Part A Statistical programming HT13

Lecturer: Geoff Nicholls

University of Oxford

Notes and Problem sheets are available at

www.stats.ox.ac.uk\ ∼nicholls\PartAStatisticalProgramming

Course structure

- 6 2hr sessions in HT weeks 3-8
- Lectures 1 hour + practical 1 hour
- 2 problem sheets and 2 classes 4-5pm Weds week 8 HT13 and Weds week 1 TT13. Sign up today.
- Exam : 1 question on AS1 and 1 question in AS2.
- Equivalent to an 8hr lecture course.

What is Statistical Programming?

There are three elements: statistics, algorithms and carrying out statistical analysis on a computer.

In a typical study we begin by loading the data into the computer, and making an exploratory analysis of the data (EDA).

We look at the raw numbers, and visualize them by plotting graphs that reveal patterns in the data. We may have a model we wish to fit to the data, and a hypothesis we wish to test.

Statistics tells us what objects we need to calculate in order to make the fit and carry out the test.

An algorithm tells us how to calculate those objects. It is a sequence of operations that carry out a given task. An algorithm is a mathematical object, with properties we can prove. We give an (informally stated) algorithm to convert a decimal to binary later in this lecture.

Finally we go back to the computer and implement the algorithm (or use an implementation someone else has written). This is a computer programme and it should carry out the calculations dictated by the algorithm. Two correct implementations of the same algorithm can differ, for example in efficiency and in the range of cases they handle correctly.

What is R

R is an open-source package for Statistical Computing

It is freely available - **http://cran.r-project.org/**

It is now widely by Statisticians in Universities and Industry

- The BS1 course in Part B will use R extensively
- The Part A Statistics and Simulation courses use R to illustrate some elements of the course.

R is actively supported and updated and has many add-on packages that specialize in specific applications.

It can be extended to do new things - functions, packages

Getting started

Begin by selecting `Start > all programmes > R` The R Graphical User Interface `RGui` opens, with an internal window, the `R Console`, into which you type commands.

You will often want to keep a record of your work from an R session. You can do this by starting an R script: `File > New Script`. This opens a new file in an R editor window.

You can type R commands in the script, which can then be selected (using the mouse) and evaluated (using Ctrl-R) in the R Console.

You also make notes in the script and save the script for the future `File > Save` (you need to select the R editor window for the script you want to save using the mouse).

Arithmetic Operators

- Arithmetic operators: `+, -, /, *, ^` where `x^y` is x^y.
- `%%` for modulo reduction, `%/%` for integer division, `%*%` for matrix multiplication

Example

- Type `2 + 3 <return>`
- Type `6 * 9 + 3 / 5 <return>`
- Type `3^4 <return>`
- Type `5^(-5.6+3) <return>`
- Type `143 %% 6 <return>`

Mathematical functions

Many mathematical functions are available e.g.

```
exp(x), log(x), log10(x), sqrt(x)
sin(x), cos(x), tan(x)
asin(x), acos(x), atan(x)
sinh(x), cosh(x), tanh(x)
asinh(x), acosh(x), atanh(x)
abs(x), min(x), max(x)
```

- Type `exp(-4 * 4 / 2) / sqrt(2 * pi)<return>`

- Type `log(4)<return>`

Named storage Often we will want to do a calculation and save the result. Assignment can be done using the `=` or `<-`.

- Type `x1 = 2 + 3 <return>` (stores result in 'object' called `x1`)

- or type `x2 <- 9*exp(-3) <return>` (stores result in 'object' called `x2`)

- To see the results of these assignments type the names of the objects into the terminal.

- We can then use these objects in new calculations.
  `x1 + x2<return>`

Vectors and sequences Vectors or sequences of numbers can also be created, stored and manipulated

- They can be created manually
  `x1 = c(0.1, 0.2, 0.3, 0.4, 0.5)` **`<return>`**

  Example

if `x = (1, 2, 3, 4)` and `y = (5, 6)`, then

`x + 3 = (4, 5, 6, 7)`

`x + y = (6, 8, 8, 10)`

Logical Operators

- `==` (equal), `!=` (not equal), `>`, `<`, `>=`, `<=`

  Example

  `3 < 4 <return>`

- `!` (not), `|` (or), `||` (or), `&` (and) `&&` (and)

- Note difference between `|,&` and `||,&&`: former work on vectors, latter (a) only consider first element and (b) stop as soon as the outcome is known.

Example (a)

if `x = (TRUE, FALSE, TRUE)`
and `y = (FALSE, TRUE, TRUE)`
`x | y = (TRUE, TRUE, TRUE)`, `x || y = (TRUE)`, then
`x & y = (FALSE, FALSE, TRUE)`, `x && y = (FALSE)`

Example (b)

If `b=2` and we execute `(a=1)<0 & (b=-1)<0` then `b=-1` but if `b=2` and we execute `(a=1)<0 && (b=-1)<0` then `b=2` (still). In the first case we have a sideffect (`b` changes value). It is usually a bad idea to allow them in your code so we would not typically include an assignment `b=-1` in a test `(b=-1)<0` irrespective of whether we used `&` or `&&`. The `&&` form saves time evaluating later tests when the result is already known.

Error messages

- The R commands you enter will sometimes contain errors.
- R will either report an Error message or prompt you to complete the command.

Example
Incomplete brackets i.e. `(4+8<return>`
prints a + which means you need to enter the remaining brackets.

Example
Too many brackets i.e. `(4+8))<return>`
results in an error message.

Example
Using the wrong type of brackets i.e. `exp{4}<return>`
results in an error message.

Computer representation of numbers

- On a computer numbers are stored in binary rather than decimal.
- Consider the number 1197.625.
  In decimal we write

$$\begin{array}{|c|c|c|c||c|c|c|} \hline 10^3 & 10^2 & 10^1 & 10^0 & 10^{-1} & 10^{-2} & 10^{-3} \\ \hline 1 & 1 & 9 & 7 & 6 & 2 & 5 \\ \hline \end{array}$$

  In binary, the number is

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c||c|c|c|} \hline 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array}$$

To convert from a base-10 integer to base-2 (binary), the number is divided by two, and the remainder is the least-significant bit. The (integer) result is again divided by two, its remainder is the next most significant bit. This process repeats until the result of further division becomes zero.

$$1197 = 2 * 598 + 1 \rightarrow 1$$
$$598 = 2 * 299 + 0 \rightarrow 0$$
$$299 = 2 * 149 + 1 \rightarrow 1$$
$$149 = 2 * 74 + 1 \rightarrow 1$$
$$74 = 2 * 37 + 0 \rightarrow 0$$
$$37 = 2 * 18 + 1 \rightarrow 1$$
$$18 = 2 * 9 + 0 \rightarrow 0$$
$$9 = 2 * 4 + 1 \rightarrow 1$$
$$4 = 2 * 2 + 0 \rightarrow 0$$
$$2 = 2 * 1 + 0 \rightarrow 0$$
$$1 = 2 * 0 + 1 \rightarrow 1 \quad \rightarrow 10010101101$$

To convert from a base-10 fraction to base-2 (binary), the number is multiplied by two, if the result is > 1 the most-significant bit is 1 otherwise 0. The (fractional) remainder is again multiplied by two and compared to 1. This process repeats until the remainder is zero.

$$0.625 * 2 = 1.25 \rightarrow 1$$
$$0.25 * 2 = 0.5 \rightarrow 0$$
$$0.5 * 2 = 1 \rightarrow 1 \quad \rightarrow .101$$

So, $1197.625_{10} = 10010101101.101_2$

- In decimals, some fractions are recurring. The same can be true in binary. Consider 0.8

$$0.8 * 2 = 1.6 \rightarrow 1$$
$$0.6 * 2 = 1.2 \rightarrow 1$$
$$0.2 * 2 = 0.4 \rightarrow 0$$
$$0.4 * 2 = 0.8 \rightarrow 0$$

  etc

  So, $0.8_{10} = 0.\overline{1100}_2$

- Fractions in binary only terminate if the denominator has 2 as the only prime factor.

- Modern computers use 64 bits to represent numbers so some numbers (like 0.8) must be approximated.

- Care is needed when testing whether 2 numbers are the same:

  - `x <- seq(0, 0.5, 0.1)` ##generate a sequence from 0 to 0.5 in steps of 0.1
  - type `x` ##Look at x
  - is `x equal to (0, 0.1, 0.2, 0.3, 0.4, 0.5)`?
  - To find out, type
    `x==c(0, 0.1, 0.2, 0.3, 0.4, 0.5)`

    `[1] TRUE TRUE TRUE FALSE TRUE TRUE`

Rounding problems Tiny inaccuracies can accumulate:

- the sample variance of a vector `x` is often calculated as $var(x) = (\sum x^2 - n\bar{x}^2)/(n-1)$
  or, in R
  `(sum(x^2) - n * mean(x)^2) / (n - 1)`
- Try it with `x <- seq(1:100)`
- Now with `x <- seq(1:100) + 10000000000`
- compare with `var(x)`
- Can you see why there was a problem?

Moral "Worry, but, if you use R, don't worry too much, because R has worried for you."

Try it yourself

Begin by selecting `Start` > `all programmes` > `R`

The R Graphical User Interface `RGui` opens, with an internal window, the `R Console`, into which you type commands.

You can open the lecture script: `File > Open Script`. The script you want is `L1.R` on the `H:` drive.

After 5 minutes we will continue with the lecture.

Basic Types of Variables Variables are the equivalent of memories in your calculator. But you can have unlimited (almost!) quantities of them and they have names of your choosing. And different types. The basic types are

- integer
- double
- character
- logical: these take one of the two values `TRUE` or `FALSE` (or `NA`, see later)
- factor or categorical

More Specialised Classes

As well as the basic types of variables, R recognises many more complicated objects such as

- vectors, matrices, arrays: groups of objects all of the same type. The practical will show you how to use these.

- lists of other objects which may be of different types

- Specialised objects such as Linear Model fits

Special values of objects

There are some types of data which need to be treated specially in calculations:

- `NA` The value `NA` is given to any data which R knows to be missing. This is not a character string, so a character string with value '`NA`' will be treated differently from one with the value `NA`.

- `Inf` The result of e.g. dividing any non-zero number by zero

- `NaN` The result of e.g. attempting to find the logarithm of a negative number.

Factors: Factors are variables which can only take one of a finite set of discrete values. They naturally occur as vectors, and can be

- numeric e.g. drug doses with values 1mg, 2mg, 5 mg
- or character e.g. voting intention with values Liberal Democrat, Conservative, Labour, Other

Although factors are stored as numbers, along with the label corresponding to each number, they cannot be treated as numeric. Would it make sense to ask R to calculate `mean(voting intention)`?

A more useful function for factors is `table` which will count how many of each value occur in the vector.

More about Factors

- Ordered Factors

  Some factor variables have a natural ordering. Drug doses do, but voting intentions usually do not. R will treat the two types differently. It is important not to allow R to treat non-ordered factors as ordered ones, since the results could be meaningless.

- Creating factors

  Use `cut` to create factor variables from continuous ones:

Example

```
age <- runif(100) * 50
table(cut(age, c(0, 10, 20, 30, 40, 50)))

 (0, 10] (10, 20] (20, 30] (30, 40] (40, 50]
      17       19       19       24       21}
```

Data frames

- For storing data which is a collection of observations (rows) of a set of variables (columns). E.g. book titles and prices.

- Similar to a matrix but variables in different columns can have different types.

- Always the same number of entries in each row, although some may be missing (`NA`).

- Can be formed by reading in data e.g. from a spreadsheet, or constructed using the function `data.frame`.

Chick weights data frame

	weight	feed
1	179	horsebean
2	160	horsebean
3	136	horsebean
4	227	horsebean
.		
.		
.		

Lists A data frame is a kind of list, which is a vector of objects of possibly different types. For example, an employee record might be created by

```
Empl <- list(employee = "Anna", spouse = "Fred", children
= 3, child.ages = c(4, 7, 9))
```

Components are always numbered and may be referred to by number. e.g. `Empl[[2]]`. If they are named, can also be referred to by name using the $ operator eg. `Empl$spouse`

Note that `Empl[4]` is a list of length 1, while `Empl[[4]]` is a numeric vector of length 3.

Keeping track of objects

- Once you have created some objects, how do you remind yourself what you called them?
- Use a function such as `ls()` or `str()`.
- `ls()` lists the names of all the objects in your workspace.
- `str(Empl)` gives a little information about the object `Empl`.

```
str(Empl)
 List of 4
 $ employee    :  chr "Anna"
 $ spouse      :  chr "Fred"
 $ children    :  num 3
 $ child.ages  :  num [1:3] 4 7 9
```

More functions `paste`

- `paste` adds together character vectors: `paste(c(1, 2), c('x', 'y', 'z'))` is a character vector of length 3
  `[1] "1 x" "2 y" "1 z"`
  Notice the recycling of the first argument.

- The pieces of the result can be joined together using the argument `collapse`:
  `paste(c(1, 2), c('x', 'y', 'z'), collapse=' ')` is a character vector of length 1,
  `"1 x 2 y 1 z"`

Yet more functions `sort,table`

- `sort(x)` sorts the vector `x` into increasing order
- `sort(x, decreasing=TRUE)` sorts the vector `x` into decreasing order
- `table(x)` creates a table showing the count of elements equal to each value. Most useful where `x` is a vector of factors or integers
- e.g. `table(rpois(20, 5))` gives

```
2 3 4 5 6 7 8
3 2 2 6 3 3 1
```

Installing R on your own machine

- Visit CRAN at **http://cran.r-project.org/**

- Find the appropriate binary
  - Windows:
    http://cran.r-project.org/bin/windows/base/release.htm
  - Mac: select as appropriate
  - Linux: select as appropriate.

- After installation, create a shortcut which starts in your folder (using right-click/properties)

- Update packages via the packages menu or using `update.packages()`

Getting Help

- `help(command)` or `?command` if you know the command name
- Type `??variance` or `help.search("variance")` for a subject
- `?‘&&‘` for operators or words such as `if`
- `help.start()` for R online documentation
- Help menu in Windows gives still more options

Books

In addition to the extensive documentation and help system that is included in R there are three main books that we recommend.

'A First Course in Statistical Programming with R' by W. John Braun and Duncan Murdoch, ISBN 0-521-69424-8

'Introductory Statistics with R' by Peter Dalgaard, ISBN 0-387-95475-9

- a very good introduction to R that includes many biostatistical examples and covers most of the basic statistics covered in this course.

'Modern Applied Statistics with S' by Bill Venables and Brian Ripley, ISBN 0- 387-95457-0

- a comprehensive text that details the S-PLUS and R implementation of many statistical methods using real datasets.

Practical

- Practical is a mixture of getting you to enter commands and solving problems
- Always use the same computer as files will be stored on it and you may need them from week to week. So note down the number of the computer you have used.
- Use a script and save the script to the H: drive. Save it with a file name that includes your name.
- May want to bring a memory stick to make a copy of your script or email the script to yourself.
- Web browsing, facebook etc are banned from practical sessions.
- To quit R type `q()`. Then select No when asked if you want to save the session.

Part A Statistical programming HT13

Geoff Nicholls

Lecture 2: Handling Data

Overview

1. Reading data.
2. Attaching data frames to the search path.
3. Further operations on data: `apply()` and subsetting.
4. Plotting data.
   (a) Histograms and overlays.
   (b) Boxplots and the R Formula notation.
   (c) Barplots
   (d) Point plots.

Getting Data into R

Data come to us in many different formats. R has functions to read them.

`hellung.txt` is text file of data from an experiment on the growth of Tetrahymena cells.

The cell concentration (**`conc`**) was set at the beginning of the experiment and the average cell diameter (**`diameter`**) was measured for two groups of cell cultures where glucose was either added (**`glucose=1`**) or not added (**`glucose=2`**) to the growth medium. A theoretical model predicts diameter proportional to the log of concentration. The effect of glucose on the cell diameter is of interest.

The data are a table, with one row for each observation and a column for each variable. The first row gives variable names.

We can use the `read.table()` function to read it. This table has a header line, so we tell R to expect a header.

```
hd=read.table('hellung.txt',header=TRUE)
```

creates a `data.frame` with cases corresponding to rows and variables to columns in the file.

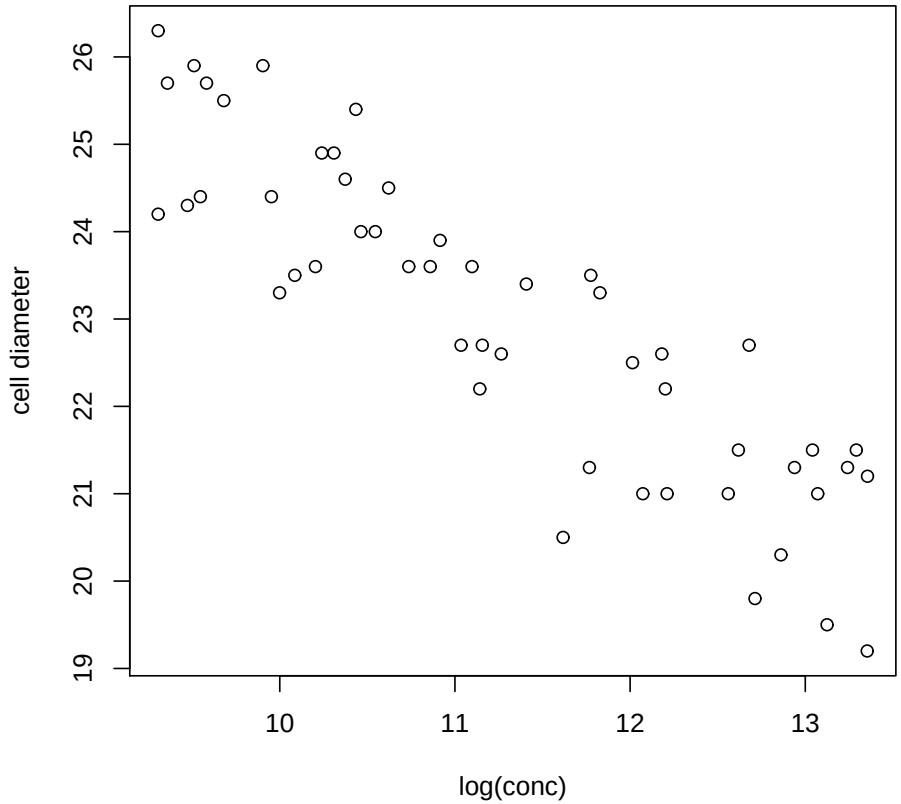
How many observations are there, and what are the variable names? `dim(hd), names(hd)`

Show me the first few rows, and basic properties of the variables!
`head(hd),str(hd), summary(hd)`

Show me the data!

```
#basic plotting
x=log(hd$conc); y=hd$diameter;
plot(x,y,xlab='log(conc)',ylab='cell diameter')
```

There are R-packages (of which more anon) for reading specialized formats such as xls and image formats.



The function and variable search path

When you type a function or variable name, R searches for the object in a list called the search path.

`search()` shows you the databases in which R searches. The first, ".GlobalEnv", is your R console workspace. The last, "package:base" has all the basic R functions.

If you `attach()` a data frame to the search path, the variables inside the data frame can be found.

For example, you must type `hd$glucose` to get the `glucose` numbers. Following `attach(hd)`, you can access it with just `glucose`. Now `search()` shows `hd` in the path.

```
plot(log(conc),diameter) #an error
attach(hd)
plot(log(conc),diameter)
```

You can `detach('hd')` an object to remove it from the path. You might do this if you wanted avoid a conflict with something else with the same name. In many commands you can specify where to look for the variables, using the `data=` option. This is often easier read, and shows explicitly which data are being used.

Data summaries Here are some useful functions.

```
mean(), median()
sd(), var(), cov(),cor()
range(), quantile(), summary()
min(), max(), pmin(), pmax()
which.max(), which.min()
sum(),cumsum(),cumprod()
```

Many of these functions have an argument na.rm which needs to be set to TRUE in order to remove NAs from the data.

There are no NA's in these data so here is a simple example: If `eg=c(1,2,NA,3,4)` then `mean(eg)` gives an error but `mean(eg,na.rm=T)` gives 2.5, the mean of $(1, 2, 3, 4)$. R is forcing you to be explicit about the NA-handling.

Applying functions over data, and subsetting

The apply() command is also very useful for applying functions to all rows or all columns of an array or data frame. For example, `apply(hd,2,max)` finds the maximum of each variable.

Sometimes we want to work on subsets of a data frame. Boolean variables are converted to array indices when they appear as the index to a vector.

For example, to compare the mean cell diameters for cells with and without glucose we might calculate `mean(diameter[glucose==1])` and `mean(diameter[glucose==2])`

We can create a new data frame from the subset of observations of interest: for example, `hd.g1=hd[glucose==1,]` pulls out the rows of `hd` that satisfy our condition, and takes all columns.

Plotting Data: Histograms

Graphical data displays provide insight to data. Think carefully about how to display data to highlight features of interest. We begin with simple exploratory plots.

`hist(diameter,freq=FALSE)` gives a histogram of the numbers in the variable `diameter`. The y-axis shows counts.

`hist(diameter,freq=TRUE)` gives a histogram with area scaled to one, like a probability density.

`par(mfrow = c(1,2)); hist(diameter); hist(conc)` puts two entire plots in one window, in 1 row and 2 columns.

You may need to adjust the number of histogram bins. The default 'Sturges' rule gives too few on large data sets. Try the 'Scott' rule, fix the number yourself, or give a vector of bin locations.

```
n = 100000; x = rnorm(n)
par(mfrow = c(2,2));
hist(x, main = "Sturges")
hist(x, breaks = "Scott", main = "Scott")
hist(x, breaks = sqrt(n),main='sqrt(n)')
hist(x, breaks = seq(from=-5,to=5,length.out=200),
     main='my breaks')
```

Overlays

The `lines()`, `points()` and `text()` commands can be used to overlay lines, points and text on an existing plot (already created with `plot()` or `hist()`).

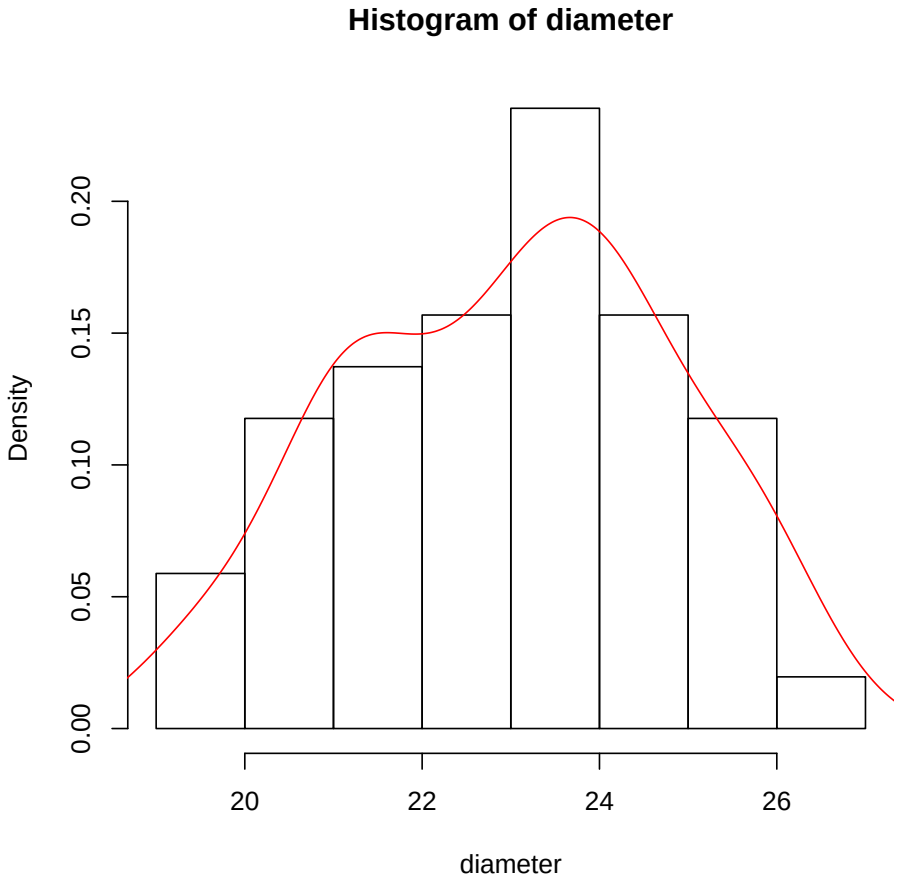
For example, we can overlay a density estimate (computed using the `density()` function) on a histogram.

```
hist(diameter,freq=FALSE);
lines(density(diameter),col=2)
```

Here `lines` will recognize `density()` output and done something sensible.

```
hist(diameter,freq=FALSE);
d=density(diameter)
lines(d$x,d$y,col=2)
```

has the same effect.



Boxplots and the R formula notation

These are useful for plotting a continuous variable against one or more discrete variables.

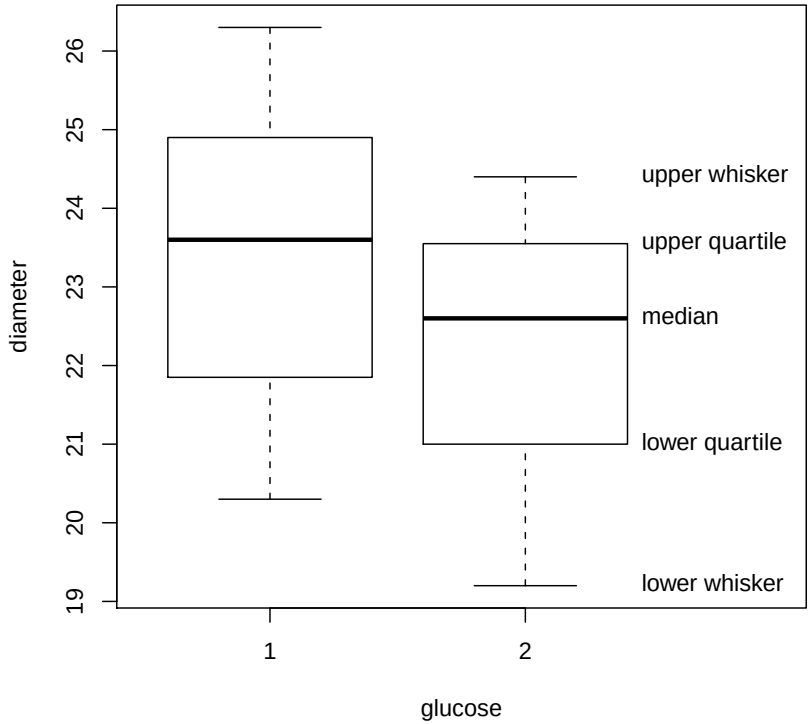
The R formula notation is useful here. The formula

```
variable1~variable2
```

means variable1 is a response variable and depends on variable2.

For example, we expect the distribution of cell diameters to be different depending on whether or not glucose was used.

```
boxplot(diameter~glucose,xlab='glucose',ylab='diameter')
```



Points beyond the outer whiskers are possible outliers.

Barplots are useful for showing the distributions of categorical variables.

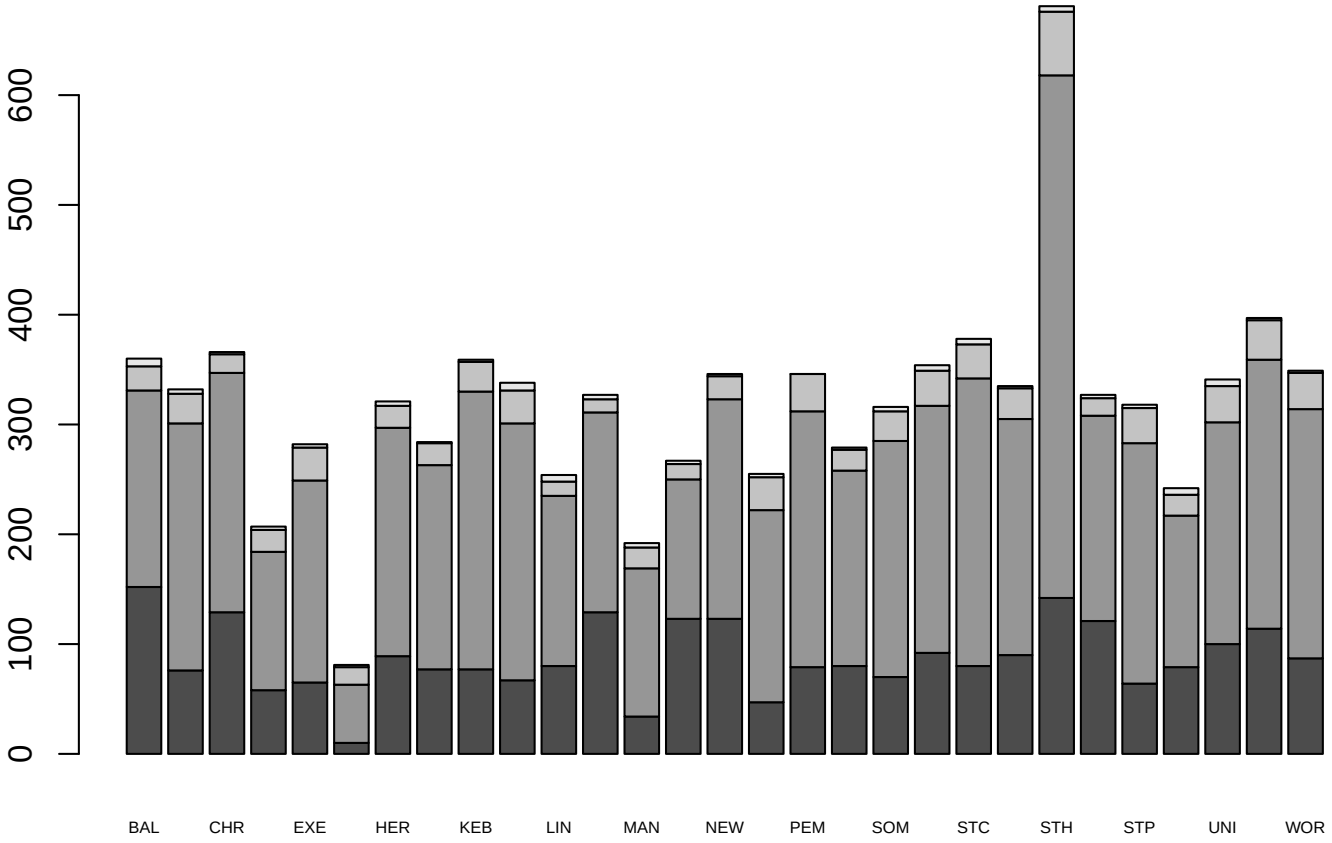
For example **Oxford_data_anon.csv** is a csv file giving (anonymous) FHS results from 2006-2008.

```
ox=read.csv("Oxford_data_anon.csv", header = TRUE)
```

For each (unnamed) student the file gives Year, College, Subject and Result.

`(ox.tab=table(ox$Result,ox$College))` gives a table of counts by result and college. The first row `ox.tab[1,]` is the count of firsts by college, so barplot(ox.tab[1,],cex.names=0.5) simply plots raw counts.

`barplot(ox.tab,cex.names=0.5)` plots the counts in each column as a bar, made up of the counts in each class.



Scatter plots

We started with an elementary plot. By varying color (`col`) and point character `pch` from point to point, you can highlight different subsets of data within the plot. For example

```
plot(log(conc), diameter, col=glucose, pch=glucose)
```

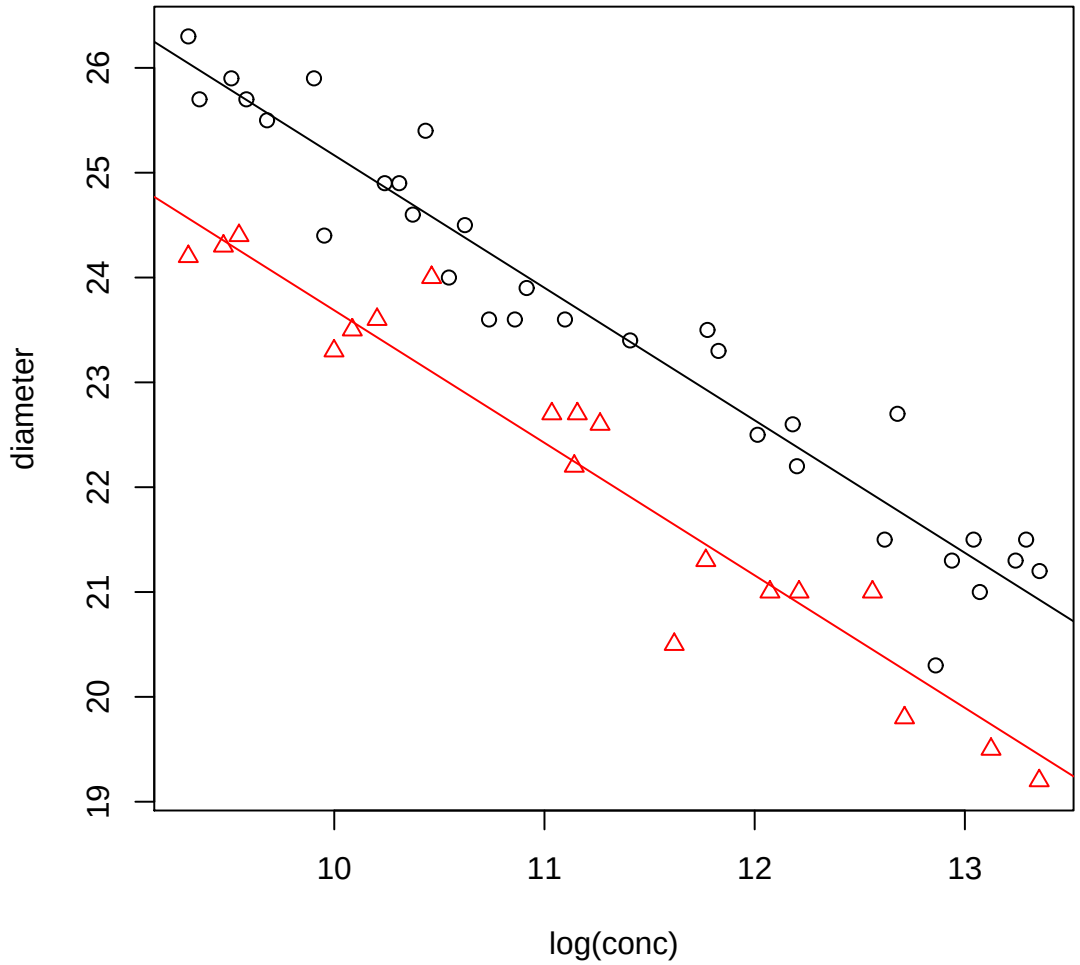
The color and shape of each point is decided by its glucose value.

We can overlay a line using the `abline(intercept,slope)` command, which simply takes the intercept and slope.

```
abline(37.806,1.264)
abline(36.327,1.264,col=2)
```

We can plot or overlay a function using `curve()`

```
curve(37.806-1.264*x, from=8 , to=15, add=T)
curve(36.327-1.264*x, from=8 , to=15, add=T, col=2)
```



Part A Statistical programming HT13

Geoff Nicholls

Lecture 3: Functions and flow control

Overview for lecture 3

1. R functions, function arguments and scoping
2. Flow control with `for()`, `if()-else` and `while()`.
3. Examples (Sieve of Eratosthenes and Newton's method).

Functions: what and why

What: functions take input, perform operations on the input and return output. Functions help us break down a big problem into modules. We can get each function working accurately and build up a working system. Functions help us avoid repeating the same sequence of commands in lots of different places.

The general definition of a function is :

```
MyFunction = function(arguments) { statements }
```

The arguments can defined with or without defaults. When the function is called the arguments are passed to the statements.

A function computing the p-norm $\left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$ of $x \in R^n$:

```
g<-function(x,p=2) {
    #my pnorm function
    y=abs(x)
    z=sum(y^p)
    return(z^(1/p))
}
```

`g(c(3,4))` returns 5 (no `p` set so default `p=2` used).
`g(c(3,4),1)` or `g(c(3,4),p=1)` returns 7.

A function returns the value of the last statement evaluated so the `return()` command is often omitted.

Variable scope

The scope of a variable tells us where the variable is visible. Generally speaking you want a variable to be visible only where it is needed, to avoid clashes with other variables that have the same name.

In R, the variables in a function are local to the function. The environment calling the function (for example, the R-console workspace, or another function) cant see the variables inside the function (only the return values). On the other hand the called function can see the variables in the environment that called it (the parent environment).

For example, if we define a function `f()` in the R-console

```
f<-function(x) {
   a=b*x^2
   return(a)
}
```

set `a=2`, `b=1` and call `f(5)` the function returns 25.

The function found `b` in the workspace that called it.

In the console `a` is still `2` because the function created its own local variable `a`.

`for` loops

A `for`-loop repeats some commands a fixed number of times.

Example: Plan and write a function computing the first n terms in the Fibonacci sequence `1 1 2 3 5 8 13 21 ...`.

We will build up the sequence in a vector $x = (x_1, x_2, ..., x_n)$. In order to calculate the ith entry we will add x_{i-1} and x_{i-2}, the previous two elements of the vector, and write the result into the ith entry in x.

```
fibonacci = function(n) {
    #evalute first n terms in Fibonacci sequence
     x = numeric(n)
     x[1:2] = 1
     for(i in 3:n) {x[i] = x[i-2] + x[i-1]}
     return(x)
}
```

The general form is

```
for (variable in sequence) { statements }
```

`if` and `if else` statements

The `if()` statement controls which statements are executed.

```
if (x > 2) {
    y = 2 * x
} else {
    y = 3 * x
}
```

sets `y=3` when `x = 1` and sets `y=8` when `x=4`.

Can have a simple `if()` without the `else`. The following gives a warning when `x=1` and `y=3` (so `z=NaN`)

```
z=(x-1)/(y-3)
if (is.nan(z)) warning('z is not a number')
```

The `while()` loop

Repeat a set of statements until a condition is satisfied.

Write an R-function to simulate a standard normal random variable conditioned to be greater than a for given real a.

```
Z<-function(a) {
   z=rnorm(1) #rnorm(1) simulates 1 N(0,1) rv
   while (z<a) {
      z=rnorm(1)
   }
   z
}
```

The code repeats the simulation until the condition is satisfied. Now `Z(2)` simulates $Z|Z > 2$ for $Z \sim N(0, 1)$.

The Newton method for root finding

Suppose we wish to find a root of an algebraic equation

$$f(x) = 0.$$

If $f(x)$ has a derivative $f'(x)$, then the following iteration will in general converge to a root if started close enough to the root.

$$\begin{aligned} x_0 &= \text{initial guess} \\ x_n &= x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \end{aligned}$$

The algorithm runs until $|f(x_n)| < \epsilon$.
The idea is based on the Taylor approximation

$$f(x_n) \approx f(x_{n-1}) + (x_n - x_{n-1}) f'(x_{n-1})$$

NOTE : this method may fail to converge.

Newton-Raphson Example

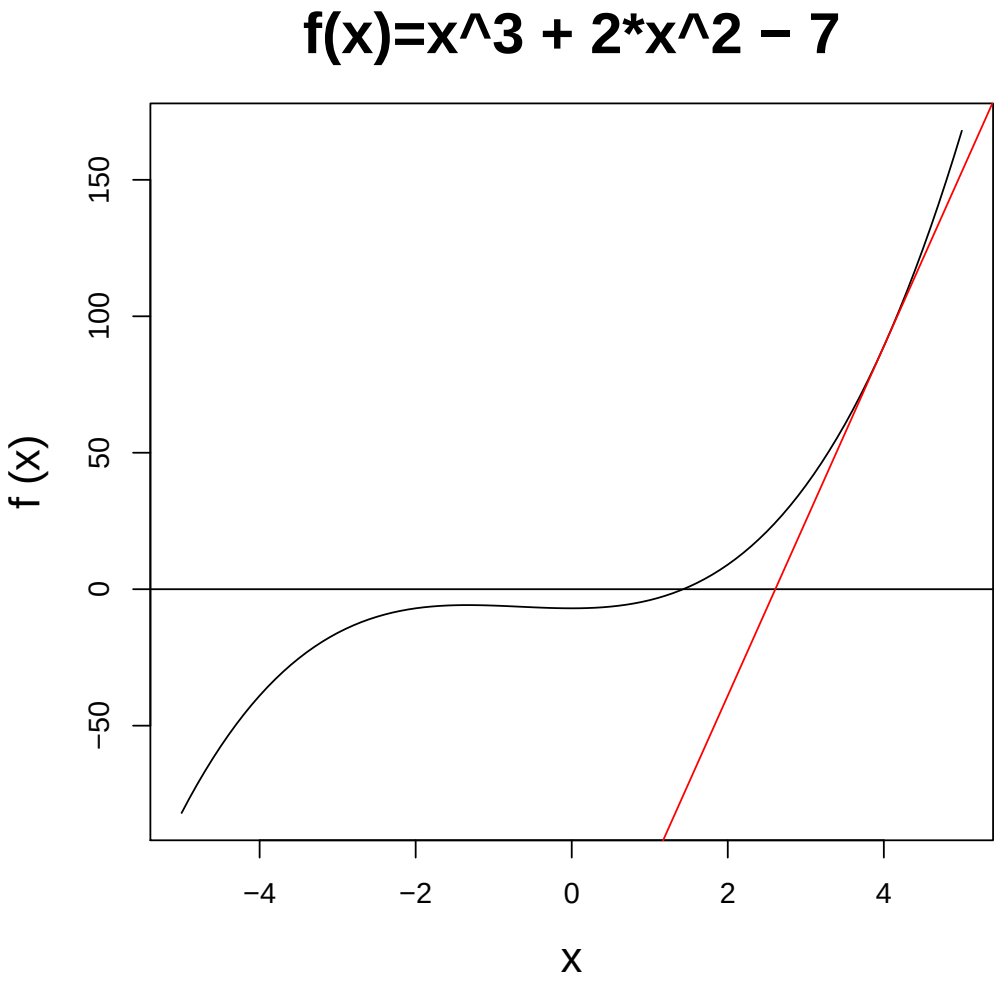
Suppose $f(x) = x^3 + 2x^2 - 7$.

```
f <- function(x) {x^3 + 2*x^2 - 7}
f.prime <- function(x) {3*x^2 + 4*x}

nr <- function(x, tol = 0.001) {
   #Newton-Raphson iteration for f
   while(abs(f(x)) > tol) {
      x = x - (f(x) / f.prime(x))
   }
   return(x)
}
```

The return value for `nr(4)` is approx. 1.428820. Compare 1.428817702 (Maple, numerical evaluation of exact solution to cubic).

Example (first iteration shown starting at $x_0 = 4$).



Part A Statistical programming HT13

Geoff Nicholls

Lecture 4: Recursion, Debugging, Efficiency
(with the sorting problem used as a running example).

Overview for lecture 4

1. Recursive evaluation
2. Sorting Algorithms
3. Runtime analysis
4. Debugging

Recursion

Recursive programmes call themselves.

Example: Plan and write a recursive function for $f(x) = x!$.

$$f(1) = 1, \qquad f(x) = xf(x-1) \qquad \text{for } x > 1.$$

Our factorial function returns $x! = 1$ on input $x = 1$ and otherwise calls itself to evaluate $(x-1)!$ and multiplies this by x.

```
factorial<-function(x) {
   if (x==1) return(1)
   if (x>1) return(x*factorial(x-1))
   stop('x must be a positive integer')
}
```

Each function in the nested sequence of calls to `factorial()` has its own variable environment with its own distinct version of the local variable `x`.

Recursive algorithms are often shorter and clearer than the corresponding implementation via `for` or `while`. However, they may be demanding of memory, if each level of recursion makes its own copy of local variables.

Sorting algorithms

Let $f(x)$ be a function which sorts the numeric input vector $x = (x_1, x_2, ..., x_n)$ into numerically increasing order $f(x) = (x_{(1)}, x_{(2)}, ..., x_{(n)})$. Several algorithms do this, with varying efficiency.

Simple sort: find the smallest element $x_{(1)}$. Suppose it is the kth element. Remove the kth element from the list, so $y = (x_1, ..., x_{k-1}, x_{k+1}, ..., x_n)$. Return the vector $(x_{(1)}, f(y))$.

Bubble sort: sweep through the vector, swapping x_i and x_{i+1} if $x_i > x_{i+1}$. Repeat this till the vector is in order. After i sweeps the last i elements $x_{(n-i)}, ..., x_{(n)}$ must be in their correct places so the algorithm terminates after n sweeps at most.

Runtime analysis

We measure the runtime in units of operations. This might be the number of additions, subtractions, divisions and multiplications. For a sorting algorithm we can count the number of comparisons.

We typically give the asymptotic run time - as a function of the input size, for large values of the input. We give the order of the function - quadratic, cubic etc. More efficient algorithms have (asymptotically at least) smaller run times.

We can give the worst case (for any input) or the average case (usually more interesting but harder to calculate).

Here is an algorithm to find the smallest entry of $n > 1$ numbers.

```
my.min<-function(x) {
   a=x[1]
   for (k in 2:length(x)) {
      if (x[k]<a) a<-x[k]
   }
   a
}
```

Let $g(x)$ be the number of comparisons. Clearly $g(x) = n - 1$ independent of x, so the runtime is $O(n)$.

Look at Simple sort. It repeatedly finds the smallest entry in a shrinking vector, of length n, $n-1$,...,1. Its runtime will be $n - 1 + (n - 2) + ... + 1$ comparisons, irrespective of the input, so it is an $O(n^2)$ sorting algorithm.

Merge sort

Split x into two halves $y = (x_1, ..., x_{\lfloor n/2 \rfloor})$ and $z = (x_{\lfloor n/2 \rfloor + 1}, ..., x_n)$.
Sort the two halves using Merge sort so $y' = f(y)$ and $z' = f(z)$.
Interleave the two sorted vectors y' and z' to form their sorted union.

```
mergesort = function(x) {
    # Terminate recursion - the trivial vector is pre-sorted
    # split vector into 2 halves
    # sort two halves by calling function on each half
    # merge the two sorted halves
}
```

Merge sort

split x into two halves $y = (x_1, ..., x_{\lfloor n/2 \rfloor})$ and $z = (x_{\lfloor n/2 \rfloor + 1}, ..., n)$.
Sort the two halves using Merge sort so $y' = f(y)$ and $z' = f(z)$.
Interleave the two sorted vectors to form their sorted union.

```
mergesort = function(x) {
    n = length(x)
    if(n < 2) { # check that vector doesnt need sorting
        result = x
    } else {
        # split vector into 2 halves
        # sort two halves by calling function on each half
        # merge the two sorted halves
    }
    return(result)
}
```

```
mergesort = function(x) {
    n = length(x)
    if(n < 2) { # check that vector doesnt need sorting
        result = x
    } else {
        # split vector into 2 halves
        y = x[1:(n%/%2)]
        z = x[(n%/%2 + 1):n]

        # sort two halves by calling function on each half
        y = mergesort(y)
        z = mergesort(z)

        result = my.merge(y,z)
    }
    return(result)
}
```

Merge sort: auxiliary function merging sorted vectors

Build up the sorted merged vector **`result`** one element at a time.

1) Take the smaller of `y[1]` and `z[1]`,

2) remove it from its vector,

3) tack it on the end of **`result`**.

4) Repeat steps 1-3 till one of the vectors `y` or `z` is emptied. This will be our **`while()`** condition. When this happens...

5) The remaining numbers in the surviving vector are sorted and all larger than the numbers in **`result`**, so tack them on the end.

```
my.merge<-function(y,z) {
   result = c()
   while(length(y)>0 && length(z)>0) {
      if(y[1] < z[1]) {
         result = c(result, y[1])
         y = y[-1]
      } else {
         result = c(result, z[1])
         z = z[-1]
      }
   }
   #exactly one of y and z is empty
   c(result,y,z)
}
```

Look at Merge sort. Suppose $n = 2^k$ for simplicity. Let g_k be the number of comparisons to sort this vector. Merge sort splits the vector into two vectors of length 2^{k-1}. These two sub-vectors have to be sorted, which is $2g_{k-1}$ comparisons. The number of comparisons to merge the sorted sub-vectors is 2^{k-1} so

$$g_k = 2g_{k-1} + 2^{k-1}$$

and $g_1 = 0$. The homogeneous solutions are $g_k = A2^k$ with particular solution $k2^{k-1}$. Applying the initial condition gives $g_k = (k+1)2^{k-1}$ or $g(x) = \log_2(2n)(n/2)$. We conclude that Merge sort needs $O(n \log_2(n))$ comparisons, irrespective of the input.

Debugging

Try to write your code so it is easy to check and maintain. We mentioned modularity (breaking code elements up into functions, as I did above with `my.merge()`) and information hiding (dont let a function have information it doesnt need). Careful planning and commenting help. Give variables meaningful names.

Your have just implemented an algorithm in R. It ran without reporting errors on the first input you tried. What do you do? Answer: assume it contains errors. Test it thoroughly using input for which you know the correct output.

You are implementing an algorithm in R. It crashes*. What do you do?

*terminates with an error message

When an error occurs R saves the list of active functions. traceback() prints that list. See example in `L4.R`.

You cant see the values of variables local to a function you called. You could use `print()` or `cat(sprintf())` commands to see what values variables take inside the function.

`debug()` lets you see what's going on inside a function interactively. You can examine (or change!) the value of variables, or execute any other R command, inside the function. You can also execute a debugger command

- n - next : execute the next line of code

- c - continue : let the function continue running

- Q - quit the debugger

`undebug()` can be used to turn off debugging on the function. The `browser()` command can also be put inside functions to start the debugger.

Try `debug(mergesort)` and then `mergesort(c(3,2,4,6,1,1,10))`.

This (ie `debug()`) is like desk-checking a programme. We pretend we are the computer and carry out the programme instructions by hand on a piece of paper. I usually use this as a last resort to fix a programme.

Part A Statistical programming HT13

Geoff Nicholls

Lecture 5: Solving Linear Systems. Optimization.

Overview for lecture 5

1. R commands for matrices and vectors (reference slides)
2. Solving linear systems $Ax = b$.
   (a) Forwards and Backwards substitution
   (b) Solving $Ax = b$ for full rank A using LU factorization
   (c) Regression.
   (d) Over-determined systems. Numerical stability and QR factorization.
3. Optimization. Direct and Indirect methods.
   (a) Newton Raphson. Analysis.

Solving linear systems

Suppose A is a real $n \times p$ matrix of rank p with $p \leq n$, and entries $a_{i,j}$, and b is an $n \times 1$ real vector.

Many important numerical problems reduce to

$$\text{solve } Ax = b \text{ for } x.$$

If $p < n$, then the system is over-determined. We come back to this case later. We will look at how the equations $Ax = b$ may be solved when $p = n$ so that A^{-1} exists and $x = A^{-1}b$.

R has a function `solve(A)` returning A^{-1} so we could compute

`x=solve(A)%*%b`.

We will see that this is inefficient and numerically unstable, and find that the best method depends on the properties of A.

Forward and Backward elimination

Suppose A is lower triangular so that $a_{i,j} = 0$ for $i > j$. Solve $Ax = b$ for x using forward substitution. Chop the n equations in $Ax = b$ into blocks

$$A = \begin{pmatrix} a_{11} & 0_{1\times(n-1)} \\ A_{21} & A_{22} \end{pmatrix}$$

Here $A_{21} = A_{2:n,1}$ is $(n-1) \times 1$ and $A_{22} = A_{2:n,2:n}$ is itself lower triangular and $(n-1) \times (n-1)$. Now $Ax = b$ is

$$\begin{pmatrix} a_{11} & 0_{1\times(n-1)} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_{2:n} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_{2:n} \end{pmatrix}$$

The top row of the matrix says $a_{11}x_1 = b_1$ so $x_1 = b_1/a_{11}$.

The bottom block of the matrix has $(n-1)$ rows

$$\begin{aligned}
\begin{pmatrix} A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_{2:n} \end{pmatrix} &= b_{2:n} \\
A_{21}x_1 + A_{22}x_{2:n} &= b_{2:n} \\
A_{22}x_{2:n} &= b_{2:n} - A_{21}x_1 \\
\tilde{A}\tilde{x} &= \tilde{b} \qquad \text{now } (n-1)\times(n-1)
\end{aligned}$$

We are left with a smaller version of the problem we started with.

It took $2(n-1)+1$ additions, subtractions, multiplications and divisions (called 'flops') to solve for x_1 and calculate $\tilde{A}$ and $\tilde{b}$. Since $\sum_{i=1}^{n}(2i-1) = n^2$, forward solving is n^2 flops.

R has `forwardsolve(A,b)` for forward elimination for $n \times n$ lower triangular `A` and $n \times 1$ `b`. There is `backsolve(A,b)` for backward elimination on upper triangular `A`.

LU factorization

The most efficient method for solving $Ax = b$ for a general full rank $n \times n$ square matrix is to factorize

$$A = LU$$

into a lower L and upper U triangular matrices * at a cost of $2n^3/3 + O(n^2)$ flops (we havn't proven this, it's just assertion) and then solving $LUx = b$ by setting $y = Ux$ and then

$$\text{solving } Ly = b \text{ (forwards)}$$

and then

$$\text{solving } Ux = y \text{ (backwards).}$$

The function `solve(A,b)` uses this method. The two elimination steps take $2n^2$ flops so the leading term in the number of flops is $2n^3/3$.

*if there is no LU factorization we seek $A = PLU$ with P a permutation.

Normal linear models

Consider the aids data

```
> d = read.table("AIDS.txt")
> head(d)
  cases time time.sq
1   185    1       1
2   200    2       4
3   293    3       9
4   374    4      16
5   554    5      25
6   713    6      36
> (n<-dim(d)[1])
[1] 25
```

Suppose we want to fit the normal linear regression model

$$y_i = \alpha + \beta_1 x_i + \beta_2 x_i^2 + \varepsilon_i, \quad i = 1, 2, \ldots, n$$

with y_i the number of cases in month x_i, and $\varepsilon_i \sim N(0, \sigma^2)$ iid normal errors. In vector form the model is

$$\begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ . & . & . \\ . & . & . \\ . & . & . \\ 1 & x_n & x_n^2 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta_1 \\ \beta_2 \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ . \\ . \\ . \\ \varepsilon_n \end{pmatrix}$$

or

$$y = X\theta + \varepsilon$$

with $\theta = (\alpha, \beta_1, \beta_2)^T$ etc.

The R commands to fit this normal linear model are

```
d.lm=lm(cases ~time+time.sq,data=d)
summary(d.lm)
```

Here `d.lm` is a list full of results from the model fit output by `lm()`. Notice the R formula notation `cases~time+time.sq`.

The columns of `summary(d.lm)` output give $\hat{\theta}_i$, an estimate $\hat{\sigma}_i$ of the error in $\hat{\theta}_i$, and columns for the test H0: $\theta_i = 0$.

If the model is good, the regression should interpolate the data with normal residuals $y - X\hat{\theta}$. We can check this using a normal qq-plot for the residuals, `qqnorm(residuals(d.lm)); qqline(residuals(d.lm))`.

What's inside the `lm()` box?

The equations $X\theta = y$ are *over-determined* (more equations than variables, $n > p$, we cant expect a solution), so minimize $R(\theta) = (y - X\theta)^T(y - X\theta)$; get $X\theta$ as close as we can to y.

$$\begin{aligned} R(\theta) &= \sum_{i=1}^{n}(y_i - \alpha - \beta_1 x_i - \beta_2 x_i^2)^2 \\ &= (y - X\theta)^T(y - X\theta) \\ &= (X\theta)^T X\theta - 2y^T X\theta + y^T y \end{aligned}$$

Taking partial derivatives wrt θ and imposing $\frac{\partial R}{\partial \theta} = 0$ (p equations) leads to the p normal equations

$$X^T X\theta = X^T y$$

for θ in this over-determined system. This is $Ax = b$ with $A = X^T X$, $x = \theta$ and $b = X^T y$.

Solving the normal equations using QR factorization

We could use LU factorization to solve the normal equations. However QR factorization is usually best as it is more stable numerically.

$$X = \begin{pmatrix} 1 & -1 \\ 0 & 10^{-10} \\ 0 & 0 \end{pmatrix} \qquad X^T X = \begin{pmatrix} 1 & -1 \\ -1 & 1 + 10^{-20} \end{pmatrix}$$

At machine precision $1+10^{-20}$ and 1 are equal so $X^T X$ appears to be singular. Any method (like LU) that solves $(X^T X)\theta = X^T y$ by first computing $X^T X$ will fail on this problem.

Instead, factorize $X = QR$ (Q is $n \times p$ and orthogonal, so $Q^T Q = I_{p \times p}$, and R is $p \times p$, upper triangular, and has positive

entries on the diagonal). This takes $2np^2$ flops (assertion). Since

$$X^T X = R^T Q^T Q R,$$

the normal equations

$$X^T X\theta = X^T y$$

are

$$R^T R\theta = R^T Q^T y.$$

We can solve these by

$$\text{solving } R\theta = Q^T y \text{ (backwards)}$$

($np + p^2$ flops) for an overall leading order cost of $2np^2$ flops. The functions `qr.solve(X,y)` and `lm()` use this method. LU would take np^2 but may fail.

In R,

```
X=cbind(rep(1,n),d$time,d$time.sq)
```

followed by

```
d.theta=qr.solve(X,d$cases)
```

to give the regression parameters.

Optimization via Newton Raphson

We optimized the residual sum of squares, $R(\theta)$ over θ using a direct method. It terminates in a fixed number of operations which we know in advance. Indirect methods for optimization iterate to the optimal solution (hopefully). Indirect methods come with fewer guarantees, but can be used to treat more complex problems.

We optimize a function $f(x)$ over x, and seek x such that $f'(x) = 0$. We can apply the Newton Raphson algorithm. The iteration for zeros of $f(x)$ was

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

so the iteration for zeros of $f'(x)$ is

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}.$$

We stop when $f'(x_n) < \epsilon$ and check $f''(x_n)$ to determine the type of stationary point.

The Newton-Raphson method - convergence

Proposition : if $f'(x^*) = 0$, and at all points in some neighborhood of x^*, f', f'' and f''' exist and f'' is nowhere zero then the Newton-Raphson method converges quadratically if $|x_0 - x^*|$ is sufficiently small.

Proof : Suppose x^* is a root of $f'(x) = 0$ then expanding $f'(x^*)$ in a Taylor series around the nth iteration of the Newton-Raphson scheme, x_n, gives

$$0 = f'(x^*) = f'(x_n) + (x^* - x_n) f''(x_n) + (x^* - x_n)^2 \frac{f'''(\lambda)}{2} \quad (1)$$

for some λ in between x^* and x_n.

By definition

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \Rightarrow f'(x_n) = f''(x_n)(x_n - x_{n+1}) \quad (2)$$

Combining (1) and (2) gives

$$0 = f''(x_n)(x^* - x_{n+1}) + \frac{f'''(\lambda)}{2}(x^* - x_n)^2.$$

Now let $\epsilon_n = x^* - x_n$ and $\epsilon_{n+1} = x^* - x_{n+1}$ be the errors at the nth and $(n+1)$th iterations.

$$\begin{aligned} \Rightarrow 0 &= f''(x_n)\epsilon_{n+1} + \frac{f'''(\lambda)}{2}\epsilon_n^2 \\ \Rightarrow \epsilon_{n+1} &= -\frac{f'''(\lambda)}{2f''(x_n)}\epsilon_n^2 \\ \Rightarrow \epsilon_{n+1} &\propto \epsilon_n^2 \end{aligned}$$

i.e. the error at the $(n+1)$th iteration is proportional to the square of the error at the nth iteration.

Comments about Newton-Raphson method

- Some functions have more than one stationary point and Newton-Raphson won't necessarily find the best one.
- If $f''(x_n) = 0$ (or is very close to 0) at any iteration then the next iterate is undefined (or very far from x_n).
- Otherwise, the Newton-Raphson iteration converges to a local stationary point, provided the start point is close enough to a stationary point.
- The Bisection method works under weaker conditions, but has slower (linear) convergence. The error is halved each iteration.

Part A Statistical programming HT13

Geoff Nicholls

Lecture 6: Simulation

Overview for lecture 6

1. Estimating an expectation by simulating the random variable
2. Simulating familiar distributions using R
3. Monte Carlo Integration
4. Pseudo-random numbers and random number seeds.
5. Markov Chains and Random Walks
   (a) Simulating simple discrete distributions
   (b) Simulating simple Markov chains
   (c) Simulating Hitting Times

Estimating Expectations using simulation

Suppose $X_i \sim f_X, i = 1, 2, ..., n$ are iid r.v. with pdf f_X and we want $\mu_f = \mathbb{E}(g(X))$. Assume $\sigma_g^2 = \mathsf{Var}(f(X))$ is finite. Let

$$\bar{g}_n = \frac{1}{n} \sum_{i=1}^{n} g(X_i)$$

and

$$S_g^2 = \frac{1}{(n-1)} \sum_{i=1}^{n} (X_i - \bar{g}_n)^2.$$

By the CLT, and approximately at large n, $\bar{g}_n \sim N(\mu_g, \sigma_g^2/n)$, so $\hat{\mu}_f = \bar{f}_n$ estimates μ_g and $\bar{g}_n \pm z_{1-\alpha/2} S/\sqrt{n}$ is an approximate level-α CI for μ_g (z_p the p-quantile of a standard normal).

Example: suppose $X \sim Exp(1)$, and we need an estimate of

$$\mathbb{E}\left(\frac{1}{1+\sin^2(X)}\right) = \int_0^\infty \frac{1}{1+\sin^2(x)} e^{-x} dx.$$

We want $X_1, X_2, ..., X_n$ iid like Exp(1) and then compute

$$\bar{g}_n = \sum_{i=1}^{n} \frac{1}{1+\sin^2(X_i)}$$

```
x=rexp(1e6); fx=1/(1+sin(x)^2)
fbar=mean(fx); Sqn=sqrt(var(fx)/n)
Za=qnorm(1-0.025); ci=c(fbar-Za*Sqn,fbar+Za*Sqn)
```

At $n = 10^6$, I found $\bar{g}_n$=0.758126, $S/\sqrt{n}$=0.0001787730 and level-0.05 confidence interval c$(0.7578, 0.7585)$.

Simulating familiar distributions

`rDBN(),dDBN(),pDBN(),qDBN()` are random numbers, the pdf/pmf, the cdf and quantiles for the distribution `DBN`.

`rexp(n=10,rate=1)` gives 10 Exp(1) rv.

`pnorm(q,mean=0,sd=1)` is $P(Z \leq q)$ for $Z \sim N(0,1)$

`dpois(x,lambda)` is $\exp(-\lambda)\lambda^x/x!$

If `q=qt(p=0.975,df=4)` then $0.975 = P(X \leq q),\ X \sim t(4)$.

What does this return? `rchisq(n=10,df=5)`

Monte Carlo Integration

Suppose we have an integral

$$I = \int_\Omega h(x)dx$$

to evaluate. If we can find a density f_X on Ω and a (nice) function $g(x)$ so that $h(x) = g(x)f_X(x)$ then

$$\int_\Omega h(x)dx = \int_\Omega g(x)f_X(x)dx$$

so $I = \mathbb{E}(g(X))$ and we can use simulation to get $\hat{I}$.

Example:

$$\begin{aligned} I &= \int_2^4 \log(\log(x))dx \\ &= \int_a^b (b-a)\log(\log(x)) \times \frac{1}{b-a}dx, \end{aligned}$$

with $a = 2, b = 4$. But

$$f_X(x) = (b-a)^{-1}$$

is the density of a uniform $X \sim U(a,b)$, for $a < X < b$, so

$$\begin{aligned} I &= \int_2^4 2\log(\log(x))]f_X(x)dx \\ &= \mathbb{E}(2\log(\log(X))). \end{aligned}$$

Simulate $X_i \sim U(a,b)$ and report $\hat{I} = n^{-1}\sum_i 2\log(\log(X_i))$.

```
> n=10000
> X=runif(n,min=2,max=4)
> gx=2*log(log(X))
> (Ihat=mean(gx))
[1] 0.1155751
> (Sqn=sqrt(var(gx)/n))
[1] 0.003855409
```

Report $\hat{I} = 0.116(4)$. Compare $\hat{I} = 0.117141566$ (quadrature).

In this example we are integrating in dimension one. For integrals in dimensions less than about 15 (depending on the problem) methods based on quadrature will be more efficient (smaller error for given runtime). Monte Carlo methods are a 'heavy hammer' for high dimensional problems, for integrals with awkward constraints, or sometimes simply because they are simple and 'good enough'.

Pseudo random numbers

How do the `rexp()`, `rnorm()` etc functions work? How do they simulate random variables from the given distribution?

Many take transforms of $U \sim U(0,1)$ rv. For example, if we want to simulate a r.v. X with CDF $F(x)$ then use $X = F^{-1}(U)$:

$$P(X < x) = P(F^{-1}(U) < x) = P(U < F(x)) = F(x).$$

Example: Exp(1) has CDF $F(x) = 1 - \exp(-x)$ so

$$X = -\log(1-U) \quad \Rightarrow X \sim \text{Exp}(1)$$

and that is how `rexp()` simulates $\text{Exp}(1)$ distributed rv. But ... we need a supply of $U(0,1)$ rv to begin with.

Let integers b (small) and m (large) be given.

1. choose a seed x_0 between 1 and m.
2. iterate

   i $x_i = bx_{i-1} (\text{mod}\, m)$
   ii $u_i = x_i/m$

   If b and m are carefully chosen, the deterministic sequence $u_1, u_2, \ldots$ behaves like an iid $U[0,1]$ sequence.

The default R `runif()` sequence is a Mersenne-Twister. The seed is set using using the clock i.e. the seed depends on when we start R.

The seed can be set using the set.seed() function. Handy for debugging when working with rv.

Simulating simple discrete distributions

Suppose $X \in \{1, 2, ..., K\}$ and the pmf is $P(X = i) = p_i$, so $p = (p_1, p_2, ..., p_K)$ is a vector of probabilities summing to one.

Simulate $X \sim p$ using the following algorithm.

1) Let $q = (q_1, q_2, ..., q_K)$ with $q_1 = p_1, q_2 = p_1 + p_2, ...q_K = 1$.

2) Simulate $U \sim U(0, 1)$.

3) Let i be the index of the smallest entry in q exceeding U.

The algorithm returns $X = i$. The probability to get $X = i$ is the probability for the event $q_{i-1} < U \leq q_i$ which is $q_i - q_{i-1} = p_i$. The value of X returned is equal i with probability p_i.

Simulating Markov Chains

Consider a Markov chain, $X_1, X_2, ...$ with K possible states, and transition matrix P,

$$P(i,j) = P(X_t = j | X_{t-1} = i) \quad i,j \in \{1,2,...,K\}.$$

For example if $X_1 = 1$ and

$$P = \begin{pmatrix} 1/2 & 0 & 3/8 & 1/8 \\ 0 & 1/2 & 0 & 1/2 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 1/2 & 0 & 1/2 \end{pmatrix}$$

then X_2 is simulated using the distribution in row X_1, that is $X_2 \sim (1/2, 0, 3/8, 1/8)$. In general

$$X_{t+1} \sim (P(X_t, 1), P(X_t, 2), ..., P(X_t, K))$$

and we iterate this to simulate as many steps as we need.

Here is some code for you to complete in the prac.

```
Markov<-function(P,x=1,n=1) {
   #simulate n steps of a MC with transition matrix P
   #starting from x
   X=rep(0,n)
   X[1]=x
   for (t in 2:n) {
      #assign a vector p to be row X[t-1,] of P
      #simulate X[t] with pmf p
   }
   return(X)
}
```

Random walks and hitting times

In the gamblers ruin, $X_k \in \{0, 1, 2, ..., N\}$. The gambler starts with $X_k = m$. At each step we add one with probability p and otherwise remove one. The process stops at the first k such that $X_k \in \{0, N\}$.

This is a Markov chain (can you write down the transition matrix?). However it is more straightforward to simulate a coin toss at each step (Heads with probability p) and add or subtract one from the total as the coin is Heads or Tails.