Statistical Programming

Robin Evans

Hilary Term 2018

Administration

You can access the course page from my website.

`http://www.stats.ox.ac.uk/~evans/teaching.htm`

I will post these slides and the practical worksheets there.

You will also find my old MSc R Programming course notes there, which may be useful.

Using R

To obtain R, start by going to `cran.r-project.org`

There are various interfaces for using R that you can choose from; I recommend RStudio or R GUI.

- **RStudio**. Powerful, well thought out and widely used, works on all major operating systems. Getting better all the time.
- **R GUI** (*Windows, OS X*). Simple, robust, effective.
- **Emacs Speaks Statistics** (*Linux*). Only recommended if you're already a huge Emacs fan: can be powerful if you know what you're doing.
- **Command Line** (*OS X, Linux*). The minimalist's choice.

Getting Help

R has extensive documentation accessed with the `?` command, but it's not always that easy to read.

There are a huge number of other resources available. If you get stuck, try searching for answers online. Google is your friend!

Books are a bonus, but not necessary.

- *First Course in Statistical Programming with R* by Braun and Murdoch.
- *Introductory Statistics with R* by Dalgaard.
- *Modern Applied Statistics with S* by Venables and Ripley. Classic text, lots of statistical examples, and data sets are available in the `MASS` package.

This Course

The Statistical Programming lectures are designed to get you comfortable with using R, and familiar with using computers to solve numerical problems in a statistical context.

Here are some tips on making the most of the course.

- Learning R isn't too hard, but to be successful you **must practice**!
- **Don't just read** these notes, type the commands into R yourself. Play around with them. It may be helpful to have a laptop open while I'm lecturing.
- Try to **understand why** a sequence of commands does what it does, otherwise it will be impossible to reproduce them.
- Learn to 'think like a computer'. The abstraction of mathematics is very useful, but the computer has to solve lots of practical problems that we usually ignore. It's important to know how it does this.

The Command Line

R is a **scripting language**, which means it can run commands as soon as you type them. This is done at the **prompt**, represented by > .

When you type a complete command at the prompt and press enter, it runs, does any computation, and prints the answer.

```
> 3 + 4

[1] 7
```

If you start a command but don't finish it, the prompt changes to a +, indicating that it expects more:

```
> 5*(3 + 3
+ )

[1] 30
```

If this happens by mistake, **press escape** to cancel the command.

Arithmetic

The symbols +, −, ∗, /, ^, () do the usual things in the usual order

```
> 6 + 9 - 3
[1] 12

> (10 + 2) / 3
[1] 4

> 3^-1
[1] 0.3333333
```

You can also use %% for modular arithmetic and %/% for integer division

```
> 12 %% pi    # pi is a built-in constant
[1] 2.575222
```

Variables

You can define and set variables with = or <-. Typing an object's name at the command line causes it to be printed.

```
> x <- 15  # set x to be 15
> x - 1

[1] 14

> y = 4
> y

[1] 4

> x <- x + y  # LHS set to old value of RHS
> x

[1] 19
```

Vectors

Vectors are very important in R, and easy to work with. To create a vector, use `c()` separating entries by commas:

```
> x <- c(1,4,9)
> x
[1] 1 4 9
```

You can also paste together vectors

```
> c(x, 5, x)
[1] 1 4 9 5 1 4 9
```

and select particular elements

```
> x[3]        # third element of x
[1] 9
```

Use `length(x)` to get the total number of elements.

Sequences

You can get some simple sequences as a vector using `:`

```
> -2:5
[1] -2 -1  0  1  2  3  4  5

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

`seq()` gives general arithmetic sequences

```
> seq(2, 4, by=0.5)
[1] 2.0 2.5 3.0 3.5 4.0

> seq(to=4, by=0.5, length.out=5)
[1] 2.0 2.5 3.0 3.5 4.0
```

You can specify any three of `from`, `to`, `by`, `length.out`.

Functions

`seq()` is an example of a **function**. Everything in R is done with functions, even things that don't look like functions (e.g. `:`, `?`).

If you type a function's name you can see its code.

```
> seq

function (...)
UseMethod("seq")
<bytecode: 0x7f8d45793060>
<environment: namespace:base>
```

`seq` is a **method**, so it does different things depending on what sort of object you give it as input. Try typing `seq.default` to see what `seq` does for most cases.

Many basic functions in R use lower-level code for speed. To find out what a function does, use `?`

```
> ?seq
```

Replicating

Besides `seq()`, another useful command is `rep()`:

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

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

> rep(1:4, each=3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4

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

Vectorization

R uses pointwise arithmetic for vectors:

```
> c(1,2,3) + c(10,20,40)
[1] 11 22 43

> c(1,2,3)*c(10,20,40)   # not a dot product!
[1]  10  40 120

> 2^(0:9)
 [1]   1   2   4   8  16  32  64 128 256 512
```

R will ‘recycle’ shorter vectors to match longer ones:

```
> c(1,2) + c(10,20,30,40)
[1] 11 22 31 42
```

These are both features of R’s **vectorization**.

Some Other Useful Mathematical Functions

`exp, log, log2, log10`

`sqrt, abs, min, max`

`sin, cos, tan, asin, acos, atan`
`sinh, cosh, tanh, asinh, acosh, atanh`

`sum, prod, cumsum`

Most of these are vectorized.

```
> x <- c(0, pi/2, pi)
> sin(x)

[1] 0.000000e+00 1.000000e+00 1.224647e-16
```

Notice that if numbers are different orders of magnitude they are given in scientific notation: `1.224647e-16` is 1.22×10^{-16}.

Random Numbers

There are a lot of functions for generating independent random numbers. `runif()` gives uniforms, and `rnorm()` normals.

```
> runif(10)

 [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
 [6] 0.0455565 0.5281055 0.8924190 0.5514350 0.4566147

> rnorm(8)

[1]  1.7150650  0.4609162 -1.2650612 -0.6868529 -0.4456620
[6]  1.2240818  0.3598138  0.4007715
```

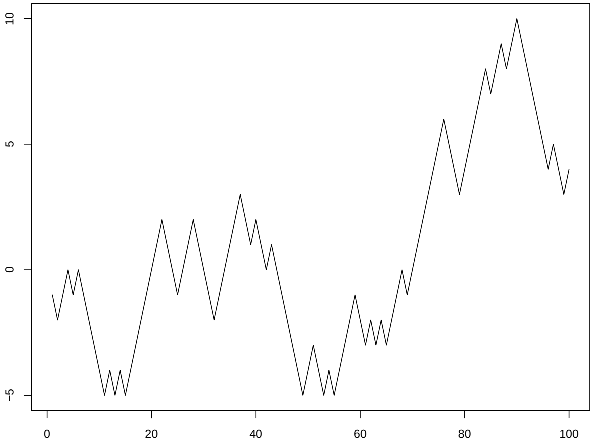
Computers generate **pseudorandom** numbers by applying a complicated function to some seed quantity, usually the exact time. For most purposes this is fine. You can 'fix' the random seed for replication:

```
> set.seed(10529)   # put any positive integer
```

Random Walks

sample() takes samples from a vector

```
> x <- sample(c(1,-1), 100, replace=TRUE)
> s <- cumsum(x)
> plot(s, type="l")  # that's a lowercase letter L
```



Logic

So far we have only seen numeric vectors, but there are also **logical** vectors, which contain `TRUE` or `FALSE`.

```
> x <- -2:3
> x^2 < 3

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

We can manipulate using `|` (or), `&` (and), `!` (not):

```
> (x^2 < 3) | (x == 3)

[1] FALSE  TRUE  TRUE  TRUE FALSE  TRUE

> !(x^2 < 3)

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

You can get the indices of true values using `which()`:

```
> which(x^2 < 3)

[1] 2 3 4
```

Comparisons

Logical Comparisons:

`==`	is equal to
`!=`	is not equal to
`<`	is less than
`>=`	is greater than or equal to

Logical Operators:

`&`	and
`|`	or
`!`	not

These are all vectorized, but there are versions of ‘and’ and ‘or’ that are not: `&&` and `||`.

Subsetting

It's extremely useful to be able to extract particular elements of a vector. There are several ways to do this:

```
> x <- 2^(0:9)
> x[c(2,4,5)]  # vector of indices elements

[1]  2  8 16

> x[-c(1:3)]   # indices not wanted

[1]   8  16  32  64 128 256 512
```

```
> x[x < 100]   # vector of logicals

[1]  1  2  4  8 16 32 64
```

A vector of logicals should be the same length, otherwise R recycles

```
> x[c(TRUE, FALSE)]   # vector of logicals

[1]   1   4  16  64 256
```

Matrices

R has built-in methods for working with matrices.

```
> matrix(1:6, 3, 4)

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

Notice:

- the entries go down the first column, only then across to the next: this is called **column major order**;
- the number of rows is specified first in the order of arguments;
- entries in the vector are recycled to fill the matrix.

The first index of the entries a_{ij} is changing fastest; in an $r \times c$ matrix, the order is

$$a_{11}, a_{21}, \ldots, a_{r1}, \quad a_{12}, a_{22}, \ldots, \quad a_{1c}, \ldots, a_{rc}.$$

Matrices

Matrices use pointwise arithmetic as though they were vectors.

```
> A <- matrix(1:6, 3, 4)
> b <- c(2,4,6,8)
> A*b

     [,1] [,2] [,3] [,4]
[1,]    2   32    6   16
[2,]    8   10   16   30
[3,]   18   24    6   48
```

To get **matrix multiplication**, use %*%.

```
> A %*% b

     [,1]
[1,]   56
[2,]   76
[3,]   96
```

Matrices

Since matrices have rows and columns we usually specify their entries with two co-ordinates. But because they are also vectors, we can use just one...

```
> A[2,3]

[1] 2

> A[8]  # same entry as (2,3)

[1] 2

> A[2,]  # leave entry blank to get everything

[1] 2 5 2 5
```

```
> dim(A)

[1] 3 4
```

Try also: `length(A)`, `nrow(A)`, `ncol(A)`.

Lists

A **list** is a bit like a vector, but the entries are completely arbitrary, and don't have to be a single number. They can be vectors themselves, or even other lists.

```
> mylist <- list(TRUE, 1:4, list())
> mylist

[[1]]
[1] TRUE

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

[[3]]
list()
```

Most more complicated objects in R are lists, possibly with some extra structure added.

Lists

You can access single entries in a list using double brackets:

```
> mylist[[2]]

[1] 1 2 3 4
```

To get a sublist, use single brackets as for a vector:

```
> mylist[c(1,3)]

[[1]]
[1] TRUE

[[2]]
list()
```

Functions

A function in R is much like a mathematical function. It takes inputs (**arguments**) and then applies some code to them (**the body**). From this it obtains an output (the **return** value).

```
addNumbers <- function(x, y) {
  # body of code here
  z <- x + y
  return(z)
}
addNumbers(3,5)
```

```
[1] 8
```

Function for $\|\cdot\|_p$

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

```
## Function to calculate p-norm of a vector
p_norm <- function(x, p=2) {
    modx <- abs(x)
    z <- sum(modx^p)
    return(z^(1/p))
}
p_norm(c(3,4))
```

```
[1] 5
```

```
p_norm(c(3,4), 1)
```

```
[1] 7
```

Note that if `p` not specified the default `p=2` used.

Unless told otherwise, a function returns the value of the **last statement evaluated**, so could replace `return(z^(1/p))` by just `z^(1/p)`.

Planning a Function

Think through how you expect the computer to behave before starting to write a function. Things that seem trivial (such as special cases) need to be considered carefully.

Let's write a function to extract every other element of a vector (the even numbered elements). We need to:

- find the length of the vector;
- construct a sequence of even numbers up to that length;
- use subsetting on the sequence.

```
## Function to get even numbered entries in vector
evenElements <- function(x) {
  len <- length(x)
  sq <- seq(from=2, to=len, by=2)
  return(x[sq])
}
```

```
> evenElements(c(1,3,5,7,9,11))

[1]  3  7 11
```

Planning a Function

What happens if the vector has length 0 or 1?

```
evenElements(4)

Error in seq.default(from = 2, to = len, by = 2):  wrong
sign in 'by' argument
```

We need to deal with the special cases:

```
## Function to get even numbered entries in vector
evenElements <- function(x) {
  len <- length(x)
  if (len < 2) return(numeric(0))
  sq <- seq(from=2, to=len, by=2)
  return(x[sq])
}
evenElements(4)

numeric(0)
```

`numeric(0)` is a numeric vector of length 0.

Flow Control

There are three main forms of **flow control** in R. That is, methods for getting your code to do particular things depending upon the situation it is in and the inputs it receives.

- `if()` and `if()...else` for conditional code.
- `for()` if code is to be run a given number of times.
- `while()` if code is to be run until a condition is met.

These are standard to most programming languages.

Caveat:

- There is often more than one way to do something in R.
- Using loops (i.e. `for()` and `while()`) is often not the quickest way, either to code or to run.

Flow Control: `if()`

Conditional code is only executed if a particular condition is met. The most basic way to do this is with an `if() else` statement.

```
mod <- function(x) {
  if (x < 0) {
    out <- -x
  }
  else {
    out <- x
  }
  out
}
mod(-4)

[1] 4
```

The condition in brackets must be a **single** logical value (`TRUE` or `FALSE`). Vectorization makes it easy to give a vector by mistake, in which case you'll see the error:

`'the condition has length > 1 and only the first element will be used'`

`for()` loops

Computer code often involves simple but laborious repetition.

```
factorial2 <- function(n) {
  out <- 1
  for (i in 1:n) {
    out <- out*i
  }
  out
}
factorial2(10)
```

```
[1] 3628800
```

The syntax is always `for (x in y)`, where `x` is a variable name to be used as a counter, and `y` is a vector to iterate over.

The code is repeated once for each entry in `y`, with `x` taking that value.

Note: in R, there are very often much better ways to do things than with a `for()` loop. What does `prod()` do?

`while()` loops

If a piece of code needs to be executed for an arbitrary number of steps until a condition is met, it may make more sense to use a `while()` loop.

```
rTruncNorm <- function(a) {
  z <- a - 1
  while (z < a) {
    z <- rnorm(1)
  }
  z
}
rTruncNorm(2)

[1] 2.338004
```

Why wouldn't this work well if I type `rTruncNorm(10)`?

Flow Control: `break`

The command `break` can be used to escape from a `for()` or `while()` loop early.

```
isPr <- function(x) {
  M <- floor(sqrt(x))   # what does this do?
  out <- TRUE
  for (i in seq(2, M)) {
    if (x %% i == 0) {
      out <- FALSE
      break
    }
  }
  out
}
isPr(5)

[1] TRUE

isPr(8)

[1] FALSE
```

Algorithms

Functions themselves can be passed as arguments to other functions.

```
newtonRaphson <- function(f, f.prime, x, tol = 1e-8) {
  #Newton-Raphson iteration for f
  while (abs(f(x)) > tol) {
    x = x - (f(x) / f.prime(x))
  }
  return(x)
}

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

newtonRaphson(f, f.prime, 2)
```

```
[1] 1.428818
```

Scope

A very important issue in programming in any language is the issue of **scope**. In order to make functions easier to understand on their own, they are allowed to give the same name to their own 'local' variables as other functions do for different variables.

Any local copies are destroyed as soon as the function finishes running. In order for a value to escape it must be **returned**.

```
add <- function(x, y) {
  z <- x + y
  return(z)
}
add(3,5)
```

```
[1] 8
```

```
z
```

```
Error in eval(expr, envir, enclos):  object 'z' not found
```

R functions only return a single item, but you can use a list if you need more than one variable.

Scope

```
> a <- 5
>
> f <- function() {
+   cat("a =", a, "\n")  # cat() prints out its arguments
+   a <- a + 1
+   cat("a =", a, "\n")
+   return(a)
+ }
>
> f()

a = 5
a = 6
[1] 6


> a

[1] 5
```

`f`'s local copy of `a` was set to 6, but the value of the original is still 5.

Note: **do not** use printing as a substitute for returning a value!

Modularity

Functional programming means programming based on functions that can be understood and used on their own.

It can be very difficult to understand what a function does if it makes substantial use of variables that are defined in a higher level frame.

- pass all variables used **explicitly** as arguments;
- your functions will be easier to understand;
- bugs will be easier to find, and code easier to reuse later;

If your code does the same thing in more than one place, do it by **calling a function**. This means that

- you only have to write the code once;
- it's less likely you'll make a mistake, and easier to find it if you do;
- if you need to change the code you can do it in one place;
- it's easier to read the code.

Bad

```
n <- 10

simNorms <- function(mu, sigma) {
  rnorm(n, mu, sigma)
}

doMCMC <- function() {
  k <- 1e3
  for (n in 1:k) {
    ### some code...
    x <- mean(simNorms(0,1))
  }
  x
}
```

This sort of code is particularly common with data sets (see next time).

Better

```
n <- 10

simNorms <- function(n, mu, sigma) {
  rnorm(n, mu, sigma)
}
```

```
doMCMC <- function() {
  k <- 1e3
  for (n in 1:k) {
    ### some code...
    x <- mean(simNorms(n=100,0,1))   # pass explicit argument
  }
  x
}
```

Readability

Other tips to make code easy to follow:

- **Indent** your code. You can do indenting automatically in RStudio with Ctrl+I (or Cmd+I).
- Use sensible **spacing**:

```
x<-1
x < -1
x <- 1
```

- use proper **commenting**; if you comment as you write it's no chore.

Comments

```r
## Function to find roots of functions by Newton-Raphson method
##
## Inputs:
## f       : function of one argument whose root to be found
## f.prime : derivative of f
## x       : starting point for algorithm
## tol     : numerical tolerance for solution
##
## Return:
## numerical root of f
##
newtonRaphson <- function(f, f.prime, x, tol = 1e-8) {
  ## while f is (numerically) non-zero, iterate
  while (abs(f(x)) > tol) {
    ## perform a Newton-Raphson step
    x = x - (f(x) / f.prime(x))
  }
  return(x)
}
```

Another Scoping Example

Can you see what's happening here?

```
z <- 0
f1 <- function () {
  # you can define functions inside other functions
  f2 <- function() {
    print(z)
  }
  z <- 1
  f2()
}
f1()
```

```
[1] 1
```

What would change if `f2()` were defined outside `f1`?

More functions

In Practical 2, Question 6, I ask you to write some code to simulate a normal by rejection from a double exponential. Recall the algorithm

1. simulate Y from $f_Y(y) \propto \exp(-|y|)$ and $U \sim U(0,1)$

2. if $U < \exp(-Y^2/2 + |Y| - \frac{1}{2})$ accept $X = Y$ and stop. Otherwise repeat 1.

We break this job down into manageable pieces.

Smaller Function

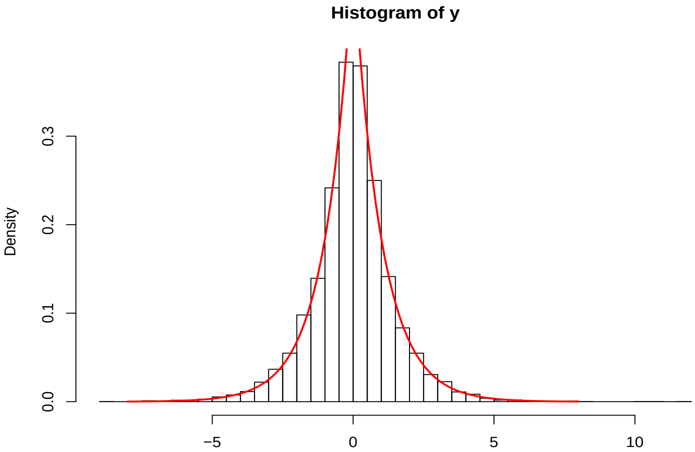
First write a function to sample Y from the density $f_Y(y) \propto \exp(-|y|)$, and test it!

```
r_double_exp <- function(n=1) {
  # simulate exp(-|x|)
  X <- log(runif(n))
  Y <- sample(c(-1,1), n, replace=TRUE)

  return(X*Y)
}

r_double_exp(n=4)
```

```
[1]  1.2712885 -4.3631841 -0.5586488 -0.2053247
```

Testing

```
y <- r_double_exp(1e4)
hist(y, breaks=50, freq=FALSE)
f <- function(x) exp(-abs(x))/2
plot(f, -8, 8, col=2, lwd=2, add=TRUE)
```



Rejection Sampler

Then write a function implementing the rejection sampler.

```
## simulate random normal by rejection
my_rnorm <- function() {
  finished <- FALSE

  while (!finished) {
    ## keep simulating double exponentials
    ## until rejection condition is satisfied
    y <- r_double_exp(n = 1)
    u <- runif(n = 1)
    p_over_Mq <- exp(-y^2/2 + abs(y) - 0.5)
    finished <- (u < p_over_Mq)
  }  # while (!finished)
  return(y)
}
```

Some Principles of Good Coding

- **modularity**: break your code up into self-contained functions, and avoid repeating code to perform the same job;
- **information hiding**: don't let a function have information it doesn't need;
- careful **planning**: try writing some pseudocode down first to see how you should structure your program;
- **commenting**: the most likely collaborator is yourself in the future—treat them nicely;
- give variables **meaningful names**, it makes it much easier to read code.

Loops

Try to avoid loops when you code in R. They often run slowly and may be harder to read.

```
# get the sum of the first n square numbers
sumSquare <- function(n) {
  tot <- 0
  for (i in 1:n) tot <- tot + i^2
  tot
}
```

Try to find a shorter, vectorized way:

```
sumSquare2 <- function(n) {
  sq <- seq_len(n)
  sum(sq^2)
}
```

apply()

The `apply()` function is useful in this context. It applies a function to every row (or column) in a matrix-like object.

```
> X <- matrix(c(1,2,3,4),2,2)
> X

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

> apply(X, 1, sum) #sum rows

[1] 4 6
```

```
> apply(X, 2, max) #sum columns

[1] 2 4

> # how many entries in each column less than 2.5?
> apply(X, 2, function(x) sum(x<2.5))

[1] 2 0
```

Simpler Code

We can often use `apply()` to replace for-loops

```
# function to simulate n Gamma(a,b) r.v.s for a integer
my_rgamma <- function(n, a, b) {
  X <- numeric(n)

  for (i in 1:n) {
    tot <- 0
    for (j in (1:a)) {
      tot <- tot - log(runif(1))/b
    }
    X[i] <- tot
  }
  X
}

my_rgamma2 <- function(n, a, b) {
  Us <- matrix(runif(a*n), a, n)
  M <- -log(Us)/b
  X <- apply(M, 2, sum)
  X
}
```

Object Types in R

We have already seen two kinds of data in R, `numeric` and `logical`. If you want to know what sort of object something is, you can use the `class()` function:

```
class(pi)

[1] "numeric"

class(TRUE)

[1] "logical"
```

Another important type is `character`, i.e. bits of text.

A **string** is a piece of text surrounded by quotes.

```
x <- "Hello"
x

[1] "Hello"

class(x)

[1] "character"
```

Character Vectors

We can form vectors of `character` objects just as with numbers:

```
x <- c("This", "is", "the first", "day",
       "of the rest", "of your life!")
x[3]
```

```
[1] "the first"
```

There are functions that can be used to manipulate strings:

```
paste(x, collapse=" ")
```

```
[1] "This is the first day of the rest of your life!"
```

To display a string nicely on the screen, use `cat()`

```
cat(x, sep=" ")
```

```
This is the first day of the rest of your life!
```

Factors

Factors are vectors that contain data on categories. The different categories are called **levels**. For example:

- a factor Gender with levels 'male' and 'female';
- a factor Age with levels 'under 25', '25-39', '40-54', '55+'.

R treats factors differently to character vectors.

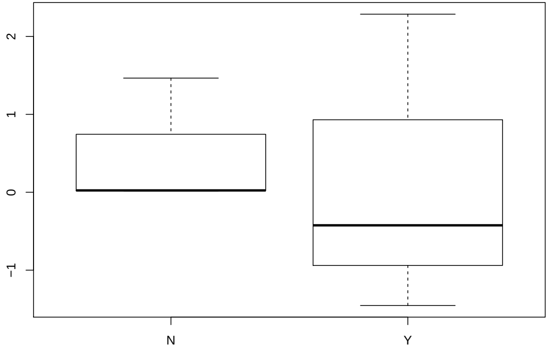
```
> agree <- c("Y", "N", "Y", "Y", "N", "N")
> Agree <- factor(agree)
> agree

[1] "Y" "N" "Y" "Y" "N" "N"

> Agree

[1] Y N Y Y N N
Levels: N Y
```

Factors

```
class(Agree)
```

```
[1] "factor"
```

```
x <- rnorm(6)
plot(Agree, x)
```



External Data

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

```
glucose conc diameter
1 631000 21.2
1 592000 21.5
1 563000 21.3
1 475000 21
1 461000 21.5
1 416000 21.3
1 385000 20.3
1 321000 22.7
1 302000 21.5
1 199000 22.2
...
```

The cell concentration (`conc`) was set at the beginning of the experiment and the average cell diameter (`diameter`) was measured for two groups of cell cultures where glucose was either added (`glucose = 1`) or not added (`glucose = 2`) to the growth medium.

Reading In Data

We can use the `read.table()` function to read it. We can read from the internet or from a local file. If we use a local file we have to give the path, or set the working directory os the directory containing the file.

This table has a header line, so we tell R to expect a header.

```
hd <- read.table("hellung.txt", header=TRUE)
```

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

How many observations are there, and what are the variable names?

```
dim(hd)

[1] 51  3

names(hd)

[1] "glucose"  "conc"     "diameter"

length(hd)

[1] 3
```

Data Frames

A data frame looks like a matrix, but is really a list.

```
head(hd, 3)
```

```
  glucose   conc diameter
1       1 631000     21.2
2       1 592000     21.5
3       1 563000     21.3
```

```
hd$diameter[1:10]
```

```
 [1] 21.2 21.5 21.3 21.0 21.5 21.3 20.3 22.7 21.5 22.2
```

It can be subsetted like a matrix.

```
hd[2,2]
```

```
[1] 592000
```

```
hd[3:4,]
```

```
  glucose   conc diameter
3       1 563000     21.3
4       1 475000     21.0
```

Summaries

```
str(hd)
```

```
'data.frame': 51 obs. of  3 variables:
 $ glucose : int  1 1 1 1 1 1 1 1 1 1 ...
 $ conc    : int  631000 592000 563000 475000 461000 416000 385000 3210
 $ diameter: num  21.2 21.5 21.3 21 21.5 21.3 20.3 22.7 21.5 22.2 ...
```

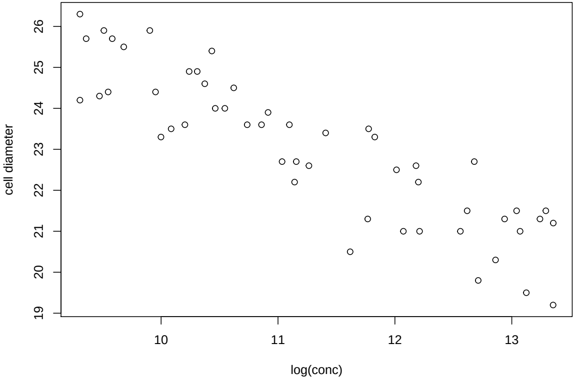
```
summary(hd)
```

```
    glucose           conc           diameter
 Min.   :1.000   Min.   : 11000   Min.   :19.20
 1st Qu.:1.000   1st Qu.: 27500   1st Qu.:21.40
 Median :1.000   Median : 69000   Median :23.30
 Mean   :1.373   Mean   :164326   Mean   :23.00
 3rd Qu.:2.000   3rd Qu.:243000   3rd Qu.:24.35
 Max.   :2.000   Max.   :631000   Max.   :26.30
```

Plotting

Show me the data!

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



Scope in Data Frames

You must type `hd$glucose` to get the `glucose` numbers.

```
plot(log(conc), diameter)
plot(log(hd$conc), hd$diameter)
```

The `with()` command can be used to allow access to the dataframe's environment.

```
with(hd, plot(log(conc), diameter))  # this works
```

In many commands you can specify where to look for the variables, using the `data=` option. This is often easier read, and shows explicitly which data are being used.

Applying functions over data, and subsetting

The `apply()` command is also very useful for applying functions to *all* rows/columns of a matrix or data frame. For example,

```
apply(hd, 2, max)
```

```
 glucose     conc diameter
     2.0 631000.0     26.3
```

Sometimes we want to work on subsets of a data frame. This works the same way as with vectors and matrices.

```
mean(hd$diameter[hd$glucose==1])
```

```
[1] 23.50625
```

```
hd[hd$conc > 550000,]
```

```
   glucose   conc diameter
1        1 631000     21.2
2        1 592000     21.5
3        1 563000     21.3
33       2 630000     19.2
```

Setting Values and Modifying Data Frames

We can use the subsetting to set values as well as to get them:

```
hd[1,3] <- 21
hd[1,]
```

```
  glucose   conc diameter
1       1 631000       21
```

We can add variables

```
hd$random1 <- rnorm(51)
head(hd, 3)
```

```
  glucose   conc diameter    random1
1       1 631000     21.0 -0.6883676
2       1 592000     21.5  0.4512159
3       1 563000     21.3 -0.9118234
```

Another way:

```
hd <- cbind(hd, random2 = rnorm(51))
```

Missing Data

Missing data are common in statistics, so R has its own special type for dealing with them. Missing values are represented in R as `NA`.

```
x <- c(NA, 5, 9, NA, 7)
x
```

```
[1] NA  5  9 NA  7
```

Many functions have an argument `na.rm` to choose to ignore missing values.

```
mean(x)
```

```
[1] NA
```

```
mean(x, na.rm=TRUE)
```

```
[1] 7
```

Data summaries

Here are some useful functions.

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

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

Computational Cost

Computations that are mathematically equivalent are not necessarily computationally equivalent. Consider

$$\sum_{i=1}^{k}\sum_{j=1}^{l} a_i b_j = (a_1 + \cdots + a_k)(b_1 + \cdots + b_l).$$

Computing the left-hand side involves kl multiplications and $kl - 1$ additions. The right-hand side $k + l - 2$ additions and 1 multiplication.

If $k = l = 10^6$ then this really matters!

Computational Cost

We tend to measure complexity in terms of the number of arithmetic operations (+, −, ∗, /) and just take the leading terms.

We are usually interested in the leading term of computational cost. Recall that

$$f(n) = O(g(n)) \quad \text{means} \quad \frac{f(n)}{g(n)} \leq M, \quad \text{for sufficiently large } n.$$

The operations on the previous slide have respective complexity $O(kl)$ and $O(k+l)$.

We can consider either:

- **worst case cost**: gives an upper bound on how long a problem can take to solve;
- **average case cost**: perhaps more useful, but often much harder to compute (and sometimes to define).

Example: Matrix Multiplication

Multiplying matrices: let A be $n \times p$ and B be $p \times m$ matrices, and $C = AB$.

To calculate $c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$ needs $2p - 1$ operations.

So to calculate C is $nm(2p - 1) = O(pmn)$ operations.

What does that mean if we have to calculate $AB\boldsymbol{x}$ for $A, B \in \mathbb{R}^{n \times n}$ and $\boldsymbol{x} \in \mathbb{R}^n$?

$(AB)\boldsymbol{x}$	$A(B\boldsymbol{x})$
$O(n^3) + O(n^2)$	$O(n^2) + O(n^2)$
$O(n^3)$	$O(n^2)$

Again, if $n = 1000$ this is very important!

Matrices in R

Here are some useful functions for matrices in R.

Try them out and understand what they do.

```
A <- matrix(1:9, 3, 3)  # create matrix
dim(A)
t(A)                    # transpose of A
det(A)                  # determinant of A
A %*% c(3,5,7)          # matrix multiplication
A[2]
A[2,]                   # subsetting
A[2,,drop=FALSE]        # keeps answer as a matrix
rbind(1, A, A)          # see also cbind()

col(A)                  # matrix of column numbers
row(A)
diag(A)                 # diagonal entries
diag(A) <- c(9,9,4)
upper.tri(A)
A[upper.tri(A)] = 0     # what's happened to A?
```

Recursion

Recursive programmes call themselves.

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

$$f(0) = 1, \qquad f(x) = xf(x-1) \qquad \text{for } x \in \mathbb{N}.$$

Our factorial function returns $x! = 1$ on input $x = 0$ and otherwise calls itself to evaluate $(x-1)!$ and multiplies this by x.

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

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

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

Example: Determinant

```
getDet <- function(M) {
  out <- 0
  n <- nrow(M)
  if (n == 1) return(M[1,1])  ## base case n = 1

  for (i in 1:n) {
    ## get determinants of each minor
    tmp <- M[1,i] * getDet(M[-1, -i, drop=FALSE])
    out <- out + (-1)^(i-1) * tmp
  }
  out
}
```

How complicated is this? Let $g(n)$ be complexity for $n \times n$ matrix:

$$g(n) = n(4 + g(n-1)),$$

so an inductive argument shows that, for any $\epsilon > 0$ we have $g(n) = O(\lceil n(1+\epsilon) \rceil !)$. This is 'slightly worse' than $O(n!)$.

This is **very high** complexity, and in fact is a very poor way of calculating the determinant, as we shall see.

Example: Cholesky Factorization

Recall simulation for the multivariate normal, $X \sim N_n(\mu, A)$, and A a $n \times n$ symmetric positive definite covariance matrix.

(Recall that A is positive definite if $x^T Ax > 0$ for all $x \neq 0$.)

Let L be a matrix such that

$$A = LL^T.$$

If $Z = (Z_1, Z_2, ..., Z_n)$ with $Z_i \sim N(0, 1), i = 1, 2, ..., n$ and we set

$$X = \mu + LZ,$$

then $X \sim N_n(\mu, A)$.

There are many choices for L. The **Cholesky decomposition** in which L is lower triangular is particularly neat. The existence of such an L is guaranteed by A being symmetric and positive definite.

L is also unique if we insist on the diagonal entries being positive.

Example: Cholesky Factorization

Here is a recursive algorithm for L. Chop A and L up into blocks

$$A = \left(\begin{array}{c|c} a_{11} & A_{21}^T \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} 1 \times 1 & 1 \times (n-1) \\ \hline (n-1) \times 1 & (n-1) \times (n-1) \end{array}\right)$$

Here $A_{21} = A_{2:n,1}$ is $(n-1) \times 1$ and $A_{22} = A_{2:n,2:n}$ is $(n-1) \times (n-1)$.

Similarly

$$L = \left(\begin{array}{c|c} \ell_{11} & 0_{1\times(n-1)} \\ \hline L_{21} & L_{22} \end{array}\right)$$

Since L is lower triangular:

- all the entries in the top row (except the first) are zero;
- L_{22} is also lower triangular.

Example: Cholesky Factorization

Since $A = LL^T$,

$$
\begin{aligned}
\begin{pmatrix} a_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} \ell_{11} & 0_{1\times(n-1)} \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \ell_{11} & {L_{21}}^T \\ 0_{(n-1)\times 1} & L_{22}^T \end{pmatrix} \\
&= \left(\begin{array}{c|c} \ell_{11}^2 & \ell_{11} {L_{21}}^T \\ \hline \ell_{11} L_{21} & L_{22} {L_{22}}^T + L_{21} {L_{21}}^T \end{array}\right)
\end{aligned}
$$

so $\ell_{11} = \sqrt{a_{11}}$, $L_{21} = A_{21}/\sqrt{a_{11}}$, and the A_{22} block gives

$$
\begin{aligned}
A_{22} - L_{21} {L_{21}}^T &= L_{22} {L_{22}}^T \\
\tilde{A} &= \tilde{L}\tilde{L}^T \qquad \text{this is } (n-1) \times (n-1)
\end{aligned}
$$

To solve for L_{22}, we need the Cholesky factorization of the $(n-1) \times (n-1)$ matrix $\tilde{A} = A_{22} - L_{21} {L_{21}}^T$, so we have reduced the problem by one dimension.

Finally, if $n = 1$ so A is a scalar, $L = \sqrt{A}$ terminates the recursion.

Runtime Analysis

How complicated is this method of finding a Cholesky decomposition?
Let $g(n)$ be the complexity for an $n \times n$ matrix.

calculation	operations
$\ell_{11} = \sqrt{a_{11}}$	1
$L_{21} = A_{21}/\ell_{11}$	$n-1$
$\tilde{A} = A_{22} - L_{21}L_{21}^T$	$2(n-1)^2$

This gives us

$$g(n) = 2(n-1)^2 + n + g(n-1) = \sum_{i=1}^{n} \left\{2(i-1)^2 + i\right\}.$$

Since $\sum_{i=1}^{n} i^2 = n(n+2)(2n+1)/6$ this implementation has approximately $g(n) \simeq 2n^3/3$ operations or $O(n^3)$.

If we had exploited symmetry we could get this down to about $n^3/3$ but we can't change the order (still $O(n^3)$).

Example: Determinant

Recall that, for a (lower or upper) triangular matrix L, the determinant is the product of the diagonal elements.

So, given the Cholesky decomposition $A = LL^T$, the determinant of A is just

$$\det A = (\det L)^2 = \prod_i \ell_{ii}^2.$$

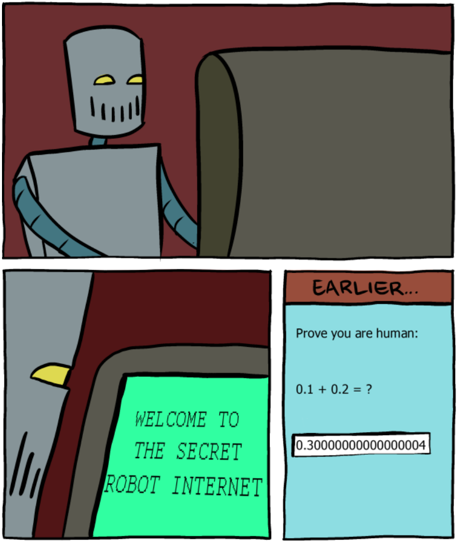
This only requires $O(n^3)$ calculations.

In fact, this is how R calculates determinants: in general it finds lower and upper triangular matrices L, U such that

$$A = LU$$

This is the **LU decomposition**, and we will use it again next time.

Rounding



www.smbc-comics.com/?id=2999

Numerical Precision

Computers store numbers as **floating point** objects.

This is a binary version of scientific notation, and consists of a **significand** and an **exponent**:

$$\underbrace{11000110}_{\text{significand}} \times 2^{-4} = 1100.0110 = 8 + 4 + \frac{1}{4} + \frac{1}{8} = 12.375$$

The significand is 53 bits long, so any significant digits that are more than this away from the leading digit will be lost.

```
(1 + 2^-52) - 1

[1] 2.220446e-16

(1 + 2^-53) - 1

[1] 0
```

Numerical Error

This means that we cannot expect numerical answers given by R to have more relative accuracy than about 10^{-15} at an absolute maximum.

```
> x <- 0.1 + 0.2 - 0.3
> x

[1] 5.551115e-17
```

This causes problems when testing for equality:

```
> x == 0

[1] FALSE
```

The `all.equal()` function can be used to deal with this (though it doesn't always return a logical).

```
> isTRUE(all.equal(x, 0))

[1] TRUE

> abs(x) < 1e-12

[1] TRUE
```

Solving linear systems

Many important numerical problems reduce to

$$\text{find} \quad x \quad \text{such that } Ax = b,$$

for $A \in \mathbb{R}^{n \times p}$ a matrix of rank p (so $p \leq n$), and $b \in \mathbb{R}^n$.

Suppose $p = n$, so that A is a square non-singular matrix.

R has a function `solve()` returning the inverse of a matrix.

```
> x <- solve(A) %*% b      ## so this does A^{-1} b
```

How does it work? We will see that the best method for finding x depends on the properties of A.

If $p < n$, then the system is **overdetermined**. We come back to this case later.

Forward Substitution

Suppose A is lower triangular so that $a_{ij} = 0$ for $i < j$.

We can solve $Ax = b$ for x using **forward substitution**.

Chop the n equations in $Ax = b$ into blocks

$$A = \begin{pmatrix} a_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix}$$

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

Now $Ax = b$ is

$$\begin{pmatrix} a_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_{2:n} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_{2:n} \end{pmatrix}$$

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

Forward Substitution

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

$$\begin{pmatrix} A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_{2:n} \end{pmatrix} = b_{2:n}$$

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

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

It took $2(n-1)+1$ subtractions, multiplications and divisions to solve for x_1 and calculate $\tilde{b}$. Since $\sum_{i=1}^{n}(2i-1) = O(n^2)$, forward solving uses $O(n^2)$ operations.

R has `forwardsolve(A,b)` for forward substitution on lower triangular `A`, and `backsolve(A,b)` for backward substitution on upper triangular `A`.

LU factorization

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

$$A = LU$$

into lower triangular L and an upper triangular U at a cost of $2n^3/3 + O(n^2)$ operations (we haven't proven this, it's just assertion) and then solving $L(Ux) = b$:

- find y such that $Ly = b$ (forwards);
- then find x such that $Ux = y$ (backwards).

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

(NB: If there is no LU factorization we seek $A = PLU$ with P a permutation. This always exists.)

Linear Models

Suppose we want to fit the normal linear regression model

$$Y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i, \quad i = 1, 2, \ldots, n.$$

with $(\beta_1, \ldots, \beta_p)$ unknown parameters and $\varepsilon_i \sim N(0, \sigma^2)$ i.i.d. normal errors.

In vector form the model is

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & & & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

or

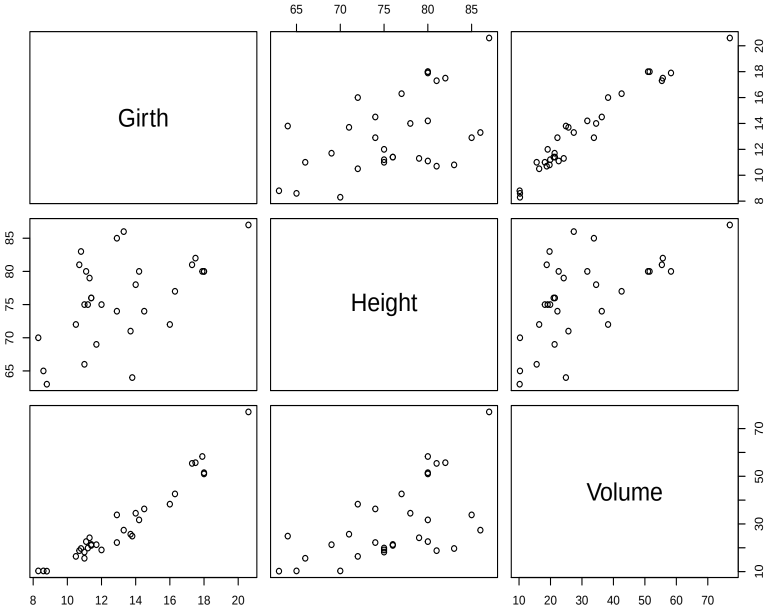
$$Y = X\boldsymbol{\beta} + \varepsilon.$$

Example: Trees Data

Consider the Trees data in R:

```
> data(trees)
> head(trees)

  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7

> nrow(trees)

[1] 31

> pairs(trees)
```

Trees Data



Girth
Height
Volume
65 70 75 80 85
8 10 12 14 16 18 20
65 70 75 80 85
10 30 50 70
8 10 12 14 16 18 20
10 20 30 40 50 60 70

Trees Data

Consider the model that the volume Y_i is a linear function of the height x_i and girth z_i.

The R commands to fit this normal linear model are

```
> lm1 <- lm(Volume ~ Height + Girth, data=trees)
> summary(lm1)
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -57.9877     8.6382  -6.713 2.75e-07 ***
Height        0.3393     0.1302   2.607   0.0145 *
Girth         4.7082     0.2643  17.816  < 2e-16 ***
```

Notice the R formula notation `Volume ~ Height + Girth`.

The columns of the `summary(lm1)` output give $\hat{\beta}_i$, an estimate $\mathrm{se}(\hat{\beta}_i)$ of the error of $\hat{\beta}_i$ (similar to s^2), and two columns for the t-test of $H_0 : \beta_i = 0$.

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

The equations $X\boldsymbol{\beta} = Y$ are *overdetermined* (more equations than variables, $n > p$, so we can't expect a solution),

Instead we can minimize

$$\begin{aligned} R(\boldsymbol{\beta}) &= (Y - X\boldsymbol{\beta})^T(Y - X\boldsymbol{\beta}) \\ &= \sum_{i=1}^{n}(Y_i - \beta_1 x_{i1} - \cdots - \beta_p x_{ip})^2 \end{aligned}$$

this gets $X\boldsymbol{\beta}$ as close as we can to Y.

The solution to the minimisation problem (see Examples Sheet 4) is $\boldsymbol{\beta}$ satisfying

$$X^T X\boldsymbol{\beta} = X^T Y.$$

This is a well determined set of p linear equations $Ax = b$ with $A = X^T X$, $x = \boldsymbol{\beta}$ and $b = X^T Y$.

In fact, the solution is also the MLE under Gaussian errors.

Fitting Linear Models

We could use LU factorization to solve the normal equations. However it is not very stable numerically. Suppose

$$X = \begin{pmatrix} 1 & -1 \\ 0 & 10^{-10} \end{pmatrix},$$

then

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

Now, at machine precision $1 + 10^{-20}$ and 1 are equal so $X^T X$ appears to be singular.

Any method (like LU) that solves $(X^T X)\beta = X^T Y$ by first computing $X^T X$ will fail on this problem.

QR factorization

Instead, we use **QR factorization**: the QR factorization of X is

$$X = QR$$

where

- Q is $n \times p$ and orthogonal;
- R is $p \times p$, upper triangular, and has positive entries on the diagonal.

This takes $2np^2$ operations (assertion). Then

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

and so the equations are

$$\begin{aligned} X^T X\boldsymbol{\beta} &= X^T Y \\ R^T R\boldsymbol{\beta} &= R^T Q^T Y \end{aligned}$$

QR For Linear Models

We can solve these by solving

$$R\beta = Q^T Y \qquad \text{(backwards)}$$

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

In R,

```
> X = cbind(1, trees$Height, trees$Girth)
```

followed by

```
> beta = qr.solve(X, trees$Volume)
> beta

[1] -57.9876589   0.3392512   4.7081605
```

gives the regression parameters.

Numerical Overflow

We saw last time that the precision of floating point numbers is finite and should be accounted for when performing numerical calculations.

In addition to this, there are limits on how large or small floating point numbers can be.

```
> 2^1023

[1] 8.988466e+307

> 2^1024

[1] Inf

> 2^(-1074)

[1] 4.940656e-324

> 2^(-1075)

[1] 0
```

This is called numerical overflow or underflow.

Likelihood Ratios

For most practical purposes you might think this doesn't matter.
However, we often deal with **products** of small or large quantities, which can quickly become too small or large:

Suppose $X_i \overset{\text{i.i.d.}}{\sim} N(\mu, \sigma^2)$.

$$L(\mu, \sigma^2; y_1, \ldots, y_n) = \prod_i f(y_i; \mu, \sigma^2),$$

```
> x <- rnorm(1000)
> Lx <- dnorm(x)
> min(Lx)    # likelihood for each observation is positive

[1] 0.0001071485


> prod(Lx)   # total likelihood appears to be zero

[1] 0
```

```
> sum(log(Lx))   # log-likelihood is managable

[1] -1469.746
```

Logarithmic Scales

The usual solution is to work on a log-scale whenever possible, and calculate things in a way that avoids potentially very large or small numbers.

```
> log(exp(1000) - exp(999) + exp(998) - exp(997))

[1] NaN
```

But of course,

$$\begin{aligned}\log(e^{1000} - e^{999} + e^{998} - e^{997}) &= \log\left\{(e^3 - e^2 + e^1 - 1)e^{997}\right\}\\ &= \log(e^3 - e^2 + e^1 - 1) + 997.\end{aligned}$$

So in reality, this is a perfectly managable number:

```
> log(exp(3) - exp(2) + exp(1) - exp(0)) + 997

[1] 999.6683
```

Logarithmic Scales

Lots of built-in functions in R either have 'log-scale versions' or optional arguments to obtain the logarithm of the output directly.

```
> log(factorial(500))

Warning in factorial(500):  value out of range in 'gammafn'

[1] Inf

> lfactorial(500)

[1] 2611.33
```

Similarly `gamma()` and `lgamma()` for the gamma function.

```
> dnorm(10, 0, 1)

[1] 7.694599e-23

> dnorm(10, 0, 1, log=TRUE)

[1] -50.91894
```

Recall the Metropolis Hastings MCMC algorithm

Consider an MH algorithm targeting $p(x) = \tilde{p}(x)/Z_p$ using proposal $q(y|x)$.

1. Set $X_0 = x_0$ (such that $p(x_0) > 0$).

2. For $t = 1, \ldots, N$: Let $X_t = x$.

   2.1 Draw $y \sim q(\cdot|x)$ and $u \sim U[0,1]$.

   2.2 Set
$$\alpha(y|x) = \min\left\{1, \frac{\tilde{p}(y)q(x|y)}{\tilde{p}(x)q(y|x)}\right\}$$

   2.3 If $u \leq \alpha(y|x)$ then set $X_{t+1} = y$, otherwise set $X_{t+1} = x$.

We initialise this with $X_0 = x_0$ (such that $p(x_0) > 0$) and iterate for $t = 1, 2, \ldots, n$.

Bayesian Posterior Distributions

In practice we are often interested in sampling from a posterior distribution $\pi(\theta \mid \boldsymbol{z}) \propto \pi(\theta)L(\theta; \boldsymbol{z})$ where $\pi(\theta)$ is the prior and $L(\theta; \boldsymbol{z})$ the likelihood given some data $\boldsymbol{z} = (z_1, \ldots, z_n)$.

The acceptance probability becomes

$$\alpha(\theta'|\theta) = \min\left\{1, \frac{\pi(\theta')L(\theta'; \boldsymbol{z})q(\theta|\theta')}{\pi(\theta)L(\theta; \boldsymbol{z})q(\theta'|\theta)}\right\}$$

For i.i.d. data, the likelihood is of the form

$$L(\theta; \boldsymbol{z}) = \prod_{i=1}^{n} L(\theta; z_i).$$

If L tends not to be that close to 1, it's easy for this quantity to be too big or too small for a computer to store properly.

Bayesian Posterior Distributions

You will implement an M-H algorithm in the practical.

When dealing with the acceptance ratio α, work on a log-scale to avoid overflow errors.

Bad

```
U <- runif(1)
alpha <- (prior(y)*lik(y))/(prior(x)*lik(x))
if (U < alpha) {
  x <- y
}
```

Good

```
U <- runif(1)
logalpha <- logprior(y) - logprior(x) + loglik(y) - loglik(x)
if (log(U) < logalpha) {
  x <- y
}
```

Gamma Distribution

In the practical you will consider data $Y_1, \ldots, Y_n \overset{\text{i.i.d.}}{\sim} \text{Gamma}(\alpha, \beta)$ with priors

$$\alpha, \beta \sim \text{Exp}(1) \text{ independently.}$$

Using Bayes rule,

$$\begin{aligned}
\pi(\alpha, \beta \mid y_1, \ldots, y_n) &\propto \pi(\alpha) \cdot \pi(\beta) \cdot L(\alpha, \beta; y_1, \ldots, y_n) \\
&\propto e^{-\alpha-\beta} \cdot \prod_{i=1}^{n} \frac{\beta^\alpha y_i^{\alpha-1} e^{-\beta y_i}}{\Gamma(\alpha)} \\
&\propto e^{-\alpha} \frac{\beta^{n\alpha} \left(\prod_i y_i\right)^{\alpha-1} e^{-\beta(1+\sum_i y_i)}}{\Gamma(\alpha)^n}
\end{aligned}$$

The $\prod_i y_i$ term in particular is likely to lead to over/underflow, but:

$$\begin{aligned}
&\log \pi(\alpha, \beta \mid y_1, \ldots, y_n) \\
&= -\alpha + n\alpha \log \beta + \alpha \sum_i \log y_i - \beta(1 + \sum_i y_i) - n \log \Gamma(\alpha)
\end{aligned}$$

is manageable.

Gamma Distribution

In R this means that instead of writing functions to evaluate the likelihood:

```
lik <- function(alpha, beta, y) {
  prod(dgamma(y, alpha, beta))
}
```

You should instead use the log-likelihood:

```
logLik <- function(alpha, beta, y) {
  sum(dgamma(y, alpha, beta, log=TRUE))
}
```

The Ising model

Denote by $\Omega = \{0,1\}^{n^2}$ the set of all $n \times n$ binary matrices $X = (x_{ij})$, $x_{ij} \in \{0,1\}$.

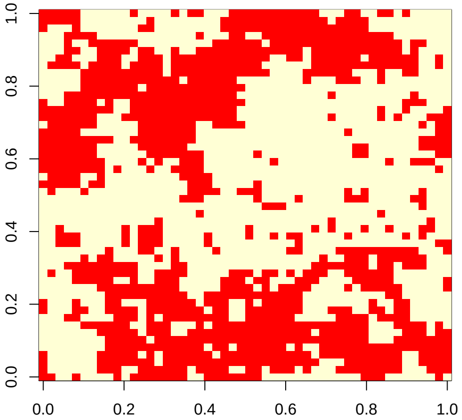
The **Ising model** over Ω has distribution:

$$\pi(X) = \exp\left\{\theta \sum_{(i,j)\sim(i',j')} \mathbb{I}\{x_{ij} = x_{i'j'}\}\right\} / Z_\theta.$$

Here θ is a **smoothing parameter** which is usually taken to be greater than zero and Z_θ is a normalizing constant.

Sample from Ising Model

$n = 50$, $\theta = 0.8$. Obtained by running M-H for 5×10^5 iterations.



MCMC for the Ising Model

Start with $X^{(0)}$ where pixels are chosen as independent $\{0, 1\}$s.

Let $X^{(t)} = x$. $X^{(t+1)}$ is determined in the following way.

1. Choose a pixel (i, j) uniformly from $\{1, \ldots, n\}^2$.

2. Set $x' = x$ except $x'_{ij} = 1 - x_{ij}$.

3. Let

$$\begin{aligned}\alpha(x'|x) &= \min\left\{1,\ \frac{\pi(x')q(x|x')}{\pi(x)q(x'|x)}\right\} \\ &= \min\left\{1,\ \exp(-\theta(d_{ij} - 2a_{ij}))\right\}\end{aligned}$$

where d_{ij} is the number of neighbours for (i, j) and a_{ij} is the number of agreements with x_{ij}.

4. With probability $\alpha(x'|x)$ set $X^{(t+1)} = x'$ and otherwise set $X^{(t+1)} = x$.

Notice: we don't even have to recalculate π, just calculate d_{ij} and a_{ij}.

Bayesian image recovery

Let Y be an unknown true image. Suppose $Y \sim \text{Ising}(\theta)$ with θ known, so the prior for Y is $\pi_\theta(y)$.

Suppose we observe Y through a 'noisy channel'. At each pixel i, j we observe

$$X_{ij} = \begin{cases} Y_{ij} & \text{with probability } p \\ 1 - Y_{ij} & \text{otherwise} \end{cases}$$

The likelihood for Y is

$$L(y; x) \propto \prod_{i,j} p^{\mathbb{I}(x_{ij} = y_{ij})} (1-p)^{\mathbb{I}(x_{ij} \neq y_{ij})} = p^K (1-p)^{n^2 - K}$$

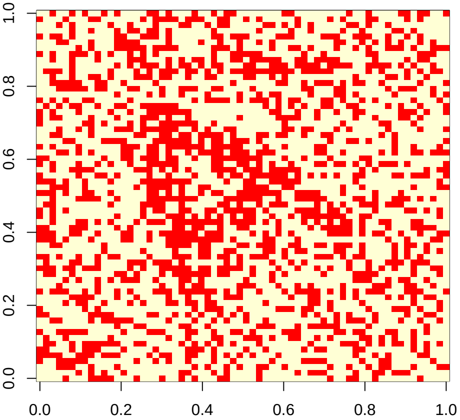
where K is the number of pairs such that $x_{ij} = y_{ij}$.

If we observe X the probability that the unknown true image Y equals y is

$$\begin{aligned} \pi(y|x) &\propto L(y; x)\pi_\theta(y) \\ &\propto p^K (1-p)^{n^2 - K} \pi_\theta(y) \end{aligned}$$

We will simulate $Y \sim \pi(y|x)$ and use the samples to estimate $E(Y_{ij}|X = x)$ for each cell i, j.

Noisy Images



MCMC for Image Recovery

We modify our MCMC to target $\pi(x|y)$ instead of $\pi(x)$.

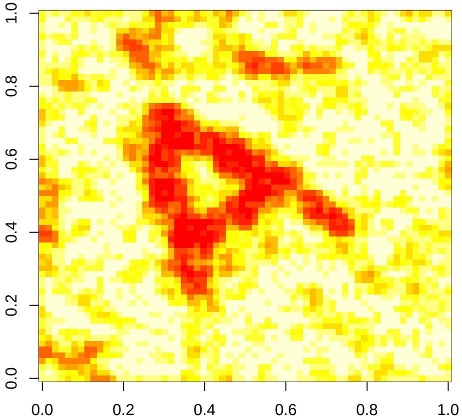
Essentially all we have to do is modift the acceptance probability to account for changes in the likelihood:

$$\begin{aligned}\alpha(y'|y) &= \min\left\{1, \frac{\pi(y'|x)q(y|y')}{\pi(y|x)q(y'|y)}\right\} \\ &= \min\left\{1, \ \frac{\pi(y')L(y' \mid x)q(y \mid y')}{\pi(y)L(y \mid x)q(y' \mid y)}\right\}.\end{aligned}$$

Choosing a symmetric proposal, the log of the ratio becomes

$$-\theta(d_{ij} - 2a_{ij}) + \mathbb{I}(y_{ij} = x_{ij}) \log\left(\frac{1-p}{p}\right).$$

Cleaned Up



The Truth