Statistical Computing (MATH10093: L01)

- Course organiser and Lecturer:
  Finn Lindgren, `finn.lindgren@ed.ac.uk`, JCMB 4615
- Additional computer lab tutors:
  Ruben Amoros Salvador, Serveh Sharifi Far, Graeme Auld, Adrian Casey, Javier Garrido Guillen, Junho Lee, Nestor Sanchez, Christine Simpson
- Lectures: w1–4 and w6–9, Tue 16:10–17:00
  JCMB Lecture Theatre B
- Computer labs: w1–4 and w6–9, Wed 09:00–10:50
  Murchison House LG.12
- Computer lab office hours (prelim): w5, 10, and 11, Wed 09:00–10:50
  Murchison House LG.12
- Coursework:
  w4–6 (50%), w9–"12" (50%)
  Handouts & hand-ins on the Wednesdays of w4/w9 and w6/w12
  Marks returned 3 weeks after each hand-in

Topics and approximate lecture & lab plan

- The R language, structured programming, debugging
- Likelihood optimisation techniques
- Model and method assessment: scoring rules
- Simulation and simulation studies
- Bootstrap methods
- Robust numerical methods for statistical computations

Functions are recipes

Solve for $\boldsymbol{x}$ in $(\boldsymbol{A} + \boldsymbol{A}^\top)\boldsymbol{x} = \boldsymbol{b}$:

Script style code

```r
A <- matrix(rnorm(10 * 10), 10, 10)
b <- rnorm(10)
x <- solve(A + t(A), b)
```

Structured, reusable code

```r
solve_my_problem <- function(A, b) {
  solve(A + t(A), b)
}

big_matrix <- matrix(rnorm(10 * 10), 10, 10)
some_vector <- rnorm(10)

x <- solve_my_problem(big_matrix, some_vector)
x2 <- solve_my_problem(5, 2)
x3 <- solve_my_problem(matrix(runif(400), 20, 20), rnorm(20))
x4 <- solve_my_problem(A = big_matrix, b = some_vector)
```

Basic R maths programming

The basic flow of an piece of R code is

1. Assign values to some *variables*
2. Perform transformation and/or calculations using the variables using *operators* and *functions*
3. Present the results (as text and/or figures)

Basic *functions* follow similar principles to regular maths functions

```r
radius <- 4
angle <- pi / 3
radius * c(cos(angle), sin(angle))

## [1] 2.000000 3.464102
```

- Predefined operators:
  - Assignment operator: `<-`
  - Arithmetic operators: `/`, `*`
- Predefined constant: `pi`
- Predefined functions: `cos()`, `sin()`, `c()` (`c` for *combine*)

The output above is prefixed by the comment character `#`

Basic R object data types

- *Numeric* and *logical* types: `integer`, `double`, `logical`
- `character`
- `factor`

The function `str()` displays the internal structure of an object

```
str(4)

##  num 4

str(2.4)

##  num 2.4

str(TRUE)

##  logi TRUE

str("Text")

##  chr "Text"

str(factor(c("Red", "Blue")))

##  Factor w/ 2 levels "Blue","Red": 2 1
```

Object collection data types

- `vector`: A sequence of objects of the same type
- `matrix`: A 2D grid of objects of the same type
- `list`: An ordered collection of objects of the same or different types
- `data.frame`: A collection of vectors of equal length

```
str(1:5) ## Sequence operator :

##  int [1:5] 1 2 3 4 5

str(c(TRUE, FALSE, TRUE))

##  logi [1:3] TRUE FALSE TRUE

matrix(1:6, nrow = 2, ncol = 3)

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

Indexing and vectorised operations

- Vector indexing: `x[i]`
- Logical vector indexing: `x[something > 0]`
- If `i` is itself a vector, a sub-vector is extracted
- Basic maths functions operate on each element of a vector

```
a <- cos((0:5) * 2 * pi / 6)
a

## [1]  1.0  0.5 -0.5 -1.0 -0.5  0.5

a[4]

## [1] -1

a[4:5]

## [1] -1.0 -0.5

a[a < 0]

## [1] -0.5 -1.0 -0.5
```

Indexing and vectorised operations

- Matrix indexing: `x[i, j]`
- If `i` or `j` are integer or logical vectors, a sub-matrix is extracted
- Basic maths functions operate on each element of a matrix

```
b <- exp(matrix(a, 3, 2))
b

##           [,1]      [,2]
## [1,] 2.7182818 0.3678794
## [2,] 1.6487213 0.6065307
## [3,] 0.6065307 1.6487213

b[2, ] # Extract an entire row as vector

## [1] 1.6487213 0.6065307

b[2:3, 2, drop = FALSE] # Extract two rows of a column and keep as a matrix

##           [,1]
## [1,] 0.6065307
## [2,] 1.6487213
```

Data type: list

A list stores a collection of (usually *named*) variables with possibly different types and size.

```
x <- list(temperature = c(4, 7, 42, 10, 23, 25),
          shoe_size = c(11, 7, 5, 10.5),
          room = c("B", "3210", "5205"))
x

## $temperature
## [1]  4  7 42 10 23 25
##
## $shoe_size
## [1] 11.0  7.0  5.0 10.5
##
## $room
## [1] "B"    "3210" "5205"

str(x)

## List of 3
##  $ temperature: num [1:6] 4 7 42 10 23 25
##  $ shoe_size  : num [1:4] 11 7 5 10.5
##  $ room       : chr [1:3] "B" "3210" "5205"
```

Data type: list

- Sub-list indexing: `x[indices]`
- List element indexing: `x[[index]]`, `x[[name]]`, `x$name`

```
x[2:3]

## $shoe_size
## [1] 11.0  7.0  5.0 10.5
##
## $room
## [1] "B"    "3210" "5205"

x[["room"]]

## [1] "B"    "3210" "5205"

x$temperature

## [1]  4  7 42 10 23 25

names(x) ## Extract the names

## [1] "temperature" "shoe_size"   "room"
```

Data type: data.frame

A data.frame stores a collection of *named* variables as columns, with each row corresponding to a joint observation.

```
x <- data.frame(Temperature = c(4, 7, 10, 23, 25),
                Failure = c(FALSE, TRUE, TRUE, FALSE, FALSE))
x

##   Temperature Failure
## 1           4   FALSE
## 2           7    TRUE
## 3          10    TRUE
## 4          23   FALSE
## 5          25   FALSE

colnames(x) ## Extract the column names

## [1] "Temperature" "Failure"

str(x)

## 'data.frame': 5 obs. of  2 variables:
##  $ Temperature: num  4 7 10 23 25
##  $ Failure    : logi  FALSE TRUE TRUE FALSE FALSE
```

A `data.frame` can be indexed both as a matrix *and* as a list:

```
x[2:3, ]

##   Temperature Failure
## 2           7    TRUE
## 3          10    TRUE

x[, "Failure"] ## Note: x[, "Failure", drop=FALSE] returns a data.frame

## [1] FALSE  TRUE  TRUE FALSE FALSE

x$Failure

## [1] FALSE  TRUE  TRUE FALSE FALSE

x[["Failure"]]

## [1] FALSE  TRUE  TRUE FALSE FALSE

name <- "Failure"
x[[name]]

## [1] FALSE  TRUE  TRUE FALSE FALSE
```

Data type conversion

- Functions of the form `as.`*`type`*`()` attempt to convert an object into the given type.
- Common conversions: `as.vector()`, `as.data.frame()`, `as.matrix()`, `as.list()`, `unlist()`

```
x <- matrix(1:6, 2, 3)
as.vector(x)

## [1] 1 2 3 4 5 6

as.data.frame(x)

##   V1 V2 V3
## 1  1  3  5
## 2  2  4  6

a <- as.data.frame(x)
colnames(a) <- c("A", "B", "C")
a

##   A B C
## 1 1 3 5
## 2 2 4 6
```

Creating empty objects and removing elements

- In most situations, `NULL` acts as *nothing*
- Pre-allocation of sized or empty objects can be useful

```r
x <- list(1, 2, 3:4, 5)
unlist(x[-3]) ## Extract all elements except number 3 and turn into a vector

## [1] 1 2 5

x[3] <- NULL ## Remove list element 3. Only works for lists
unlist(x)

## [1] 1 2 5

numeric(4)

## [1] 0 0 0 0

logical(2)

## [1] FALSE FALSE
```

Formulas for linear models

- R has many code packages implementing various types of generalised linear models
- The most basic is `lm()`, which estimates models of the form

$$y_i = \sum_{k=0}^{K} z_{ik}\beta_k + e_i \quad \text{(elementwise definition)}$$

$$\boldsymbol{y} = \boldsymbol{Z}\boldsymbol{\beta} + \boldsymbol{e} \quad \text{(matrix form)}$$

- In R, the `formula` object type is used to define linear model structures

```
form1 <- y ~ covariate ## y_i = beta_0 + covariate_i beta_k
form2 <- SnowDepth ~ Temperature ## SnowDepth_i = beta_0 + Temperature_i beta_T
str(form <- c(form1, form2))
```

```
## List of 2
##  $ :Class 'formula'  language y ~ covariate
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ :Class 'formula'  language SnowDepth ~ Temperature
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

Unless $\sim$ `-1` is used, the *intercept* is included automatically.

More advanced vectorisation

- `lm()` is not a basic mathematical function that can be applied to a list of formulas
- We can vectorise model estimation by applying arbitrary functions to each element of a list, with `lapply()`
- For simpler cases, `vapply()` does the same for each element of a vector object.

```
mydata <- data.frame(SnowDepth, Temperature, ShoeSize)
form <- c(SnowDepth ~ Temperature, SnowDepth ~ ShoeSize)
models <- lapply(form, function(x) lm(x, data = mydata))
lapply(models, function(x) x$coef)

## [[1]]
## (Intercept) Temperature
##   2.5000000  -0.1285714
##
## [[2]]
##  (Intercept)     ShoeSize
## 1.000000e+00 3.172727e-16
```

Functions

- In the previous example, we used temporary *functions* to simplify the code.
- A common use of functions is to structure the code into more easily understood pieces, while simultaneously hiding complexity from the code that *calls* the functions.
- A major benefit is the reduced need for *copy&paste*, and the ease of *changing* the global behaviour of a script by only modifying code inside a function.

```r
lm_list <- function(x, ...) { # ... are parameters to pass through
  lapply(x, function(x) lm(x, ...)) # Local variable names override others
}
## Use the function:
models <- lm_list(form, data = mydata)
lapply(models, function(x) x$coef)

## [[1]]
## (Intercept) Temperature
##   2.5000000  -0.1285714
##
## [[2]]
##  (Intercept)     ShoeSize
## 1.000000e+00 3.172727e-16
```

Plotting functions

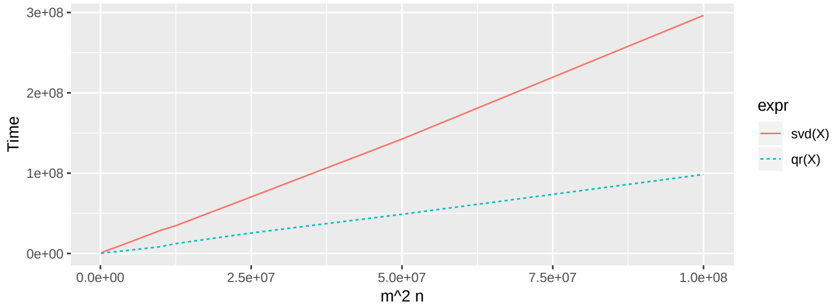
- The simplest plotting mechanism is called *base graphics*.
- Plot a vector or vectors: `plot(x)`, `plot(x, y)`
- Plot a curve from an expression: `curve(cos(x), left, right)`
- Common optional parameters: `plot(x, ...)`
  - Override default axis limits:
    `xlim=c(left, right)`, `ylim=c(lower, upper)`
  - Override default axis labels:
    `xlab="Temperature (c)"`, `ylab="Snow depth (mm)"`
- Useful helper functions:
  Data range: `range(x)` $= [\min(x_i), \max(x_i)]$
  Sequences: `seq(`*`start`*`, `*`end`*`, length.out=`*`length`*`)`
- Add points or lines to an existing figure:
  `points(x, y, ...)`, `lines(x, y, ...)`
  `abline(`*`intercept`*`, `*`slope`*`, ...)`,
  `abline(h = `*`vertical location of horizontal line`*`, ...)`,
  `abline(v = `*`horizontal location of vertical line`*`, ...)`
- Common optional parameters:
  `col=`*`colour name`*, `lty=`*`line type nr`*, `pch=`*`point symbol nr`*

RStudio recommendations

- Use *projects* (*Organisation*)
- Write code in a script file so that you rerun it easily (*Reproducibility*)
- Use keyboard shortcuts to run code from the file:
  (don't copy paste; high risk of *cut* and or):
  Ctrl-Enter runs the current line and steps to the next
- Lab 1 deals with general R coding and evolving structured programs
- Later we'll look at tools to help debug code with errors
- Recommended (not enforced) style guide for R code:
  `http://style.tidyverse.org/`
- `install.packages("styler")` gives a nice "Addins" tool to RStudio that can reformat code to fit the style guide. RStudio also has builtin code checking that can be activated to give hints in the margin about potential problems.
- Use `?name` or *Help* pane search to see the documentation/help/examples for a function!

Computational cost

- For large models, computational speed and memory usage are vital issues.
- Choosing the right algorithm can mean the difference of waiting for a few seconds and waiting for several weeks!
- Example: X is a n-by-m matrix, $n > m$. Compute the SVD and QR factorisations.



Both methods take $\propto m^2 n$ operations to compute, but QR is consistently around a factor 3 faster than SVD.
In R, the `lm()` function uses QR decomposition internally to solve a least squares problem.