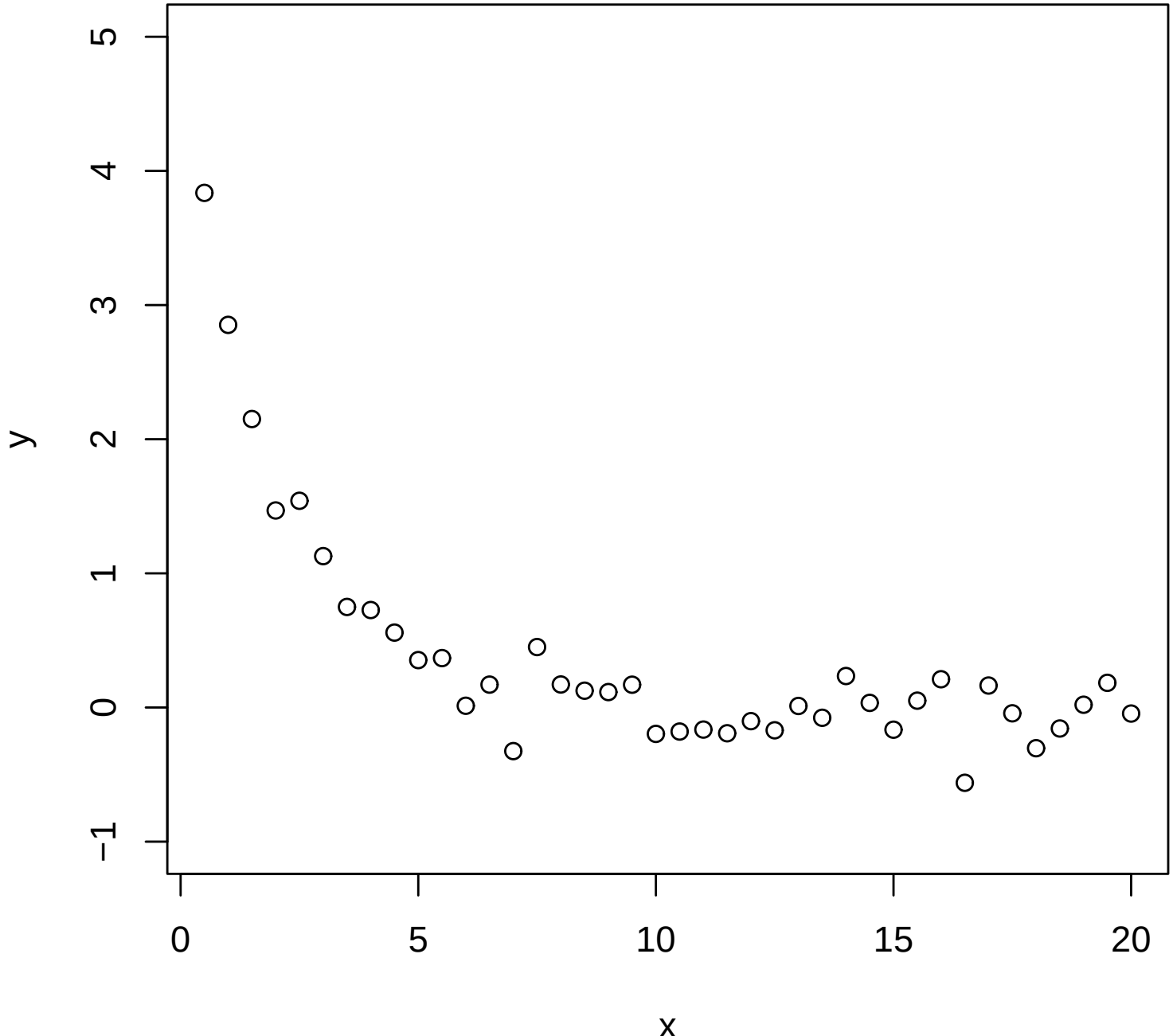
```
my.function.alt<-function(x,th) {

  y<-th[1]*exp(-th[3]*x)
  y<-y+th[2]*exp(-(th[3]+th[4])*x)
  return(y)

}
x<-seq(from=0,to=10,length=101)
theta<-c(1.0,2.0,0.1,0.2)
fx<-my.function.alt(x,theta)
plot(x,fx,type="l",ylim=range(0,4))
```

Exercise

We will simulate some data using the function above and try to recover the parameters

```
set.seed(300905)
theta<-c(3.0,2.0,0.5,0.2)
x<-c(1:40)/2
expected.y<-my.function.alt(x,theta)
y<-expected.y + rnorm(length(x),sd=0.2)
plot(x,y,ylim=range(-1,5))
```

Example Data



Least-squares Fit

We will slightly change the defined function so that the R minimization function `nlm` can be used to find the best fit.

```
my.function.new<-function(th,xvals,yvals){
  fy<-th[1]*exp(-th[3]*xvals)
  fy<-fy+th[2]*exp(-(th[3]+th[4])*xvals)
  ssq<-sum((yvals-fy)^2)
  return(ssq)
}

th.start<-c(3.0,2.0,0.5,0.2)
nlm(f=my.function.new,p=th.start,xvals=x,yvals=y)
```

Results

```
>nlm(f=my.function.new,p=c(3.0,2.0,0.5,0.2),xvals=x,yvals=y)

$minimum [1] 1.415276

$estimate [1] 2.9522862733 1.9203908648 0.5204927933 0.0002269101

$gradient [1] 7.890393e-07 -7.270481e-07 -9.703349e-08 7.016610e-08

$code [1] 2

$iterations [1] 31
```

This means that the line of best fit is obtained when

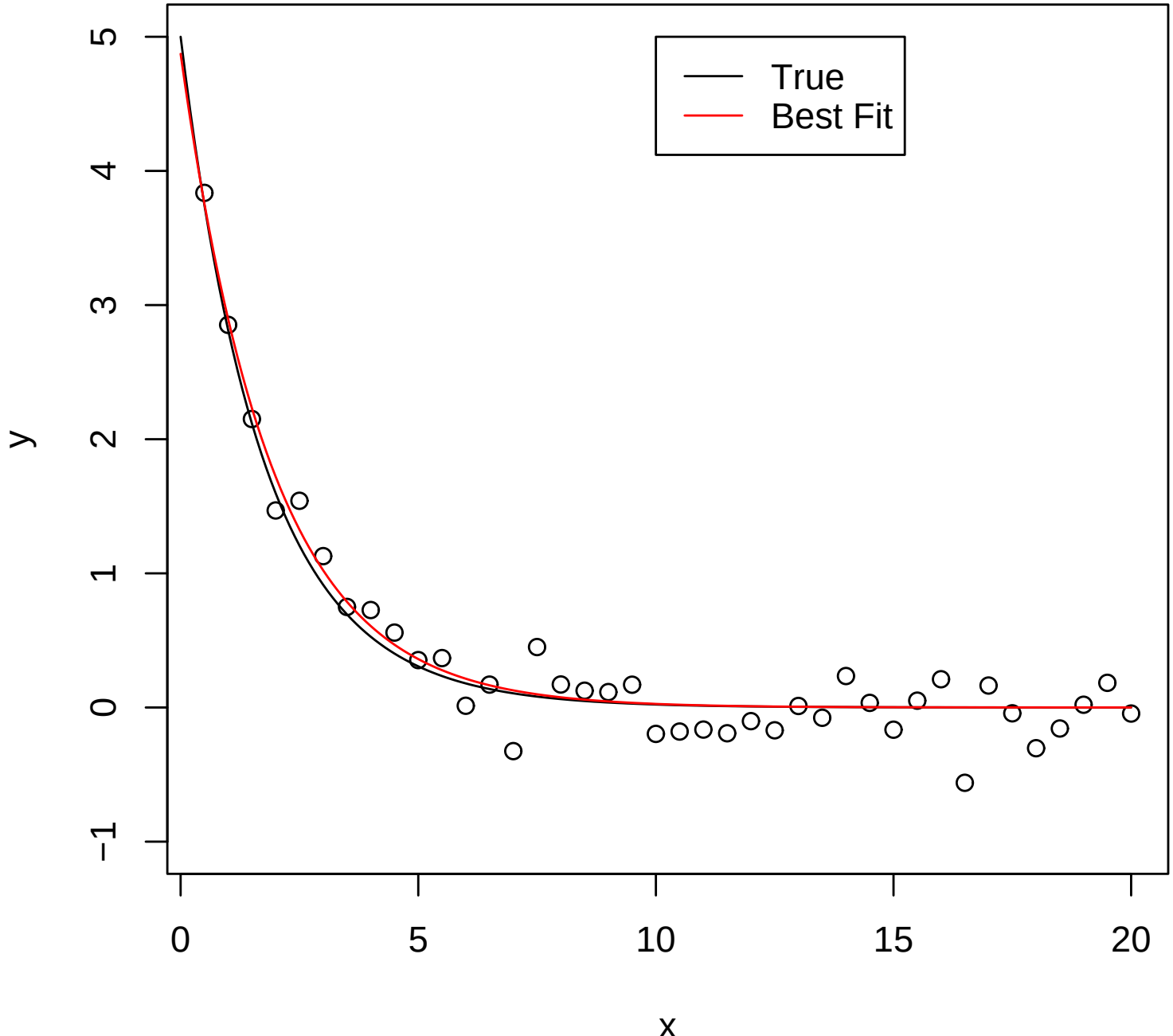
$$\lambda_1 = 2.952 \qquad \lambda_2 = 1.920 \qquad \alpha_1 = 0.520 \qquad \alpha_2 = 0.0002$$

Best Fit ?

```
my.fit<-nlm(f=my.function.new,p=c(3.0,2.0,0.5,0.2),xvals=x,yvals=y)
param.estimates<-my.fit$estimate
```

```
xv<-c(0:200)/10
true.y<-my.function.alt(xv,theta)
fitted.y<-my.function.alt(xv,param.estimates)
plot(x,y,ylim=range(-1,5))
lines(xv,true.y)
lines(xv,fitted.y,col="red")
legend(10,5,c("True","Best
Fit"),lty=c(1,1),col=c("black","red"))
```

Best Fit !



Extended Example

In this example, we will

- simulate a large data set
- automate its analysis
- process and plot the results

The key components will be the use of the function `apply` to a numerical matrix.

Simulated Microarray data

Microarrays are a high-throughput technology for the analysis of the function of genes.

Typically, thousands of genes are processed simultaneously.

Interest lies in distinguishing genes that are “differentially expressed" in two tissue types.

In this experiment we will simulate some appropriate data.

Simulation

```
#Number of genes
ngenes<-7500

#Number of samples
N0<-20
N1<-37

#Select the genes that are differentially expressed
Ndiff<-50
gene.list<-sample(c(1:ngenes),size=Ndiff,rep=F)

#The amount of up-regulation
Up.mean<-2.0
Y0<-matrix(rnorm(N0*ngenes),ncol=N0,nrow=ngenes)
Y1<-matrix(rnorm(N1*ngenes),ncol=N1,nrow=ngenes)
Y1[gene.list,]<-Y1[gene.list,]+Up.mean
Y<-cbind(Y0,Y1)
```

Method I

#One method of analysis

```
date()
test.results<-numeric(ngenes)
for(igene in 1:ngenes){
    y0<-Y0[igene,]
    y1<-Y1[igene,]
    test.igene<-t.test(y1,y0,var.equal=T)
    test.results[igene]<-test.igene$statistic
}
date()
hist(test.results)
```

Method II

Using `apply`

```
my.test<-function(x){
    n0<-x[1]
    n1<-x[2]
    y0<-x[3:(2+n0)]
    y1<-x[(2+n0+1):(2+n0+n1)]
    t.res<-t.test(y1,y0,var.equal=T)
    return(t.res$statistic)
}
tmp.Y<-cbind(rep(N0,ngenes),rep(N1,ngenes),Y)
date()
my.test.results<-apply(tmp.Y,1,my.test)
date()
```

R Programming : Control Structures

To release the power of the programming language, we need to learn about the language constructs the provide *control structures*, that provide the capacity for selection and iteration.

Think back to the `ChickWeight` example: had we required to consider all 4 diet groups, we would have had to write effectively the same piece of code 4 times. An alternative would be to write a new function that conducts the computation for selected data.

First we will look at control structures.

Selection

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

```
if (condition)
        true.branch
else
        false.branch
```

We have already seen a variety of logical comparisons that can serve as `condition`. If `condition` evaluates as `TRUE`, then `true.branch` is followed otherwise `false.branch` is followed. If condition evaluates as `NA`, an error occurs.

Selection

Example

```
if (x >3) {
    y <- 1
    z <- 2
} else {
    y <- 2
    z <- 1
}
```

Note the use of curly braces allow us to deliver compound (that is multi-line) statements. Also note the use of indenting to try to clarify structure.

Selection

The `else` part of an `if` statement is optional. As regular parts of the `R` language, `if` statements can occur within the branches of `if` statements – that is, they can be nested. For example

```
if (x > 2)

        if (y < 3)
                count <- count + 1
        else
         ...
```

Note `R` is quite fussy about placement of symbols in scripts. For a more elegant alternative to multiply nested `if` statements, use the `switch` function.

Selection

It is often useful to have compound conditions with an `if` statement. We can combine conditions with the logical operators && (for AND) and || (for OR). Note these are different to the single character vector operators. For example, in an optimisation problem we may have

```
if (iterations > max.it && abs(error) < tol)
        converged <- T
```

We will sometimes need to use brackets to clarify compound conditions. Note also order of evaluation for %% and ||.

Be careful with conditions. If the condition evaluates to a vector, the first element is used (and could be coerced to logical).

Selection

For selection operations on vectors, use the `ifelse` function. The general form of `ifelse` is

```
ifelse(test,true.value,false.value)
```

Here, all the arguments are vectors. `test` is a comparison operation applied to each element of a vector, `true.value` is returned in positions where the comparison is `TRUE`, and `false.value` is returned otherwise. For example

```
ifelse(1:10 < 5,0,1)
```

This should be efficient even for large vectors, and is to be preferred over explicit looping wherever possible. ifelse switch

Iteration

We can distinguish two types of iteration construct: count controlled loops, provided by the `for` statement, and variable length loops, provided by the `while` and `repeat` statements.

WARNING: bad use of loops is the most common source of inefficient `R` code. This is particularly true of nested loops. Always think hard about how use functions like `apply` rather than using a loop. Of course, sometimes it is unavoidable.

Iteration

The general format of the `for` statement is

`for (variable in sequence) statement` And note

that `statement` can be compound. `variable` is the counter variable, that will take consecutive values in `sequence`. As a simple example, of something NOT to do, consider the following

```
for(i in 1:length(x)) { y[i] <- sin(x[i]) }
```

In the exercises, you will see just how inefficient this is, compared to using a vectorised function. We can nest `for` loops, but do this with caution. Be careful not to change the vale of the the counter variable.

Iteration

The general format of the `while` statement is

```
while (condition) statement
```

Note that a while loop may never execute the statement. The statement is executed repeatedly until `condition` becomes false. In contrast, a `repeat` loop, with general format

```
repeat statement
```

will execute at least once, and continue until it is explicitly interrupted with a `break` statment. In fact, `break` will immediately exit from any loop structure. This can be useful for diagnostic purposes.

R Demo: Bracketing

The function

$$x^2 - 1$$

has a single zero in the interval $(0, 1)$.

A simple approach to finding zeros is *bracketing*, where we find an interval containing the zero, evaluate the function in the middle of the interval, and restrict attention to the half interval containing the zero (as is indicated by the sign of the function at the three points). This process is repeated until the width of the interval is smaller than a specified tolerance.

R Demo: Bracketing

```
# Start values
hi <- 1
lo <- 0
f.hi <- hi*hi-1/2
f.lo <- lo*lo-1/2
# set tolerance
tol <- 1e-9
found <- abs(hi-lo) < tol
# iteration counter

its <- 0
```

R Demo: Bracketing

```
# search
while (!  found){
      mid <- (hi+lo)/2
      f.mid <- mid*mid-1/2
      if (sign(f.mid) == sign(f.hi))
      {
            f.mid <- f.hi
            hi <- mid
             }
      else
            {
            f.mid <- f.lo
            lo <- mid
            }
      its <- its + 1
      if ((its %% 3) ==0)
       cat("Iteration ",its," hi= ",hi," lo= ",lo," mid=", mid,"\ n")
      found <- abs(hi-lo) < tol

}
```

R Demo: Bracketing

Of course, this is not the only way of implementing this procedure. We may have wanted to stop after a certain number of iterations. We could achieve this by modifying the convergence criterion

```
found <- (abs(hi-lo) < tol) && (its <= maxit)
```

Or, by using a `break` statement

```
if (its > 10) break()
```

Note that we could embody this code in a function, and that this function could be made to deal with arbitrary equations.

Be aware that `while` and `repeat` loops may never stop - the condition may not be satisfied.

Miscellany

In this section, we will look at a some miscellaneous features of R . Some or all of them might be regarded as useful for automation. In particular, we have often found it useful to automatically examine a collection of data files in a directory.

R Calling the Operating System

The R command for interfacing with the operating system is

```
shell(cmd, shell, flag="/c", intern=FALSE,
   wait=TRUE,translate=FALSE, mustWork=FALSE,
   ...)
```

Here cmd is a system command, enclosed in brackets. We have seen one example already

```
shell("dir")
```

By default, this will run under the DOS shell, although other shells can be specified. Note that the effect here is to display the directory listing – it is not stored in `.Last.value`.

Calling the Operating System

To manipulate what is returned (if anything) by the system call as an object, set the argument `intern=T`. In this case, the result of `dir` is stored as a vector of class character, where each line of the result of `dir` is stored as an element.
a typical line might be

```
[124] "09/27/2005 11:36 AM 731 sol8.R"
```

I have often found it useful to access the names of files of a specific type in a directory. This is best done with

```
files <- shell("dir /b *.type",intern=T)
```

Calling the Operating System

Now, suppose we want to read the contents of a number of sensibly formatted files, into different data frames, that share the name but not extension of the file. We can readily determine the filenames (let's assume they will make legal `R` object names). Two tasks remain (i) reading the file and (ii) creating the named object.

The first stage is often easy. Assume that `read.table` will do the job, as follows

```
read.table(files[1])
```

We now need to truncate the filename, and creating the object.

String Handling

R has a range of string handling facilities. For the immediate purpose of dividing a string like "hills.txt", we simply use the substr function

```
substring(files[1],1,nchar(files[1])-4)
```

to produce the character vector hills. Here, nchar counts the number of characters. This takes advantage of the fact we know the length of the extension. If more complicated handling were required, functions such as strsplit would be useful.

Now, we have computed the name we wish to use for the object. It is not obvious from what we have seen so far how to do this.

Assigning to Computed Names

We use the `assign` function as follows

```
assign(substring(files[1],1,
  nchar(files[1])-4),read.table(files[1]),1)
```

This complicated looking call associates the characters in the first argument as the name of the object in the second argument. The object is assigned to the database indicated by the third argument. In this case we write to the workspace - the database at position 1 (actually, things are more complicated that this, but it serves our purposes).

Handling Characters

Suppose the file names are such that they would not make legal object names - by beginning with a number, say. Here we can use the `paste` function, which is used to join characters together. For example

```
paste("a","b") paste("a","b",sep="")
```

produces

```
"a b"
"ab"
```

respectively. Now we simply proceed as before, but with the more complicated (that is modified with `paste`) first argument.

Resources

- WEB
  - `http://www.r-project.org/`
  - `http://cran.uk.r-project.org/`
- BOOKS
  - John M. Chambers. *Programming with Data*. Springer, New York, 1998.
  - William N. Venables and Brian D. Ripley. *Modern Applied Statistics with S*. Fourth Edition. Springer, 2002.
  - William N. Venables and Brian D. Ripley. *S Programming*. Springer, 2000.
  - Peter Dalgaard. *Introductory Statistics with R*. Springer, 2002.
  - John Maindonald and John Braun. *Data Analysis and Graphics Using R*.
  - Julian J. Faraway. *Linear Models with R*. Chapman & Hall/CRC, Boca Raton, FL, 2004
  - Paul Murrell. *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL, 2005.

available on web.