

Scientific Programming with R

Stephen Eglen
Laurent Gatto

September 25, 2018

Aims of course

This course aims to teach R as a general-purpose programming language. Issues specific to Computational Biology (e.g. Bioconductor packages) are covered in other course modules.

In part 1, topics to be mastered in this course include:

- Interactive use of R .
- Basic data types: `vector`, `matrix`, `list`, `data.frame`, `factor`, `character`.
- Writing scripts.
- Graphical facilities.
- Writing your own functions.
- File input/output.
- Control-flow statements, looping.
- Vectorization.
- Numerics issues.
- Debugging.

Books and online help

- Introductory Statistics with R (Springer, Dalgaard).
- A first course in statistical programming with R (CUP, Braun and Murdoch).
- Computational Genome Analysis: An Introduction (Springer, Deonier, Tavaré and Waterman).
- S programming (Springer, Venables and Ripley).
- R programming for Bioinformatics (CRC Press, Gentleman).
- Scientific programming and simulation using R (CRC, Jones, Maillardet and Robinson).
- The art of R programming (No Starch Press, Matloff).
- Writing Scientific Software (WSS) (CUP, Oliveira and Stewart).
- www.r-project.org, www.rseek.org, www.r-bloggers.com
- R -help mailing list.
- Eglen (2009) <http://www.ploscompbiol.org/doi/pcbi.1000482>

1 / 191

2 / 191

Part 2: Scientific computing issues

In part 2 of the course¹, we will explore various other topics, building on core knowledge of R .

- Numerical integration
- Phase plane analysis
- Handling large files/data bases
- String processing (e.g. for genomic data)
- Advanced graphing / presenting results
- Reproducible research
- Future directions (R and generally)
- (Object-oriented programming)
- (Package development)
- (R profiling)

¹tentative

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

Scientific programming/software

- Different from software engineering (but should try to adhere to SE best practice, of course).
- Moving target.
- Domain scientists write the code (some argue this is a weakness).
- Has of course to be accurate, user-friendly (no GUI vs CLI ranting here), usable and useful, flexible, efficient and open, owned by the community, facilitate reproducible research.
- Contribute to users education (i.e not be a black box), in terms data requirements, the data processing and result interpret. Importance of documentation.

See Gentleman et al. (2004) Genome Biology (Bioconductor paper) for an example of successful scientific software.

5 / 191

6 / 191

Trustworthy software

The complexity of the data processes and of the computations applied to them mean that those who receive the results of modern data analysis have limited opportunity to verify the results by direct observation. Users of the analysis have no option but to trust the analysis, and by extension the software that produced it. Both the data analyst and the software provider therefore have a strong responsibility to produce a result that is trustworthy, and, if possible, one that can be *shown* to be trustworthy.

This places an obligation on all creators of software to program in such a way that the computations can be understood and trusted.

John M. Chambers, *Software for Data Analysis* (Springer)

What is R ?

- Statistical computing environment, and programming language.
- Very popular in many areas of statistics, computational biology.
- “Programming with data” (Chambers)
- Approach: command-line for one-liners; interactive usage; write scripts/functions for larger work (edit/run cycle); develop package for consolidation and distribution.

7 / 191

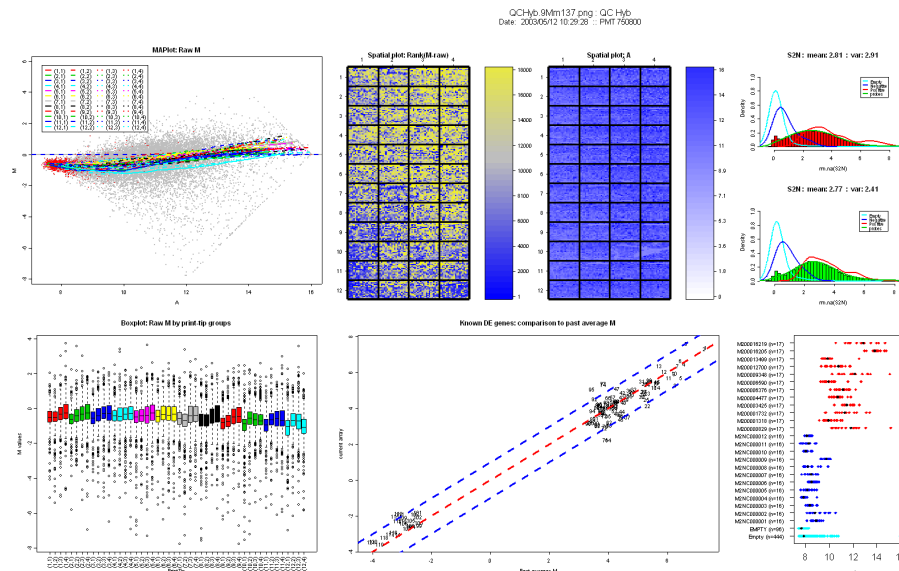
8 / 191

- S language came from Bell Labs (Becker, Chambers and Wilks). Commercial version S-plus (1988).
- R emerged as a combination of S and Scheme: Ross Ihaka and Robert Gentleman (NZ).
- 1993: first announcement.
- 1995: 0.60 release, now under GPL.
- 2018-07-02: release 3.5.1. Stable, multi-platform. Major release every April.
- R-core now 21 people, key academics in field, including John Chambers.

- GPL'd, available on many platforms.
- Excellent development team with yearly release cycle.
- Source always available to examine/edit.
- Fast for vectorized calculations.
- Foreign-language interface (C/Fortran) when speed crucial, or for interfacing with existing code.
- Good collection of numerical/statistical routines.
- Comprehensive R Archive Network (CRAN) ~ 11539 packages [2017-10-03] (cf 1000 in April 2007).
- On-line doc, with examples.
- High-quality graphics (pdf, postscript, quartz, x11, bitmaps). Often used just for plotting . . .

9 / 191

Graphics example



Jean YH Yang; gpQuality

<http://bioinf.wehi.edu.au/marray/ibc2004/lect1b-quality.pdf>

11 / 191

Weaknesses of R

- Loops are slow. Learn how to vectorize solutions.
- No fast compiler yet, and unlikely to happen due to nature of language. Byte compiler available in **compiler** package.
- No (decent) endorsed GUI built-in to R . Tk is available within base R , and packages for other graphical toolkits (e.g. Gtk2, Qt) are also available.
“Programming Graphical User Interfaces with R ”, M. F. Lawrence and J. Verzani

10 / 191

12 / 191

Brief comparison to matlab

- Flexible language, similar to matlab, but definitely not “everything is a matrix”. Frames, lists, vectors ...
- From matlab to R :
<http://cran.r-project.org/doc/contrib/R-and-octave.txt>
- Comprehensive matlab and R guide: <http://www.math.umaine.edu/faculty/hiebeler/comp/matlabR.html>
- Use `x[i]` not `x(i)` for indexing vectors.
- Making vectors: `x <- c(10, 9, 5, 1)`
- Assignment: best to use `<-` rather than `=`.

13 / 191

My very first R session

```
x <- rnorm(50, mean=4)
x
mean(x)
range(x)
hist(x)
## check help -- how to change title?
?hist
hist(x, main="my first plot")
q()
```

15 / 191

Using R

- Start-up: type ‘R’ at command line.
- Type commands interactively, and get results.
- Type commands into a file; `source('myfile.R')`; edit file ...
- Mac/Win has a GUI for interactive use, with internal editors.
- All platforms have a command-line interface
- Many external editors have support for R , including
 - Emacs through ESS (<http://ess.r-project.org>),
 - Eclipse IDE (<http://www.walware.de/goto/statet>),
 - Rstudio (<http://www.rstudio.org>),
 - Jupyter (<https://irkernel.github.io/>)
 - ...

14 / 191

Interacting with R

- Can use up/down arrow keys to go through command history. Within a command, use left/right arrow keys to edit.
- History can be saved over sessions (`?history`).
- Multiple commands can be put onto one line, using `;` as separator between lines, e.g. `x<-10; y<-3; a <- 5`.
- TAB can do object/file completion.

16 / 191

Objects and Functions

R manipulates objects. Each object has a name and a type (vector, matrix, list, ...)

Name of an object: letters (upper/lower case are distinct), digits, period. Start with a letter.

Objects set by way of assignment. Use the `<-` assignment operator rather than `=` wherever possible. (Does `i = i+1` make sense?)

Objects and Functions

```
> x <- 200
> half.x <- x/2
> threshold <- 95.0
> age <- c(15, 19, 30)
> age[2]      ## [] for accessing element.

[1] 19

> length(age) ## () for calling function.

[1] 3
```

17 / 191

What's up with the assignment and underscore? (Advanced)

Historically, underscore was used in S for assignment (because an old system keyboard had a key equivalent to the ASCII underscore that generated a back arrow). Hence underscore was not used within variables.

More recently, `=` is now available as an assignment operator (similar to languages like C), but is frowned upon as it can be confusing.

What does `i = i+1` imply mathematically?

Better to stick to `i <- i + 1` and use equals just within calls to functions, e.g. `runif(max=3)`.

Note also that assignments return values:

```
> y <- 1 + ( x <- 9 )
> a <- b <- 0
```

<http://developer.r-project.org/equalAssign.html>

19 / 191

18 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

20 / 191

Vectors

Vectors are a fundamental object for R . Scalars are treated as vector of length 1.

```
> y <- c(10, 20, 40)
> y[2]

[1] 20

> length(y)

[1] 3

> x <- 5
> length(x)

[1] 1
```

21 / 191

Generating vectors

Many short hand methods for regular sequences; c() for irregular.

```
> x <- seq(from=1, to=9, by=2)
> y <- seq(from=2, by=7, length=3)
> z <- 4:8
> a <- seq.int(5)  ## fast for integers
> b <- c(3, 9, 2)
> d <- c(a, 10, b)
> e <- rep( c(1,2), 3)
> f <- integer(7)
```

23 / 191

Vectors

Some operations work element by element, others on the whole vector, compare:

```
> y <- c(20, 49, 16, 60, 100)
> min(y)

[1] 16

> range(y)

[1] 16 100

> sqrt(y)

[1] 4.472136 7.000000 4.000000 7.745967 10.000000

> log(y)

[1] 2.995732 3.891820 2.772589 4.094345 4.605170
```

22 / 191

Accessing and setting elements

```
> x <- seq(from=100, by=1, length=20)
> x[3]      ## just element 3.

[1] 102

> x[c(12,14)] ## element 12 and 14

[1] 111 113

> x[1:5]

[1] 100 101 102 103 104

> bad <- 1:4
> x[-bad]    ## exclude elements

[1] 104 105 106 107 108 109 110 111 112 113 114 115 116 117
[15] 118 119
```

24 / 191

Accessing and setting elements

Can also provide a logical vector of same length as vector (logical values explained later).

```
> x <- c(5, 2, 9, 4)
> v <- c(TRUE, FALSE, FALSE, TRUE)
> x[v]

[1] 5 4
```

25 / 191

Recycling rule (Advanced)

Recycling is convenient, but dangerous; when vectors are of different lengths, the shorter one is often recycled to make a vector of the same length.

```
> a <- c(1,5) + 2
> x <- c(1,2); y <- c(5,3,9,2)
> x + y

[1] 6 5 10 4

> x + c(y,1) ## odd recycling, warning.
```

Warning in `x + c(y, 1)`: longer object length is not a multiple of shorter object length

```
[1] 6 5 10 4 2
```

27 / 191

Accessing and setting elements

Elements can be set in several ways

```
> x <- rep(0,10)
> x[1:3] <- 2
> x[5:6] <- c(-5, NA)
> x[7:10] <- c(1,9) ## recycling.
```

26 / 191

Recycling rule (Advanced)

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

28 / 191

Recycling rule (Advanced)

```
> x <- 1:10
> y <- x * 2
> z <- x^2
> y + z

[1] 3 8 15 24 35 48 63 80 99 120

> x + 1:2

[1] 2 4 4 6 6 8 8 10 10 12

> x + 1:3

Warning in x + 1:3: longer object length is not a multiple
of shorter object length

[1] 2 4 6 5 7 9 8 10 12 11
```

29 / 191

Naming indexes of a vector

```
> joe["height"] == joe[2]

height
TRUE
```

Referring to index by name rather than by position can make code more readable, and flexible. Cannot do things like `x[1:4]` easily though, since you need to name all four elements you want. Although extremely useful, names have a cost when processing large objects.

Note: in second use of `names()` above, we are actually using the *replacement function* `names<-`, see later.

Naming indexes of a vector

```
> joe <- c(24, 1.70)
> joe

[1] 24.0 1.7

> names(joe)

NULL

> names(joe) <- c("age", "height") ## replacement function
> joe

   age height 
24.0   1.7
```

30 / 191

Common functions for vectors

- `length()`
- `rev()`
- `sum()`, `cumsum()`, `prod()`, `cumprod()`
- `mean()`, `sd()`, `var()`, `median()`
- `min()`, `max()`, `range()`, `summary()`
- `exp()`, `log()`, `sin()`, `cos()`, `tan()` (radians, not degrees)
- `round()`, `ceiling()`, `floor()`, `signif()`
- `sort()`, `order()`, `rank()`
- `which()`, `which.max()`
- `any()`, `all()`

31 / 191

32 / 191

Functions as function args

Functions can be called within function calls; the following are equivalent:

```
> x <- c(3, 2, 9, 4)
>
> y <- exp(x); z1 <- which(y > 20) ## case 1
> z2 <- which ( exp(x) > 20)      ## case 2
>
> all.equal(z1, z2)

[1] TRUE
```

33 / 191

Default values for function arguments

A function will error if not all required arguments are provided. Some functions have both required and optional arguments. If the optional arguments are not provided, they are either ignored, or they take a default value.

Usage:

```
round(x, digits = 0)
```

35 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

34 / 191

Default values for function arguments

```
> x <- c(2.091, 4.126, 7.925)
> round()      ## required arg is missing
```

```
Error in eval(expr, envir, enclos): 0 arguments passed to
'round'which requires 1 or 2 arguments
```

```
> round(x)
```

```
[1] 2 4 8
```

```
> round(x, digits = 2)
```

```
[1] 2.09 4.13 7.92
```

Let's see how this works in more detail.

36 / 191

Argument matching

R has a flexible method for specifying arguments to function. We can either provide an actual value for a formal argument, or give arguments as key=value (or formal=actual).

As an example, let's look at help for seq:

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

(NB: in seq(from=x), from is the **formal argument** of the function, and here x is the actual value.)

The ... notation allows for other arguments to be passed, which are not used by this function.

37 / 191

... in function calls (Advanced)

Why do some functions, like sqrt, require only one argument, yet others take many arguments?

Functions like c, cbind, have ... in the arguments:

Usage:

```
c(..., recursive=FALSE)
```

Arguments:

```
...: objects to be concatenated.
```

The ... indicate any number of objects may be passed, not just (say) one or two.

The result of c() is to combine them all into one long vector, taking into account if the keyword "recursive" is provided (when args are first flattened). The ... can also indicate that other arguments can be provided which are not processed directly by this function, but may be useful for other functions (e.g. popular when plotting).

39 / 191

Argument matching

Typical calls are as follows:

```
> seq(1, 3, 0.5)           ## positional matching
[1] 1.0 1.5 2.0 2.5 3.0

> seq(1, 5, length.out=3) ## can skip args (e.g. by)
[1] 1 3 5

> seq(to=5)                ## order not important.
[1] 1 2 3 4 5

> seq(f=5, t=1)            ## abbrev tags.
[1] 5 4 3 2 1

> seq(len=5, 1, 2)         ## tags removed before positional matching
[1] 1.00 1.25 1.50 1.75 2.00
```

38 / 191

Replacement functions (Advanced)

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

[1] 1 2 3 4 5

> length(x)

[1] 5

> length(x) <- 2
> x

[1] 1 2
```

40 / 191

Normally `length(x)` would return a value, rather than you assigning a value to the function! These are **replacement functions**, see help page:

Usage:

```
length(x)
length(x) <- value
```

Help pages

- What you can expect to find:
 - Description – one line summary
 - Usage – formal arguments
 - Arguments – interpretation of arguments
 - Details – what the function does
 - Value – return value.
 - References – documentation
 - See also – helps you find related pages
 - Examples – guaranteed to run: `example(hist)`

- `help(hist)` to see help file (or `?hist`).
- `args(hist)` to see arguments of a function.
- `example(boxplot)` run examples in help page.
- `options(help_type="html")` will then use web-browser for help.
- `help.search("histogram")`
- `demo()` to list all demos, e.g. `demo(graphics)`

NB: In the R terminal `?command` works as shorthand for `help("command")` except for a small number of commands, e.g. `if`, `while`. Use the longhand for these.

41 / 191

Numbers and special values

- numeric (floating-point, double): 12, 4.92, 1.5e3 – `is.numeric()` (integers converted to f.p.)
- integers 1L – `is.integer()`
- complex: 3+2i – `is.complex()`

```
> typeof(1)
[1] "double"

> typeof(1L)
[1] "integer"

> is.integer(1)
[1] FALSE

> is.integer(1L)
[1] TRUE
```

42 / 191

Numbers and special values

Special values:

- NA: not available. (Often used to represent missing data point) – `is.na()`
- NaN: not a number. e.g. `0/0` – `is.nan()`
- Inf, -Inf: $\pm\infty$ – `is.finite()`

You will also meet:

- NULL: often, list of zero length – `is.null()`

Numbers and special values

```
> typeof(NA)
[1] "logical"
> typeof(NaN)
[1] "double"
> typeof(Inf)
[1] "double"
> typeof(NULL)
[1] "NULL"
```

45 / 191

Operator precedence ?Syntax

```
> 3 * 4 + 2 != 3 * (4 + 2)
> 2^3+1 != 2^(3+1)
> 1:5-1
```

Operator precedence ?Syntax

```
> 3 * 4 + 2 != 3 * (4 + 2)
[1] TRUE
> 2^3+1 != 2^(3+1)
[1] TRUE
> 1:5-1
[1] 0 1 2 3 4
```

46 / 191

47 / 191

48 / 191

Operator precedence ?Syntax

Subset taken from ?Syntax, see that page for full list. Highest precedence at top.

'['['	indexing
'\$ @'	component / slot extraction
'^'	exponentiation (right to left)
'- +'	unary minus and plus
':'	sequence operator
'%any%'	special operators
'* /'	multiply, divide
'+ -'	(binary) add, subtract
'< > <= >= == !='	ordering and comparison
'!'	negation
'& &&'	and
' '	or
'<- <<-'	assignment (right to left)
'?'	help (unary and binary)

Bottom line: use parentheses to order preference.

When things go wrong

Syntax errors are those where you've just made a typing mistake.
Logical errors are harder to find!

Common problems:

- missing close bracket leads to continuation line.

```
> x <- (1 + (2 * 3)
+
```

Hit Ctrl C (below) or keep typing!

- too many parens:

```
2 + (2*3))
```

- wrong/mismatched brackets (see next slide).
- Likewise, do not mix double quotes and single quotes.
(Unless you need to quote within quotes.)
- ...
- wrong variable name (not syntax error)
- When things seem to take too long, try C-c [Ctrl and C, together]

Operators

Most operators will be familiar, but some may not:

```
x <- 10
x == 4    ## test for equality
x != 10   ## not equal?
7 %/% 2   ## division, ignoring remainder. (3)
7 %% 2    ## remainder (1)
x <- 9    ## assignment
x <<- 9   ## assign x to 9 in the global env. (BAD)
## Raising to a power can be done in two ways.
all.equal( 10.1 ** 2.5, 10.1^2.5 )
```

Types of parentheses

- f(3,4) – call the function f, with arg1=3, arg2=4.
- a + (b*c) – use to enforce order over which statements are executed.
- { expr1; expr2; ...; exprn } – group a set of expressions into one compound expression. Value returned is value of last expression; used in looping/conditionals.
- x[4] – get the 4th element of the vector x.
- 1[[3]] – get the 3rd element of some list 1, and return it. (compare with 1[3] which returns a list with just the 3rd element inside – will see list objects later).

From interactive to source files

- Typing in commands interactively is good for one-liners, but soon you will want to switch to putting your sequence of commands into a script file, and then ask R to run (source) those commands.
- This leaves to a rapid edit–run–edit cycle.
- e.g. type these commands into a file trig.R:

```
x <- seq(from=0, to=2*pi, length=100)
y <- sin(x)
z <- cos(2*x)
z ## will not appear when source'd
print(y[1:10]) ## should use print()
plot(x, y, type='l')
lines(x, z, type='l', col='red')
```

- Eval within R using source('trig.R').

53 / 191

Scripts

- Use source('trig.R', echo=TRUE) to see commands and output. Or use print(x) to print an object within a script.
- Keep your code open in the editor in one window, and keep R running in another window.
- Are you in the right directory? Check that you can see your script file in the same directory as where R is currently. Check dir(), and setwd, see later.
- On unix, the initial directory is the directory from where you started R . On windows, the initial directory might be “My Documents”. You may need to change directory (setwd) first.
- Use a good editor that helps you spot mistakes (e.g. paren matching, syntax highlighting). Examples: Emacs/ESS, gedit, Rstudio.
- Use .R or .r as the filename suffix. Avoid any temptation to put spaces (although R does not mind) in your filenames!

55 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

54 / 191

Why are scripts a good thing?

- You don't have to remember what commands you ran, they are saved in the file.
- This corresponds to the “source is real” philosophy of using S and R .
- You can easily give your work to others, by passing them the file.
- You can eventually run your scripts in BATCH, i.e. non-interactively. Good for long jobs which you can leave overnight.

56 / 191

Running scripts in batch (Advanced)

- At the command line, type `R CMD BATCH trig.R`. R will start up, process your commands and then quit.
- Output is stored in the file `trig.Rout`
- If there were no errors, the last line of the output file shows the time taken to run the script.
- Any output is not shown on the screen but sent to a PDF called `Rplots.pdf`.
- This is a GREAT way of testing your scripts, since R starts with an empty workspace, you will see if you have all the steps needed.
- Aim to always leave your scripts in a working state at the end of a session, so that a few days later you don't have to remember why it wasn't working!

57 / 191

Commenting your work

- Do not be shy when putting comments into your code.
- Meaningful variable names help, but do document. At a bare minimum, each file should state at the top what the purpose of the file. Important variables and functions should be clearly documented.
- You may think it obvious how your code works, but try looking at it a week or a month later and then see if you clearly understand it. If in doubt, document it.
- Describe *what* your code is doing, not *how* it is doing it (WSS, p79). Compare the following two:

```
> s <- s + 1    ## prepare to process next subject
> j <- j + 1    ## increment j by 1.
```

- Comments can be put before commands, if you temporarily do not want to run that command; remove the comments when you want to run the command again, or delete the line.

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

59 / 191

Rscript

- Rscript works as an interpreter. It is quicker to start than R (fewer packages are loaded?).
- Can use interactively or for standalone scripts.

```
$ Rscript -e 'round(runif(10))'
[1] 0 1 0 0 1 0 1 1 0 0
```

```
#!/usr/bin/env Rscript
args <- commandArgs(TRUE)

stopifnot(length(args)==3)
args = as.numeric(args)
n = args[1]; mean = args[2]; sd = args[3]
rnorm(n, mean, sd)
```

```
$ ./simple_rnorm.R 5 10 2
[1] 8.651807 13.372094 10.063347 10.703155 6.351886
```

58 / 191

Line wrapping

- Line-wrapping. Do not write beyond around column 72, for readability. You can break long expressions at suitable points.
- End of line should not look like end of an expression. Compare:

```
x <- sqrt( c(100, 200, 300, 400, 500) ) + 10
x <- sqrt( c(100, 200, 300, 400, 500) )
+ 10
x <- sqrt( c(100, 200, 300, 400, 500) ) +
10
x <- sqrt( c(100, 200, 300, 400, 500) ) +
10
```

60 / 191

Line wrapping

```
> ## 1: ok - all fits onto one line, just.
> (x <- sqrt( c(100, 200, 300, 400, 500) ) + 10)

[1] 20.00000 24.14214 27.32051 30.00000 32.36068

> ## 2: not okay -- first line is seen as complete.
> x <- sqrt( c(100, 200, 300, 400, 500) )
> + 10

[1] 10

> x

[1] 10.00000 14.14214 17.32051 20.00000 22.36068
```

Line wrapping

```
> ## 3: solved, by moving the operator (+) up.
> x <- sqrt( c(100, 200, 300, 400, 500) ) +
+ 10
> x

[1] 20.00000 24.14214 27.32051 30.00000 32.36068

> ## 4: as 3, but indentation makes it clearer.
> x <- sqrt( c(100, 200, 300, 400, 500) ) +
+      10
> x

[1] 20.00000 24.14214 27.32051 30.00000 32.36068
```

61 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

Matrices

A matrix is just a vector with some additional markup to reformat it. Matrix stored in column-major order (like fortran, unlike C).

```
> x <- 1:6
> is.matrix(x)

[1] FALSE

> dim(x) <- c(2,3)
> is.matrix(x)

[1] TRUE

> x

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

> dim(x)

[1] 2 3
```

62 / 191

63 / 191

64 / 191

Matrices

```
> x[2,2] ## extracting a value.

[1] 4

> x[1,] ## extracting row

[1] 1 3 5

> x[1:2, 2:3]

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

> x[,2] ## not column vector!

[1] 3 4
```

65 / 191

Typical matrix construction methods

- `matrix()`
- `cbind()`
- `rbind()`

```
> m <- matrix( floor(runif(6, max=50)), nrow=3)
> x <- rbind( c(1,4,9), c(2,6,8), c(3,2,1))
> y <- cbind( c(1,2,3), 5, c(4,5,6))
```

67 / 191

Matrices

```
> x[, 2, drop=TRUE] ## default

[1] 3 4

> x[, 2, drop=FALSE] ## gotcha!

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

66 / 191

Typical matrix construction methods

```
> matrix( floor(runif(6, max = 50)), nrow = 3 ) ## ncol = 2

      [,1] [,2]
[1,]   19   16
[2,]   49   13
[3,]    1   11
```

68 / 191

Typical matrix construction methods

```
> rbind( c(1,4,9), c(2,6,8), c(3,2,1) )
```

	[,1]	[,2]	[,3]
[1,]	1	4	9
[2,]	2	6	8
[3,]	3	2	1

Typical matrix construction methods

```
> cbind( c(1,2,3), 5, c(4,5,6) ) ## recycling again
```

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

69 / 191

Typical matrix construction methods

Note that matrix indices can also be named:

```
> dimnames(m) <- list(student = c("ann", "bob", "joe"),  
+                       exam = c("math", "french"))  
> m
```

	exam	
student	math	french
ann	45	13
bob	19	33
joe	28	38

```
> m["bob", ] ## get bob's scores
```

	math	french
	19	33

71 / 191

Common matrix operations

- diagonal: `diag(x)` – watch if `x` matrix or scalar!
- matrix multiplication: `%*%` vs `*` (element-wise)

```
> x <- matrix(1:4, 2,2)  
> i <- diag(2) ## 2 x 2 identity matrix  
> x %*% i      ## should be x
```

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

```
> x * i      ## not x!
```

	[,1]	[,2]
[1,]	1	0
[2,]	0	4

70 / 191

72 / 191

Common matrix operations

- transpose: `t(x)`
- `dim`, `nrow`, `ncol`
- inverse: `solve(x)`,

```
> (x %*% solve(x)) == diag(nrow(x))
```

```
      [,1] [,2]  
[1,] TRUE TRUE  
[2,] TRUE TRUE
```

Arrays

Arrays as extension of matrices to multiple dimensions.

```
> array(1:12, c(2,2,3))
```

```
, , 1  
      [,1] [,2]  
[1,]     1     3  
[2,]     2     4  
  
, , 2  
      [,1] [,2]  
[1,]     5     7  
[2,]     6     8  
  
, , 3  
      [,1] [,2]  
[1,]     9    11  
[2,]    10    12
```

73 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

Boolean values – ?logical

Logical values TRUE/FALSE (avoid abbrev to T/F).

TRUE/FALSE equivalent to 1/0

```
> as.integer(TRUE)
```

```
[1] 1
```

```
> as.integer(FALSE)
```

```
[1] 0
```

```
> as.logical(1)
```

```
[1] TRUE
```

```
> as.logical(0)
```

```
[1] FALSE
```

74 / 191

75 / 191

76 / 191

Boolean values – ?logical

```
> d <- c(3.2, 1.0, 4.0, 9.2, 2.3, 8.1, 6.3)
> d > 5.0

[1] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE

> d[d > 5.0]

[1] 9.2 8.1 6.3

> which(d > 5.0)

[1] 4 6 7
```

Boolean values – ?logical

Key operators for handling boolean values:

```
> !TRUE                ## negation: swap T -- F.

[1] FALSE

> TRUE & FALSE          ## and: both must be true.

[1] FALSE

> FALSE | TRUE          ## or: one must be true.

[1] TRUE

> xor(TRUE, TRUE)       ## xor: only one is true.

[1] FALSE
```

Boolean values – ?logical

```
> d[which(d > 5.0)]

[1] 9.2 8.1 6.3

> medium.sized <- (d > 3.0) & (d < 5.0)
> d[medium.sized]

[1] 3.2 4.0

> d[!medium.sized]

[1] 1.0 9.2 2.3 8.1 6.3

> ifelse(d > 3.0, 1.0, 0.0) ## same as as.numeric(d > 3)

[1] 1 0 1 1 0 1 1
```

Boolean logic: issues

`a & b` (same for `a | b`) is an **elementwise** operation, with a result the same length as the longer of `a`, `b` (recycling is used if one vector is shorter).

`a && b` examines only the **first** element of `a` and `b`, returning one logical value. **Lazy evaluation** is used: we calculate only what's needed to determine result.

```
> TRUE || some.long.computation()

[1] TRUE

> TRUE && stop("no")

Error in eval(expr, envir, enclos): no

> FALSE && stop("no")

[1] FALSE
```

R uses *Lazy evaluation*, which delays the evaluation of an expression (here the argument) until its value is actually required ^[2]:

```
> f <- function(x) { 10 }
> system.time(f(Sys.sleep(3)))

   user  system elapsed 
    0.00    0.00     0.00 

> f <- function(x) { force(x); 10 }
> system.time(f(Sys.sleep(3)))

   user  system elapsed 
0.0000  0.0000   3.0030
```

²example from Hadley Wickham's devtools

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

Boolean logic: issues

Comparing numbers: When testing numbers for equality, can use `x == y` when `x`, `y` are integers, otherwise use `all.equal(x,y)`. See later on numerics.

Avoid using `F`

```
> F <- 3
> F == FALSE

[1] FALSE
```

What is a list?

A list is used to collect a group of objects of different sizes and types. Very flexible. Often returned as the result of a complex function (e.g. model fit) to return all relevant information in one object.

```
> l <- list(who='joe', height=1.70, dob=c(1960, 12, 1))
> l

$who
[1] "joe"

$height
[1] 1.7

$dob
[1] 1960    12     1
```

What is a list?

```
> length(l)

[1] 3

> names(l) ## show components

[1] "who"      "height" "dob"

> l$height ## access an element.

[1] 1.7
```

What is a list?

List elements can either be accessed by name (e.g. `l$height` or `l[['height']]` – if named list) or by position (`l[[2]]`).

When using numbers to index list, compare `l[2]` (a list with one element) with `l[[2]]`. You can therefore do `l[2:3]` but not `l[[2:3]]`.

85 / 191

What is a list?

```
> unlist(l) ## opposite of list

  who height  dob1  dob2  dob3
"joe"  "1.7" "1960"  "12"   "1"

> str(l) ## structure of l

List of 3
 $ who   : chr "joe"
 $ height: num 1.7
 $ dob   : num [1:3] 1960 12 1
```

Modifying lists (Advanced)

We can append new items to list either by making a new list from the old one (e.g. 1) , or directly by assigning new element (e.g. 2):

```
> l1 <- list(who="fred")
> l1 <- c(l1, height=1.8) ## e.g. 1
> l1[["dob"]] <- c(1965, 10, 17) ## e.g. 2
```

87 / 191

86 / 191

88 / 191

Modifying lists (Advanced)

Deleting list items:

```
> l1["height"] <- NULL
> str(l1)
```

```
List of 2
 $ who: chr "fred"
 $ dob: num [1:3] 1965 10 17
```

Modifying lists (Advanced)

Finally, for completeness, here is a way to predefine a list of given length and gradually fill it in:

```
> empty <- vector("list", 3) ## Prealloc to given length.
> names(empty) <- c("who", "height", "dob")
> empty[["height"]] <- 1.8
```

89 / 191

Initialisation

Initialising R object (list, vectors, ...) is **much** faster than creating and extending these objects on the fly

```
> n <- 1e4
> l <- list()
> system.time(for (i in 1:n) l[[i]] <- rnorm(1e3) )

   user  system elapsed 
1.398   0.109   1.508 

> l <- vector("list", n)
> system.time(for (i in 1:n) l[[i]] <- rnorm(1e3) )

   user  system elapsed 
0.895   0.010   0.906
```

91 / 191

Data frames

A data frame is like a matrix, but each column can be of a different type.

Data frame is stored as a list, with each element a vector of same length. Useful for reading in tabular data from a file (see `read.csv`).

```
> nms <- c("joe", "fred", "harry")
> a <- c(24, 19, 30)
> ht <- c(1.7, 1.8, 1.75)
> s <- c(TRUE, FALSE, TRUE)
```

90 / 191

92 / 191

Data frames

```
> cbind(a, ht)      ## matrix, preserves
```

```
      a  ht
[1,] 24 1.70
[2,] 19 1.80
[3,] 30 1.75
```

```
> class(cbind(a, ht))
```

```
[1] "matrix"
```

Data frames

```
> cbind(nms, ht)    ## matrix, ht col becomes strings.
```

```
      nms    ht
[1,] "joe"  "1.7"
[2,] "fred" "1.8"
[3,] "harry" "1.75"
```

```
> class(cbind(nms, ht))
```

```
[1] "matrix"
```

93 / 191

94 / 191

Data frames

```
> d <- data.frame(name = nms,
+                 age = a,
+                 height = ht,
+                 student = s)
> class(d)
```

```
[1] "data.frame"
```

Data frames

```
> d$age ## same as d[, "age"]
```

```
[1] 24 19 30
```

```
> names(d)
```

```
[1] "name"    "age"      "height"   "student"
```

```
> d[2,] ## access 2nd row.
```

```
  name age height student
2 fred  19   1.8   FALSE
```

95 / 191

96 / 191

Data frames

Compare how a data frame (d) is printed, compared to printing `as.list(d)`.

```
> d
  name age height student
1  joe  24   1.70     TRUE
2 fred  19   1.80    FALSE
3 harry 30   1.75     TRUE

> as.list(d)

$name
[1] joe  fred  harry
Levels: fred harry joe

$age
[1] 24 19 30

$height
[1] 1.70 1.80 1.75

$student
[1] TRUE FALSE TRUE
```

97 / 191

Factors (Advanced)

(Mostly seen when reading in data frames)

Factors internally code categorical variables with a number. e.g. 1=Sunday, 2=Monday, ... 7=Saturday. For large vectors, this is more efficient storage, especially when character strings repeat. Can also make code more readable.

Also useful in many statistical functions, using the formula interface.

99 / 191

Outline

[Introduction](#)

[Vectors](#)

[Calling functions](#)

[Scripts](#)

[Matrices](#)

[Boolean logic](#)

[Lists](#)

Factors

[Character arrays](#)

[Objects in your environment](#)

[Basic plotting](#)

[Reading/writing data to file system](#)

[Writing functions](#)

[Conditionals and looping](#)

[Vectorization](#)

[Random number generation](#)

[Debugging](#)

98 / 191

Factors (Advanced)

```
> scores1 <- c('good', 'poor', 'bad', 'poor',
+             'bad', 'bad', 'good')
> scores <- factor(scores1)
> scores

[1] good poor bad  poor bad  bad  good
Levels: bad good poor

> levels(scores)

[1] "bad" "good" "poor"

> as.integer(scores) ## integer representation

[1] 2 3 1 3 1 1 2

> as.character(scores) ## show strings

[1] "good" "poor" "bad" "poor" "bad" "bad" "good"
```

100 / 191

Factors (Advanced)

```
> which(scores1 == 'bad')

[1] 3 5 6

> ## Can do further comparisons with an ordered factor.
> ## Levels are now ordered, as shown by "<" in levels.
>
> s2 <- factor(scores1,
+             levels = c('poor', 'bad', 'good'),
+             ordered = TRUE)
> s2[1] > s2[2]

[1] TRUE
```

101 / 191

Strings / character arrays

Character arrays are vectors of strings.

Use single (') or double (") quotes to mark strings, but don't mix:

```
> x <- 'good'
> z <- "no" # need to match"
> z <- "it's working"
```

103 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

102 / 191

Strings / character arrays

Within a script, easy way to generate output:

```
> cat("Now computing the steady-state\n")

Now computing the steady-state

> x <- 134
> cat("sqrt of", x, "is", sqrt(x), "\n")

sqrt of 134 is 11.57584

> cat("sqrt of", x, "is", sqrt(x), "\n", sep='__')

sqrt of__134__is__11.57584__
```

See also message, warning and stop for communicating diagnostic messages – cat and print are generally used when displaying an object.

104 / 191

Strings / character arrays

backslash characters allow you to generate control characters, importantly:
newline: `\n`, tab: `\t`.

```
> cat("5\t9\n_")
```

```
5 9
```

```
-
```

`paste()` returns a string, e.g. for assignment.

```
> x <- 1:5; exp.dir <- '/home/stephen/res'
> file <- paste(exp.dir, '/expt_res', x, '.dat', sep='')
> file
```

```
[1] "/home/stephen/res/expt_res1.dat"
[2] "/home/stephen/res/expt_res2.dat"
[3] "/home/stephen/res/expt_res3.dat"
[4] "/home/stephen/res/expt_res4.dat"
[5] "/home/stephen/res/expt_res5.dat"
```

105 / 191

Strings

- Just as R stores vectors of numbers, it also stores vectors of strings.
- Pattern matching facilities are available, based on Unix terms (grep, regular expressions). These are worth learning:

```
> s <- c('apple', 'bee', 'cars', 'danish', 'egg')
> nchar(s)
```

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

```
> substr(s, 2,3)
```

```
[1] "pp" "ee" "ar" "an" "gg"
```

paste0

Don't forget `paste0()` where `sep=""`.

(Humour): [http://simplystatistics.org/2013/01/31/](http://simplystatistics.org/2013/01/31/paste0-is-statistical-computings-most-influential-contribution)

[paste0-is-statistical-computings-most-influential-contribution](http://simplystatistics.org/2013/01/31/paste0-is-statistical-computings-most-influential-contribution)

106 / 191

Strings

```
> grep('e', s)
```

```
[1] 1 2 5
```

```
> grep('^e', s) ## regexps...
```

```
[1] 5
```

```
> sub('e', '_', s)
```

```
[1] "appl_" "b_e" "cars" "danish" "_gg"
```

```
> gsub('e', '_', s) ## global sub, watch "bee"
```

```
[1] "appl_" "b__" "cars" "danish" "_gg"
```

107 / 191

108 / 191

Strings

```
> toupper(s)

[1] "APPLE" "BEE" "CARS" "DANISH" "EGG"

> sprintf('name %s len %d', s, nchar(s)) ## C users!

[1] "name apple len 5" "name bee len 3"
[3] "name cars len 4" "name danish len 6"
[5] "name egg len 3"
```

109 / 191

Environments (Advanced)

An environment is a *frame*, or collection of named objects (variables), and a pointer to an *enclosing environment*.

The working environment of your interactive R session is the `<R_GlobalEnv>`.

```
> x <- 2
> ls()

[1] "x"

> ## current environment
> environment()

<environment: R_GlobalEnv>
```

111 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

110 / 191

```
> e <- new.env()
> e

<environment: 0x4f87aa8>

> ls(e)

character(0)

> parent.env(e)

<environment: R_GlobalEnv>

> e$x <- 1
> ls(e)

[1] "x"

> e$x != x

[1] TRUE
```

112 / 191

Inspecting variables and the environment

```
x <- 9
y<- c(2,4,5)
m <- matrix(2:5, 12,10)
objects()      ## what vars do I have?
ls()           ## shorthand for objects.
str(m)         ## Display compact representation
head(m)
tail(m)
rm(list = ls()) ## clear up the working environment
ls()
```

113 / 191

What is an object?

- An object is typically either a variable or a function.
- You can use the same name for a function and a variable, and R uses context to decide which you mean:

```
> sum <- 3 + 4 + 5
> total <- sum(1:4)
> total

[1] 10

> sum

[1] 12

> sum(sum)  ## can get confusing!

[1] 12
```

115 / 191

Converting an object from one type to another

Some ways of finding out what kind of object you have and converting.

```
mode(y)
typeof(y)
class(y)
object.size(y)
is.vector(y)  ## is.xyz() check if type XYZ
is.matrix(y)
as.vector(m)  ## convert object to type XYZ
l <- list(x=c(1,4), y=c(6,9,12))
unlist(l)
```

114 / 191

OO programming in R

Data abstraction; object manipulation becomes independent of the implementation details.

There are several OO programming frameworks in R . The main ones, supported in base R are S3, S4 and S4 ReferenceClasses. S6 are lightweight reference classes.

Bioconductor provides many ad hoc classes to store, manipulate and process microarray, RNA Seq, proteomics, flow cytometry, ... data.

If there is interest, we could have a session about OO programming at the end of the course.

116 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

Basic plotting

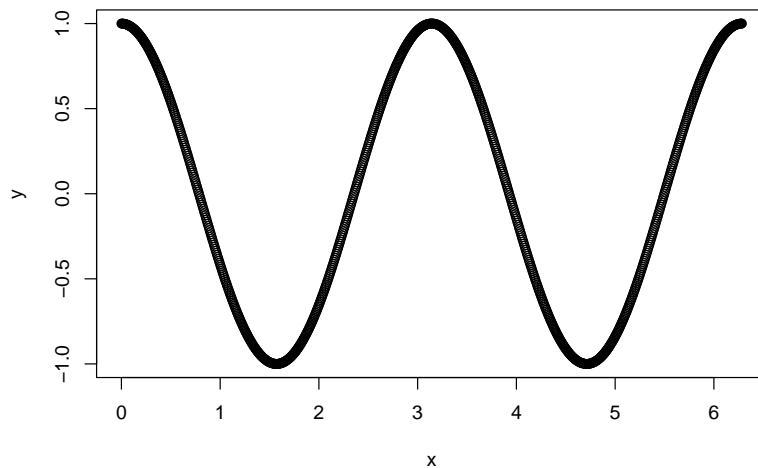
- Basic x,y plots (scatter plots)
- Multiple plots in one figure
- Saving your plots

This section will just introduce the mechanics of making basic plots, rather than worry about interpreting them.

117 / 191

Basic plotting

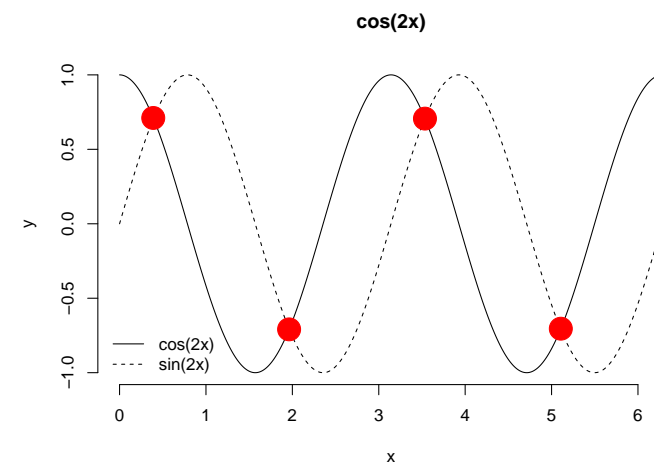
```
> x <- seq(from=0, to=2*pi, len=1000)
> y <- cos(2*x)
> plot(x,y) ## just provide data; sensible labelling
```



119 / 191

Basic plotting

```
> ## Expand on previous plot ...
> plot(x,y, main='cos(2x)', type='l', lty=1, bty='n')
> y2 <- sin(2*x)
> lines(x,y2, type='l', lty=2)
> same <- which( abs(y - y2) < 0.01)
> points(x[same], y[same], pch=19, col='red', cex=3)
> legend('bottomleft', c("cos(2x)", "sin(2x)"), bty='n', lty=c(1,2))
```



118 / 191

120 / 191

Options controlling the plot

`par()` outputs the (long) list of options that control plotting behaviour.

Read `?par` for all the details!

Common options to explore:

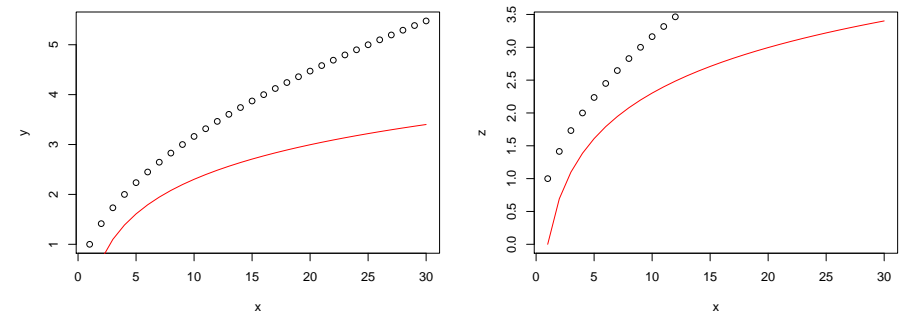
- `mfrow`, `mfcol`: multiple plots in figure
- `mar`, `oma`: margins around plot and figure.
- `ask`: whether to hit RETURN between pages of figures.

Multiple data sources on one plot

When you wish to have multiple data sources on one plot (e.g. two time-series plots), the approach is to draw the first using `plot` and then draw subsequent features using `lines` or `points`.

Axes are not rescaled, so draw the bigger plot first.

```
> x <- 1:30
> y <- sqrt(x); z <- log(x)
> plot(x,y); lines(x,z, col='red')
> plot(x,z, type = "l", col = "red"); points(x,y) ## some data missing
```

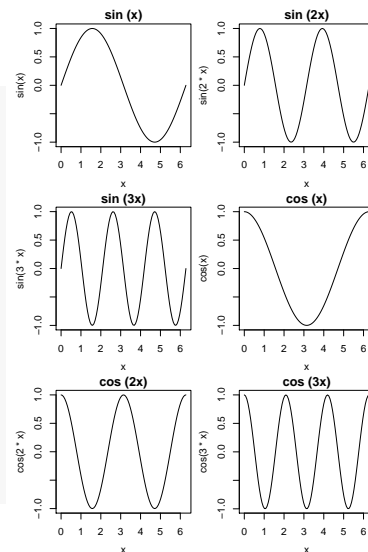


121 / 191

Multiple plots in one figure

`mfrow` and `mfcol` are useful parameters within `par()`, but margins often need to be changed to maximise space.

```
pdf(file = 'mfrow_eg.pdf',
    width = 4, height = 6)
par(mfrow = c(3,2))
par(mar = c(3.5, 3.5, 1.5, 0.5),
    mgp = c(2.5, 1, 0))
x <- seq(from = 0, to = 2*pi,
    len = 100)
plot(x, sin(x), main = "sin (x)",
    type = 'l')
plot(x, sin(2*x), main = "sin (2x)",
    type = 'l')
plot(x, sin(3*x), main = "sin (3x)",
    type = 'l')
plot(x, cos(x), main = "cos (x)",
    type = 'l')
plot(x, cos(2*x), main = "cos (2x)",
    type = 'l')
plot(x, cos(3*x), main = "cos (3x)",
    type = 'l')
dev.off()
```



123 / 191

Saving your plots

R can save plots in many formats, including PDF, postscript, PNG, JPEG.

Best to use vector formats (PDF, postscript) for graphs and bitmap formats (png, jpeg) for images.

R has output devices, only one of which is active, `dev.cur()`.

```
> dev.list()
> pdf(file='hist.pdf', width=7, height=7) ## inch
> dev.list()
> hist( rnorm(9999) )
> dev.off() ## close device
> png(file='hist.png', w=600, h=600) ## pixels
> hist( rnorm(9999) )
> dev.off()
```

Zoom in on text of PNG to see limitations of this format.

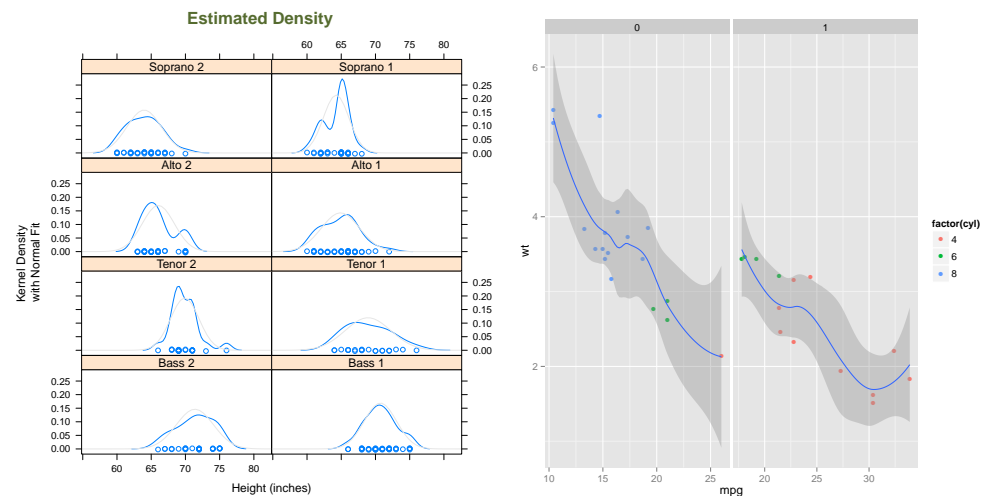
122 / 191

124 / 191

Next steps with plotting (Advanced)

R has a vast range of functions for plotting particular data types. You may read about different packages for plotting:

- base graphics (or “traditional”)
- lattice/grid (lattice is built upon grid)
- ggplot2 – <http://had.co.nz/ggplot2/>



125 / 191

Here are some starting points to explore:

- `demo(graphics)` to see diversity of plots.
- low-level functions: `symbols()`, `rect()`, `segments()`, `abline()`.
- `curve(x*log(x))`
- R graphics gallery
<http://www.r-graph-gallery.com>

127 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

126 / 191

128 / 191

- What's my current directory? `dir`, `getwd`, `setwd`
- `scan`, `readLines`
- `read.csv`, `read.table`, `write.table`
- RData files – save and load
- Further I/O functions

129 / 191

Scan, write, readLines

For basic reading/writing of data, use `scan`/write. Filenames are specified relative to current directory. Can even give URL as a file. Files often have a header which can be skipped over.

```
> x <- scan('files/ages.dat', skip=1)
> summary(x)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
14.00	19.25	27.25	31.58	39.25	65.00

```
> s1 <- readLines('files/ages.dat') ## treats as strings
> summary(s1)
```

Length	Class	Mode
5	character	character

```
> h <- scan('http://damtp.cam.ac.uk/user/sje30/r/heights.dat') ## heights.dat
> summary(h)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
136.2	158.3	164.9	163.0	169.2	178.7

131 / 191

- where am I currently? `getwd()`
- change me to a new directory: `setwd("/tmp")`
(GUIs have chooser for interactively changing directory.)
- What files are in my [current] directory?

```
> dir()
> dir("/tmp")
> dir(pattern="\\.R$") ## regexps, see later.
```

130 / 191

Scan, write, readLines

```
> rand.vals <- round( runif(100, min=5, max=10), 2)
> tf <- tempfile()
> tf
```

```
[1] "/tmp/Rtmp1XqNw/file8a107b78d5b8"
```

```
> write(rand.vals, tf)
> s <- scan(tf)
> all.equal(s, rand.vals)
```

```
[1] TRUE
```

132 / 191

read.table et al.

If data are tabular, `read.table` or `read.csv` is often useful. (Useful for importing spreadsheets; just save as a comma separated value file, CSV.)

```
x <- read.table('./files/players.dat', sep = '\t',
               header = TRUE)

names(x)
head(x)
x[2,]
x$Goals
is.data.frame(x)
tf <- tempfile()
write.csv(x, tf, row.names=FALSE)
## sort by goals scored.
x[order(x$Goals, decreasing = TRUE), ]
```

See `?read.table`.

133 / 191

Saving your workspace with .RData files

When you quit R, you are asked:

```
> q()
Save workspace image? [y/n/c]:
```

If you answer y, all objects (variables and functions) in your global environment are saved for future use, using `save.image`. From `?save`:

'`save.image()`' is just a short-cut for "save my current workspace", i.e., '`save(list = ls(all=TRUE), file = ".RData")`'. It is also what happens with '`q("yes")`'.

Warning: If an `.RData` file is present in the current directory when starting R, it is silently loaded. I think it can be dangerous, as you may not realise what values have been silently loaded. Better to instead be explicit:

```
> save.image(file='all.Rda')    ## keep everything
> save(x, y, z, file='key.Rda') ## or just key objects
> ## ...
> load('all.Rda')              ## reload objects
```

or start R using `R --no-restore`

135 / 191

Rdata files

Text files are useful for portably storing data, so that they can be read across applications. R has its own format for *efficiently* storing objects. Files much smaller than text files. However, this format is not universally known.

```
> n <- 99999; x <- rnorm(n)
> txt.file <- tempfile()
> rda.file <- tempfile()
> write(x, n, file = txt.file)
> save(x, n, file = rda.file)
>
> ## Compare sizes of files with the object.
> object.size(x)
> file.info(txt.file)$size
> file.info(rda.file)$size    ## compressed
>
> rm(x,n)
> load(rda.file) ## reload data.
```

134 / 191

Further I/O functions (Advanced)

R has many facilities for I/O. See for example the following help topics.

- `?connections` — interface to files, pipes, sockets, compressed files ...
- `?sink` — divert R output to a connectin
- `?dget` / `?dput` — read/write ASCII representation of an R object.

Also: XML, DBMS, SQLite, netCDF, hdf5, ...

136 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

Writing functions: overview

- Why bother?
- How to write (local args, return value; cannot change value)
- Example: computing std. deviation
- Local variables within functions
- Recursion.

137 / 191

Functions

- Functions promote code reuse.
- Black-box approach; given inputs, what output should I expect? This requires good documentation of what your function does. Can it be described without having to look at the code?
- Finding the right level of definition for a function is hard, and how to modularise comes with experience. Typically rewrite many times before getting final solution

139 / 191

Functions

- How to define a new function:

```
> my.fun <- function(arg1, arg2) {  
+   ## Doc string here.  
+   x <- arg1 * 2  
+   y <- sqrt(arg2) + 5  
+   z <- x * y  
+   ## last value is the return value of the function.  
+   ## Use a list to return several items.  
+   z ## same as return(x)  
+ }
```

138 / 191

140 / 191

Example of writing a new function

Compute the standard deviation of a vector of numbers:

$$\text{std.dev} = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}} \quad \text{where} \quad \bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

```
> std.dev <- function(x) {  
+   ## Return std dev of X.  
+   n <- length(x)  
+   xbar <- sum(x)/n  
+   diff <- x - xbar  
+   sum.sq <- sum( diff^2)  
+   var <- sum.sq / (n-1)  
+   ## last value calculated is return value.  
+   sqrt(var)  
+ }
```

141 / 191

- Any change to formal args within a function does not change value of actual argument outside the function:

```
> sum.sq <- function(x) {  
+   x <- x^2   ## change internally  
+   sum(x)  
+ }  
> y <- c(4, 5, 6)  
> sum.sq(y)  
[1] 77  
> y  
[1] 4 5 6
```

143 / 191

Terminology of variables within functions

- In `std.dev`, `x` is the name of a **formal argument**. In the following, `y` is called the **actual argument** (doesn't have to be named `x` – can be named however you wish).

```
> n <- 5  
> y <- c(9, 2, 7, 10)  
> std.dev(y)  
[1] 3.559026  
  
> print(n) ## should still be 5, not 4.  
[1] 5
```

- Local variables** within function are not available outside of function.

142 / 191

Handling unbound variables

Variables created by assignment within a function are known as local variables (e.g. `y` below). If a variable is not local, or a formal argument, it is an **unbound variable**. It may be found in the enclosing environment (typically the global workspace), or an error is generated – this is **bad practice!**

```
> fn1 <- function(x) {  
+   y <- x^2  
+   res <- sum( (y - thresh)^2 )  
+   res  
+ }  
> dat <- 1:5  
> fn1(dat)      ## case 1
```

Error in `fn1(dat)`: object 'thresh' not found

```
> thresh <- 10  
> fn1(dat)      ## case 2  
[1] 379
```

144 / 191

Handling unbound variables (2)

In this case, better to define `thresh` as an argument of the function, and provide a default value:

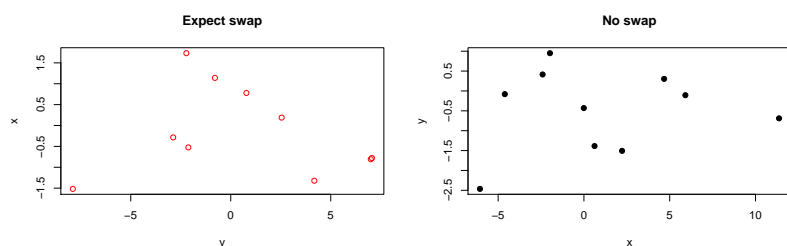
```
> fn1 <- function(x, thresh=10) {  
+   y <- x^2  
+   res <- sum( (y - thresh)^2 )  
+   res  
+ }  
> fn1(dat)      ## case 3  
  
[1] 379
```

Advanced: use `codetools::checkUsage()` to find unbound vars; `codetools::findGlobals()` for globals.

145 / 191

The ... arguments

```
> p <- function(x, y, ...) {  
+   if (diff(range(x)) > diff(range(y))) {  
+     plot(x, y, ...)  
+   } else {  
+     plot(y, x, ...)  
+   }  
+ }  
> p(rnorm(10, 0, 1), rnorm(10, 0, 5), main = "Expect swap",  
+   col = "red")  
> p(rnorm(10, 0, 5), rnorm(10, 0, 1), main = "No swap", pch = 19)
```



147 / 191

The ... arguments

From R language definition

The ... argument is special and can contain any number of arguments. It is generally used if the number of arguments is unknown or in cases where the arguments will be passed on to another function.

See `?cat` for an example of *number of arguments is unknown*.

146 / 191

Writing a replacement function (Advanced)

Convention for a replacement function is that the name should end with `<-`. The last argument of the replacement function must be called `value` and is the RHS of the assignment.

```
> "threshold<-" <- function(x, value) {  
+   ## X is the object to update  
+   ## VALUE is the value on the RHS.  
+   y <- ifelse(x > value, 1, 0)  
+   return(y)  
+ }  
> x <- c(0.3, 0.1, 0.6, 0.7, 0.9, 0.2)  
> threshold(x) <- 0.4  
> x  
  
[1] 0 0 1 1 1 0
```

In general, replacement functions are used to set variable attributes or object slots.

148 / 191

Tips for writing functions

- Can you think of a way to break down the problem so that a team can work on the problem, with each person assigned to a independent piece? “Divide + conquer”.
- Each function should be easy to test, then you can “freeze” it. Write test cases, which can be automatically checked.

```
> all.equal(my.fun(100,200), 300)
```

- Rule of thumb: each function should be no more than a page or two of code.
- For large projects, avoid mixing computation and plotting in the same function – separate the two jobs; this makes it easier to run in batch.

```
> res <- some.computation(par1, par2, par3)
> plot.results(res)
```

149 / 191

Control-flow constructs

- if
- switch
- for
- while
- Vectorization
- simple applications – numerics

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

150 / 191

if / if ... else ...

```
> x <- 8;
>
> if (x > 10) {
+   ## condition was true
+   cat("x is bigger than 10\n")
+ } else {
+   cat("x is 10 or less\n")
+ }
x is 10 or less
```

Notes:

“else ...” can be omitted if you do not need it.

if returns a value, which can be assigned, e.g. `y <- if (x <10) 40 else 20`. A better solution in this case however is the vectorized form `y <- ifelse(x<10, 40, 20)`

151 / 191

152 / 191

Braces in conditional constructs

Curly braces not needed if there is only one expression in the if clause:

```
if ( x > 10 ) {  
  y <- 1  
}  
  
if ( x > 10 )  
  y <- 1  
##
```

But braces are needed in multiline if/else statement:

```
if ( x > 10 ) {  
  y <- 1  
} else {  
  y <- 0 ## OK  
}  
  
if ( x > 10 )  
  y <- 1  
else  
  y <- 0 ## NOT OK  
##
```

From ?Control: Note that it is a common mistake to forget to put braces ('{ .. }') around your statements, e.g., after 'if(..)' or 'for(...)'. In particular, you should not have a newline between '}' and 'else' to avoid a syntax error in entering a 'if ... else' construct at the keyboard or via 'source'. For that reason, one (somewhat extreme) attitude of defensive programming is to always use braces, e.g., for 'if' clauses.

153 / 191

Recursive functions

Here is an example of using conditionals with a divide and conquer approach; quicksort in a few lines (albeit not very efficient).

```
> qsort <- function(data) {  
+   ## Sort data into ascending order.  
+   n <- length(data)  
+   if (n <= 1) {  
+     data  
+   } else {  
+     pivot <- data[floor(n/2)]  
+     less <- data[which(data < pivot)]  
+     equal <- data[which(data == pivot)]  
+     greater <- data[which(data > pivot)]  
+     c( qsort(less), equal, qsort(greater))  
+   }  
+ }  
> all(replicate(99, {  
+   data <- runif(2000, max=10)  
+   all.equal(qsort(data), sort(data)) }))  
  
[1] TRUE
```

155 / 191

switch (Advanced)

Nested if ... else commands can get a bit messy. Like other languages, R has a switch construct. From ?switch:

```
> centre <- function(x, type) {  
+   switch(type,  
+     mean = mean(x),  
+     median = median(x),  
+     trimmed = mean(x, trim = .1))  
+ }  
> x <- rcauchy(10)  
> centre(x, "mean")  
  
[1] -0.1566859  
  
> centre(x, "median")  
  
[1] -0.1831462  
  
> centre(x, "trimmed")  
  
[1] -0.1473678
```

154 / 191

Looping constructs

Looping constructs allow you to repeat calculations as many times as you wish. This is why computers are so useful – it is just as easy (usually) to repeat something 1000 times as 10 times.

e.g. if you want to simulate flipping a (biased) coin 100 times, and counting the number of heads, no problem. If you want to repeat this process 1000 times, no problem. (See later.)

156 / 191

for loops

for (var in seq) command

seq is a vector; var is set in turn to each value in the vector, and then command executed. Multiple commands can be given within braces.

e.g.

```
> x <- 6
> for (i in 1:3) {
+   res <- x * i
+   cat(x, "*", i, "=", res, "\n")
+ }

6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
```

157 / 191

Breaking out of loops

repeat expr will repeatedly execute expr until you break out of the loop.

```
> i <- 3
> repeat {
+   if (i == 5) {
+     break
+   } else {
+     cat("i is", i, "\n")
+     i <- i+1
+   }
+ }

i is 3
i is 4
```

159 / 191

while loops

```
while (condition) {
  command
  command
}
```

So the commands are executed until the condition is no longer true.

Typically then one of the commands will change the condition.

e.g. print all the Fibonacci numbers ($f[i] = f[i-1] + f[i-2]$) less than 100.

```
> n1 <- 0; n2 <- 1
> while (n2 < 100) {
+   print(n2)
+   old <- n2
+   n2 <- n2 + n1
+   n1 <- old
+ }
```

158 / 191

Breaking out of loops

next allows you to skip to next iteration of a loop. Both next and break can be used within other loops (while, for).

```
> for (i in 1:10) {
+   if ((i %% 2) == 0)
+     next
+   print(i)
+ }

[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
```

160 / 191

A word on indentation

Indentation helps you see the flow of the logic, rather than flattened version. (Use tab key to indent). Reformatting tools are available (e.g. within Emacs).

```
## version 1.
i <- 3
repeat {
  if (i==10) {
    break
  } else {
    cat("i is", i, "\n")
    i<- i+1
  }
}
```

```
## version 2.
i <- 3
repeat {
  if (i==10) {
    break
  } else {
    cat("i is", i, "\n")
    i<- i+1
  }
}
```

Indentation helps to show structure, and match braces.

161 / 191

Vectorization

When possible, operate on vectors, rather than using for loops. Rewrite code, but beware sometimes not possible (Fibonacci). Compute difference between times of events, *e*. Given *n* events, there will be *n*-1 inter-event times. `interval[i] <- e[i+1] - e[i]`

```
> diff1 <- function(e) {
+   n <- length(e)
+   interval <- rep(0, n-1) ## good to pre-alloc!
+   for (i in 1:(n-1)) {
+     interval[i] <- e[i+1] - e[i]
+   }
+   interval
+ }
> diff2 <- function(e) {
+   n <- length(e)
+   e[-1] - e[-n]
+ }
> e <- c(2, 5, 10.2, 12, 19)
> diff2(e)

[1] 3.0 5.2 1.8 7.0

> all.equal(diff1(e), diff2(e))

[1] TRUE
```

Advantages: shorter, more readable and faster (no loops).

163 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

162 / 191

Vectorization example

Q: Flip a biased coin [*p*=0.6 of heads] 100 times; how many heads do you get? Repeat this for 1000 trials.

```
> n <- 100    ## number of coin flips in trial
> p <- 0.6    ## prob of getting heads
> ntrials <- 1000
```

```
> trial1 <- function(n, p.heads) {
+   count <- 0
+   for (i in 1:n) {
+     if (runif(1) < p.heads)
+       count <- count + 1
+   }
+   count
+ }
>
> res <- rep(0, ntrials)
> for (j in 1:ntrials) {
+   res[j] <- trial1(n, p)
+ }
> hist(res)
```

164 / 191

Vectorization example

Q: Flip a biased coin [$p=0.6$ of heads] 100 times; how many heads do you get? Repeat this for 1000 trials.

```
> n <- 100    ## number of coin flips in trial
> p <- 0.6    ## prob of getting heads
> ntrials <- 1000
```

```
> trial1 <- function(n, p.heads) {
+   count <- 0
+   for (i in 1:n) {
+     if (runif(1) < p.heads)
+       count <- count + 1
+   }
+   count
+ }
> res <- rep(0, ntrials)
> for (j in 1:ntrials) {
+   res[j] <- trial1(n, p)
+ }
> hist(res)
```

```
> trial2 <- function(n, p.heads) {
+   rand.vals <- runif(n)
+   sum( rand.vals < p.heads)
+ }
> res <- replicate(ntrials,
+                  trial2(n, p))
> hist(res)
```

165 / 191

apply family

e.g. how to compute sum of each row of a matrix? `sum(A)` will normally return the sum of all elements of `A`.

`apply(X, MARGIN, FUN, ...)`
`MARGIN` = 1 for row, 2 for cols.
`FUN` = function to apply
`...` = extra args to function.

```
> A <- matrix(1:6, 2, 3)
> row.means <- apply(A, 1, mean) ## or rowMeans
> col.sums <- apply(A, 2, sum, na.rm=TRUE) ## colSums
```

167 / 191

Vectorization example

Q: Flip a biased coin [$p=0.6$ of heads] 100 times; how many heads do you get? Repeat this for 1000 trials.

```
> n <- 100    ## number of coin flips in trial
> p <- 0.6    ## prob of getting heads
> ntrials <- 1000
```

```
> trial1 <- function(n, p.heads) {
+   count <- 0
+   for (i in 1:n) {
+     if (runif(1) < p.heads)
+       count <- count + 1
+   }
+   count
+ }
> res <- rep(0, ntrials)
> for (j in 1:ntrials) {
+   res[j] <- trial1(n, p)
+ }
> hist(res)
```

```
> trial2 <- function(n, p.heads) {
+   rand.vals <- runif(n)
+   sum( rand.vals < p.heads)
+ }
> res <- replicate(ntrials,
+                  trial2(n, p))
> hist(res)
```

In this case, `hist(rbinom(1000, 100, 0.6))` would also work!

166 / 191

apply family

Other functions: `lapply` (apply to list), `sapply` (simplify), `replicate`, ...

```
> lapply(ls(), object.size)
> sapply(ls(), object.size)
> tapply(1:20, factor(rep(letters[1:5], each = 4)), sum)
> ## see also ?by
> mapply(sum, 1:5, 1:5, 1:5)
> hist( replicate(200, mean(rnorm(100))) )
```

How to `*apply` – <http://stackoverflow.com/questions/3505701>

Exercise: Why is this better than writing a for loop?
(Eglen, 2009); parallel package: `mclapply()`, `parLapply()`, ...

168 / 191

Sometimes you don't want to pollute name space by defining a new function, so just use an "anonymous function", i.e. a function without a name. Particularly useful e.g. in an apply call.

```
> my.mat <- matrix(1:10, ncol=5)
> apply(my.mat, 2, function(x) { sum(x^2) + 10 })
```

Since functions are just objects, anonymous functions are just objects without names, similar to 'anonymous numbers' like $a+b$ in an expression $a+b+c$.

169 / 191

Efficiency

"premature optimization is the root of all evil" (Knuth, 1974).

Examples adopted from www.mathworks.com/res/code_segments
f1 is bad; should pre-allocate vector, rather than rely on R to allocate memory repeatedly³.

```
> f1 <- function() {
+   n <- 1e4; decay <- 0.9995
+
+   out <- 1.0
+   for (i in 2:n)
+     out[i] <- out[i-1] * decay
+   out
+ }
> system.time(o1 <- f1())
```

user	system	elapsed
0.012	0.000	0.010

```
> f2 <- function() {
+   n <- 1e4; decay <- 0.9995
+   out <- rep(0, n) ##pre-alloc
+   out[1] <- 1.0
+   for (i in 2:n)
+     out[i] <- out[i-1] * decay
+   out
+ }
> system.time(o2 <- f2())
```

user	system	elapsed
0.008	0.000	0.008

³This is not optimisation, but rather good coding practice.

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$f[n] = f[n-1] + f[n-2]$$

How to vectorize?

Exercise: write a function, fibonnaci(n) that returns the n^{th} element of the sequence. Assume that fibonnaci(1) == 0, fibonnaci(2) == 1.

Exercise: use fibonacci() to estimate the golden ratio.

170 / 191

A final word on efficiency

- Rule 1 of optimization: don't bother (Kernighan).
- For loops are not always a bad thing. See last example in Help Desk article (May 2008).

http://cran.r-project.org/doc/Rnews/Rnews_2008-1.pdf

Numerics issues

Although integer arithmetic is reliable, floating-point arithmetic is to be treated with care! (All R 's calculations are in what C programmers call “double precision”.) Compare:

```
> 1.0 + 2.0 == 3.0
```

```
[1] TRUE
```

```
> 0.1 + 0.2 == 0.3
```

```
[1] FALSE
```

From FAQ (7.31?)

```
> a <- sqrt(2)
> a * a == 2
```

```
[1] FALSE
```

```
> a * a - 2
```

```
[1] 4.440892e-16
```

173 / 191

How big is infinity?

Use while loop to estimate it:

```
> x <- 1
> while ( is.finite(x*2) ) {
+   x <- x*2
+ }
>
> x ## 8.988466e+307
```

```
[1] 8.988466e+307
```

```
> x*2 ## Inf
```

```
[1] Inf
```

```
> (x*2)/2 ## Inf
```

```
[1] Inf
```

```
> .Machine$double.xmax
```

```
[1] 1.797693e+308
```

175 / 191

Numerics issues

Solution - testing *near equality*

```
> all.equal( 0.1 + 0.2, 0.3)
```

```
[1] TRUE
```

If there is a difference, you get a summary of differences. Within a function, you might prefer just to test whether result is TRUE or not:

```
> x = 1:4
> y = rnorm(4,sd=0.2)
> all.equal(x,y)
```

```
[1] "Mean relative difference: 1.013383"
```

```
> isTRUE(all.equal(x,y))
```

```
[1] FALSE
```

174 / 191

How small is epsilon?

How big can ϵ be such that $1 + \epsilon = 1$? (Taken from Goldberg (1991) ACM article, p220).

```
> eps <- 1
> while (eps + 1 > 1) {
+   eps <- eps * 0.5
+ }
> eps ##1.110223e-16
```

```
[1] 1.110223e-16
```

```
> 1 + eps ## 1
```

```
[1] 1
```

```
> (1 + eps == 1) ## TRUE
```

```
[1] TRUE
```

```
> 1 + (2*eps) ##1
```

```
[1] 1
```

```
> (1 + (2*eps) == 1) ## FALSE
```

```
[1] FALSE
```

176 / 191

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

Outline

Introduction

Vectors

Calling functions

Scripts

Matrices

Boolean logic

Lists

Factors

Character arrays

Objects in your environment

Basic plotting

Reading/writing data to file system

Writing functions

Conditionals and looping

Vectorization

Random number generation

Debugging

Random number generation

Computers usually generate “pseudo-random numbers”. They are generated based on some iterative formula:

$$x_{new} = f(x_{old}) \mod N$$

where modulo operation provides the “remainder” division.

To generate the first random number, you need a **seed**.

Setting the seed allows you to reliably generate the same sequence of numbers, which can be useful when debugging programs.

R has many routines for generating random samples from various distributions, but for now we will just use `runif()`, (and maybe `rnorm()`).

Exercise: write a random number generator. See: “Randu: a bad random number generator”. <http://physics.ucsc.edu/~peter/115/randu.pdf>

Exercise: Apply the central limit theorem to generate samples from a normal distribution by adding together samples from a uniform distribution.

177 / 191

Debugging (Advanced)

See **An introduction to the Interactive Debugging Tools in R**, Roger D Peng for detailed usage.

<http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf>

For a more modern approach <https://www.rstudio.com/products/rstudio/release-notes/debugging-with-rstudio/>

- warnings vs errors; converting warnings to errors; `stopifnot()`.
- what to do when I get an error: `traceback()`
- simple print statements are often useful.
- Use of `browser()` at key points in code.
- `debug(fn)`, `undebug(fn)`
- Using `recover()` rather than `browser()`

179 / 191

178 / 191

180 / 191

Warnings and errors

- A warning is softer than an error; if a warning is generated your program will still continue, whereas an error will stop the program.

```
> log(c(2, 1, 0, -1, 2))  
  
Warning in log(c(2, 1, 0, -1, 2)): NaNs produced  
[1] 0.6931472 0.0000000 -Inf      NaN 0.6931472  
  
> xor( c(TRUE, FALSE))  
  
Error in xor(c(TRUE, FALSE)): argument "y" is missing,  
with no default
```

- If you try to isolate warnings, you can change warnings to errors: `options(warn=2)`. See `?options` for further details.
- Add warnings and errors to your code using `warning()`, `stop()`.
- Can add “assertions” into your code to check that certain values hold.

```
> stopifnot(x>0)
```

- Other useful safety checks: `all(x>0)`, `any(x>0)`

181 / 191

Single-stepping through your code

Use `browser()` to single-step through your code. Place it within your function at the point you want to examine (e.g.) local variables.

Can use `debug(function.name)` to step through entire function.

`undebug()` will remove that debug call.

Within the browser, you can enter expressions as normal, or you can give a few debug commands:

- n: single-step
- c: exit browser and continue
- Q: exit browser and abort, return to top-level.
- where: show stack trace.

Debug on `stddev.R`

183 / 191

Traceback

When your program generates an error, use `traceback()` to find out where it went wrong:

```
> start <- function() { go( sqrt(10)) }  
> go <- function(x) { inner(x, '-13')}  
> inner <- function(a, b) {  
+   c <- sqrt(b)  
+   a * log(c)  
+ }  
> start() ## error  
> traceback() ## postmortem debugging
```

182 / 191

Safety-checks: browser

Here's a possible usage of `browser()` that I have in my code:

```
find.high <- function(x, t) {  
  ## Return samples in x bigger than t.  
  ## (Better to use x[x>t] in real-life!)  
  max.length <- 100 ## should be upper limit...  
  results <- rep(0, max.length)  
  counter <- 0  
  for (i in x) {  
    if (i > t) {  
      counter <- counter + 1  
      if (counter > max.length) {  
        browser()  
      } else {  
        results[counter] <- i  
      }  
    }  
  }  
  results[1:counter]  
}  
x <- rnorm(100)  
find.high(x, 0.7)  
  
x <- rnorm(1000)  
(1- pnorm(0.7)) * length(x) ## expected.  
find.high(x, 0.7)
```

184 / 191

recover

`recover()` is like `browser()`, except you can choose which level to inspect, rather than the level at which `browser` was called.

Following allows `recover()` to be launched when you hit an error:

```
options(error=recover)
```

Here we simply tell R that when an error is generated, we call the function “recover”. The default is NULL, in which case stop is called.

From `?options`:

*Note that these need to be specified as e.g.
`options(error=utils::recover)` in startup files such as `.Rprofile`.*

CRAN: Comprehensive R Archive Network

CRAN: Site(s) for downloading R, and also its many contributed *packages*.

Mac/Win have a GUI for installing packages, or it can be done on the command line:

```
install.packages(c("splancs", "sp"))  
R CMD INSTALL mypackage.tar.gz ## from shell
```

If asked to select a CRAN mirror, in UK use:

<http://mirrors.ebi.ac.uk/CRAN/>.

For Bioconductor

```
> source("http://www.bioconductor.org/biocLite.R")  
> library("BiocInstaller")  
> biocLite("Biobase")
```

Packages

- R has a packaging system for external code.
- A **package** is loaded from a **library** using `library("pkg.name")`.
- Beware: don't call a package a library! A library is a group of folders where packages are stored ...

```
> library() ## view available packages  
> library(help="cluster") ## what's in this?  
> library("cluster") ## load package  
> example(pam) ## can use pam and friends.  
> detach("package:cluster") ## remove pkg.
```

185 / 191

Managing libraries

If you do not have write access to the default library, R will create a local one in your home directory. Multiple libraries are supported:

```
> .libPaths()  
[1] "/mhome/damtp/cbi/sje30/NOBACKUP/RLIB-ds"  
[2] "/alt/applic/user-maint/sje30/COMPBIO/RLIB"  
[3] "/alt/applic/user-maint/sje30/COMPBIO/RLIB/bioc-3.6"  
[4] "/alt/applic/lib/R/site-library"  
[5] "/usr/local/lib/R/site-library"  
[6] "/usr/lib/R/site-library"  
[7] "/usr/lib/R/library"
```

You can also set R libraries as a global shell environment in your `.bashrc` file (e.g. linux)

```
export R_LIBS=$HOME/NOBACKUP/RLIB
```

(Be careful! Check that you are not overwriting an existing `R_LIBS` setting.)

186 / 191

187 / 191

188 / 191

A success story of R . Started 2001 with aims to:

- provide access to stat/graphical methods for analysis of *genomic* data.
- link seamlessly to on-line databases (PubMed/GenBank).
- allow rapid development of extensible software.
- provide training in methods (short courses).
- promote software with high quality docs and reproducible research (vignettes) ...
- Gentleman et al. (2004) Genome Biology 5:R80.
<http://genomebiology.com/2004/5/10/R80>

- S3, S4 and S4 Reference classes for OOP.
- Building your own packages. Useful for packaging up your code, data sets and documentation. You may wish to do this for large projects that you wish to share with others. Read *Writing R Extensions* manual and see `package.skeleton` to get started.
- Access to databases. Computational Biology datasets are often quite large, and you might wish to access data via databases. R package DBI provides common interface to SQLite, MySQL, Oracle. See Gentleman (2008), Chapter 8.

189 / 191

190 / 191

Reproducible research: Sweave, knitr and vignettes

- Use one file to store code and document. Best shown by way of example... `estimate.Rnw`
- Vignettes often used in Bioconductor to document packages.
- Interactively explore vignettes.

```
> library("tkWidgets")
> vExplorer()
```

- knitr is the natural successor to Sweave. It rocks!
- Jupyter documents also becoming more popular.