R

David J. Scott

d.scott@auckland.ac.nz

Department of Statistics, University of Auckland

Outline

- Introduction: the **S** language
- Basic **S** concepts
- Practicalities: help, commands, running **R**, ESS
- Data structures
  - Numbers and vectors
  - Objects, modes and attributes
  - Factors
  - Arrays and matrices
  - Lists and data frames
- Vectors
- Matrices
- Text handling in **R**
- Importing and exporting data

Outline (continued)

- Programming in **S**
- Dates and times
- Miscellanea
  - Scope
  - Customising the environment
  - Installing packages
  - Accessing source code
  - Mathematical expressions
  - Probability distributions
  - Optimisation
- An extended example: the Weibull distribution

Outline (continued)

- **S** graphics
- Trellis graphics and lattice
- Sweave
- Writing functions and packages
- S4 methods

Resources

- CRAN (Comprehensive R Archive Network), local mirror
  `http://cran.stat.auckland.ac.nz/`
- An Introduction to R
  `http://cran.stat.auckland.ac.nz/doc/manuals/R-intro.pdf`
- A Reference Card for R from Tom Short
  `www.stat.auckland.ac.nz/~dscott/782/RrefcardShort.pdf`
- The ESS Manual
  `http://www.stat.auckland.ac.nz/~dscott/782/essmanual.pdf`
- A Reference Card for ESS
  `www.stat.auckland.ac.nz/~dscott/782/essrefcard.pdf`
- The Sweave User Manual
  `www.stat.auckland.ac.nz/~dscott/782/Sweave-manual-20050103.pdf`

Introduction: The S Language

The S Language

S is a language and computational environment designed specifically for carrying out “statistical” computations.

- It was designed by statisticians so that they could easily develop and deploy new statistical methodology.
- It brings good ideas on numerical methods, graphics and language design together in a single useful package.
- It represents a philosophy of *turning ideas into programs* quickly and easily.

History

- The **S** language was developed at AT&T/Lucent Bell Laboratories by *John Chambers*, with the assistance of *Rick Becker*, *Allan Wilks* and *Duncan Temple Lang*.
- The development of the language was aided by collaboration of the AT&T researchers with a much wide group of academics and practitioners.
- The developers of **S** language recognise a number of distinct *versions* of **S** which came about during the development of the language.

S Version 1

- Released in 1980.
- The first version of **S** released outside of Bell Labs.
- Used primarily in North American universities.
- It is described in the book

  S : A Language and Environment For
  Data Analysis and Graphics

  (also known as the “brown” book).

S Version 2

- Released in 1987.
- Based initial experiences developing **S**, Chambers developed a new environment called QPE.
- This added the ability for users to define their own extensions to the language. This version of **S** is described in the book

  The NEW S Language

  (also known as the “blue” book).

S Version 3

- Released in 1990.
- Added some basic object-oriented facilities, which were seen as essential for providing good modelling facilities.
- Included “State of the Art” modelling software.
- These extensions are described in the book

  Models in S

  also known as the “white” book.

S Version 4

- Released in 2000.
- Contains a rather more sophisticated version of object-oriented programming and a number of other developments which support programming activities with S.
- The initial versions of S4 ran very slowly, and this has tended delay its acceptance by the **S** programming community.
- This version of **S** is described in the book

  Programming with Data

  (also known as the “green book”).

The ACM Software Systems Award

- The importance of John Chambers’ work on **S** was recognised in 1998, when he was awarded the ACM award for software systems.
- The ACM award is the world’s most prestigious award for software engineering.
- Other winners of the ACM award include Thompson and Ritchie for the UNIX operating system and Don Knuth for the TEX typesetting system.

S Implementations

S	This was the version of S used internally for research purposes at Bell Labs. A small group of external researchers had access to this version.
S-PLUS	In the late 1980s AT&T sold the rights to sell a commercial version of S to Seattle-based *Statistical Sciences*. The company has since been renamed twice and is now known as *Insightful*.
R	In the early 1990s Robert Gentleman and Ross Ihaka University of Auckland developed an alternative free **S** implementation. It is called **R** after its original developers: Robert Gentleman and Ross Ihaka.

R

- **R** is now developed by the R-Core Development Team
- New releases are produced around every six months
- In addition to base **R** there are a number of recommended packages which comprise the basic **R** distribution
- There are around 1000 additional packages produced by users of **R**
- The source code and compiled versions of **R** for Windows, Macintosh and various Linux distributions (debian, redhat, ubuntu, . . .) are available from CRAN
- CRAN also has user-produced packages and documentation

Basic S Concepts

Basic S Concepts

- **S** is a computer language which is processed by a special program called an *interpreter*
- The interpreter reads and evaluates **S** language *expressions*, and prints the values determined for the expressions
- The interpreter indicates that it is expecting input by printing its *prompt* at the start of a line
- By default the **S** prompt is a *greater than* sign >.
- On UNIX or LINUX machines you can start Splus by typing "`Splus`" and R by typing "`R`"

Using S as a Calculator

- A user types *expressions* to the **S** interpreter
- **S** responds by computing and printing the answers

```
> 1 + 2
[1] 3
> 1/2
[1] 0.5
> 17^2
[1] 289
```

Grouping and Evaluation

- Normal arithmetic rules apply,

```
> 1 + 2 * 3
[1] 7
```

- But evaluation can be controlled with parentheses

```
> (1 + 2) * 3
[1] 9
```

Built-in Functions

- Many mathematical and statistical functions are available.

```
> sqrt(2)
[1] 1.414214
> log(10)
[1] 2.302585
> qnorm(0.975)
[1] 1.959964
```

Assignment

- Values are stored by *assigning* them a name
- The resulting name-value pair is called a *variable*.
- The statements:

  ```
  > z = 17
  > z <- 17
  ```

  store the value `17` under the name `z`.
- You can also write the assignment in the other direction
  `17 -> z`

Using Variables

- Variables can be used in expressions in the same way as numbers
- For example,

```
> z <- 17
> z <- z + 23
> z

[1] 40
```

Special Values

- In addition to ordinary numbers, **S** has special codes which indicate infinite and undefined numerical values

```
> 1/0
[1] Inf
> -1/0
[1] -Inf
> 0/0
[1] NaN
```

Special Values

- Infinities and NaNs have expected properties

```
> 100 + Inf
[1] Inf
> 100 + NaN
[1] NaN
> 100/Inf
[1] 0
> Inf/Inf
[1] NaN
```

Practicalities

Help

- Help using `help` or preceding with `?`
  `help(solve)` or `?solve`
- Other help from `help.start`, `help.search`, and `example`, e.g.

```
help.search("unix")
help(package="HyperbolicDist")
example(system.time)
```

Commands

- **R** is *case-sensitive*
- Names are locale dependent, alphanumeric characters allowed, plus ’.’ and ’_’
- Commands are either *expressions* or *assignments*
- Expressions are evaluated and printed
- An assignment evaluates an expression and passes the value to a variable: the result is not automatically printed
- An assignment in brackets is printed
- # denotes the start of a comment

Commands

```
> 1:10

 [1]  1  2  3  4  5  6  7  8  9 10

> x <- 1:10
> (x <- 10:20)

 [1] 10 11 12 13 14 15 16 17 18 19 20
```

Running R

- Can run in a terminal window: command recall and editing is possible
- Best environment is XEmacs and ESS
- On Windows WinEdt plays very nicely with **R**
- Run commands from a file with `source`
  ```
  > source("commands.R")
  ```
- Put output to a file with `sink`
  ```
  > sink("commands.out")
  ```
- Return to console output with `sink()`
- See objects in current workspace with `objects()` or `ls()`
- Remove objects with `rm`
  ```
  > rm(x,n,mydata,temp)
  ```

XEmacs and ESS

- To start **R** from within XEmacs use M-x R
- Most important is to evaluate a region C-c C-r
- This leaves you in the commands file
- C-c M-r evaluates the region but leaves you in **R**
- Evaluate a line with C-c C-j
- To ensure XEmacs uses ESS, put the line `(require 'ess-site)` in the file init.el
- Note that init.el is accessible in XEmacs from the Options menu: select "Edit Init File"

Workspace, History, Batch Mode

- All objects can be saved at the end of a session
- Saved in `.RData` in the current directory
- Objects will be reloaded if a later session is started in this directory
- To avoid confusion of objects, use different directories for different projects
- To run in batch mode, on the Unix command line:
  `R CMD BATCH` *`filename.R`*
- Output will be in *`filename.Rout`*
- Can be done in the background or with `nice` or `nohup`
- See `?BATCH`
- There are other commands like this, e.g. `R CMD INSTALL`

Data Structures

Data Structures

- Vectors: can be numeric, character, or logical
- Matrices: just vectors with dimensions added and possibly row and column names—elements are all of the same type
- Factors : for categorical data
- Lists: like a general form of vector
  - elements can be of different types
  - elements can be vectors or lists
  - used for returning results from functions
- Data frames: like matrices, rectangular
  - columns can be of different types
  - common input to modeling functions
- Functions

Objects

- **R** operates on objects
- Vectors are atomic—all of one type or *mode*
- Lists are recursive structures
- Functions and expressions are also recursive structures
- Mode and length are intrinsic attributes of objects
- Can change mode—coerce objects
- Once an object exists its length can be changed
- Change attributes with `attributes`
- Finding out about objects: `str`, `attributes`, `head`, `tail`, and for functions `args`

Objects

- Objects may have a class—used for object-oriented programming
- Determines the behaviour of `plot`, `summary`, and printing
- These are generic functions
- If no class, default is used, e.g. `plot.default` is used if you try and plot the object
- Remove the class using `unclass`
- See the available methods with `methods`
- See function code for non-visible functions by using `getS3method("`*`genericname`*`","`*`classname`*`")` or `getAnywhere(`*`functionname`*`)`
- Alternatively if the namespace is known, use *`namespace`*`:::`*`functionname`*

Factors

- Specifies a grouping
- Can be ordered or unordered
- Used in modeling to do traditional analysis of variance
- Use `factor` or `ordered` to create factors
- See examples from `?factor`
- `table` allows construction of tables using factors

Lists

- An ordered collection of components
- To create:

```
Lst <- list(name="Fred",wife="Mary",no.children=3,
            child.ages=c(4,7,9))
```

- Refer to components using number in double brackets:
  `Lst[[3]]`
  by name using the separator `$`:
  `Lst$name`
  or by name using the quoted name in double brackets:
  `Lst[["name"]]`
- `Lst[[1]]` is different from `Lst[1]`
- Lists are constructed using `list`

Data Frames

- A list of class `data.frame`
- Components are vectors, factors, numeric matrices, lists, or other data frames
- Matrices, lists, and data frames provide as many variables to the new data frame as they have columns, elements, or variables, respectively
- Numeric vectors, logicals and factors are included as is
- Character vectors are coerced to factors
- Vector structures in the data frame must all have the same length
- Matrix structures must have the same row size

Data Frames

- Create with `data.frame`

```
accountants <- data.frame(home=statef, loot=incomes,
                          shot=incomef)
```

- Coerce for example matrices using `as.data.frame`
- Instead of using $ notation, such as `accountants$statef` use `attach()` and `detach()`

```
> table(home)
Error in table(home) : Object "home" not found
> attach(accountants)
> table(home)
home
act nsw  nt qld  sa tas vic  wa
  2   6   2   5   4   2   5   4
> detach(accountants)
```

Vectors

Vectors

- **S** works naturally with vectors of values
- The simple way to combine scalar values into a vector is using the `c()` function

```
> x <- c(1, 2, 4, 3, 1)
> x
[1] 1 2 4 3 1
```

- Vectors can be manipulated just like scalar values

```
> 2 * x
[1] 2 4 8 6 2
> x/4
[1] 0.25 0.50 1.00 0.75 0.25
```

Properties of Vectors

- The functions `length` and `mode` return information about the values stored in vectors

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

[1] 5

> mode(x)

[1] "numeric"
```

Simple Subsetting

- Individual vector elements can be accessed with a subsetting mechanism

```
> x <- c(1, 2, 4, 3, 1)
> x[5]

[1] 1

> x[3]

[1] 4
```

Vector Subsets

- Extracting vector subsets:

```
> x <- c(1, 2, 4, 3, 1)
> x[c(1, 2, 3)]

[1] 1 2 4

> x[-5]

[1] 1 2 4 3
```

- When subscripts are all negative, all elements except those are extracted

Logical Subsetting

- Values can be extracted using logical conditions
- For example, the command `x[x>2]` extracts all elements from `x` which are greater than 2

```
> x <- c(1, 2, 4, 3, 1)
> x[x > 2]

[1] 4 3
```

- Logical subsetting can be very expressive.

```
> mean(income[gender == "male"])
```

Modifying Subsets

- It is possible to change the values in a subset of a vector by using subsetting in conjunction with assignment
- For example, the expression

  ```
  > x[1:2] <- 10
  ```

  will change the first two elements of the vector `x` to `10`.

Patterned Sequences

- **S** has a number of ways of generating vectors containing special sequences of values
- The one that is most commonly used is the sequence operator "`:`"
- The expression $n_1 : n_2$ returns the sequence of integer values from n_1 to n_2.

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
```

Patterned Sequences Using `:`

- The vectors created with `:` can be quite large, and when printed they may span several lines.

```
> 1:50
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
[13] 13 14 15 16 17 18 19 20 21 22 23 24
[25] 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48
[49] 49 50
```

- The value in brackets at the start of each line gives the index of the first value on the line
- This can make it easier to locate particular values

Patterned Sequences Using `seq`

- More general sequences can be created by using the “`seq`” function
- The expression `seq(0,5,by=0.2)` generates the sequence of values from 0 to 5 in steps of 0.2

```
> seq(0, 5, by = 0.2)

 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6
[10] 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4
[19] 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0
```

- It is also possible to create sequences of a specified length

```
> seq(0, 4, length = 9)

[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

Patterned Sequences Using `rep`

- Another function which is useful for creating patterned sequences is the function “`rep`”, which repeats its first argument according to the value of its second
- The second argument can either be a single count, giving the number of times to repeat the first

```
> rep(1:3, 5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

- Or it can be a vector of counts, indicating how many times to repeat each element of the first argument

```
> rep(1:3, c(3, 4, 3))
 [1] 1 1 1 2 2 2 2 3 3 3
```

Arithmetic on Vectors

- **S** has very general capabilities for vector arithmetic
- Arithmetic operations are carried out on vectors according to an *element recycling rule*
- Under this rule, when vectors of different lengths are combined in an arithmetic operation, the shorter vector is first enlarged to match the length of the longer vector by recycling its elements
- Then the vectors are combined element by element

Recycling Rule Example

Consider adding the vectors `1:3` and `1:6`

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} \xrightarrow{\textit{recycle}} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} \xrightarrow{\textit{add}} \begin{bmatrix} 2 \\ 4 \\ 6 \\ 5 \\ 7 \\ 9 \end{bmatrix}$$

This is how the result appears when computed with **S**

```
> 1:3 + 1:6

[1] 2 4 6 5 7 9
```

Summary Operations on Vectors

- The following functions return single-value numerical summaries for the values in a vector or collection of vectors

`sum(x)`	the sum of the elements of `x`
`prod(x)`	the product of the elements of `x`
`min(x)`	the minimum of the elements of `x`
`max(x)`	the maximum of the elements of `x`

- The `range` function computes a two value summary.

```
> range(1:10)
[1]  1 10
```

Examples

- The `prod` function can be used to compute factorials
- For example $10!$

```
> prod(1:10)
[1] 3628800
```

- Beware that very large factorials can't be represented with finite precision arithmetic ($200!$ is too large)

```
> prod(1:200)
[1] Inf
```

- The following computation shows how to compute binomial coefficients.

```
> prod(1:10)/prod(1:3, 1:7)
[1] 120
```

Cumulative Summaries

- There are also cumulative versions called `cumsum`, `cumprod`, `cummax`, and `cummin`
- These produce a vector which consists of the summary computed for; the first element, first two elements, the first three elements and so on

```
> cumsum(1:10)

 [1]  1  3  6 10 15 21 28 36 45 55

> cumprod(1:10)

 [1]       1       2       6      24
 [5]     120     720    5040   40320
 [9]  362880 3628800
```

Statistical Summaries

- There are a number of summaries which compute quantities of statistical interest
- The most important of these are:

`mean(x)`	the mean of the elements of `x`,
`median(x)`	the median of the elements of `x`,
`var(x)`	the variance of the elements of `x`.

- In addition to these, **R** also has

`sd(x)`	the s.d. of the elements of `x`.

Sample Quantiles

- The `quantile` function computes summaries based on the percentiles of the values in its argument
- The simplest use of `quantile` computes the median, upper and lower quartiles, and extremes

```
> quantile(1:10)
```

```
   0%    25%    50%    75%   100% 
 1.00   3.25   5.50   7.75  10.00 
```

- An optional second argument to `quantile` can be used to specify a different set of quantiles

```
> quantile(1:10, 0:10/10)
```

```
  0%  10%  20%  30%  40%  50%  60%  70% 
 1.0  1.9  2.8  3.7  4.6  5.5  6.4  7.3 
 80%  90% 100% 
 8.2  9.1 10.0 
```

Character Strings

- Any value in quotes is regarded by **R** as being a character string.

```
> greeting <- "hello, world"
> greeting

[1] "hello, world"
```

- This is true, even when the value appears to be numeric

```
> number <- "21"
> number

[1] "21"
```

- Character strings can be combined to form vectors

```
> c(greeting, number)

[1] "hello, world" "21"
```

Type Coercion

- It is possible to combine numbers and character strings into a single vector using "`c`"
- In this case, the numbers are first converted to strings.

```
> c(greeting, 100)
[1] "hello, world" "100"
```

- This an example of automatic *type coercion*
- Arithmetic on a mixture of strings and numbers does not work, however

```
> "100" + 10
Error in "100" + 10 : non-numeric argument
                      to binary operator
```

Naming Vector Elements

- Individual elements of a vector can be given names

```
> x <- 1:5
> x
[1] 1 2 3 4 5
> l <- c("a", "b", "c", "d", "e")
> l
[1] "a" "b" "c" "d" "e"
> names(x) <- l
> x
a b c d e
1 2 3 4 5
```

- Element names are printed above the elements, indexing information at the start of a line is dropped

Names and Subsetting

- Names are preserved during subsetting

```
> x[4:5]
d e
4 5
```

- It is also possible to extract subsets by using the names as subscripts

```
> x[c("a", "d")]
a d
1 4
```

Matrices

Matrices

- In addition to vectors, **S** has a wide range of data structures.

  ```
  > A <- matrix(1:6, nrow = 3, ncol = 2)
  ```

  creates a 3×2 matrix
- The value can be viewed as follows:

  ```
  > A
  ```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

- Notice that the elements are inserted into the matrix in *column major order*

Row-Major Order

- By default, values are inserted into matrices in column major order
- It is also possible to specify that the matrix be filled in row major order

```
> B <- matrix(1:6, nrow = 3, ncol = 2,
+       byrow = TRUE)
> B
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Dimensioning Information

- Matrix dimensions can be obtained in a number of ways

```
> A <- matrix(1:6, nrow = 3, ncol = 2)
> nrow(A)
[1] 3
> ncol(A)
[1] 2
> dim(A)
[1] 3 2
```

Diagonal Matrices

- The function `diag` can be used to create diagonal matrices and to extract the diagonals from matrices

```
> A <- diag(1:3)
> A

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3

> diag(A)

[1] 1 2 3
```

Special Matrix Forms

- There is a special form of call to `diag` which can be used to create identity matrices

```
> diag(3)

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

- It is also easy to construct a block matrix of ones

```
> matrix(1, nrow = 3, ncol = 3)

     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

Matrix Arithmetic

- The standard arithmetic operations are all defined for matrices, and take place elementwise

```
> A <- matrix(1:6, nrow = 3, ncol = 2)
> B <- matrix(1:6, nrow = 3, ncol = 2,
+        byrow = TRUE)
> A + B
```

```
     [,1] [,2]
[1,]    2    6
[2,]    5    9
[3,]    8   12
```

- Note that `A * B` is the elementwise product of `A` and `B`, not the matrix product

Other Matrix Operations

- The function `t` computes the transpose of its argument

```
> t(A)

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

- Matrix multiplication can be performed with the `%*%` binary operator

```
> t(A) %*% B

     [,1] [,2]
[1,]   22   28
[2,]   49   64
```

Solving Linear Equations

- The `solve` function an be used to solve systems of linear equations
- For example the linear system

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

can be solved as follows

```
> A <- matrix(c(1, 3, 2, 4), ncol = 2)
> b <- c(1, 1)
> solve(A, b)
```

```
[1] -1  1
```

Matrix Inversion

- Called with a single argument, `solve` computes the matrix inverse

```
> solve(A)

     [,1] [,2]
[1,] -2.0  1.0
[2,]  1.5 -0.5
```

- **But** computers only work to a finite precision—all computations are subject to roundoff error

```
> A %*% solve(A)

     [,1]         [,2]
[1,]    1 1.110223e-16
[2,]    0 1.000000e+00
```

Regression

- The general linear model can be written in matrix form as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}.$$

- The least-squares estimates of the parameters are given by

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

  and the residuals by

$$\hat{\boldsymbol{\varepsilon}} = \mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}.$$

- The estimated dispersion matrix of $\hat{\boldsymbol{\beta}}$ is

$$\mathcal{D}[\hat{\boldsymbol{\beta}}] = \frac{\hat{\boldsymbol{\varepsilon}}'\hat{\boldsymbol{\varepsilon}}}{n-p}(\mathbf{X}'\mathbf{X})^{-1}.$$

Regression in S

- The equations for regression analysis can be translated directly into **S** statements

```
> n <- nrow(x)
> p <- ncol(x)
> betahat <- solve(t(x) %*% x, t(x) %*%
+      y)
> epsilonhat <- y - x %*% betahat
> sigmahat2 <- sum(epsilonhat^2)/(n -
+      p)
> D <- sigmahat2 * solve(t(x) %*%
+      x)
```

- **Warning**: This is **not** the best way of computing the results

Matrix Decompositions

- There are many **S** functions which support computations on matrices
- The most useful are:

`eigen`	eigenvalues and eigenvectors
`svd`	singular-value decomposition
`qr`	QR decomposition

Subsetting

- It is possible to extract submatrices from matrices

```
> A <- matrix(1:6, ncol = 3)
> A

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> A[1, 2]

[1] 3

> A[1:2, 1:2]

     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Subsetting Shorthands

- An empty subscript field is equivalent to the full range of possible subscripts
- For example:

```
> A[, 2:3]

     [,1] [,2]
[1,]    3    5
[2,]    4    6

> A[2, ]

[1] 2 4 6
```

- Subscripting matrices by using logical indices and row or column labels is also possible, but tends to be far less common

Row and Column Indices

- The two functions `row` and `column` return matrices containing the index for each row an column

```
> A <- matrix(1:6, ncol = 3)
> row(A)

     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2

> col(A)

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
```

Example: Zeroing A Lower Triangle

- `row` and `col` can be used together with logical subscripting to operate on upper and lower triangles of matrices

```
> A <- matrix(1:9, ncol = 3)
> A[row(A) > col(A)] <- 0
> A

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    0    5    8
[3,]    0    0    9
```

Row and Column Labeling

- Matrices can be made rather more useful by using row and column labels

```
> A <- matrix(1:6, nrow = 3)
> dimnames(A) <- list(c("sex", "drugs",
+      "rock&roll"), c("this", "that"))
```

- The primary benefit of labeling can be seen when the matrix is printed

```
> A

          this that
sex          1    4
drugs        2    5
rock&roll    3    6
```

Labeling The Labels

- In **R**, but not **S**, it is possible to “label the labels.”

```
> A <- matrix(1:6, nrow = 3)
> dimnames(A) <- list(what = c("sex",
+       "drugs", "rock&roll"), which = c("this",
+       "that"))
> A

           which
what         this that
  sex           1    4
  drugs         2    5
  rock&roll     3    6
```

Multiway Arrays

- Multiway arrays generalise the notion of matrices.
  - Arrays are created with the `array` function and their subsetting and labeling methods parallel those of matrices.
  - The only major difference is that the notion of transpose must be generalized.
  - The function `aperm` provides such a generalized transpose operation.

Multiway Arrays

```
> (A <- array(1:5,c(2,3,1)))
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    1

> aperm(A)
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5

, , 2

     [,1] [,2] [,3]
[1,]    2    4    1
```

Multiway Arrays

```
> aperm(A,c(2,1,3))
, , 1

     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    1
```

Text Handling in R

Printing

- Complete **S** objects can be displayed by using the `print` function

```
> print("A Title")

[1] "A Title"

> print(matrix(1:4, nc = 2))

     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

- Only very basic control of formatting is possible when the `print` function is used
- Other functions can be used when finer control is necessary

Control Of Printing

- Print formating can be controlled by a number of optional arguments

`digits=`	controls the number of digits printed to the right of the decimal point. The default is the value of the system option `digits`.
`quote=`	controls whether strings are printed with or without quotes. The default value is `TRUE`.
`na.print=`	the character string used to indicate `NA` values. This is printed without quotes.

Examples Using `print`

```
> print(c("Hello", "there"), quote = FALSE)
[1] Hello there
> print(1/1:5, digits = 2)
[1] 1.00 0.50 0.33 0.25 0.20
> print(c(1, NA, Inf), na.print = ".")
[1]   1   . Inf
```

The `cat` function

- `cat` is a function which can be used to concatenate and then print character strings passed to it as arguments
    - Objects other than character strings are automatically converted to character strings by `cat`
    - By default, the strings being concatenated are separated by a space character. Can be overridden with the `sep=` argument.
    - Certain sequences of characters have a special interpretation (shared with C, C++, Java and other languages)
    - `\n` represents a *newline* character, `\t` represents a *tab* character, and `\\` a *backslash* character

Examples Using `cat`

```
> R2 <- 0.7863
> cat("aaa", "bbb", "\n")
aaa bbb
> cat("aaa", "bbb", "\n", sep = "")
aaabbb
> cat("a", "b", "c", "d", "\n", sep = " - ")
a - b - c - d -
> cat("R-squared =", R2, "\n")
R-squared = 0.7863
```

Vector Arguments to `cat`

- When the arguments to `cat` are vectors, their elements are treated as though they were separate arguments.

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
[10] "j" "k" "l" "m" "n" "o" "p" "q" "r"
[19] "s" "t" "u" "v" "w" "x" "y" "z"
```

```
> cat("The alphabet: ", letters,
+     "\n", sep = "")
```

```
The alphabet: abcdefghijklmnopqrstuvwxyz
```

Rounding

- There are a variety of **S** functions which help to format numbers for use with `cat`
- The function `round` rounds its argument to a specified number of decimal digits

```
> round(runif(5), 2)

[1] 0.81 0.26 0.36 0.25 0.53
```

- The function `signif` rounds its argument to a specified number of significant digits

```
> signif(runif(5), 2)

[1] 0.10 0.11 0.93 0.95 0.28
```

Formatting

- The **R** function `formatC` provides general formatting of numeric values as strings
- The function is very flexible and provides better control than `format`
- `formatC` has arguments which control whether scientific or standard formatting is used
- It also has control for the number of digits appearing in results

```
> formatC(1/3, format = "f", digits = 4)
[1] "0.3333"
> formatC(1/3, format = "e", digits = 4)
[1] "3.3333e-01"
```

- It provides facilities for printing numbers in European formats or for printing dollar amounts

The `paste` *Function*

- `paste` provides a very flexible way of pasting strings together
- Useful in conjunction with `cat`
- `paste` obeys the vector recycling rule—useful for tasks like creating labels

```
> paste("Var", 1:4)
[1] "Var 1" "Var 2" "Var 3" "Var 4"
```

- `sep=` and `collapse=` arguments control how strings are glued together
- They make it possible to glue all the results into a single string

```
> paste("Var", 1:4, sep = "-", collapse = ":")
[1] "Var-1:Var-2:Var-3:Var-4"
```

LaTeX Tables

- The next slide shows a very simple **S** function which can be used to print tables suitable as input for LaTeX
- This is just a sketch of how a real `latex.table` function could be written

```
> x

  c d
a 1 3
b 2 4

> latex.table(x)

\begin{tabular}[lrr]
  & c & d \\
a & 1 & 3 \\
b & 2 & 4 \\
\end{tabular}
```

Source Code

```
> latex.table <- function(x) {
+     rlabs <- dimnames(x)[[1]]
+     clabs <- dimnames(x)[[2]]
+     fmt <- c("l", rep("r", length(clabs)))
+     cat("\\begin{tabular}[", fmt,
+         "]\n", sep = "")
+     cat("  ", clabs, sep = " & ")
+     cat(" \\\\\n")
+     for (i in 1:nrow(x)) {
+         cat(rlabs[i], x[i, ], sep = " & ")
+         cat(" \\\\\n")
+     }
+     cat("\\end{tabular}\n")
+ }
```

Generalisations

- The `latex.table` function is very basic and can be enhanced in a number of ways
  - The function fails if its argument does not have row and column labels
  - The function should be changed so that the labels are optional, or could be specified on the command line
  - It might be useful to be able to be able to centre or left justify some columns
  - It might be useful to round or format columns differently
- Packages are available for converting **R** output into a form where it may be directly incorporated into a LaTeX file
  - `xtable` will format tables
  - `latex` from the package Hmisc will format output from some **R** modeling functions

Text Processing

- There are a number of functions which allow manipulation of character strings in **R**

`nchar`	computes string lengths
`substr`	extracts substrings
`substring`	extracts substrings
`strsplit`	splits strings
`tolower`	converts to lower case
`toupper`	converts to upper case
`casefold`	character case conversion
`chartr`	carries out character conversions

Substring Examples

```
> substr("abcdef", 2, 4)
[1] "bcd"
> substring("abcdef", 1:6, 1:6)
[1] "a" "b" "c" "d" "e" "f"
> substr(rep("abcdef", 4), 1:4, 4:5)
[1] "abcd" "bcde" "cd"   "de"
> substr("abcdef", 1:4, 4:5)
[1] "abcd"
> substring("abcdef", 1:4, 4:5)
[1] "abcd" "bcde" "cd"   "de"
```

Substring Replacement

```
> x <- "123456789"
> substring(x, 2) <- "aaa"
> x
[1] "1aaa56789"
> x <- "123456789"
> substring(x, 2, 3) <- "aaa"
> x
[1] "1aa456789"
```

Splitting Strings

```
> strsplit("hello there world", " ")

[[1]]
[1] "hello" "there" "world"

> strsplit("hello  there\tworld",
+      " ")

[[1]]
[1] "hello"         ""
[3] "there\tworld"

> strsplit("hello  there\tworld",
+      "[[:space:]]+")

[[1]]
[1] "hello" "there" "world"
```

Pattern Matching

- The `grep` function can be used to carry out pattern matching

```
> x <- "all the king's men"
> x <- unlist(strsplit(x, " "))
> x
[1] "all"    "the"    "king's" "men"
> grep("e", x)
[1] 2 4
> x[grep("e", x)]
[1] "the" "men"
```

Pattern Substitution

```
> x <- "The color of money colors my decision"
> sub("color", "colour", x)

[1] "The colour of money colors my decision"

> gsub("color", "colour", x)

[1] "The colour of money colours my decision"

> x <- "all the king's men"
> x <- unlist(strsplit(x, " "))
> gsub("(.*)", "\\1=\\1", x)

[1] "all=all"         "the=the"
[3] "king's=king's" "men=men"
```

Importing and Exporting Data

Exporting Data to Text Files

- Use `write.table`

```
> args(write.table)
function (x, file = "", append = FALSE, quote = TRUE,
    sep = " ", eol = "\n", na = "NA", dec = ".",
    row.names = TRUE, col.names = TRUE,
    qmethod = c("escape", "double"))
```

- **R** prefers the header line to have no entry for the row names

```
             dist climb   time
Greenmantle   2.5   650 16.083
Carnethy      6.0  2500 48.350
  ...
```

- `col.names=NA` puts an empty entry in the space for the row names
- To transfer to Excel, the usual method is to write a comma-separated file: `write.csv` is a wrapper which selects the appropriate options

Reading Data from Text Files

- Use `read.table`

```
> args(read.table)
function (file, header = FALSE, sep = "", quote = "¨’", dec = ".",
    row.names, col.names, as.is = !stringsAsFac-
tors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors())
```

- The header commonly has entries for the columns only, not for the row labels, so one field is shorter than the remaining lines
- If the file has (possibly empty) header field for the row labels, use

```
read.table("file.dat", header = TRUE, row.names = 1)
```

- Column names (which override the header line) can be given with `col.names`

Reading Data from Text Files

- Files exported from a spreadsheet can have trailing empty fields and their separators omitted. Use `fill=TRUE`
- `read.table` tries to select a suitable class for each variable in the data frame working through `logical`, `integer`, `numeric` and `complex`
- The final choice is `factor`
- `colClasses` and `as.is` provide greater control
- `as.is` suppresses conversion of character vectors to factors
- There are special functions for comma-separated files (`read.csv`) and fixed width format files (`read.fwf`)
- Reading files uses `scan` which reads a character at a time, and can be used directly
- See also `readLines` to read whole lines

Importing Data

- The package foreign allows import of data written in other formats
- Functions include `read.epiinfo`, `read.mtp`, `read.xport`, `read.S`, `read.spss` and `read.systat` for EpiInfo, Minitab portable, SAS Transport, S-PLUS, SPSS, Stata and Systat files
- A very useful option is to read data from a relational database
- Several packages support database access such as RMySQL, ROracle, and RODBC which is quite generic
- A later section of the course introduces the database mySQL and shows how to use it in conjunction with **R**

Connections

- Connections replace use of filenames with a flexible interface to file-like objects
- For example `gzfile` will read and write from `gzip` files
- `url` will read from URLs of types `http://`, `ftp://`, and `file://`

Excel Spreadsheets

- Best approach is to export the data in tab-delimited or comma-separated form
- You can cut and paste from the display of a spreadsheet with `read.table(file("clipboard"), header = TRUE, sep = "\t", dec = ".")`
- You can also read the clipboard with `readClipboard`
- On Windows the function `odbcConnectExcel` in the package RODBC can select rows and columns from sheets in an Excel spreadsheet file

Data

- There are data sets with **R** in the package `datasets`
- See a list with `data()`
- Load a data set with `data(infert)`
- Find data or load data from other packages

```
data(package="HyperbolicDist")
data(resistors,package="HyperbolicDist")
```

- Edit data or create a data frame in a spreadsheet format with `edit`

```
data(mamquam,package="HyperbolicDist")
mamquamNew <- edit(mamquam)
new.df <- edit(data.frame())
```

- Some statisticians advise against this usage since an *ad hoc* change to the data like this is not able to be done using batch processing and is thus not reproducible

Programming in S

Programming Concepts

- We write programs to solve problems, e.g. to simulate a queue
- A **program** is an organized list of instructions that, when executed, causes the computer to behave in a predetermined manner
- A program is like a recipe. It contains
  - a list of ingredients—called variables; and
  - a list of directions called statements or commands
- Directions tell the computer what to do with the variables
- Variables can represent numeric data, text, or graphical images

Programming Concepts

- A **programming language** is a vocabulary and a set of grammatical rules for instructing a computer to perform specific tasks. Examples are **Matlab**, **R** or **C**
- A **procedure** is a section of a program that performs a specific task, or a sequence of instructions for performing some action
- A **procedure** is called an **algorithm** if it is finite, definite, and effective, with some output
- Computer programs should be algorithms, but there are many procedures not related to computers—knitting patterns, recipes, choreography

Algorithms

- An **algorithm** if it is finite, definite, and effective, with some output
  - finite means it ends after a finite number of steps
  - definite means each step is clearly defined—”beat until fluffy” in a recipe doesn’t satisfy this criterion
  - effective means able to be carried out
  - output is required otherwise the result of implementing the algorithm will remain unknown.
- To solve a problem, the clearest, most reliable way is to first develop a suitable algorithm, then to program or code the algorithm in a suitable programming language.

Representing Algorithms

- How is the algorithm to be specified, if not in a programming language?
  - in natural language—English
  - by using a flowchart—using diagrams is often helpful
  - by writing pseudocode—this is recommended by computer scientists and software developers

Pseudocode

- **Pseudocode** is an outline of a program, written in a form that can easily be converted into real programming statements
- Enables the programmer to concentrate on the algorithms without worrying about the syntactical details of the programming language
- Can be made progressively more detailed as the algorithm is developed
- Can be used as comments for the final program

Flow of Control

- To be useful, a programming language needs to have a sufficiently rich collection of **control structures**
- **Control structures** determine the order in which programming instructions are executed, called **flow of control**
- Programming languages typically allow for
  - sequence
  - alternation or selection
  - iteration, repetition or looping
  - jumping
  - interrupting

Flow of Control

- **sequence**—this is the default order, instructions being executed in the order in the order in which they appear
- **alternation** or **selection**—implemented as `if ...then ...else ...` or `case` or `switch`
- **iteration, repetition** or **looping**—implemented as `while` or `for` or `repeat`
- **jumping**—implemented as `goto` or `break` or `stop`
- **interrupting**—used for mouse operations for example

Flow of Control

- Important **R** commands, or **control structures** which affect the flow of control are:
  - `if` together with `else` executes one group of **R** commands if some logical expression is true, and another set if it is not true
  - `for` executes a group of **R** commands for a particular set of values of some index
  - `while` executes a group of **R** commands an indefinite number of times while some expression is true.
  - `switch` is used to choose which of a number of different groups of code will be executed.

Compound Expressions

- Compound expressions give a way of treating a sequence of expressions as a single expression
- The general form of a compound expression is:

  $\{ \ \mathit{expression}_1 \ ; \ \ldots \ ; \ \mathit{expression}_n \ \}$
- A compound expression is evaluated by evaluating each of its component expressions in turn and taking the value of the last one as the value of the compound
- Note that the statements here are separated by semi-colons, but newlines will also serve as separators

`if, else`

- The syntax is

```
if (expr_1)
     expr_2
else
     expr_3
```

- Here *expr_1* must evaluate to a logical value
- The other expressions may be a collection of **R** statements enclosed in curly brackets $\{, \}$
- Indentation is used to highlight flow of control

Example of `if, else`

```
# Solving a quadratic equation
# Specify coefficients a, b, c = const
Delta <- b^2 - 4*a*const    # the discriminant
if ( Delta < 0){
  soln <- c(NA,NA)
  cat("No real solutions to this quadratic equation\n")
}else{
  soln <- c(-b + sqrt(Delta), -b - sqrt(Delta))/(2*a)
}
soln
```

- Informative names should be used
- Some functions have one-letter names: `c, t, q`
- `x, y, z` often used for data, `i, j, k` for counters

Example of `if , else`

For $a = 1, \quad b = 2, \quad c = 1$

```
> a <- 1; b <- 2; const <- 1
> Delta <- b^2 - 4*a*const    # the discriminant
> if ( Delta < 0){
+   soln <- c(NA,NA)
+   cat("No real solutions to this quadratic equation\n")
+ }else{
+   soln <- c(-b + sqrt(Delta), -b - sqrt(Delta))/(2*a)
+ }
> soln
[1] -1 -1
```

Example of `if`, `else`

For $a = 1, \quad b = 2, \quad c = 4$

```
> a <- 1; b <- 2; const <- 4
> Delta <- b^2 - 4*a*const    # the discriminant
> if ( Delta < 0){
+   soln <- c(NA,NA)
+   cat("No real solutions to this quadratic equation\n")
+ }else{
+   soln <- c(-b + sqrt(Delta), -b - sqrt(Delta))/(2*a)
+ }
No real solutions to this quadratic equation
> soln
[1] NA NA
```

Example of `if`, `else`

For $a = 1, \quad b = 2, \quad c = -4$

```
> a <- 1; b <- 2; const <- -4
> Delta <- b^2 - 4*a*const    # the discriminant
> if ( Delta < 0){
+   soln <- c(NA,NA)
+   cat("No real solutions to this quadratic equation\n")
+ }else{
+   soln <- c(-b + sqrt(Delta), -b - sqrt(Delta))/(2*a)
+ }
> soln
[1]  1.236068 -3.236068
```

`ifelse`

- Used for elementwise change to a vector
- The syntax is

$$\boxed{\texttt{ifelse}\,(test,\, yes,\, no)}$$

- Here $test$ is a logical vector
- yes gives the value to be returned if the test value is true
- no gives the value to be returned if the test value is false

```
> x <- c(3:-3)
> sqrt(x)                                  # gives warning
[1] 1.732051 1.414214 1.000000 0.000000      NaN      NaN      NaN
Warning message:
NaNs produced in: sqrt(x)
> sqrt(ifelse(x >= 0, x, NA))        # no warning
[1] 1.732051 1.414214 1.000000 0.000000       NA       NA       NA
```

Relations and Operators

- Relations are tests, or comparisons, which are either ’TRUE’ (= 1), or ’FALSE’ (= 0)

Relational Operator	Comparison
`<`	less than
`>`	greater than
`<=`	less than or equal
`>=`	greater than or equal
`==`	equal
`!=`	not equal

- Note that "==" is used in *relation* statements

Relations and Operators

- Relation statements may be combined using logical operators `&`, `|` and `!`

Relational Operator	Resulting Comparison
`&`	logical *and*
`|`	logical *or*
`!`	logical *not*

- For example `(3 != 5)|(2 > 4)`, is `(TRUE) | (FALSE)` and is thus `TRUE`, so **R** will have the value 1
- Use brackets to make sure the relation specified is the one intended

`for`

- The syntax is

```
for (name in expr_1)
    expr_2
```

- The index *name* starts out equal to the first element of *expr_1*
- Each time the `for`-loop is executed, *name* is set equal to the next entry in *expr_2*, etc.

Example of Use of `for`

```
# Calculate and print the first 6 elements
# of the Fibonacci series
fibonacci <- numeric(6)
fibonacci[1] <- 1
cat("The 1st Fibonacci number is: ", fibonacci[1], "\n")
fibonacci[2] <- 1
cat("The 2nd Fibonacci number is: ", fibonacci[2], "\n")
for (i in 3:6){
  fibonacci[i] <- fibonacci[i - 1] + fibonacci[i - 2]
  cat("The ",i,"th Fibonacci number is: ", fibonacci[i], "\n'
}
```

Example of Use of `for`

```
> fibonacci <- numeric(6)
> fibonacci[1] <- 1
> cat("The 1st Fibonacci number is: ", fibonacci[1], "\n")
The 1st Fibonacci number is:  1
> fibonacci[2] <- 1
> cat("The 2nd Fibonacci number is: ", fibonacci[2], "\n")
The 2nd Fibonacci number is:  1
> for (i in 3:6){
+   fibonacci[i] <- fibonacci[i - 1] + fibonacci[i - 2]
+   cat("The ", i, "th Fibonacci number is: ", fibonacci[i],
+ }
The  3 th Fibonacci number is:  2
The  4 th Fibonacci number is:  3
The  5 th Fibonacci number is:  5
The  6 th Fibonacci number is:  8
```

`while`

- The syntax is

```
while (condition)
    expr
```

- *expr* is executed while *condition* remains true
- Initialisation is usually required prior to entering the loop, and some change which will affect the logical expression within the loop

Example of Use of `while`

```
# Toss a die until a 6 is obtained
toss <- ceiling(6*runif(1))  # ceiling function
count <- 1
while (toss != 6){
    toss <- ceiling(6*runif(1))
    cat('Toss ', count, ' was a ', toss, '\n')
    count <- count + 1
}
cat('There were ', count-2, ' tosses before the first 6\n')
```

Example of Use of `while`

```
> toss <- ceiling(6*runif(1))  # ceiling function
> count <- 1
> while (toss != 6){
+     toss <- ceiling(6*runif(1))
+     cat('Toss ', count, ' was a ', toss, '\n')
+     count <- count + 1
+ }
Toss  1  was a  1
Toss  2  was a  5
Toss  3  was a  6
> cat('There were ', count-2, ' tosses before the first 6\n')
There were  2  tosses before the first 6
```

Functions

- New functionality is added to **S** by defining new *functions*
- As a very simple example, define a function which squares its argument as follows:

  ```
  > square <- function(x) x * x
  ```

- The expression `function(x) x * x` creates a function which is assigned as the value of the variable `square`
- The function has a single argument x and the value is multiplied by itself to provide the value of the function
- We can use this function in exactly the same way as any other **S** function

  ```
  > square(10)
  [1] 100
  ```

Vectorisation

- Because the operation $*$ acts element-wise on vectors, the new square function will also

```
> square(1:10)
 [1]   1   4   9  16  25  36  49  64  81
[10] 100
```

- Using this fact we can write a simple sum-of-squares function

```
> sumsq <- function(x) sum(square(x))
> sumsq(1:10)
[1] 385
```

General Functions

- In general, an **S** function has the form:

  `function (` *arglist* `)` *body*

  where *arglist* is a (comma separated) list of variable names known as the *formal arguments* of the function, and *body* is a simple or compound expression known as the body of the function
- The general rule for evaluating a call to a function is to temporarily create a set of variables by associating the arguments passed to the function with the variable names in *arglist*, and then to use these variable definitions to evaluate the function body

Example

- Consider the function defined by:

  ```
  > hypot <- function(a, b) {
  +     sqrt(a^2 + b^2)
  + }
  ```

  and suppose we make a call to this function by typing:

  ```
  > hypot(3, 4)
  [1] 5
  ```

- This function call is evaluated as follows:
  - Temporary variables, `a` and `b`, are created with the values 3 and 4.
  - These variable definitions are used to evaluate the expression `sqrt(a^2 + b^2)` to obtain the value `5`.
  - When the evaluation is complete the temporary definitions of `a` and `b` are removed.

Optional Arguments

- **S** has a notion of default argument values which makes it possible for the writer of a function to specify reasonable default values for arguments, while still providing the flexibility users the option of overriding these defaults
- As an example, consider the following sum-of-squares function

```
> sumsq <- function(x, about = 0) {
+      sum((x - about)^2)
+ }
```

- The function definition provides a default definition for the `about` argument

Example

- When invoked with just a single argument the function returns the sum of the squared values in that argument

```
> sumsq(1:10)
[1] 385
```

- When provided with a value for the `about` argument, the function computes the sum of squared deviations about the specified value

```
> sumsq(1:10, mean(1:10))
[1] 82.5
```

Argument Matching

- Because it is not necessary to specify all the arguments to **S** functions, it is important to be clear about which argument corresponds to which formal parameter of the function
- This can be done by providing a name for the argument explicitly.

```
> sumsq(1:10, about = mean(1:10))
[1] 82.5
```

- When names are provided for arguments, they are used in preference to position which matching up formal arguments and arguments
- For example

```
> sumsq(about = mean(1:10), 1:10)
[1] 82.5
```

  returns the same answer as the function call above

The Argument Matching Process

- The general rule for matching formal and actual arguments is as follows.
  1. Use any names provided with the actual arguments to determine the formal arguments associated with the named arguments. Partial matches are acceptable, unless there is an ambiguity.
  2. Match the unused actual arguments, in the order given, to any unmatched formal arguments, in the order they appear in the function declaration.

Argument Matching Example

- If `sumsq` is defined by

```
> sumsq <- function(x, about = 0) {
+       sum((x - about)^2)
+ }
```

then all the following calls to `sumsq` are equivalent.

```
> sumsq(1:10, mean(1:10))
> sumsq(1:10, about=mean(1:10))
> sumsq(1:10, a=mean(1:10))
> sumsq(x=1:10, mean(1:10))
> sumsq(mean(1:10), x=1:10)
```

Dates and Times

Dates and Times

- Dates and times are needed for time series, survival analysis etc
- Problems: time and date formats, time zones, daylight saving
- Classes Date, chron, POSIX (classes: POSIXct, and POSIXlt)
- Date is only dates, no times, chron is a bit old, POSIX classes are comprehensive but complicated
- Tasks
  - read dates and times in different formats: `as.Date`, `as.POSIXct`, `as.POSIXlt`, `strptime`
  - print dates and times in different formats: `format.Date`, `strftime`
  - operate on dates and times: normal functions such as `mean`, `cut`, `-`, plus `difftime`

Dates and Times

- Origin
  - Date: days since January 1, 1970
  - chron: days since January 1, 1970, fractions of days give times within days
  - POSIXt: number of seconds since the beginning of 1970 in the GMT timezone (includes 22 leap seconds, see the object `.leap.seconds`)
- Date has no times, chron has times and dates, no timezones, POSIXt has times, dates and timezones
- Suggestion: use the simplest possible class which supports your needs
- Time zones are very tricky

Dates and Times

- To read a date, use `as.Date`, `as.chron`, `as.POSIXct`, with optional format
- Formats are provided by `format`, and for POSIXct by `strptime`: see `?strptime` for these
- Check what has been read in using `unclass` and `as.character`
- POSIXt is a super class containing POSIXct and POSIXlt
- POSIXct is a number of seconds
- POSIXlt is the number of seconds broken down into a list of 9 items: year, month, day of the month, hour, minute, second and a daylight saving flag

Miscellanea

Scope

- A major difference between **S-PLUS** and **R**
- Symbols in the body of a function are:
  - formal parameters: occurring in the argument list, values determined by binding actual function arguments to formal parameters
  - local variables: values determined from expressions in the function body
  - other variables called free variables: become local if values are assigned to them

```
f <- function(x){
  y <- 2*x
  print(x)
  print(y)
  print(z)
}
```

- `x` is a formal parameter, `y` a local variable, `z` a free variable

Scope

- In **R** the free variable bindings are resolved by looking in the environment in which the function was created: called *lexical scope*
- Consider a function `cube`

```
cube <- function(n){
  sq <- function() n*n
  n*sq()
}
```

- Then `cube(2)` in **R** gives the answer 8.
- Useful for maximum likelihood determination using `optim()` or `nlm()` (see later).

Customizing the Environment

- **R** can be invoked with a number of options: the following assumes that it is started with just the defaults
- For details, check the manual *An Introduction to R* `http://cran.stat.auckland.ac.nz/doc/manuals/R-intro.pdf` or use `help(Startup)`
- Note that this area has changed recently and there may be differences between versions of **R**
- There are also differences between operating systems
- Customisation is possible at the site level, the user level, and the directory level
- Note that **R** is installed as a directory tree at a particular location in the file system which is referred to as R_HOME and may be pointed to by the environment variable $R_HOME

Customizing the Environment

- The startup routine is basically as follows:
  - The site file is read—either as pointed to by R_ENVIRON or at `R_HOME/etc/Renviron.site`
  - The site-wide startup profile is read—either as pointed to by R_PROFILE or at `R_HOME/etc/Rprofile.site`
  - **R** searches for `.Rprofile` in the current directory or in the user's home directory (in that order) and sources it
  - It loads a saved image from `.RData` if there is one
  - It executes a `.First` function if there is one. This function can be specified in the startup profiles, or in `.RData`

Installing Packages

- Details regarding installation are given in the manual *R Installation and Administration*

  `http://cran.stat.auckland.ac.nz/doc/manuals/R-admin.pdf`
- Note that this area has changed recently and there may be differences between versions of **R**
- There are also differences between operating systems
- Packages are installed in libraries
- **R** comes with a single library `R_HOME/library` which contains the standard and recommended packages
- The location of the main library is given in the character string `.Library`

```
> .Library
[1] "/usr/local/lib/R/library"
```

Installing Packages

- First of all you must create a directory where packages will be installed: suppose it is called RLIBS
- Then you must set the environment variable R_LIBS to point to that directory: In bash use

  ```
  export R_LIBS=RLIBS
  ```

- Include this in your `.bashrc` so that your R_LIBS will be automatically set when you start a shell
- Now start up **R**. To install the package xtable, use

  ```
  install.packages("xtable",repos="http:/cran.stat.auckland.ac.nz")
  ```

- Note that this will install from the Statistics Department's local CRAN mirror site so will not incur any costs for internet use
- From 2.5.0, **R** can have a site-specific libraries and user libraries

Mathematical Annotation

- Pass an *expression* rather than a character string to `text`, `mtext`, `axis`, or `title`
- Will give Greek and other symbols, super and subscripts
- Like a poor man's TEX
- In an expression
  - `alpha`, `beta` etc. will give the corresponding Greek letter
  - Superscripts obtained using ˆ
  - Subscripts obtained using square brackets [ ]
- See the `plotmath` function

Probability Distributions

- For many distributions there are functions to calculate the density function, cumulative distribution function, quantile function, and to generate random observations
- The function name is comprised of the abbreviated distribution name and a prefix
- The prefixes are `d`, `p`, `q`, and `r`
- For example for the Poisson we have `dpois`, `ppois`, `qpois`, and `rpois`

Optimisation

- This is a common task: e.g. for maximum likelihood
- Two functions `optim()` and `nlm()`
- `optim()` has a number of optimisation methods, can use derivatives if available

An Extended Example: the Weibull Distribution

Weibull example from Devore (2000)

Let $X_1, \ldots, X_n$ be a random sample from a Weibull pdf

$$f(x;\alpha,\beta) = \begin{cases} \frac{\alpha}{\beta^\alpha} x^{\alpha-1} e^{-(x/\beta)^\alpha} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Writing the likelihood and ln(likelihood), then setting both $(\partial/\partial\alpha)[\ln(f)] = 0$ and $(\partial/\partial\beta)[\ln(f)] = 0$ yields the equations

$$\alpha = \left[\frac{\sum x_i^\alpha \ln(x_i)}{\sum x_i^\alpha} - \frac{\sum \ln(x_i)}{n}\right]^{-1} \qquad \beta = \left(\frac{\sum x_i^\alpha}{n}\right)^{1/\alpha}$$

These two equations cannot be solved explicitly to give general formulas for the mle's $\widehat{\alpha}$ and $\widehat{\beta}$. Instead, for each sample $x_1, \ldots, x_n$, the equations must be solved using an iterative numerical procedure.

Weibull example in R

- Regard the problem of calculating MLEs as an optimization problem. Usually it is easier to optimize the log-likelihood rather than the likelihood itself. The parameter values that optimize the log-likelihood are also the MLEs.
- Probability density functions (probability functions for discrete distributions) have names starting with d — dnorm, dbinom, dunif, dweibull.
- All density functions take an optional argument log which, when TRUE causes evaluation of the log-density.
- The recipe becomes:

  Regard the parameters as the variables and sum the log density for your distribution using your data.

An R session on the Weibull example

```
> library(Devore5)
> data(xmp04.30)     # import the data
> str(xmp04.30)      # examine the structure
'data.frame':   10 obs. of  1 variable:
 $ lifetime: num   282  501  741  851 1072 ...

> # Reasonable starting estimates are shape = 1, scale = 1000
> # Do a simple evaluation at this set of parameters
> sum(dweibull(xmp04.30$lifetime, shape=1, scale=1000, log=TRUE))
[1] -80.47655

> # Optimization functions minimize so use negative log-likelihood
> llfunc <- function(x) {   # express as a function
+  -sum(dweibull(xmp04.30$lifetime, shape=x[1], scale=x[2], log=TRUE))
+ }
> mle <- nlm(llfunc, c(shape = 1, scale = 1000), hessian = TRUE)
```

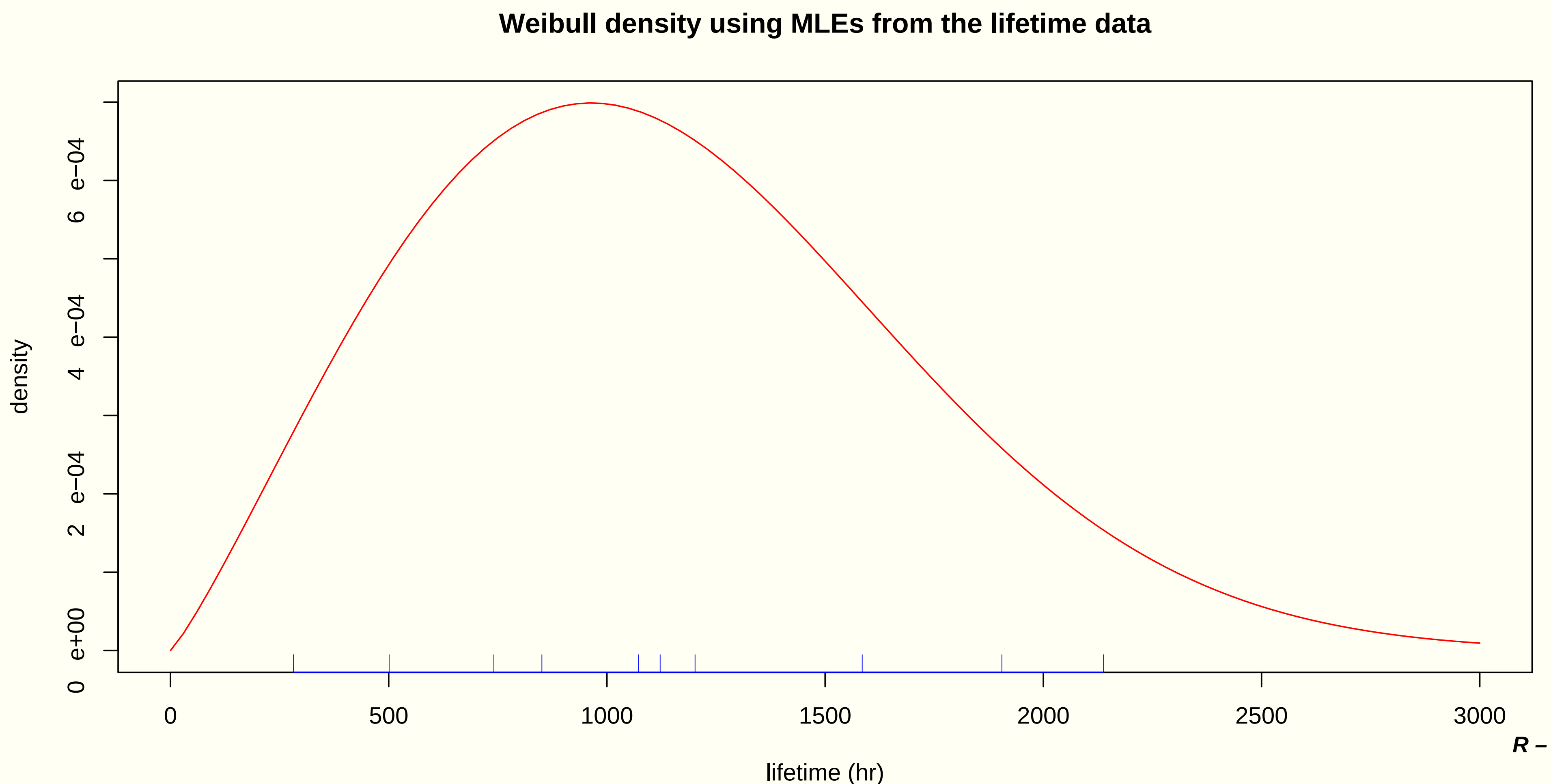
Results of the Weibull example

```
> str(mle)   # structure of the returned value
List of 6
 $ minimum   : num 77.1
 $ estimate  : num [1:2]    2.15 1289.34
 $ gradient  : num [1:2] -1.50e-05  2.98e-08
 $ hessian   : num [1:2, 1:2]  3.75e+00 -3.20e-03 -3.20e-03  2.78e-05
 $ code      : int 1
 $ iterations: int 20
> solve(mle$hessian)   # approximate variance-covariance matrix
           [,1]        [,2]
[1,]  0.2959642    34.06422
[2,] 34.0642200 39835.81998
```

We see that the maximum of the log-likelihood is -77.1, achieved at $\widehat{\alpha} = 2.15$ and $\widehat{\beta} = 1289.34$. The approximate standard errors of the estimates are $0.544 = \sqrt{0.29596}$ and $199.6 = \sqrt{39835.82}$. We can use the standard errors to determine a grid of (α, β) values for contouring the log-likelihood function.

Plotting the density at the estimates

```
> plot(function(x) dweibull(x, shape = 2.15, scale = 1289.34), 0, 3000,
+       col = "red", xlab = "lifetime (hr)", ylab = "density",
+       main = "Weibull density using MLEs from the lifetime data")
> rug(xmp04.30$lifetime, col = "blue")
```

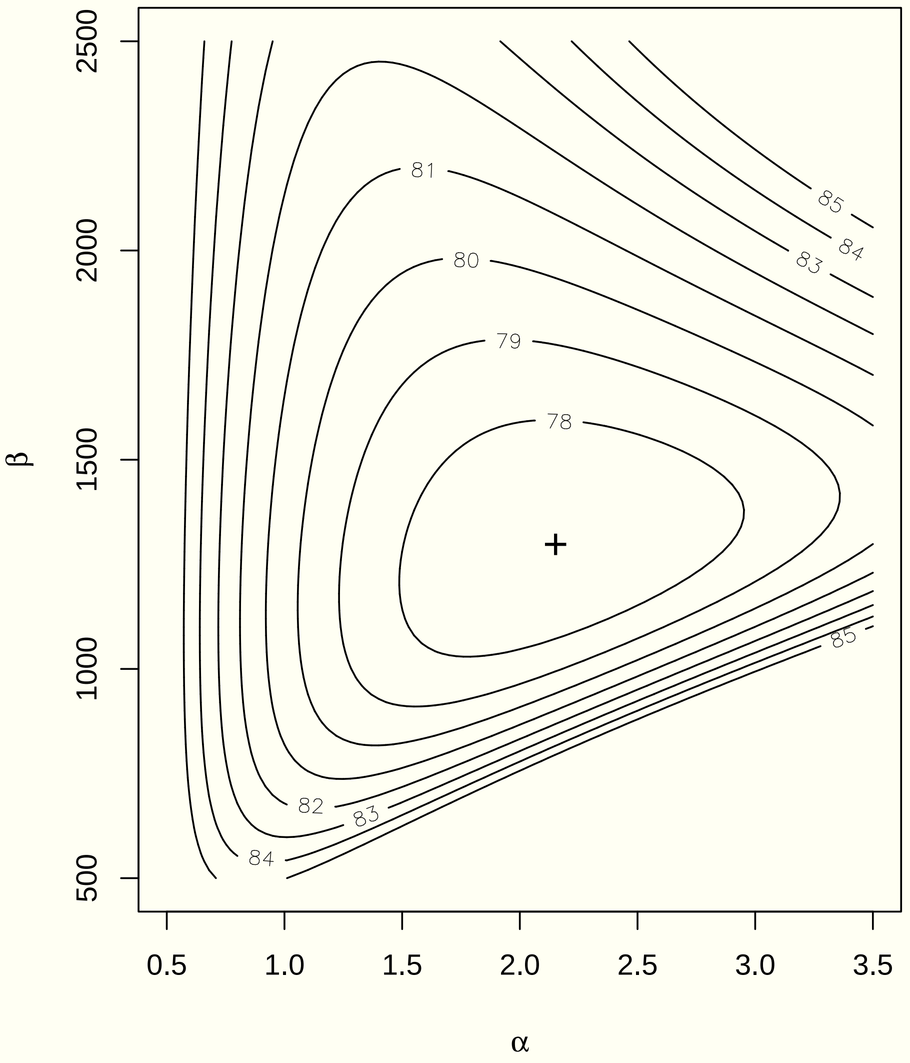


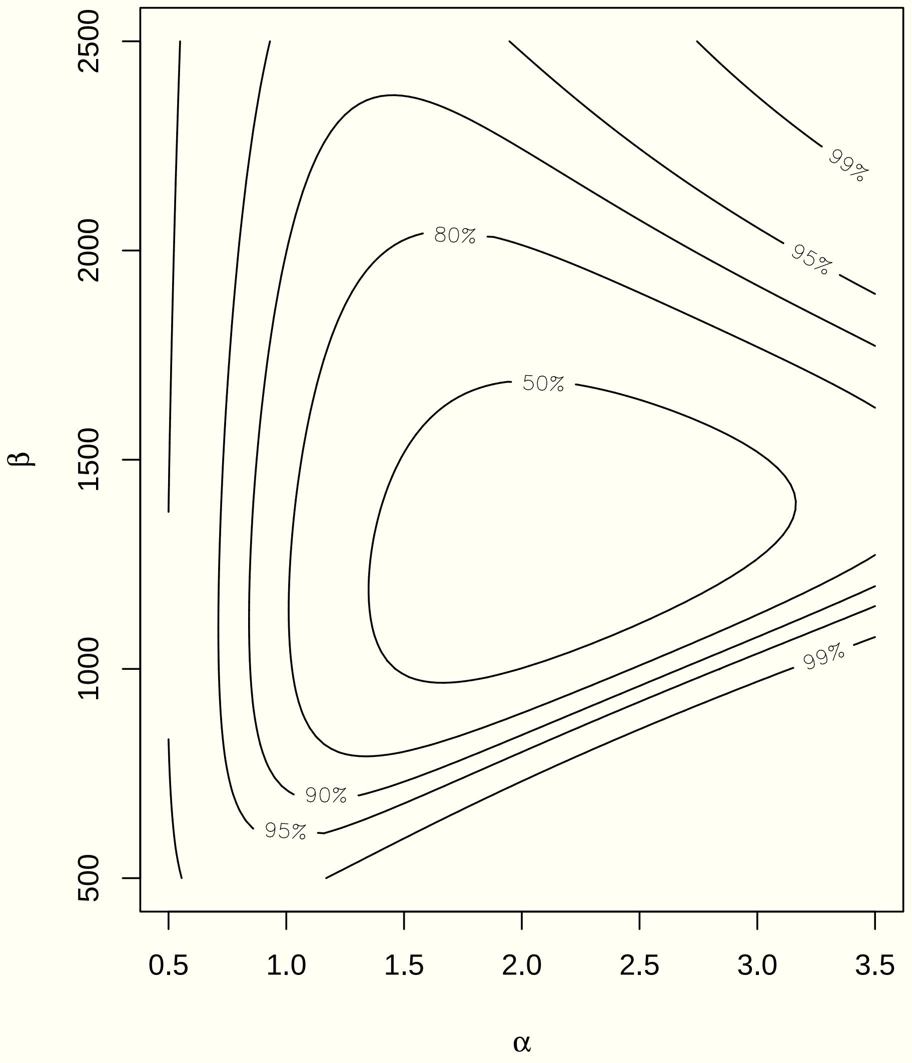
Contouring the log-likelihood function

```
> grid <- matrix(0.0, nrow = 101, ncol = 101)
> scvals <- seq(0.5, 3.5, len = 101)  # scale parameter
> shvals <- seq(500, 2500, len = 101) # shape parameter
> for (i in seq(along = scvals)) {
+    for (j in seq(along = shvals)) {
+        grid[i,j] <- llfunc(c(scvals[i], shvals[j]))
+    }
+ }
> contour(scvals, shvals, grid, levels = 77:85)
> points(mle$estimate[1], mle$estimate[2], pch = "+", cex = 1.5)
> title(xlab = expression(alpha), ylab = expression(beta))

> # Or use levels calculated from the chi-square distribution
> contour(scvals, shvals, grid,
+   levels = mle$min + qchisq(c(0.5,0.8,0.9,0.95,0.99), 2),
+   labels = paste(c(50,80,90,95,99), "%", sep = ""))
```

Log-likelihood contours - Weibull





Lessons from the Weibull example

- The likelihood function is the same as the probability density but with the parameters varying and the data fixed.
- For a random sample, the log-likelihood is

```
sum(d<distname>(<data>, par1, par2, ..., log = TRUE))
```

- We minimize the negative of the log-likelihood

```
llfunc <- function(x)
       -sum(d<distname>(<data>, par1 = x[1], ..., log = TRUE))
mle <- nlm(llfunc, <starting estimates>, hessian = TRUE)
```

- The inverse of the hessian provides an estimate of the variance-covariance matrix.
- For two-parameter models we can evaluate a grid of log-likelihood values and get contours.
- Standard errors from the inverse hessian are not always realistic indications of the variability in the parameter estimates.

S Graphics

Ross Ihaka on Graphics

Ross has a set of slides on graphics which I will use at this point

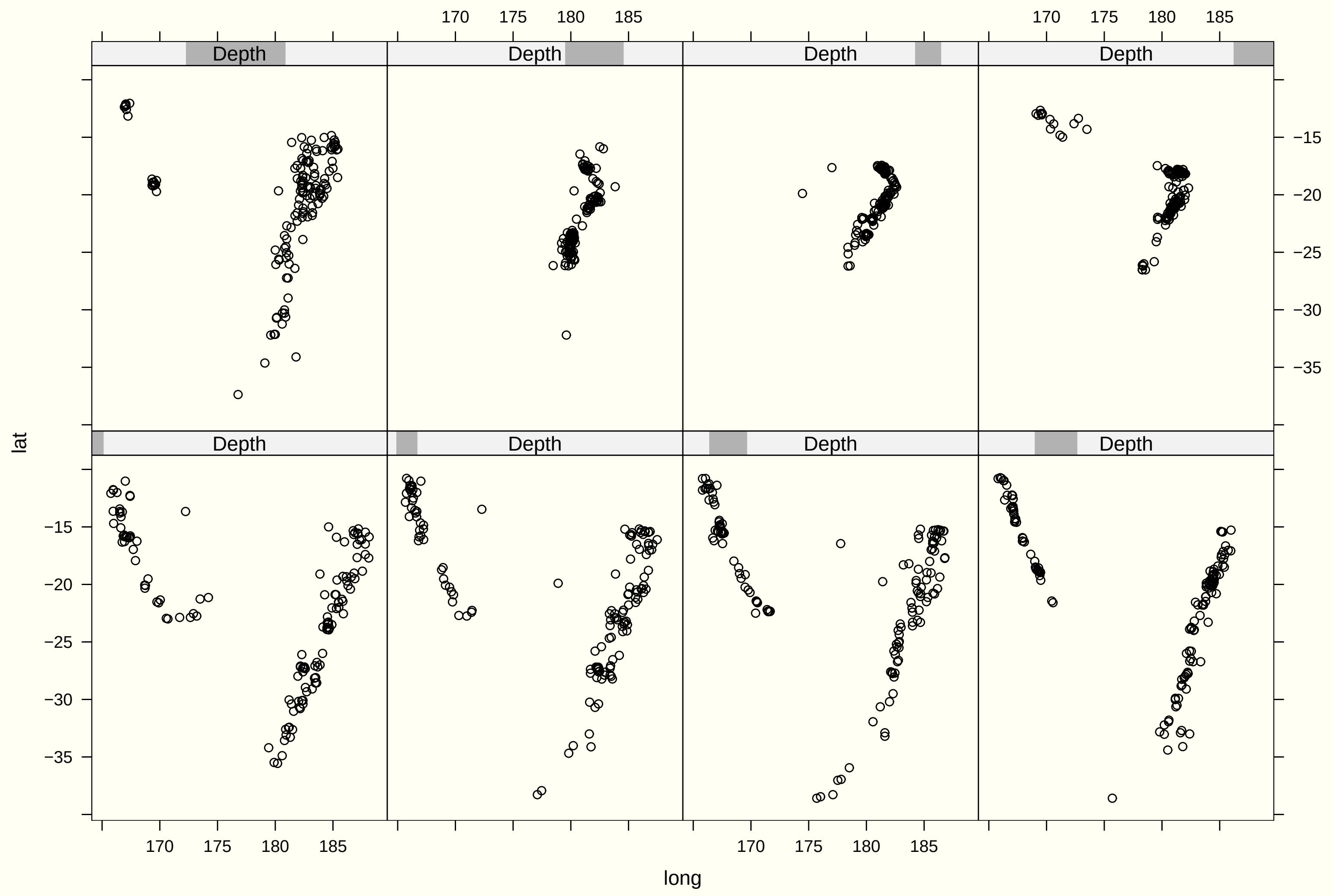
Trellis Graphics with lattice

Trellis Graphics

- A new form of graphical display used especially for grouped data
- Trellis displays are plots with one or more panel arranged on a regular grid-like structure
- Panels are the same type of graph for a subset of the data
- Subsets are chosen by conditioning on continuous or discrete variables in the data
- This account draws **very** heavily on *A Tour of Trellis Graphics* by Richard A. Becker, William S. Cleveland, and Ming-Jen Shyu, and includes verbatim quoting from that document.
- Cleveland is probably the driving force behind Trellis Graphics
- In **R** Trellis Graphics are implemented using the package `lattice`, which in turn is based on `grid`, a new graphical engine.

Trellis Graphics Example

- Example showing the latitude and longitude at different depths for earthquakes near Fiji. See `?quakes`.



Trellis Graphics

- Graphics is a strong point of **R** and the **S** language
- Standard graphics consist of high-level plotting functions (`boxplot, hist, qqplot`) and low-level functions to augment existing plots `lines, segments, points, text`)
- It is possible to produce multiple plots using `mfrow, mfcol` and `layout`
- A coordinated set of plots with control over aspect ratios and axes is more difficult
- Trellis provides a unifying framework for multipanel displays based on *conditioning*
- Also excellent for single panel displays
- Executing a trellis expression produces a trellis object which is usually displayed (unless assigned a name or used in further calculations)

Example

- A simple trellis plot can be produced as follows

```
xyplot(NOx ~ E, data=ethanol)
```

- The `ethanol` data set relates the production of nitrous oxide (NOx) to the equivalence ratio (a measure of richness of the fuel/air mixture)
- First argument is a formula
- A modification produces a number of similar plots conditional on the compression ratio (`C`) which has 5 values

```
xyplot(NOx ~ E|C, data=ethanol)
```

- We can change the layout of the panels

```
xyplot(NOx ~ E|C, data=ethanol,layout=c(2,3,1))
```

  produces 2 columns and 3 rows on 1 page

Example

- If we try and condition on a variable which takes a lot of values we get a lot of very small uninformative graphs
- We can condition instead on intervals from the range of a variable
- `equal.count` constructs *shingles* from the data with a specified number of intervals, and a specified overlap between successive intervals

  ```
  EE <- equal.count(ethanol$E, number = 9,overlap = 1/4)
  ```
- We can then plot against `EE`

  ```
  xyplot(NOx ~ C|EE, data=ethanol,aspect=2)
  ```
- `subset` allows restriction of the plots to part of the data

  ```
  xyplot(NOx ~ C|EE, data=ethanol, subset=C > 8)
  ```

Example

- `barley` data gives yield of barley for 6 sites for 2 years

```
dotplot(variety ~ yield| year*site,
        data = barley,
        xlab = "Barley Yield (bushels/acre)")
```

Panel Functions

- For standard graphs, panel functions are provided: `panel.xyplot, panel.dotplot, panel.histogram`
- Panels can also be constructed specially, for example by combining two panels
- For example, here is a plot which has a loess smooth added

```
xyplot(NOx ~E|C, data = ethanol,
       panel = function(x,y)
               panel.xyplot(x,y)
               panel.loess(x,y)

       )
```

Practicalities

- In a `for` loop, printing is suppressed to avoid excessive output, explicit `print` or `cat` statements must be used
- Likewise in a `for` loop trellis plots are not produced unless an explicit `print` command is used
- The colors for trellis plots have been optimised for screen display
- If you intend to include a trellis plot in a printed document it is recommended that other colours be used
- The usual way to do this is via

```
trellis.par.set(theme = col.whitebg())
```

- This is an example of setting parameters for trellis plots which differs from ordinary graphics
- See `?trellis.par.set` or `?trellis.par.get`

Sweave

Sweave

- Traditionally, to produce a report of a statistical analysis would require the following steps:
  - Analyse the data, obtaining summaries, tables, graphs etc
  - Write the report in LaTeX copying the numeric results into the document, and using `\includegraphics` to incorporate the graphs or figures
- This process might be made easier by using `xtable` to format tabular output to make it suitable for LaTeX
- Sweave enables reports to be prepared using a single document which contains the **R** code to carry out the analysis, and the LaTeX formatting to produce the report

Sweave

- The source file containing the R code and LaTeX formatting is run through **R** first which results in all the data analysis output being including in the resulting LaTeX file
- This file is then processed by LaTeX or pdfLaTeX in the usual manner to produce the final formatted report
- There are many advantages:
  - The report can be readily updated, if for example new data is obtained
  - Regular reports can be easily run each month say, just by supplying new dates and new data
  - Research is readily reproducible with all the material required in a single file
  - The tedious and error-prone procedure of transferring output from **R** into a LaTeX document is avoided

Sweave

- Sweave source files are noweb files with some additional syntax to allow control over the final output
- The usual extension for them is `.Rnw`
- Nonweb is a programming tool which allows program source code and documentation to be included in a single file
- Noweb files consist of segments of code and of documentation called chunks
- Different programs are used to extract the code (“*tangle*”) or typeset documentation together with the code (“*weave*”)
- `<<...>>=` at the start of a line marks the start of a code chunk
- `@` at the start of a line marks the start of a documentation chunk

Sweave

- In the noweb syntax shown above, options can be included in the angled brackets to either show or not show the code, and to show or not show the results
- By default both code and results are shown
- To hide the code, use `echo=false`, to hide results, use `results=hide`
- Then both are hidden using `<<echo=false,results=hide>>`
- To include an S object in the text (as opposed to in displayed text), use `Sexpr{}`

Sweave

- As an alternative to the noweb sysntax described above for chunks of code, LaTeX syntax can be used
- Here is an example from the article by Friedrich Leisch who is responsible for Sweave

```
\begin{Scode}{echo=false,results=hide}
library(lattice)
library(xtable)
data(cats, package="MASS")
\end{Scode}
```

- XEmacs provides support for `.Rnw` files: most importantly, to include a code chunk, use M-n i

Processing Sweave Documents

- In **R** use

```
library(tools)
Sweave("sweavefile.Rnw")
```

- Run LaTeX on the resulting document
- It is possible to use a Makefile also which processes the document as a batch job submitted to **R** and then runs LaTeX

```
sweave: $(FILENAME).Rnw
        echo "library(tools); Sweave(\"$(FILENAME).Rnw\")" |
             R --no-save --no-restore --slave
        latex $(FILENAME)
        dvips -o $(FILENAME).ps $(FILENAME).dvi
        ps2pdf $(FILENAME).ps $(FILENAME).pdf
```

- Note that the line beginning with `echo` should not be broken after the pipe symbol in the actual file

Writing R Functions and Packages

Package Structure

- Packages provide a set of functions along with documentation which can extend the capability of **R**.
- Typically the functions provided form a coherent set of procedures: for example HyperbolicDist provides a set of functions which dealt with the hyperbolic distribution
- A package consists of a subdirectory containing a file called `’DESCRIPTION’` and the sub-directories `’data’`, `’demo’`, `’exec’`, `’inst’` `’man’`, `’po’`, `’src’`, `and` `’tests’` not all of which need be present.
- The subdirectory may also contain files `’INDEX’`, `’NAMESPACE’`, `’configure’`, `’cleanup’` and `’COPYING’`
- Files like `’README’`, `’NEWS’`, and `’ChangeLog’` are ignored by **R** but are possibly useful to users

Files

- Files 'configure' and 'cleanup' are Bourne shell script files executed before and after installation on Unix
- 'COPYING' contains a copy of the license to the package: should be just a reference to copies elsewhere (in share/licenses')
- The subdirectory name should be the same as the package name
- Names may not include '"','*','/','<','>','?','\' and '|'
- The function `package.skeleton` can be used to create the structure for the new package

The DESCRIPTION ***File***

- This file should have the form

```
Package: pkgname
Version: 0.5-1
Date: 2004-01-01
Title: My first collection of functions
Author: Joe Developer <Joe.Developer@some.domain.net>, with
  contributions from A. User <A.User@whereever.net>.
Maintainer: Joe Developer <Joe.Developer@some.domain.net>
Depends: R (>= 1.8.0), nlme
Suggests: MASS
Description: A short (one paragraph) description of what
  the package does and why it may be useful.
License: GPL version 2 or newer
URL: http://www.r-project.org, http://www.another.url
```

Package Subdirectories

- The `R` subdirectory contains **R** code in one or more files preferably with the extension `.R`
- It should be possible to read in the files using `source()` so **R** objects should be created by assignments
- The `man` subdirectory should contain documentation files for the objects in the package in *R documentation* (Rd) format
- File names must start with a letter and have the extension `.Rd`
- C, C++, and FORTRAN code goes in the `'src'` directory
- Source code is compiled and linked using Make when `R CMD INSTALL` is run

Package Subdirectories

- The `'data'` subdirectory is for data files which can be loaded using `data()`
- Data files can be plain **R** code (with extension `.R`), tables (with extension `'.tab'`, `'.txt'`, or `.csv`), or `save()` images (with extension `'.Rdata'` or `'.rda'`)
- The `'demo'` subdirectory is for **R** scripts, and needs a `'00Index'` file with one line for each demo, giving its name and description
- Subdirectory `'tests'` is for test code

Checking Packages

- `R CMD check` is the package checker. It checks such things as
  - missing cross references and duplicate aliases in help files
  - valid filenames (for all operating systems)
  - valid syntax in **R** files
  - correct syntax and meta data in Rd files
  - no missing documentation entries
- Also various code is exercised
  - examples in documentation are run
  - tests are run
  - if a working LaTeXis present, a `'.dvi'` version of the package's manual is created

Building Packages

- `R CMD build` builds a package from its sources
- The package is created in gzipped tar format
- Some checks and cleanups are performed

Name Spaces

- This is a mechanism to prevent interference of two packages which happen to use objects of the same name
- Specifies which variables in the package are to be *exported* to make them available to package users, and which are to *imported* from other packages
- Implemented by placing a `'NAMESPACE'` file in the top level package directory
- `'NAMESPACE'` file contains *name space directives*
- Packages with name spaces are loaded and attached to the search path by calling `library` but only the exported variables are placed in the attached frame

Name Spaces

- The `export` directive is of the form
  `export(f,g)`
  where `f` and `g` are variables to be exported
- A regular expression can be used with `exportPattern`
- Imports are specified with `import`:
  `import(foo,bar)`
  which imports all variables from packages `foo` and `bar`
- `importFrom` imports from a named package:
  `importFrom(foo,f,g)`
- The usual method of fully specifying a variable can also be used, as `foo::f` but this is inefficient and not recommended
- To use S3 method dispatch, methods should be registered:
  `S3method(print,foo)`
  ensures that `print.foo` will be used as a `print` method for class `foo`

S4 Classes and Methods

S4 Classes and Methods

I will use the slides of Yong Wang for this topic