MTH3045: Statistical Computing

Dr. Ben Youngman
b.youngman@exeter.ac.uk
Laver 817; ext. 2314

16/01/2024

Introduction

Module outline

MTH3045, Statistical Computing, is designed to introduce you to some important, advanced topics that, when considered during calculations, can improve using computers to fit statistical models to data. This can allow us to use more data, or to fit a model more quickly and/or more reliably, for example.

Lectures / practical classes

In-person lectures will be held at the following times:

- Tuesday 12.35 – 13.25. Laver LT3
- Thursday 11.35 – 12.25. Old Library 134
- Friday 13.35 – 14.25.
  - Forum Exploration Lab 1 (weeks 1,2,4,5,6,8,9,11)
  - Streatham Court 0.93 (weeks 3,7,10)

The first lecture takes place on Tuesday 16th January 2024. Lectures will typically involve a mix of me presenting a few slides being to introduce a method to you and then you engaging in hands-on programming so that you experience and confirm understanding of what's been introduced. To engage with MTH3045 you should be attending lectures in person.

I currently plan to use week 6 as a reading week for you. During week 6 there will be no MTH3045 lectures.

Office hours

I will hold office hours each week on Tuesdays at 13.30 - 14.30 in my office, Laver 817, or you can schedule an appointment to meet me online during that hour.

Resources

All material that will be examined for MTH3045 can be found in the lecture notes and exercises provided. A complete set of lecture notes and exercises with solutions can be found in pdf format on the module's ELE page

https://ele.exeter.ac.uk/course/view.php?id=13566

There is also an identical web-based version of the lecture notes available at

https://byoungman.github.io/MTH3045/

and a web-based version of the exercises available at

https://byoungman.github.io/MTH3045/exercises

Note that you might find the web-based versions easier for your study.

During lectures various challenges will be set. These can be found with skeleton solutions at

https://byoungman.github.io/MTH3045/challenges

However, sometimes you may find it useful to get a different perspective on things. If so, you may want to consult the following books. Specific parts of these books useful for given topics will be highlighted throughout the lecture notes.

- Banerjee, S. and A. Roy (2014). *Linear Algebra and Matrix Analysis for Statistics*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.
- Eddelbuettel, D. (2013). *Seamless R and C++ Integration with Rcpp*. Use R! Springer New York.
- Gillespie, C. and R. Lovelace (2016). *Efficient R Programming: A Practical Guide to Smarter Programming*. O'Reilly Media. https://csgillespie.github.io/efficientR/.
- Monahan, J. F. (2011). *Numerical Methods of Statistics (2 ed.)*. Cambridge University Press.

- Nocedal, J. and S. Wright (2006). *Numerical Optimization (2 ed.)*. Springer Series in Operations Research and Financial Engineering. Springer New York.
- Petersen, K. B. and M. S. Pedersen (2012). *The Matrix Cookbook*. https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf
- Press, W., S. Teukolsky, W. Vetterling, and B. Flannery (2007). *Numerical Recipes: The Art of Scientific Computing (3 ed.)*. Cambridge University Press.
- W. N. Venables, D. M. S. and the R Core Team (2021). *An Introduction to R (4.1.0 ed.)*. https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf.
- Wickham, H. (2019). Advanced R (2 ed.). Chapman & Hall/CRC the R series. CRC Press/Taylor & Francis Group. https://adv-r.hadley.nz/.
- Wood, S. N. (2015). Core Statistics. Institute of Mathematical Statistics Textbooks. Cambridge University Press. https://www.maths.ed.ac.uk/~swood34/core-statistics.pdf.

Acknowledgements

Much information used to form these notes came from the above resources. However, Simon Wood's APTS notes and Charles Geyer's Statistics 3701 notes have also proved incredibly useful for providing some additional information.

Assessment

Assessment for MTH3045 will comprise one piece of coursework (worth 50% of the module's total module mark) and an online practical exam (worth 50% of the module's total module mark). Your coursework must be your on work.

Motivating example

Suppose that we want to find $\mathbf{z}$, where $\mathbf{z} = \mathbf{ABy}$. Let's try this in `R`. We can use `system.time()` to measure how long a command takes to execute. We'll use 3000×3000 matrices, `A` and `B`, for $\mathbf{A}$ and $\mathbf{B}$, respectively, which we'll fill with arbitrary (here Normal(0, 1), henceforth $N(0, 1)$) variates, and fill the n-vector $\mathbf{y}$, called `y`, similarly.

```
> n <- 3e3
> A <- matrix(rnorm(n * n), n, n)
> B <- matrix(rnorm(n * n), n, n)
> y <- rnorm(n)
> system.time(z1 <- A %*% B %*% y)
```

```
##    user  system elapsed
##   1.706   1.084   0.190
```

We're interested in the total time, `elapsed`, which is 0.19 seconds. However, the next line gives exactly the same answer

```
> system.time(z2 <- A %*% (B %*% y))
```

```
##    user  system elapsed
##   0.051   0.126   0.016
```

which we can check with `all.equal()`

```
> all.equal(z1, z2)
```

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

yet is about 10 times faster. The point of this example is that, if we understand some of the basics behind computations that we need when fitting statistical models, we can make our fitting much more efficient. Improving efficiency can, for given computational cost, allow us to fit models to more data, for example, from which we should hope to draw more reliable inferences.

What's happened above? In first approach `R` has essentially calculated $\mathbf{C} = \mathbf{AB}$, say, and then $\mathbf{Cy}$, i.e. multiplied two $n \times n$ matrices, and then an $n \times n$ matrix by an n-vector. In the second approach, however, `R` has essentially calculated $\mathbf{x} = \mathbf{By}$ and then $\mathbf{z} = \mathbf{Ax}$, say, i.e. two multiplications of $n \times n$ matrices and n-vectors. By considering the n-vector as an $n \times 1$ matrix, we should expect multiplying together two $n \times n$ matrices to require roughly a factor of n times more calculations than multiplying an $n \times n$ matrix by an n-vector. This is a trivial, but illuminating, example, worth bearing in mind when writing matrix-based code.

Exploratory and refresher exercises

1. Generate a sample of $n = 20$ $N(1, 3^2)$ random variates, and without using `mean()`, `var()` or `sd()` write `R` functions to calculate the sample mean, $\bar{x}$, sample variance, s^2, and sample standard deviation, s, where

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

   Note than `sum()` may be used.

2. Consider computing $\Pr(Z \geq z) = 1 - \Phi(z)$ where $Z \sim \text{Normal}(0, 1)$, or, for short, $Z \sim N(0, 1)$. For $z = 0, 0.5, 1, 1.5, 2, \ldots$ compute this in `R` in three different ways using the following three commands

```
> pnorm(z, lower.tail = FALSE)
> 1 - pnorm(z)
> pnorm(-z)
```

and find the lowest value of z for which the three don't give the same answer.

3. The formula $\text{Var}(Y) = \text{E}(Y^2) - [\text{E}(Y)]^2$ is sometimes called the 'short-cut' variance formula, i.e. a short-cut for $\text{Var}(Y) = \text{E}[Y - \text{E}(Y)]^2$. Compare computing $\text{Var}(Y)$ using the two formulae above for the samples `y1` and `y2` below.

```
> y1 <- 1:10
> y2 <- y1 + 1e9
```

Statistical computing in R

Overview

In MTHM3045 we'll be using the programming language R for computation. Note that RStudio is an Integrated Development Environment (IDE) for R: it simplifies working with scripts, R objects, plotting and issuing commands by putting them all in one place. In MTH3045 we'll typically be interested in programming, so will just refer to R, but you may want to think of this as code that's run in RStudio.

Mathematics by computer

In statistical computing, we can usually rely on our computer software to take care of most things in the background. Nonetheless, it's useful to know the basics of how computers perform calculations, as, if our code isn't doing what we'd like, or what we'd expect, then such knowledge can help us diagnose any problems.

Put simply, if we issue

```
> 1 + 2
```

```
## [1] 3
```

in R, how does it get to the answer of 3?

Positional number systems

Any positive number, z, can be represented as a base-B number with the digits $\{0, 1, \ldots, B-1\}$ in the form

$$z = a_k B^k + \ldots + a_2 B^2 + a_1 B + a_0 + a_{-1} B^{-1} + a_{-2} B^{-2} + \ldots$$

for integer coefficients a_j in the set $\{0, 1, \ldots, B-1\}$. We can write this more tidily as

$$(a_k \ldots a_2 a_1 a_0 . a_{-1} a_{-2} \ldots)_B.$$

The 'dot' in this representation is called the *radix point*, or the decimal point, in the special case of base 10, which is usually referred to simply as 'decimal'.

In a *fixed point* number system, the radix point is always placed in the same place, i.e. after a_0, the coefficient of $B^0 = 1$. So when we write π as 3.14159... in decimal form, we have the decimal point after the 3, which is the coefficient of $10^0 = 1$. Fixed point number systems are easy for humans to interpret.

A historical aside on exact representations of integers

Humans now usually use base 10 for mathematics, and computers work in base two. However, humans used to primarily use base 60, known as sexagesimal, for mathematics. The sexagesimal system can be traced back to the Sumerians back in 3000BC. The many factors of 60, e.g., 1, 2, 3, 4, 5, 6, ..., 30, 60, were one of its selling points, and it's still used for some formats of angles and coordinates, and of course time! The decimal number system is attributed to Archimedes (c. 287–212 BCE).

Floating point representation

In a *floating point* number system, the radix point is free to move. Computers use such number systems. Scientific notation, e.g. Avagadro's number

$$N = 6.023 \times 10^{23}$$

is an example of such a system. More generally, we can write any *positive* number in the form

$$M \times B^{E-e},$$

where M is the *mantissa*, E is the *exponent*, and e is the *excess*, and even more generally, we can write *any* number in the form

$$S \times M \times B^{E-e},$$

where S is its *sign*. We may think of S as from the set $\{+, -\}$. Hence, for a given base, we can represent any number with the triple (S, E, e, M). In base 10 Avogadro's number can be written $N = (+, 24, 0, 0.6023)$ or $N = (+, 23, 0, 6.023)$. The latter is classed as *normalised* because its leading term is non-zero.

Single-precision arithmetic

Computers work in base $B = 2$, or *binary*, and usually consider the mantissa to be of the form $M = 1 + F$, where $F \in [0, 1)$ is the *fraction*, giving

$$S \times (1 + F) \times 2^{E-e}$$

and hence using a normalised representation. Computers use a limited number of *bits* to store numbers. As a result any number is only stored approximately. Computers vary in how they store different types of number. A commonly-used standard is IEEE 754.

Let's first consider *single-precision* arithmetic, which uses 32 bits, $b_1, \ldots, b_{32}$, say, to store numbers. Under the IEEE 754 standard, the order of bits is arbitrary. What each does is defined, so that

- 1 bit, b_1 say, gives the sign, $S = (-1)^{b_1}$,
- 8 bits, $b_2, \ldots, b_9$, give the exponent, $E = \sum_{i=1}^{8} b_{i+1} 2^{8-i}$,
- 23 bits, $b_{10}, \ldots, b_{32}$, give the fraction, $F = \sum_{i=1}^{23} b_{i+9} 2^{-i}$,

with $e = 127$ fixed.

Consider the single-precision representation

0 10000000 10010010000111111011011

and use `R` to find the number in decimal form to 20 decimal places.

We'll start with a function `bit2decimal()` for converting a bit string into a decimal. Producing such a function is beyond the scope of MTH3045, but some aspects of the function may help you with other parts of the module. The comments should give an idea of what each line of `bit2decimal()` does. Note that `bit2decimal()` has arguments `x`, the bit string we want to convert, `e` for the excess e, and `dp`, which specifies the number of decimal places we want it to return the decimal number to. These are repeated at the start of the function, which is often good practice, especially when sharing code.

```
> bit2decimal <- function(x, e, dp = 20) {
+ # function to convert bits to decimal form
+ # x: the bits as a character string, with appropriate spaces
+ # e: the excess
+ # dp: the decimal places to report the answer to
+ bl <- strsplit(x, ' ')[[1]] # split x into S, E and F components by spaces
```

```
+ # and then into a list of three character vectors, each element one bit
+ bl <- lapply(bl, function(z) as.integer(strsplit(z, '')[[1]]))
+ names(bl) <- c('S', 'E', 'F') # give names, to simplify next few lines
+ S <- (-1)^bl$S # calculate sign, S
+ E <- sum(bl$E * 2^c((length(bl$E) - 1):0)) # ditto for exponent, E
+ F <- sum(bl$F * 2^(-c(1:length(bl$F)))) # and ditto to fraction, F
+ z <- S * 2^(E - e) * (1 + F) # calculate z
+ out <- format(z, nsmall = dp) # use format() for specific dp
+ # add (S, E, F) as attributes, for reference
+ attr(out, '(S,E,F)') <- c(S = S, E = E, F = F)
+ out
+ }
```

Then we'll input the numbers in binary form, and call `bit2decimal()`.

```
> b0 <- '0 10000000 10010010000111111011011'
> sing_prec <- bit2decimal(b0, 127)
> sing_prec
```

```
## [1] "3.14159274101257324219"
## attr(,"(S,E,F)")
##           S           E           F
##   1.0000000 128.0000000   0.5707964
```

That's right: it's the single-precision representation of π.

Note that for MTH3045, you're not expected to repeat a similar calculation. The example is merely designed to show how the given single-precision representation can be converted to a number in conventional format, as shown by `R`. Also note that the function `format()` guarantees printing a number to `nsmall` decimal places.

Double-precision arithmetic

`R` usually uses *double-precision* arithmetic, which uses 64 bits to store numbers.

- 1 bit, b_1 say, for the sign, $S = (-1)^{b_1}$.
- 11 bits, $b_2, \ldots, b_{12}$, for the exponent, $E = \sum_{i=1}^{11} b_{i+1} 2^{11-i}$.
- 52 bits, $b_{13}, \ldots, b_{64}$, for the fraction, $F = \sum_{i=1}^{52} b_{i+9} 2^{-i}$.

For double-precision arithmetic $e = 1023$. Using twice as many bits essentially brings twice the precision; hence double- instead of single-precision.

Now consider the double-precision representation

0 10000000000 1001001000011111101101010100010001000010110100011000

and use R to find the number in decimal form to 20 decimal places.

```
> b0 <- '0 10000000000 1001001000011111101101010100010001000010110100011000'
> doub_prec <- bit2decimal(b0, 1023)
> doub_prec
```

```
## [1] "3.14159265358979311600"
## attr(,"(S,E,F)")
##            S            E            F
##    1.0000000 1024.0000000    0.5707963
```

Rather repetitively, it's the double-precision representation of π.

The constant π is built in to R as `pi`. We can compare our single- and double-precision approximations to that built in

```
> format(pi - as.double(sing_prec), nsmall = 20)
```

```
## [1] "-8.742278e-08"
```

```
> format(pi - as.double(doub_prec), nsmall = 20)
```

```
## [1] "0.00000000000000000000"
```

and we see that our double-precision approximation is exactly the same as that built in, whereas the single-precision version differs by 10^{-8} in order of magnitude. Note that our function `bit2decimal()` generated a character string (which let us ensure it printed a specific number of decimal places), so we use `as.double()` to convert its output to a double-precision number, which allows direct comparison with `pi`.

We can overwrite R's constants.

```
> pi <- 2
> pi
```

```
## [1] 2
```

In general this is a bad idea. The simplest way to fix it is to remove the object we've created from R's workspace with `rm()`. Then `pi` reverts back to

R's built in value.

```
> rm(pi)
> pi
```

```
## [1] 3.141593
```

Note that R also has T and F built in as aliases for TRUE and FALSE. T and F can be overwritten, but TRUE and FALSE can't. In general, for example with function arguments, it is better to use `argument = TRUE` or `argument = FALSE`, just in case T or F are overwritten with something that could be interpreted as the opposite of what's wanted, such as issuing `T <- 0`, since

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

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

Flops: floating point operations

Applying a mathematical operation, such as addition, subtraction, multiplication or division, to two floating point numbers is a *floating point operation*, or, more commonly, a *flop*[1].

The research area *numerical analysis* includes the analysis of the accuracy of flops, by virtue of numbers being approximately represented. Numerical analysis goes far beyond just analysing flops, and also includes the study of algorithms. Such analysis is far beyond the scope of MTH3045. We will merely cover a couple of examples, which will be illuminating for understanding when flops can go wrong. Such knowledge can help us avoid unnecessary errors.

The addition of floating point numbers works by representing the numbers with a common exponent, and then summing their mantissas.

Calculate 123456.7 + 101.7654 using base-10 floating point representations.

We first represent both numbers with a common exponent: that of the largest number. Therefore

[1] Note that in MTH3045 we'll use *flops* as the plural of flop. It is also often used to abbreviate floating point operations per second, but for that we'll use flop/s. For reference, ENIAC, the first (nonsuper)-computer, processed about 500 flop/s in 1946. My desktop computer can apparently complete 11,692,000,000 flop/s. The current record, set by American supercomputer Frontier in May2022, is 1.102 exaflop/s, i.e. 1,102,000,000,000,000,000 flop/s!

$$123456.7 = 1.234567 \times 10^5$$
$$101.7654 = 1.017654 \times 10^2 = 0.001017654 \times 10^5.$$

We next sum their mantissas, i.e.

$$\begin{aligned} 123456.7 + 101.7654 &= (1.234567 \times 10^5) + (1.017654 \times 10^2) \\ &= (1.234567 \times 10^5) + (0.001017654 \times 10^5) \\ &= (1.234567 + 0.001017654) \times 10^5 \\ &= 1.235584654 \times 10^5 \end{aligned}$$

Note that if the mantissa was rounded to six decimal places, the result would be 1.235584×10^5, and the final three decimal places would effectively be lost. We call the difference between the actual value and that given by the approximate algorithm **roundoff error**[2].

Consider the following calculations in R.

```
> a <- 1e16
> b <- 1e16 + pi
> d <- b - a
```

Obviously we expect that $(1 \times 10^{16} + \pi) - 1 \times 10^{16} = \pi$.

```
> d
```

```
## [1] 4
```

But `d = 4`! So, what's happened here?

Double-precision lets us represent the fractional part of a number with 52 bits. In decimal form, this corresponds to roughly 16 decimal places. Addition (or subtraction) with floating point numbers first involves making common the exponent. So above we have

$$\pi = 3.1415926535897932384626 \times 10^0,$$

[2] A rather catastrophic example of roundoff error is that of the Ariane rocket launched on June 4, 1996 by the European Space Agency. Ultimately, it caused the rocket to be destoryed on its 37th flight, which the interested reader can read more about here and on various other parts of the web. The Patriot missile is another well-known example.

but when we align its exponent to that of `a` we get

$$\pi = 0.0000000000000003 \times 10^{16}.$$

Then mantissas are added (or subtracted); hence

$$\texttt{b} = 1.0000000000000003 \times 10^{16}$$

and so `d` $= 3$. This simple example demonstrates **cancellation error**. (Note that above we had `d = 4`, because `R` did its calculations in base 2, whereas we used base 10.)

Some useful terminology

The **machine accuracy**, often written ϵ_m, and sometimes called *machine epsilon*, is the smallest (in magnitude) floating point number that, when added to the floating point number 1.0, gives a result different from 1.0. Let's take a look at this by considering $1 + 10^{-15}$ and $1 + 10^{-16}$.

```
> format(1.0 + 1e-15, nsmall = 18)
```

```
## [1] "1.000000000000001110"
```

```
> format(1.0 + 1e-16, nsmall = 18)
```

```
## [1] "1.000000000000000000"
```

The former gives a result different from 1.0, whereas the latter doesn't. This tells us that $10^{-16} < \epsilon_m < 10^{-15}$. In fact, `R` will tell us its machine accuracy, and it's stored as `.Machine$double.eps` which is 2.220446×10^{-16}. Note that $2.220446 \times 10^{-16} = 2^{-52}$, and recall that for double-precision arithmetic we use 52 bits to represent the fractional part. Also note that this is the machine accuracy for double-precision arithmetic, and would be larger for single-precision arithmetic.

By using the floating point representation for a number, we are effectively treating it as rational. We should anticipate that any subsequent flop introduces an additional fractional error of at least ϵ_m. The accumulation of roundoff errors in one or more flops is **calculation error**.

In statistics, **underflow** can sometimes present problems. Put simply, this is when numbers are sufficiently close to zero that they cannot be differentiated from zero using finite representations, which, for example, results in meaningless reciprocals. Maximum likelihood estimation gives a

simple example of when this can occur, because we may repeatedly multiply near-zero numbers together until the likelihood becomes very close to zero. The opposite is **overflow**, when we have numbers too large for the computer to deal with. This is simple to demonstrate as

```
> log(exp(700))
```

```
## [1] 700
```

works but

```
> log(exp(710))
```

```
## [1] Inf
```

doesn't, just because `exp(710)` is too big. (Here's a great example where logic, i.e. avoiding taking the logarithm of an exponential, would easily solve the problem.)

The history of R

Before we discuss the history of R, we'll consider S, which is sometimes considered to be its predecessor. S is a statistical programming language that aimed "to turn ideas into software, quickly and faithfully", according to John Chambers. Chambers, along with Rick Becker and Allan Wilks, began developing S in 1975 when working for Bell Laboratories, an American industrial research and scientific development company. This was released outside Bell Labs in 1980 as a set of macros, and in 1988 updated to be function based. A commercial implementation of the S programming language, S-PLUS, was also released in 1988, but is no longer available.

Ross Ihaka and Robert Gentleman of the University of Auckland began developing the R programming language in 1992. R is, at its core, an open-source implementation of the S programming language. Its name is partly inspired by that of S, and also by its authors' first-name initials. R was officially released in 1995, which was followed by a 'stable beta' version (v1.0) in 2000. The R Core Team, which develops R, includes Chambers, Ihaka and Gentlemen, and 17 other members. As I write this, the latest version of R is v4.3.2, which is called 'Eye Holes'[3]. According to a survey by IEEE

[3]I'd always wondered where the names for different R versions come from. Then, when putting these lecture notes together, I decided to find out, and came across Lucy D'Agostino McGowan's blog entry R release names. Anyway, the answer is that they're

Spectrum, R is currently the 11th most popular programming language (after 1. Python, 2. Java, 3. C++, 4. C, 5. JavaScript, 6. C#, 7. SQL, 8. Go, 9. TypeScript, 10. HTML). In my opinion, R would not have been as popular as it is if it weren't free and open-source, and those that develop R could have easily heavily profited financially from it, yet, commendably, their work has instead led to highly regarded and free computer software, which is now frequently and widely used for statistical programming.

Why R?

In MTH3045 we will only consider R for any computations. I could give various reasons for this. Instead, I'll give the following quote:

> "... despite its sometimes frustrating quirks, R is, at its heart, an elegant and beautiful language, well tailored for data science." — Wickham (2019)

I concur with this.

The reason for this is that R tends to be quite good – or better – at most computational aspects of statistics than other programming languages. For example, as a programming language it is relatively accessible and its computational speed is usually okay. A particularly appealing quality of R is that it is free and open-source. The entire code that comprises the software package can be viewed by anyone. This is perhaps why R has such a strong community, which includes package contributions, and helps keep R up-to-date in terms of the statistical models that its users can fit.

However, it is not the only programming language available for statistical computing: Python and MATLAB are other popular choices. Other languages, such as Julia, are growing in popularity. Browsing the internet, you'll find comparisons of these programming languages. In general the conclusion is that R is slower. However, in MTH3045, it will become apparent that statistical computing is heavily reliant on matrix computations. For these R calls on basic linear algebra subprograms, usually referred to as 'BLAS'; see http://www.netlib.org/blas/ and https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms. These typically involve incredibly advanced code for matrix algebra, which is often the time-consuming part of statistical computations. If we're doing computations that heavily rely on BLAS – or can be sensibly manipulated to rely

inspired by Peanuts comic strips, films and badges.

on BLAS - then R becomes an attractive programming language. For these reasons, we use R in MTH3045. Nonetheless, most – if not all – of what we cover in MTH3045 could be achieved in other programming languages.

Basics

To get started with R, the development team's 'An Introduction to R' manual, available from https://cran.r-project.org/manuals.html is a good starting point. You'll find the pdf version on the MTH3045 VLE page.

I'll just pick out some key information that will be vital for MTH3045.

How R works

R is an *interpreted* programming language. The console in RStudio, the R GUI, or when using R from a terminal, is a command-line interpreter. This is essentially a program that converts what you request into something of the form that can be passed to another programming language. With R, this is to the 'low-level' programming languages C or Fortran. Calculations are based on code written in these languages. So, when we issue

```
> 1.5 + 2.3
```

```
## [1] 3.8
```

R has very cleverly interpreted (for C in this case) that we've passed two numeric vectors, each of length one, and that we want to sum them, and that the result should be a vector of length one.

How functions work

I'm quite a fan of R's help files, but others are not. If you're ever unsure of how a function works, I suggest first consulting its help file. For example `help(lm)`, or equivalently `?lm`, gives help on R's `lm()` function. The function is titled 'Fitting Linear Models'. It then has

- `Description`, describing what the function does;
- `Usage`, detailing how we use the function does, i.e. what command(s) to issue;
- `Arguments`, explaining what each argument is, and in what format it should be;
- `Details`, where present, goes into a bit more detail;
- `Value`, states what we'll get when we call the function; and then

- `Examples`, where present, gives examples of usage (which are often incredibly useful).

Note that we can also get help on basic arithmetic functions: e.g. `help('*')` for multiplication and `help('%*%')` for matrix multiplication.

Data structures

`R` can handle various different data structures. Here we'll introduce a few of the more commonly used ones. Familiarity with these will be helpful for various aspects of MTH3045.

Vectors

We'll ignore scalars, and start with the `vector`. Issuing

```
> vec <- c(1, 2, 3)
> vec
```

```
## [1] 1 2 3
```

creates the *column vector* $(1, 2, 3)^{\mathrm{T}}$, where $^{\mathrm{T}}$ denotes transpose. We may note that this is equivalent to calling `1:3`, which is also equivalent to calling `seq(1, 3)`. `seq()` is a useful function. It generates *regular* sequences and its usage is

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

Typically we specify `from` and `to`, i.e. the start and end points of the sequence, and then specify its length with `length.out`, or specify its 'jumps' with `by`. Note that when calling a function, we don't need to give argument names if we give them in the right order, and we can shorten argument names if the shortenings are unique:

```
> seq(0, 1, 0.2)
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

```
> seq(0, 1, by = 0.2)
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

```
> seq(0, 1, b = 0.2)
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Another useful function for creating a `vector` is `rep()`, which replicates a `vector`. It can be used to repeat a `vector` one after the other, such as

```
> rep(vec, 3)
```

```
## [1] 1 2 3 1 2 3 1 2 3
```

or to replicate each element of the supplied `vector` the same number of times

```
> rep(vec, each = 3)
```

```
## [1] 1 1 1 2 2 2 3 3 3
```

or a different number of times

```
> rep(vec, c(2, 1, 3))
```

```
## [1] 1 1 2 3 3 3
```

Matrices

We create a matrix with `matrix()`. Issuing

```
> mat <- matrix(1:6, 2, 3)
> mat
```

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

specifies a matrix with first row $(1, 3, 5)$ and second row $(2, 4, 6)$. The second and third arguments to `matrix()` are `nrow` and `ncol`, its numbers of rows and columns, respectively. Provided the vector that we specify is equal in length to the product of the matrix's numbers of rows and columns, we only need to supply one of `nrow` and `ncol`, as the other can be determined. `R` will recycle the supplied vector if its length is below the product of `nrow` and `ncol`. Note that `R`, by default, fills matrices column-wise. We can fill row-wise with argument `byrow = TRUE`:

```
> mat2 <- matrix(1:8, ncol = 4, byrow = TRUE)
> mat2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

Arrays

We may think of a `vector` as a one-column `matrix`. We can extend this by thinking of a `matrix` as a two-dimensional `array`. An `array` can be of any dimension. Here's one of dimension $2 \times 3 \times 4$.

```
> arr <- array(1:24, c(2, 3, 4))
> arr
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

In `R` we refer to each dimension as a *margin*.

Lists

One of `R`'s particularly convenient data structures is the `list`. It is described, by `R`, as a *generic vector*: essentially as a vector in which each element can be more complicated than a scalar, which are the elements of a conventional mathematical vector. So a `list` is a collection of data structures, which might be the same, such as two `vectors`, or might be different, such as a

`vector` and `matrix`, or even a `vector` and another `list`. For example,

```
> lst <- list(vec, mat)
> lst
```

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

`lists` are therefore incredibly flexible, and hence incredibly useful. A `data.frame` is actually just a `list`, albeit one typically comprising `vectors` of the same length, and hence printable similarly to a `matrix`.

Attributes

Sometimes we might have an object that we want to add a bit more information to. We can do this by assigning it *attributes*. Suppose we're interested in the number 1×10^{20}. We could store this as 20, if we knew we were storing it in $\log_{10}$ format. We could use an attribute to remind us of this.

```
> x <- 20
> attr(x, 'format') <- 'log10'
> x
```

```
## [1] 20
## attr(,"format")
## [1] "log10"
```

The function `attributes()` will then show us all the attributes of an object

```
> attributes(x)
```

```
## $format
## [1] "log10"
```

and we can use `attr()` again to access a specific attribute

```
> attr(x, 'format')
```

```
## [1] "log10"
```

Some useful R functions

sum(), rowSums() and colSums()

You'll have encountered various R functions during your degree. For example, sum() gives the *global* sum of a vector, matrix, or even an array:

```
> sum(vec)
```

```
## [1] 6
```

```
> sum(mat)
```

```
## [1] 21
```

Note that we may want to sum a matrix or array over one or more of its margins. For this we should use rowSums() and colSums(). For a matrix, this is fairly straightforward:

```
> rowSums(mat)
```

```
## [1]  9 12
```

```
> colSums(mat)
```

```
## [1]  3  7 11
```

For an array, though, there are various options. First note that both rowSums() and colSums() have argument dims, which defaults to 1 and so the following implicitly use dims = 1. This means that first margin is regarded as the row, and the remaining margins are regarded as the columns.

```
> rowSums(arr)
```

```
## [1] 144 156
```

```
> colSums(arr)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    3   15   27   39
## [2,]    7   19   31   43
## [3,]   11   23   35   47
```

If we change dims to dims = 2, the first two margins are now the rows, and the final margin is the column. So rowSums() is over two margins, whereas colSums() is over one:

```
> rowSums(arr, dims = 2)
```

```
##      [,1] [,2] [,3]
## [1,]   40   48   56
## [2,]   44   52   60
```

```
> colSums(arr, dims = 2)
```

```
## [1]  21  57  93 129
```

Note that `rowSums()` and `colSums()` require contiguous margins. If we wanted to sum over the first and third margins of the above `array`, then we can permute its margins with `aperm()` so that the first and third margins become the first and second margins:

```
> arr2 <- aperm(arr, c(1, 3, 2))
> rowSums(arr2)
```

```
## [1] 144 156
```

```
> colSums(arr2)
```

```
##      [,1] [,2] [,3]
## [1,]    3    7   11
## [2,]   15   19   23
## [3,]   27   31   35
## [4,]   39   43   47
```

There are also functions `rowMeans()` and `colMeans()`, which are self-explanatory.

The `*apply()` family of functions

`apply()` applies a function to margins of an `array` or `matrix`. In the following

```
> apply(mat, 1, prod)
```

```
## [1] 15 48
```

we get the product, through function `prod()`, of each row of `mat`, and it returns a vector of length the number of rows of `mat`. The second argument of `apply()` is the margin over which we want to apply the specified function. Therefore `apply(mat, 2, prod)` would give product over the columns of `mat`. We can also apply functions over margins of an `array`. So

```
> apply(arr, c(1, 3), max)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    5   11   17   23
## [2,]    6   12   18   24
```

uses `max()` to find the maximum over margin 2. As margins 1 and 3 are of length 2 and 4, respectively, here `apply()` returns a 2×4 `matrix`. Note that `apply(..., ..., sum)` is inefficient in comparison to `rowSums()` and `colSums()`.

`lapply()` applies a function over a `list`. Consider

```
> lapply(lst, sum)
```

```
## [[1]]
## [1] 6
##
## [[2]]
## [1] 21
```

Note that `lapply()` returns a `list` that is the same length as that supplied to it, i.e. that same length as `lst` here. We might want the result of `lapply()` simplified and coerced to an array, if possible, which is what `sapply()` does:

```
> sapply(lst, sum)
```

```
## [1]  6 21
```

Sometimes, though, this isn't possible, and `lapply()` and `sapply()` do the same, such as if the applied function returns results of different length, such as

```
> lst2 <- list(rnorm(4), runif(6))
> lst2
```

```
## [[1]]
## [1] -0.9272323 -0.8308059  0.7838722  0.8280914
##
## [[2]]
## [1] 0.14929131 0.07922344 0.35341683 0.28141011 0.20574756 0.67235123
```

```
> sapply(lst2, sort)
```

```
## [[1]]
```

```
## [1] -0.9272323 -0.8308059  0.7838722  0.8280914
##
## [[2]]
## [1] 0.07922344 0.14929131 0.20574756 0.28141011 0.35341683 0.67235123
```

which still returns a `list`. Such a `list` is sometimes called a *ragged array*.

There are also useful functions `tapply()` and `mapply()`, which you may want to explore in your spare time. And if you like them, then take a look a `Map()` and `Reduce()`!

Other miscellaneous functions

There are lots of other handy functions in `R`. For example, I often use `split()`, `match()`, `combn()` and `expand.grid()`. I won't try and list *all* the handy functions here. This sample, however, may give you an idea of the breadth of function that `R` has to offer. Put simply, if you're looking for a function in `R`, there's a good chance that it's there. Sometimes, though, the tricky bit is knowing what to search the web for in order find what the function you want is called!

Control structures

So far we've considered executing *all* of our lines of code. Sometimes, we may want to control how our code is executed according to some logic. We can do this with *control structures*. For example, *conditional execution* only runs lines of code if one or more conditions is met, and *repeated execution* repeatedly runs lines of code until one or more stopping conditions is met.

We sometimes refer to a calculation that's repeated over and over again as a *loop*. For example,

```
> n <- 10
> x <- integer(n)
> for (i in 1:n) {
+   x[i] <- i
+ }
> x
```

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

creates an empty integer vector, `x`, of length `n`, and then `for` is a loop that changes the ith value to i, for $i = 1, \ldots, 10$. (Of course, these first five lines

of code are equivalent to `x <- 1:10`.)

Another commonly used control structure is the `if()` condition, which is often coupled with an `else` condition. Without an `else` condition, nothing happens if the `if()` condition isn't met. Here's a simple example using `if()` and a random draw from the Uniform[0, 1] distribution to mimic a coin toss.

```
> u <- runif(1)
> if (u > 0.5) {
+   x <- 'head'
+ } else {
+   x <- 'tail'
+ }
> x
```

```
## [1] "head"
```

```
> u
```

```
## [1] 0.6788044
```

We could then combine the `for()` and `if()` control flows to mimic repeated coin tosses. The following gives ten:

```
> n <- 10
> x <- character(n)
> for (i in 1:10) {
+   u <- runif(1)
+   if (u > 0.5) {
+     x[i] <- 'head'
+   } else {
+     x[i] <- 'tail'
+   }
+ }
> x
```

```
##  [1] "head" "head" "head" "tail" "tail" "head" "tail" "tail" "tail" "tail"
```

The function `ifelse()` can tidy up `if ... else ...` calls. Equivalently to above we could use `ifelse(runif(1) > 0.5, 'head', 'tail')`, so that the first argument is the `if` condition, the second is what's returned if it's `TRUE` and the third is what's returned if it's `FALSE`. Note that `for()` loops can be a bit untidy, especially if what's inside the loops is just a line of code. Then `replicate()` can be useful. Now that we know `ifelse()`, we could

have just used either of the following

```
> replicate(10, ifelse(runif(1) > 0.5, 'head', 'tail'))
```

```
##  [1] "tail" "head" "head" "head" "tail" "tail" "tail" "head" "head" "head"
```

```
> ifelse(runif(10) > 0.5, 'head', 'tail')
```

```
##  [1] "head" "head" "tail" "tail" "head" "head" "head" "head" "tail" "head"
```

which is equivalent to the above, once the randomness of the call is taken into account.

In the above, we fixed how many coin tosses we wanted, but we might want to stop when we've reached a given number of heads (or tails). For such a random stopping condition, we can use the `while()` control flow. The following stops when we reach four heads.

```
> x <- character(0)
> while(sum(x == 'head') < 4) {
+   u <- runif(1)
+   if (u > 0.5) {
+     x <- c(x, 'head')
+   } else {
+     x <- c(x, 'tail')
+   }
+ }
> x
```

```
## [1] "head" "tail" "tail" "head" "tail" "head" "head"
```

Note that above `x` started as a zero-length vector, and grew at each iteration. This is useful if we don't know how long `x` needs to be. If we do, it's better to set the vector's length at the start.

`R` also has the function `repeat()` for repeating a set of code. Somewhere this will need a stopping condition that invokes the `break` command; otherwise the code will be repeated forever!

Vectorisation

A very useful skill to adopt in `R` early on is to vectorise your code, where possible. Vectorisation typically involves programming with vectors instead of scalars, when such an approach makes sense. Often this means avoiding

writing `for()` loops. Two reasons for this, given in Wickham (2019), are:

1. It makes problems simpler. Instead of having to think about the components of a vector, you can only think about entire vectors.
2. The loops in a vectorised function are written in `C` instead of `R`. Loops in `C` are much faster because they have much less overhead.

Here are a few illuminating examples. Later we'll see what we've gained in efficiency.

Suppose we've got a `vector` comprising some `NA`s (note that we use `NA` in `R` when a value is 'not available', such as missing data) and we want to swap them all for zeros. The function `is.na()` tells us which elements in a `vector` (or `matrix` or `array`) are `NA`.

```
> x <- c(-2.09, NA, -0.25, NA, NA, 0.52, NA, 0.48, 0.29, NA)
> is.na(x)
```

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

We can then subset those elements in a vectorised way and set them to zero.

```
> x[is.na(x)] <- 0
> x
```

```
##  [1] -2.09  0.00 -0.25  0.00  0.00  0.52  0.00  0.48  0.29  0.00
```

We could easily have gone through each element with a `for()` loop, and swapped each `NA` element one at a time for a zero, but that would have taken more complicated code, and, for large problems, such as very long vectors, would be slower at performing the operation. The tidiness of vectorisation should reduce the chance of errors in code, too.

Another rather convincing example is if we have a `vector` and want to know in which interval, from a set of intervals, each of its elements fall.

```
> n <- 10
> x <- runif(10)
> x
```

```
##  [1] 0.1088417 0.1027859 0.4160449 0.1709585 0.5720126 0.9430031 0.4734595 0.15760
```

```
> intervals <- seq(0, 1, by = 0.1)
> m <- length(intervals) - 1
> which_interval <- integer(n)
> for (i in 1:n) {
+   for (j in 1:m) {
+     if (x[i] > intervals[j] & x[i] <= intervals[j + 1]) {
+       which_interval[i] <- j
+     }
+   }
+ }
> which_interval
```

```
##  [1]  2  2  5  2  6 10  5  2  7  5
```

As you might imagine, we're probably not the first person to want to perform such a calculation, and R thinks this too. Hence we can simply use its vectorised `findInterval()` function to perform the equivalent calculation to that above.

```
> findInterval(x, intervals)
```

```
##  [1]  2  2  5  2  6 10  5  2  7  5
```

A related function is `cut()`, which you may want to explore in your own time.

Good practice

Useful tips to remember when coding

Use scripts: don't type in the Console

When programming in R, work from scripts or RMarkdown documents.

Use functions for repeated calculations

If you're using a piece of code more than once, it should probably be formulated as a `function`.

Avoid repeating lines of code

If you're copying and pasting code, think about whether this can be avoided.

Here's an example of 'the bad'.

```
> n <- 5
> x <- matrix(0, nrow = n, ncol = 4)
> x[, 1] <- rnorm(n, 0, 1.0)
> x[, 2] <- rnorm(n, 0, 1.5)
> x[, 3] <- rnorm(n, 0, 2.0)
> x[, 4] <- rnorm(n, 0, 3.0)
```

Here's an example of something better, given `n` and `x` above.

```
> sds <- c(1.0, 1.5, 2.0, 3.0)
> for (i in 1:4) {
+   x[, i] <- rnorm(n, 0, sds[i])
+ }
```

An improvement would be to swap `for (i in 1:4)` with `for (i in 1:length(sds))` or `for (i in 1:ncol(x))`, as then we're not relying on remembering the length of `sds` or equivalently the number of columns of `x`. Even more tidily, we could use either of the following:

```
> x1 <- sapply(sds, rnorm, n = n, mean = 0)
> x2 <- sapply(sds, function(z) rnorm(n, 0, z))
```

For `x1` we've relied on knowing the order of the arguments to `rnorm()`, and by fixing its first and second arguments, `n` and `mean`, `R` knows that the first argument that we're supplying to `sapply()` should be `sd`, its third argument, i.e. `rnorm()`'s first free argument. This approach relies on good knowledge of a function's arguments.

Write with style

In `R` we use the `<-` symbol to assign an object to a name. The left-hand side of the `<-` symbol is the name we're giving the object, and the right-hand side is the code that defines the object. Wickham (2019, sec. 5.1.2) states: 'Variable and function names should be lowercase. Use an underscore to separate words within a name.' Particularly usefully, Wickham (2019, sec. 5.1.2) also states: 'Strive for names that are concise and meaningful (this is not easy!)'. It is not easy, but worth aiming towards, especially if you're re-using an object multiple times.

Spacing is particularly useful for helping the appearance of code. For example, the two lines of code

```
> x1 <- sapply(sds, rnorm, n = n, mean = 0)
> x1<-sapply(sds,rnorm,n=n,mean=0)
```

will both make the same object `x1`, but, I hope you'll agree, the first line is easier to read. In general, spacing should be used either side of `<-`, mathematical operators (e.g. `=`, `+`, `*`, and control flows (e.g. `if (...)` not `if(...)`), and after commas. Spacing can also be used to align arguments, such as

```
> x <- list(a = c(1, 2, 3,  4,  5,  6),
+           b = c(7, 8, 9, 10, 11, 12))
```

which can sometimes make code more readable.

Comment your code

What a line of code does might be self-explanatory, but might not. When it's not, add comments to explain what it does. This is particularly useful at the start of a function, when what a function does and its arguments can be stated.

```
> fn <- function(x, y, z) {
+ # function to compute (x + y) * z element-wise
+ # x, y and z can be compatible scalars, vectors, matrices or arrays
+ # returns a scalar, vector, matrix or array
+ # (depending on class of x, y and z)
+ xplusy <- x + y
+ xplusy * z
+ }
> fn(2, 3, 4)
```

```
## [1] 20
```

In the above we could – and should – have just done `(x + y) * z` on one line, but we'll find the formulation above useful for later.

Commenting code is essential if you're sharing code. You will be sharing code in MTH3045 to submit assignments. Marks will be given for sensible commenting, and may also be given if comments make clear your intentions, even if the code itself contains errors. Commenting is one of those things when coding that can take a bit of discipline. It's almost always more exciting, for example, to produce a working function as quickly as possible, than to

take a bit longer and also comment that function. I tend to think there's a balance between when to comment and when to code, and might delay commenting until a function is finished. Always comment code, though, while it's fresh in your mind.

Debugging

When we write code in R, we don't always get it right first time. The errors or, more commonly, *bugs* in our code may not be straightforward to spot. We call identifying bugs *debugging*. We want an efficient strategy to identify bugs so that we can fix them.

Let's assume that our bug lies within a `function` somewhere. We've tried to run the function, and it's failed. I tend to go through the following sequence of events.

1. Read the error message R has returned, if any. From this we may be able to deduce where the bug in our code is.
2. If 1. fails, inspect the code and hope to spot the bug (if there's any hope of spotting it); otherwise
3. Use some of R's functions to help us debug our function.

Somewhere within our function, something must not be as we expect. We can inspect what's inside a function with `debug()`. I prefer `debugonce()`, which inspects a function once, as opposed to every time. Suppose we want to debug `fn()` above. We simply issue

```
> debugonce(fn)
> fn(2, 3, 4)
```

in the Console, and then we'll end up inside the function. So if we type `x` is the Console, R will print `2`. It will also be showing the line of code that it's about to execute, which will be `xplusy <- x + y`. If we hit Enter then R will run the line of code. If we then type `xplusy` in the Console R will print `5`. Debugging lets us sequentially go through each line of a function. When debugging, R will also throw up an error once we try and execute the line of code where the bug is. This approach to debugging can help us find the offending line of code, and then we may want to debug again up to that line, in order to find what the problem is.

When our `function` has many lines of code, and we know the bug to be near the end, we may want to choose from which point of the `function` our

debugging starts. We can do this with `browser()`.

Profiling and benchmarking

You may have heard of the term *Big Data*. Essentially this means lots of data.

> "The definition of big data is data that contains greater variety, arriving in increasing volumes and with more velocity. This is also known as the three Vs.
>
> Put simply, big data is larger, more complex data sets, especially from new data sources." — (https://www.oracle.com/uk/big-data/what-is-big-data/)

The more data we attempt to fit a statistical model to, the more flops involved, and, in general, the longer it takes to fit and the more memory it needs. Typically we should try and use all the *useful* data that we can. If data aren't useful, we shouldn't use them.

Profiling is the analysis of computer code, typically of its time taken or memory used.

Let's consider a matrix-based update to `fn()` above, which we'll call `fn2()`, and computes $(A + B)C$ for matrices A, B and C.

```
> fn2 <- function(x, y, z) {
+ # function to compute (x + y) %*% z
+ # x, y and z are matrices with compatible dimensions
+ # returns a matrix
+ xplusy <- x + y
+ xplusy %*% z
+ }
> n <- 5e2
> p <- 1e4
> A <- matrix(runif(n * p), n, p)
> B <- matrix(runif(n * p), n, p)
> C <- matrix(runif(n * p), p, n)
```

We can profile with `Rprof()`. Here we'll ask it to write what it's currently doing to file `profile.txt` every 0.00001 seconds. Note that `Rprof(NULL)` ends the profiling.

```
> Rprof('profile.txt', interval = 1e-5)
> D <- fn2(A, B, C)
> Rprof(NULL)
```

Here's what's in `profile.txt`

```
## sample.interval=10
## "+" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
## "%*%" "fn2"
```

which is two-column output, as there are at most two functions active. The second column is the first function to be called and then the second is any subsequent functions. From the first column, we see that after 0.00001 seconds `R` is evaluating function `+`, i.e. matrix addition; and for the next 17 0.00001-second intervals `R` is evaluating function `%*%`, i.e. matrix multiplication. The second column tells us that `R` is evaluating `fn2` throughout. For $n \times p$ matrices, matrix addition requires np additions, whereas matrix multiplication requires p multiplications and $p-1$ additions, each repeated np times. Considering only the dominant terms, we write the computational complexity of matrix multiplication as $O(np^2)$ whereas that of matrix addition is $O(np)$, which uses so-called 'big-O' notation[4].

[4]big-O notation – Consider functions $f()$ and $g()$. We write $f(x) = O(g(x))$ if and only if there exist constants N and C such that $|f(x)| \leq C|g(x)| \; \forall \; x > N$. Put simply, this means that $f()$ does not grow faster than $g()$.

Instead of trying to interpret the output of `Rprof()`, R's `summaryRprof()` function will do that for us

```
> summaryRprof('profile.txt')
```

```
## $by.self
## [1] self.time  self.pct   total.time total.pct
## <0 rows> (or 0-length row.names)
##
## $by.total
##       total.time total.pct self.time self.pct
## "fn2"          0    100.00         0     0.00
## "%*%"          0     94.44         0    94.44
## "+"            0      5.56         0     5.56
##
## $sample.interval
## [1] 1e-05
##
## $sampling.time
## [1] 0.00018
```

by working out the percentage of information in the `Rprof()` output attributable to each unique line of output.

Profiling can be useful for finding *bottlenecks* in our code, i.e. lines that heavily contribute to the overall computational expense (e.g. time or memory) of the code.

If we find a bottleneck *and* it's unbearably slow *and* we think there's scope to reduce or eliminate it without disproportionate effort, then we might consider changing our code to make it more efficient. We'll focus on efficiency in terms of times taken for commands to execute.

Suppose that we've put together some code, which we consider to be the *benchmark* that we want to improve on. Comparison against a benchmark is called *benchmarking*. One of the simplest ways to benchmark in `R` is with `system.time()`, which we saw in the first lecture. Suppose we've got the following line in our code, based on matrix `A` above.

```
> a_sum <- apply(A, 1, sum)
```

The following tells us how long it takes to execute.

```
> system.time(a_sum <- apply(A, 1, sum))
```

```
##    user  system elapsed
##   0.049   0.021   0.069
```

This gives us three timings. The last, `elapsed`, tells use how long our code has taken to execute in seconds. Then `user` and `system` partition this total time into so-called 'user time' and 'system time'. Their definitions are operating system dependent, but this information from the `R` help file for `proc.time()` gives as idea. "The 'user time' is the CPU time charged for the execution of user instructions of the calling process. The 'system time' is the CPU time charged for execution by the system on behalf of the calling process." We'll just consider `elapsed` time for MTH3045.

For benchmarking in `R` we'll consider the `microbenchmark::microbenchmark()`. Note that this notation refers to function `microbenchmark()` within the `microbenchmark` package. If we have the `microbenchmark` installed, we can either issue `microbenchmark::microbenchmark()` to use the function or load the package, i.e. `library(microbenchmark)`, and then just use `microbenchmark()`. I'll initially use the `::` notation to introduce any functions in `R` that aren't loaded by default when `R` starts.

```
> library(microbenchmark)
> microbenchmark(
+   apply(A, 1, sum),
+   rowSums(A)
+ )
```

```
## Unit: milliseconds
##              expr       min        lq      mean   median        uq        max
##  apply(A, 1, sum) 67.307182 73.689749 74.215260 73.92187 74.193785 112.463168
##        rowSums(A)  6.879372  6.988103  7.040655  7.04430  7.096565   7.230114
##  neval cld
##    100   b
##    100  a
```

The `microbenchmark::microbenchmark()` function is particularly handy because it automatically chooses its units, which here is milliseconds. For functions that take longer to execute it might, for example, choose seconds. The output of `microbenchmark::microbenchmark()` includes `neval`, the number of evaluations it's used for each function; from these, the range and quartiles are calculated. If we compare medians, we note that the `rowSums()`

approach is about an order magnitude faster than the `apply()` approach. We should note that for either approach, timings differ between evaluations, even though they're doing the same calculation and giving the same answer. On this occasion the minimum and maximum times are between two and three factors different.

Note that we could also use `benchmark::rbenchmark()` for benchmarking, which gives similar details to `system.time()`. For MTH3045, I'll use `microbenchmark::microbenchmark()`, because I find its output more useful.

Compiled code with `Rcpp`

We've seen that vectorised functions can simplify our code (and later we'll see that they can bring considerable gains in computation time). Suppose, though, that we want a vectorised function, but that it doesn't exist for our purposes. We could write a function in `C` or `FORTRAN`. However, using `Rcpp` is much more convenient. `Rcpp` is an `R` package that efficiently and tidily links `R` and `C++`.

Let's consider a simple example of a function to calculate the sum of a vector.

```
> sum_R <- function(x) {
+ # function to calculate sum of a vector
+ # x is a vector
+ # returns a scalar
+ out <- 0
+ for (i in 1:length(x))
+   out <- out + x[i]
+ out
+ }
```

Obviously, we should use `sum()`, but if it wasn't available to us, then the above would be an alternative.

Consider the following `C++` function, which I've got stored as `sum_Rcpp.cpp`, so that the contents of the .cpp file are as below.

```
> #include <RcppArmadillo.h>
+ // [[Rcpp::depends(RcppArmadillo)]]
+
+ // [[Rcpp::export]]
+ double sum_Rcpp(arma::vec x) {
```

```
+ double out = 0.0;
+ int n = x.size();
+ for (int i = 0; i++; i < n) {
+   out += x[i];
+ }
+ return out;
+ }
```

We won't go into great detail on `Rcpp` in MTH3045. The purpose of this section is to raise awareness of its existence, should you ever need it. In the above `.cpp` file:

- `#include <RcppArmadillo.h>` and `\\ [[Rcpp::depends(RcppArmadillo)]]` point to the `RcppArmadillo` library;
- `// [[Rcpp::export]]` makes a function that is visible to `R`;
- `double sum_Rcpp(arma::vec x) {` specifies that our function returns a `double`, i.e. a single value of double-precision, is called `sum_Rcpp`, and that its one argument is a vector, hence `arma::vec`, which we're calling `x`;
- `double out = 0.0;` forms the double that will be returned, and sets its initial value to 0.0;
- `int n = x.size();` finds the length of `x` and stores it as an integer called `n`;
- `for (int i = 0; i++; i < n) {` initiates a loop: we've called our index `i` and specified that it's an integer; then we've specified that it goes up by one each time with `i++`, and stops at `n - 1` with `i < n`. (Note here that indices in C++ start at zero, whereas they start at one in `R`.)
- We then update `out` at each iteration by adding `x[i]`. (Note that `out += ...` is equivalent to `out = out + ...` .)
- We then close the loop, and specify what we return.

An important difference between `R` and C++ code is that for the latter we're specifying what's an integer and what's a double. Then `Rcpp::sourceCpp()` checks that what we've specified is okay. By not specifying these normally in `R`, time is needed to interpret the code. We avoid these overheads with C++ code. The trade-off is that C++ code usually takes a bit longer to write, because we need to think about the form of our data, whereas `R` can handle most of this for us.

To use the `cpp` function in `R`, we compile it with `Rcpp::sourceCpp()`.

We're also going to load `RcppArmadillo`, which gives access to the excellent `Armadillo` C++ library for linear algebra and scientific computing, more details of which can be found at http://arma.sourceforge.net/docs.html.

We can then perform a quick benchmark on what we've done.

```
> library(RcppArmadillo)
> Rcpp::sourceCpp('sum_Rcpp.cpp')
> x <- runif(1e3)
> microbenchmark::microbenchmark(
+   sum_R(x),
+   sum_Rcpp(x),
+   sum(x)
+ )
```

```
## Unit: nanoseconds
```

We see that `sum_Rcpp()` is typically at least an order of magnitude faster than `sum_R()`. However, it's still slower than `sum()`, because it's one of `R`'s functions that's heavily optimised for efficiency. It's great that we have such efficiency at our disposal.

Bibliographic notes

For further details on topics covered in this chapter, consider the following.

- Positional number systems: Press et al. (2007, sec. 1.1.1) and Monahan (2011, sec. 2.2).
- Fundamentals of programming in `R`: W. N. Venables and R Core Team (2021), Wickham (2019, Ch. 2) and Grolemund (2014, Ch. 1-5).
- Profiling and benchmarking: Wickham (2019, Ch. 22-24) and almost all of Gillespie and Lovelace (2016), especially Section 1.6.
- `R` coding style: Various parts of Wickham (2019) and Gillespie and Lovelace (2016).

Matrix-based computing

Motivation

Perhaps surprisingly, much of a computation that we do when fitting a statistical model can be formulated with matrices. The linear model is a prime example. In this chapter we'll explore some key aspects of matrices and calculations involving them that are important for statistical computing. A particularly useful reference for matrices, especially in the context of statistical computing, is the Matrix Cookbook (Petersen and Pedersen (2012)).

Definitions

Matrix properties

Let's consider an $n \times n$ matrix $\mathbf{A}$ and $n \times p$ matrix $\mathbf{B}$. We will let A_{ij}, for $i, j = 1, \ldots, n$ denote the (i, j)th element of $\mathbf{A}$, i.e. in row i and column j. We'll assume that $\mathbf{A}$ is stored in `R` as `A`.

```
> A %*% B # computes AB for matrices A and B
```

$\mathbf{A}$ is **real** if all its elements are real numbers. (We'll only consider real matrices in MTH3045, so 'real' may be taken as given.)

```
> !is.complex(A) && is.finite(A) # checks whether a matrix A is real
```

The **transpose** of a matrix, denoted $\mathbf{A}^{\mathrm{T}}$, is given by interchanging the rows and columns of $\mathbf{A}$.

```
> t(A) # computes the transpose of a matrix A
```

The **cross product** of matrices $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A}^{\mathrm{T}}\mathbf{B}$.

```
> crossprod(A, B) # computes t(A) %*% B
```

`crossprod(A, B)` is more efficient that `t(A) %*% B` because `R` recognises that it doesn't need to transpose `A` and can instead perform a modified matrix multiplication in which the columns of `A` are multiplied by the columns of `B`.

```
> tcrossprod(A, B) # computes A %*% t(B)
```

`crossprod(A)` is equivalent to `crossprod(A, A)` and `tcrossprod(A)` to `tcrossprod(A, A)`.

A matrix is **diagonal** if its values are zero everywhere, except for its diagonal, i.e. $A_{ij} = 0$ for $i \neq j$.

A matrix is **square** if it has the same numbers of rows and columns.

The **rank** of $\mathbf{A}$, denoted rank($\mathbf{A}$), is the dimension of the vector space generated (or spanned) by its columns. This corresponds to the maximal number of linearly independent columns of $\mathbf{A}$. A matrix is of *full rank* if its rank is equal to its number of rows.

The following apply *only* to square matrices.

- The $n \times n$ identity matrix, denoted $\mathbf{I}_n$, is diagonal *and* all its diagonal elements are one.

```
> diag(n) # creates the n x n identity matrix for integer n
```

- $\mathbf{A}$ is **orthogonal** if $\mathbf{A}^{\mathrm{T}}\mathbf{A} = \mathbf{I}_n$ and $\mathbf{A}\mathbf{A}^{\mathrm{T}} = \mathbf{I}_n$.
- $\mathbf{A}$ is **symmetric** if $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$.
- The **trace** of $\mathbf{A}$, denoted tr($\mathbf{A}$), is the sum of its diagonal entries, i.e. $\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{n} A_{ii}$. In `R`, `diag(A)` extracts the diagonal elements of `A`, and so `sum(diag(A))` computes the trace of `A`.
- $\mathbf{A}$ is **invertible** is there exists a matrix $\mathbf{B}$ such that $\mathbf{AB} = \mathbf{I}_n$. Note that $\mathbf{B}$ must be $n \times n$.
- The **inverse** of $\mathbf{A}$, if it exists, is denoted $\mathbf{A}^{-1}$.

```
> solve(A) # computes the inverse of A
```

- A symmetric matrix $\mathbf{A}$ is **positive definite** if $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} > 0$ for all non-zero $\mathbf{x}$, i.e. provided all elements of $\mathbf{x}$ aren't zero. (Changes to the inequality define positive semi-definite ($\geq$), negative semi-definite ($\leq$),

and negative definite ($<$) matrices, but in statistical computing it's usually positive definite matrices that we encounter.)

The Hilbert matrix, $\mathbf{H}_n$, is the $n \times n$ matrix with (i, j)th elements $1/(i+j-1)$ for $i, j = 1, \ldots, n$. Write a function to form a Hilbert matrix for arbitrary n. Use this to form $\mathbf{H}_3$ and then check whether the matrix that you have formed is symmetric.

There are many ways that we could write this function. We should, though, avoid a `for` loop. Here's one option.

```
> hilbert <- function(n) {
+   # Function to evaluate n by n Hilbert matrix.
+   # Returns n by n matrix.
+   ind <- 1:n
+   1 / (outer(ind, ind, FUN = '+') - 1)
+ }
> H <- hilbert(3)
> H
```

```
##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.5000000 0.3333333
## [2,] 0.5000000 0.3333333 0.2500000
## [3,] 0.3333333 0.2500000 0.2000000
```

A matrix is symmetric if it and its transpose are equal. There a various ways we can check this.

```
> H - t(H) # should be all zero
```

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

```
> all.equal(H, t(H)) # should be TRUE
```

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

Or we can turn to `isSymmetric()`, which is R's function for checking matrix symmetry.

```
> isSymmetric(H) # should be TRUE
```

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

Let $\mathbf{Y} \sim MVN_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ denote a random p-vector with a multivariate Normal (MVN) distribution that has mean vector $\boldsymbol{\mu}$ and variance-covariance matrix $\boldsymbol{\Sigma}$. Its probability density function (pdf) is then

$$f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^p |\boldsymbol{\Sigma}|}} \exp\left\{-\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{y}-\boldsymbol{\mu})\right\}. (\#eq:dmvn) \tag{1}$$

Thus note that, to compute the MVN pdf, we need to consider both the determinant and inverse of $\boldsymbol{\Sigma}$, amongst other calculations.

Write a function `dmvn1()` to evaluate its pdf in `R`, and then evaluate $\log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$ for

$$\mathbf{y} = \begin{pmatrix} 0.7 \\ 1.3 \\ 2.6 \end{pmatrix}, \qquad \boldsymbol{\mu} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \qquad \boldsymbol{\Sigma} = \begin{pmatrix} 4 & 2 & 1 \\ 2 & 3 & 2 \\ 1 & 2 & 2 \end{pmatrix}.$$

The function `dmvn1()` below evaluates the multivariate Normal pdf

```
> dmvn1 <- function(y, mu, Sigma, log = TRUE) {
+   # Function to evaluate multivariate Normal pdf
+   # y and mu are vectors
+   # Sigma is a square matrix
+   # log is a logical
+   # Returns scalar, on log scale, if log == TRUE.
+   p <- length(y)
+   res <- y - mu
+   out <- - 0.5 * determinant(Sigma)$modulus - 0.5 * p * log(2 * pi) -
+              0.5 * t(res) %*% solve(Sigma) %*% res
+   if (!log)
+     out <- exp(out)
+   out
+ }
```

although we'll later see that this is a crude attempt. Then the following create $\mathbf{y}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as objects `y`, `mu` and `Sigma`, respectively.

```
> y <- c(.7, 1.3, 2.6)
> mu <- 1:3
> Sigma <- matrix(c(4, 2, 1, 2, 3, 2, 1, 2, 2), 3, 3)
```

Then we evaluate $\log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with

```
> dmvn1(y, mu, Sigma)
```

```
##            [,1]
## [1,] -3.654535
## attr(,"logarithm")
## [1] TRUE
```

Note that above `determinant()$modulus` directly calculates `log(det())`, and is usually more reliable, so should be used when possible.

In general, it is much more sensible to work with log-likelihoods, and then if the likelihood itself is actually sought, simply exponentiate the log-likelihood at the end. This has been implemented for `dmvn1()`. This will sometimes avoid underflow.

Special matrices

Diagonal, band-diagonal and triangular matrices

The following gives examples of various special types of square matrix, which we sometimes encounter in statistical computing. These are diagonal (as defined above), tridiagonal, block diagonal, band and lower triangular matrices. Instead of defining them formally, we'll just show schematics of each. These are plotted in Figure @ref(fig:matrices) with `image()`, which plots the rows along the x-axis and columns across the y-axis. Hence, to visualise actually looking at the matrix written down on paper, each plot should be considered rotated clockwise through 90 degrees.

Sparse matrices

A matrix is **sparse** if most of its elements are zero.

The definition of a sparse matrix is rather vague. Although no specific criterion exists in terms of the proportion of zeros, some consider that the number of non-zero elements should be similar to the number of rows or columns.

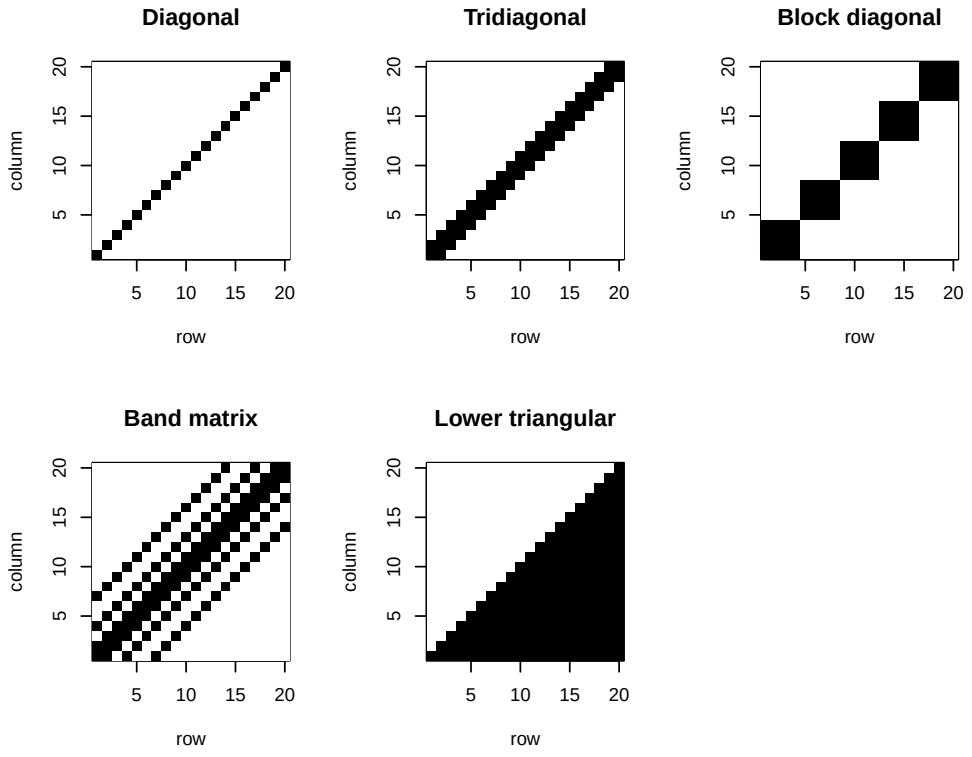


Figure 1: Schematics of diagonal, tridiagonal, block diagonal, band and lower triangular matrices.

Systems of linear equations

Systems of linear equations of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, (\#eq : Axb) \tag{2}$$

where $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{x}$ and $\mathbf{b}$ are n-vectors are often encountered in statistical computing. The multivariate Normal pdf of equation @ref(eq:dmvn) is one example: we don't need to compute $\mathbf{\Sigma}^{-1}$ and then calculate $\mathbf{z} = \mathbf{\Sigma}^{-1}(\mathbf{y} - \boldsymbol{\mu})$; instead, left-multiplying by $\mathbf{\Sigma}$, we can recognise that $\mathbf{z}$ is the solution to $\mathbf{\Sigma}\mathbf{z} = \mathbf{y} - \boldsymbol{\mu}$. R's `solve()` function can not only invert a matrix, but can also solve a system of linear equations. Given equation @ref(eq:Axb), suppose we have $\mathbf{A}$ and $\mathbf{b}$ stored as `A` and `b`, respectively, then we obtain $\mathbf{x}$, which we'll store as `x`, with `x <- solve(A, b)`.

Modify the function `dmvn1()` used in Example @ref(exm:mvn1) to give a new function `dmvn2()` in which, instead of inverting $\mathbf{\Sigma}$, the system of linear equations $\mathbf{\Sigma}(\mathbf{y} - \boldsymbol{\mu})$ is solved.

We simply need to replace `solve(Sigma) %*% res` with `solve(Sigma, res)`, giving `dmvn2()` as follows

```
> dmvn2 <- function(y, mu, Sigma, log = TRUE) {
+   # Function to evaluate multivariate Normal pdf by solving
+   # a system of linear equations
+   # y and mu are vectors
+   # Sigma is a square matrix
+   # log is a logical
+   # Returns scalar, on log scale, if log == TRUE.
+   p <- length(y)
+   res <- y - mu
+   out <- - 0.5 * determinant(Sigma)$modulus - 0.5 * p * log(2 * pi) -
+            0.5 * t(res) %*% solve(Sigma, res)
+   if (!log)
+     out <- exp(out)
+   out
+ }
```

which reassuringly gives the same answer as `dmvn1()`.

```
> dmvn2(y, mu, Sigma)
```

```
##           [,1]
```

```
## [1,] -3.654535
## attr(,"logarithm")
## [1] TRUE
```

In general, solving a system of linear equations is faster and more numerically stable than inverting and multiplying. The latter is essentially a result of reducing numerical errors, as discussed in Chapter @ref(ch2).

An **elementary row operation** on a matrix is any one of the following.

- Type-I: interchange two rows of the matrix;
- Type-II: multiply a row by a nonzero scalar;
- Type-III: replace a row by the sum of that row and a scalar multiple of another row.

A $m \times n$ matrix $\mathbf{U}$ is said to be in **row echelon form** if the following two conditions hold.

- If a row $\mathbf{u}_{i*}^{\mathrm{T}}$ comprises all zeros, i.e. $\mathbf{u}_{i*}^{\mathrm{T}} = \mathbf{0}^{\mathrm{T}}$, then all rows below also comprise all zeros.
- If the first nonzero element of $\mathbf{u}_{i*}^{\mathrm{T}}$ is the jth element, then the jth element in all rows below is zero.

Gaussian elimination is perhaps the best established method for solving systems of linear equations. In MTH3045 you won't be examined on Gaussian elimination, but it will be useful to be familiar with how it works, so that the virtues of the matrix decompositions that follow will become apparent.

The system of linear equations $\mathbf{Ax} = \mathbf{b}$ may be verbosely written as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n. \end{aligned} \qquad (\#eq : Axb2) \tag{3}$$

The aim of Gaussian elimination is to use elementary row operations to transform @ref(eq:Axb2) into an equivalent but triangular system.

Instead of algebraically writing the algorithm for Gaussian elimination, it will be simpler to consider a numerical example in which we want to solve the following system of four equations in four variables

$$\begin{array}{rcrcrcrl} 2x_1 & + & 3x_2 & & & & & = 1 \\ 4x_1 & + & 7x_2 & + & 2x_3 & & & = 2 \\ -6x_1 & - & 10x_2 & + & x_4 & & & = 1 \\ 4x_1 & + & 6x_2 & + & 4x_3 & + & 5x_4 & = 0 \end{array}$$

and for which we'll write the coefficients of the x_is and $\mathbf{b}$ in the augmented matrix form

$$\left[\begin{array}{rrrr|r} 2 & 3 & 0 & 0 & 1 \\ 4 & 7 & 2 & 0 & 2 \\ -6 & -10 & 0 & 1 & 1 \\ 4 & 6 & 4 & 5 & 0 \end{array}\right]$$

for convenience.

We start by choosing the **pivot**. Our first choice is a coefficient of x_1. We can choose any nonzero coefficient. Anything below this is set to zero through elementary operations. We'll choose a_{11} as the pivot and then perform the following elementary matrix operations: row 2 $\rightarrow$ 2 $\times$ row 1 + - 1 $\times$ row 2; row 3 $\rightarrow$ 3 $\times$ row 1 + 1 $\times$ row 3; row 4 $\rightarrow$ -2 $\times$ row 1 + row 4, which gives the following transformation of the above augmented matrix.

$$\left[\begin{array}{rrrr|r} 2 & 3 & 0 & 0 & 1 \\ 4 & 7 & 2 & 0 & 2 \\ -6 & -10 & 0 & 1 & 1 \\ 4 & 6 & 4 & 5 & 0 \end{array}\right] \rightarrow \left[\begin{array}{rrrr|r} 2 & 3 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 & 0 \\ 0 & -1 & 0 & 1 & 4 \\ 0 & 0 & 4 & 5 & -2 \end{array}\right]$$

Repeating this with the element in the position of a_{22} we get

$$\left[\begin{array}{rrrr|r} 2 & 3 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 & 0 \\ 0 & -1 & 0 & 1 & 4 \\ 0 & 0 & 4 & 5 & -2 \end{array}\right] \rightarrow \left[\begin{array}{rrrr|r} 2 & 3 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 & 4 \\ 0 & 0 & 4 & 5 & -2 \end{array}\right]$$

and then again with the element in the position of a_{33} we get

$$\left[\begin{array}{cccc|c} 2 & 3 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 & 0 \\ 0 & -1 & 0 & 1 & 4 \\ 0 & 0 & 4 & 5 & -2 \end{array}\right] \rightarrow \left[\begin{array}{cccc|c} 2 & 3 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 & 4 \\ 0 & 0 & 0 & 3 & -10 \end{array}\right]$$

The above operations have **triangularised** the system of linear equations, i.e. with an augmented matrix of the form

$$\left[\begin{array}{cccc|c} u_{11} & u_{12} & \dots & u_{1n} & b_1^* \\ 0 & u_{22} & \dots & u_{2n} & b_2^* \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & \dots & u_{nn} & b_n^* \end{array}\right]$$

which can be tidily written as $\mathbf{Ux} = \mathbf{b}^*$. It is straightforward to find $\mathbf{x}$ from such a system, because, if we turn to the above example, $x_4 = -10/3$, which can be substituted in the above line to give $x_3 = 11/3$, and so forth gives $x_2 = -22/3$ and $x_1 = 23/2$, i.e. $\mathbf{x} = (23/2, -22/3, 11/3, -10/3)^{\mathrm{T}}$.

The above is an example of **backward substitution**.

Consider the system of linear equations given by $\mathbf{Ux} = \mathbf{b}$, where $\mathbf{U}$ is an $n \times n$ upper triangular matrix. We can find $\mathbf{x}$ by **backward substitution** through the following steps.

1. Calculate $x_n = b_n/u_{nn}$.
2. For $i = n-1, n-2, \dots, 2, 1$, recursively compute

$$x_i = \frac{1}{u_{ii}} \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j \right).$$

Note that **forward substitution** is simply the analogous process of backward substitution where we find a lower triangular matrix, and then solve for x_1, x_2 given x_1, and so forth.

Gaussian elimination is the two-stage process of forming the triangular matrix and then performing backward substitution.

If we want to perform backward or forward substitution in `R` we should use `backsolve()` and `forwardsolve()`, respectively. These have usage

```
>   backsolve(r, x, k = ncol(r), upper.tri = TRUE, transpose = FALSE)
>   forwardsolve(l, x, k = ncol(l), upper.tri = FALSE, transpose = FALSE)
```

So `backsolve()` expects a right upper-triangular matrix and `forwardsolve()` expects a left lower-triangular matrix.

Confirm that $\mathbf{x} = (23/2, -22/3, 11/3, -10/3)^{\mathrm{T}}$ using `backsolve()`.

We need to input the upper-triangular matrix $\mathbf{U}$ and $\mathbf{b}^*$, which we'll call `U` and `bstar`, respectively.

```
> U <- rbind(
+   c(2, 3, 0, 0),
+   c(0, 1, 2, 0),
+   c(0, 0, 2, 1),
+   c(0, 0, 0, 3)
+ )
> bstar <- c(1, 0, 4, -10)
> backsolve(U, bstar)
```

```
## [1] 11.500000 -7.333333  3.666667 -3.333333
```

We may want to solve multiple systems of linear equations of the form $\mathbf{A}\mathbf{x}_1 = \mathbf{b}_1$, $\mathbf{A}\mathbf{x}_2 = \mathbf{b}_2$, ..., $\mathbf{A}\mathbf{x}_p = \mathbf{b}_p$, which can be written with matrices as $\mathbf{A}\mathbf{X} = \mathbf{B}$ for $n \times p$ matrices $\mathbf{X}$ and $\mathbf{B}$. In this situation we can recognise that we only triangularise $\mathbf{A}$ once, and then use that triangularisation to go through the back substitution algorithm p times.

We can find the inverse of $\mathbf{A}$ by solving $\mathbf{A}\mathbf{X} = \mathbf{I}_n$ for $\mathbf{X}$ and then setting $\mathbf{A}^{-1} = \mathbf{X}$.

A $m \times n$ matrix $\mathbf{U}$ is said to be in **reduced row echelon form** if the following two conditions hold.

1. It is in row echelon form;
2. The first nonzero element of each row is one;
3. All entries above each pivot are zero.

Consider Gaussian elimination of the 3×3 matrix $\mathbf{A}$ given by

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 1 \\ -2 & -3 & 1 \\ 3 & 5 & 0 \end{bmatrix}$$

We can go through the following steps to transform the matrix to reduced row echelon form.

$$\begin{bmatrix} 1 & 2 & 1 \\ -2 & -3 & 1 \\ 3 & 5 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 3 \\ 0 & -1 & -3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & -5 \\ 0 & 1 & 3 \\ 0 & 0 & 0 \end{bmatrix}$$

We've ended up with only two nonzero rows, and hence $\mathbf{A}$ has rank 2, and because it has three rows it is therefore **rank deficient**.

Use `R` to find $\mathbf{x}$ such that $\mathbf{Dx} = \mathbf{b}$ and $\mathbf{Lx} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{b}$ where

$$\mathbf{D} = \begin{pmatrix} -0.75 & 0.00 & 0.00 \\ 0.00 & -0.61 & 0.00 \\ 0.00 & 0.00 & -0.28 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 1.00 & -0.19 & 0.89 \\ 0.00 & 0.43 & 0.02 \\ 0.00 & 0.00 & -0.20 \end{pmatrix},$$

$$\mathbf{L} = \begin{pmatrix} -0.72 & 0.00 & 0.00 \\ 0.00 & 2.87 & 0.00 \\ -1.94 & -2.04 & 0.81 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} -2.98 \\ 0.39 \\ 0.36 \end{pmatrix}.$$

We'll start by loading $\mathbf{b}$, $\mathbf{D}$, $\mathbf{U}$ and $\mathbf{L}$ and which we'll store as `b`, `D`, `U` and `L`, respectively.

```
> D <- diag(c(-0.75, -0.61, -0.28))
> L <- matrix(c(-0.72, 0, -1.94, 0, 2.87, -2.04, 0, 0, 0.81), 3, 3)
> U <- matrix(c(1, 0, 0, -0.19, 0.43, 0, 0.89, 0.02, -0.2), 3, 3)
> b <- c(-2.98, 0.39, 0.36)
```

We can solve $\mathbf{Dx} = \mathbf{b}$ for $\mathbf{x}$ with the following two lines of code

```
> solve(D, b)
```

```
## [1]  3.9733333 -0.6393443 -1.2857143
```

```
> b / diag(D)
```

```
## [1]  3.9733333 -0.6393443 -1.2857143
```

but should note that the latter uses fewer calculations, and so is more efficient and hence better.

Then we can solve $\mathbf{Ux} = \mathbf{b}$ for $\mathbf{x}$ with the following two lines of code

```
> solve(U, b)
```

```
## [1] -1.1897674  0.9906977 -1.8000000
```

```
> backsolve(U, b)
```

```
## [1] -1.1897674  0.9906977 -1.8000000
```

and $\mathbf{Lx} = \mathbf{b}$ for $\mathbf{x}$ with the following two lines of code

```
> solve(L, b)
```

```
## [1]  4.1388889  0.1358885 10.6995765
```

```
> forwardsolve(L, b)
```

```
## [1]  4.1388889  0.1358885 10.6995765
```

and on both these occasions the latter is more efficient because it uses fewer calculations.

Matrix decompositions

Cholesky decomposition

Any positive definite real matrix $\mathbf{A}$ can be factorised as

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}} (\#eq : chol) \tag{4}$$

where $\mathbf{L}$ is a real lower-triangular matrix of the same dimension as $\mathbf{A}$ with positive diagonal entries. The factorisation in @ref(eq:chol) is the **Cholesky decomposition**[5].

In general, we won't be concerned with algorithms for computing matrix decompositions in MTH3045. However, the algorithm for computing the Cholesky decomposition is rather elegant, and so is given below. You won't,

[5]André-Louis Cholesky (15 Oct 1875 – 31 Aug 1918) was a French military officer and mathematician. He worked in geodesy and cartography, and was involved in the surveying of Crete and North Africa before World War I. He is primarily remembered for the development of a matrix decomposition known as the Cholesky decomposition which he used in his surveying work.His discovery was published posthumously by his fellow officer Commandant Benoît in the Bulletin Géodésique.

however, be expected to use it. Consider the following matrices

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \qquad \mathbf{L} = \begin{bmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix}$$

where $\mathbf{A}$ is symmetric and non-singular. The entries of $\mathbf{L}$ are given by

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}, \qquad l_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} l_{jk}}{l_{ii}}, \text{ for } i > j.$$

```
> chol(A) # computes the Cholesky decomposition of a square matrix A
```

The `chol()` function in `R` returns an upper-triangular decomposition, i.e. returns $\mathbf{U}$ for $\mathbf{A} = \mathbf{U}^{\mathrm{T}}\mathbf{U}$. To obtain $\mathbf{L}$ we just use `t(chol())`.

Compute the Cholesky decomposition of $\mathbf{\Sigma}$ from Example @ref(exm:mvn1) in upper- and lower-triangular form, and verify that both are Cholesky decompositions of $\mathbf{\Sigma}$.

```
> U <- chol(Sigma) # upper-triangular form
> all.equal(crossprod(U), Sigma)
```

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

```
> L <- t(U) # lower-triangular form
> all.equal(tcrossprod(L), Sigma)
```

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

Above, instead of `L <- t(chol(Sigma))` we've used `L <- t(U)` to avoid repeated calculation of `chol(Sigma)`. In this example, where we compute the Cholesky decomposition of a 3×3 matrix, the calculation is trivial. However, for much larger matrices, calculating the Cholesky decomposition can be expensive, and so we could gain significant time by only calculating it once.

Properties

Once a Cholesky decomposition has been calculated, it can be used to calculate determinants and inverses.

- $\det(\mathbf{A}) = (\prod_{i=1}^{n} l_{ii})^2$.
- $\mathbf{A}^{-1} = \mathbf{L}^{-\mathrm{T}}\mathbf{L}^{-1}$, where $\mathbf{L}^{-\mathrm{T}}$ denotes the inverse of $\mathbf{L}^{\mathrm{T}}$.

For $\mathbf{\Sigma}$ from Example @ref(exm:mvn1), compute $\det(\mathbf{\Sigma})$ and $\mathbf{\Sigma}^{-1}$ based on either Cholesky decomposition computed in Example @ref(exm:cholR1). Verify your results.

We'll start with the determinant

```
> det1 <- det(Sigma)
> det2 <- prod(diag(L))^2
> all.equal(det1, det2)
```

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

and then have various options for the inverse

```
> inv1 <- solve(Sigma)
> inv2 <- crossprod(solve(L), solve(L))
> all.equal(inv1, inv2)
```

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

```
> inv3 <- crossprod(solve(L))
> all.equal(inv1, inv3)
```

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

```
> inv4 <- solve(t(L), solve(L))
> all.equal(inv1, inv4)
```

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

```
> inv5 <- chol2inv(t(L))
> all.equal(inv1, inv5)
```

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

which all give the same answer, although `chol2inv()` should be our default.

Solving systems of linear equations

We can also use a Cholesky decomposition to solve a system of linear equations. Solving $\mathbf{Ax} = \mathbf{b}$ is equivalent to solving $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$ and then $\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. This might seem inefficient at first glance, because we're having to solve

two systems of linear equations. However, $\mathbf{L}$ being triangular means that forward elimination is efficient for $\mathbf{L}$, and backward elimination is for $\mathbf{L}^{\mathrm{T}}$.

Recall the multivariate Normal pdf of Example @ref(exm:mvn1) in which we needed $\boldsymbol{\Sigma}^{-1}(\mathbf{y} - \boldsymbol{\mu})$, with $\mathbf{y}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as given in Example @ref(exm:mvn1). Compute $\boldsymbol{\Sigma}^{-1}(\mathbf{y} - \boldsymbol{\mu})$ by solving $\boldsymbol{\Sigma}\mathbf{z} = \mathbf{y} - \boldsymbol{\mu}$ for $\mathbf{z}$ using using the Cholesky decomposition of $\boldsymbol{\Sigma}$ from Example @ref(exm:chol1). Verify your answer.

We'll first use `solve()` on $\boldsymbol{\Sigma}$, as in Example @ref(exm:mvn2),

```
> res1 <- solve(Sigma, y - mu)
```

and then we'll solve $\mathbf{L}\mathbf{x} = \mathbf{y} - \boldsymbol{\mu}$ for $\mathbf{x}$ followed by $\mathbf{L}^{\mathrm{T}}\mathbf{z} = \mathbf{x}$ for $\mathbf{z}$.

```
> x <- solve(L, y - mu)
> res2 <- solve(t(L), x)
```

which we can confirm gives the same as above, i.e. `res1`, with `all.equal()`

```
> all.equal(res1, res2)
```

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

We can tell `R` to use forward substitution, by calling function `forwardsolve()` instead of `solve()`, or backward substitution, by calling function `backsolve()`. Then `R` knows that one triangle of the supplied matrix comprises zeros, which speeds up solving the system of linear equations. Solving via the Cholesky decomposition is also more stable than without it. Otherwise, if we just used `solve()`, `R` performs a lot of needless calculations on zeros, because it doesn't know that they're zeros.

Repeat Example @ref(exm:chol2) by recognising that the Cholesky decomposition of $\boldsymbol{\Sigma}$ is triangular.

We want to use `forwardsolve()` to solve $\mathbf{L}\mathbf{x} = \mathbf{y} - \boldsymbol{\mu}$ and then `backsolve()` to solve $\mathbf{L}^{\mathrm{T}}\mathbf{z} = \mathbf{x}$. However, `backsolve()` expects an upper-triangular matrix, so we must use $\mathbf{L}^{\mathrm{T}}$, hence `t(L)` below.

```
> x2 <- forwardsolve(L, y - mu)
> res3 <- backsolve(t(L), x2)
> all.equal(res1, res3)
```

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

We can avoid the transpose operation by letting `backsolve()` know the format of Cholesky decomposition that we're supplying. We're supplying a

lower-triangular matrix, hence `upper.tri = FALSE`, which needs transposing to be upper-triangular, hence `transpose = TRUE`.

```
> res4 <- backsolve(L, forwardsolve(L, y - mu), upper.tri = FALSE, transpose = TRUE)
> all.equal(res1, res4)
```

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

If we begin with an upper-triangular matrix, `U = t(L)`, then we'd want to use either of the following

```
> forwardsolve(U, backsolve(U, y - mu, transpose = TRUE), upper.tri = TRUE)
> backsolve(U, forwardsolve(U, y - mu, upper.tri = TRUE, transpose = TRUE))
```

and see that the first gives the required result.

```
> res5 <- forwardsolve(U, backsolve(U, y - mu, transpose = TRUE), upper.tri = TRUE)
> all.equal(res1, res5)
```

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

Given the p-vectors $\mathbf{x}$ and $\mathbf{y}$ and a variance-covariance matrix $\mathbf{\Sigma}$, the **Mahalanobis distance** is defined as

$$D_M(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{y} - \mathbf{x})^{\mathrm{T}} \mathbf{\Sigma}^{-1} (\mathbf{y} - \mathbf{x})}.$$

We can efficiently compute the Mahalanobis distance using the Cholesky decomposition. Consider $\mathbf{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$ so that $\mathbf{\Sigma}^{-1} = \mathbf{L}^{-\mathrm{T}}\mathbf{L}^{-1}$. Then

$$\begin{aligned} \left[D_M(\mathbf{x}, \mathbf{y})\right]^2 &= (\mathbf{y} - \mathbf{x})^{\mathrm{T}} \mathbf{L}^{-\mathrm{T}} \mathbf{L}^{-1} (\mathbf{y} - \mathbf{x}) \\ &= \left[\mathbf{L}^{-1}(\mathbf{y} - \mathbf{x})\right]^{\mathrm{T}} \mathbf{L}^{-1} (\mathbf{y} - \mathbf{x}) \\ &= \mathbf{z}^{\mathrm{T}} \mathbf{z} \end{aligned}$$

where $\mathbf{z}$ is the solution of $\mathbf{L}\mathbf{z} = \mathbf{y} - \mathbf{x}$.

If we have $\mathbf{x}$, $\mathbf{y}$ and the lower-triangular Cholesky decomposition of $\mathbf{\Sigma}$ stored as `x`, `y` and `L`, respectively, then we can efficiently compute the Mahalanobis distance in `R` with

```
> sqrt(crossprod(forwardsolve(L, y - x)))
```

but may want to simplify the use of `crossprod()` and use `sqrt(sum(forwardsolve(L, y - x)^2))` instead.

Create a function `dmvn3()` that evaluates the multivariate Normal pdf, as in examples @ref(exm:mvn1) and @ref(exm:mvn2), based on a Cholesky decomposition of the variance-covariance matrix $\mathbf{\Sigma}$. Verify your function using $\mathbf{y}$, $\boldsymbol{\mu}$, and $\mathbf{\Sigma}$ given in Example @ref(exm:mvn1).

We first recognise that, if matrix $\mathbf{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$, then $\log(\det(\mathbf{\Sigma})) = 2\sum_{i=1}^{n}\log(l_{ii})$, which we'll incorporate in `dmvn3()`

```
> dmvn3 <- function(y, mu, Sigma, log = TRUE) {
+   # Function to evaluate multivariate Normal pdf by solving
+   # a system of linear equations via Cholesky decomposition
+   # y and mu are vectors
+   # Sigma is a square matrix
+   # log is a logical
+   # Returns scalar, on log scale, if log == TRUE.
+   p <- length(y)
+   res <- y - mu
+   L <- t(chol(Sigma))
+   out <- - sum(log(diag(L))) - 0.5 * p * log(2 * pi) -
+            0.5 * sum(forwardsolve(L, res)^2)
+   if (!log)
+     out <- exp(out)
+   out
+ }
```

along with the result to evaluate the Mahalanobis distance from Example @ref(exm:maha). The following confirms the value seen previously.

```
> dmvn3(y, mu, Sigma)
```

```
## [1] -3.654535
```

We can generate $\mathbf{Y} \sim MVN_p(\boldsymbol{\mu}, \mathbf{\Sigma})$, i.e. multivariate Normal random vectors with mean $\boldsymbol{\mu}$ and variance-covariance matrix $\mathbf{\Sigma}$, using the following algorithm.

- Step 1. Find some matrix $\mathbf{L}$ such that $\mathbf{L}\mathbf{L}^{\mathrm{T}} = \mathbf{\Sigma}$.
- Step 2. Generate $\mathbf{Z}^{\mathrm{T}} = (Z_1, \ldots, Z_p)$, where Z_i, $i = 1, \ldots, p$, are independent $N(0, 1)$ random variables.
- Step 3. Set $\mathbf{Y} = \boldsymbol{\mu} + \mathbf{L}\mathbf{Z}$.

In `R`, we can write a function, `rmvn()`, to implement this.

Write a function in R to generate n independent $MVN_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ random vectors and then generate six vectors with $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as in Example @ref(exm:mvn1).

The Cholesky decomposition clearly meets the criterion for $\mathbf{L}$ in Step 1.

Suppose that $n =$ `n`, $\boldsymbol{\mu} =$ `mu` and $\boldsymbol{\Sigma} =$ `Sigma`, then we can use function `rmvn()` below.

```
> rmvn <- function(n, mu, Sigma) {
+ # Function to generate n MVN random vectors
+ # mean vector mu
+ # variance-covariance matrix Sigma
+ # integer n
+ # returns p x n matrix
+ p <- length(mu)
+ L <- t(chol(Sigma))
+ Z <- matrix(rnorm(p * n), nrow = p)
+ as.vector(mu) + L %*% Z
+ }
> rmvn(6, mu, Sigma)
```

```
##           [,1]       [,2]      [,3]      [,4]       [,5]     [,6]
## [1,] 0.8457857 -0.6251981 -2.322655 0.8581701 -0.2873112 1.710739
## [2,] 2.0171233  1.5664917  1.525484 2.6164901  1.5392341 2.640589
## [3,] 4.2669704  2.5623726  2.376017 4.4479018  2.2114458 3.174201
```

Eigen-decomposition

Definition

We can write any symmetric matrix $\mathbf{A}$ in the form

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{\mathrm{T}} (\#eq:eig) \tag{5}$$

where $\mathbf{U}$ is an orthogonal matrix and $\boldsymbol{\Lambda}$ is a diagonal matrix. We will denote the diagonal elements of $\boldsymbol{\Lambda}$ by $\lambda_1 \leq \ldots \leq \lambda_n$. Post-multiplying both sides of the decomposition by $\mathbf{U}$ we have $\mathbf{AU} = \mathbf{U}\boldsymbol{\Lambda}$. Let $\mathbf{u}_i$ denote the ith column of $\mathbf{U}$. Then $\mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_i$. The λ_is are the **eigenvalues** of $\mathbf{A}$, and the columns of $\mathbf{U}$ are the corresponding **eigenvectors**. We call the decomposition in @ref(eq:eig) the **eigen-decomposition** (or sometimes *spectral decomposition*) of $\mathbf{A}$.

Properties

If $\mathbf{A}$ is symmetric, the following properties of eigen-decompositions hold.

- $\mathbf{U}^{-1} = \mathbf{U}^{\mathrm{T}}$
- $\mathbf{A}^{-1} = \mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{U}^{-1}$, and, because $\mathbf{\Lambda}$ is diagonal, so too is $\mathbf{\Lambda}^{-1}$ and its elements are $(\mathbf{\Lambda}^{-1})_{ii} = 1/\lambda_i$.
- $\det(\mathbf{A}) = \prod_{i=1}^{n} \lambda_i$.
- $\mathbf{A}$ is positive definite if all of its eigenvalues are positive.

```
> eigen(A) # computes the eigen-decomposition of a matrix A
```

Use `eigen()` in `R` to give the eigen-decomposition of the 3×3 Hilbert matrix.

We've already calculated the 3×3 Hilbert matrix and stored it as `H`, so we just need

```
> eigen(H)
```

```
## eigen() decomposition
## $values
## [1] 1.40831893 0.12232707 0.00268734
##
## $vectors
##           [,1]       [,2]       [,3]
## [1,] 0.8270449  0.5474484  0.1276593
## [2,] 0.4598639 -0.5282902 -0.7137469
## [3,] 0.3232984 -0.6490067  0.6886715
```

Note that `eigen()` returns a list comprising element `values`, a vector of the eigenvalues in descending order, and `vectors`, a matrix of the eigenvectors, in column order corresponding to the eigenvalues. We can ask `eigen()` to return only the eigenvalues by specifying `only.values = TRUE`, and stipulate that the supplied matrix is symmetric by specifying `symmetric = TRUE` (which avoids checking symmetry, and can save a bit of time for large matrices).

Confirm that the eigenvectors of the eigen-decomposition of the 3×3 Hilbert matrix form an orthogonal matrix.

If $\mathbf{U}$ denotes the matrix of eigenvectors, then we need to show that $\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}$, which the following confirms.

```
> eH <- eigen(H)
> U <- eH$vectors
> crossU <- crossprod(U) # should be 3 x 3 identity matrix
> all.equal(crossU, diag(3))
```

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

Let $\mathbf{A}$ be a positive definite matrix with eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$ and corresponding eigenvectors $\mathbf{u}_1, \mathbf{u}_2, \ldots \mathbf{u}_n$. Then $\mathbf{u}_1$ maximises $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$ *and* $\mathbf{u}_1^{\mathrm{T}}\mathbf{A}\mathbf{u}_1 = \lambda_1$. Furthermore, for $k = 1, \ldots, p < n$, given $\mathbf{u}_1, \ldots, \mathbf{u}_k$, $\mathbf{u}_{k+1}$ maximises $\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$, subject to $\mathbf{x}$ being orthogonal to $\mathbf{u}_1, \ldots, \mathbf{u}_k$ *and* $\mathbf{u}_{k+1}^{\mathrm{T}}\mathbf{A}\mathbf{u}_{k+1} = \lambda_{k+1}$.

For a proof see, e.g. Johnson and Wichern (2007, 80), but note that knowledge of the proof is beyond the scope of MTH3045.

Consider $n \times n$ matrix $\mathbf{A}$ with eigen-decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}$. The second power of $\mathbf{A}$ is $\mathbf{A}^2 = \mathbf{A}\mathbf{A}$. Show that the mth power of is given by $\mathbf{A}^m = \mathbf{U}\mathbf{\Lambda}^m\mathbf{U}^{\mathrm{T}}$.

$$\mathbf{A}^m = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}} \ldots \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}.$$

Because $\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}_n$ this reduces to

$$\mathbf{A}^m = \mathbf{U}\mathbf{\Lambda}\mathbf{\Lambda} \ldots \mathbf{\Lambda}\mathbf{U}^{\mathrm{T}} = \mathbf{U}\mathbf{\Lambda}^m\mathbf{U}^{\mathrm{T}}.$$

Consider a random vector $\mathbf{Y} = (Y_1, \ldots, Y_n)^{\mathrm{T}}$ with variance-covariance matrix $\mathbf{\Sigma}$. Then consider taking linear combinations of $\mathbf{Y}$ so that

$$\begin{array}{rcccl} Z_1 & = & \mathbf{a}_1^{\mathrm{T}}\mathbf{Y} & = & a_{11}Y_1 + a_{12}Y_2 + \ldots + a_{1n}Y_n, \\ Z_2 & = & \mathbf{a}_2^{\mathrm{T}}\mathbf{Y} & = & a_{21}Y_1 + a_{22}Y_2 + \ldots + a_{2n}Y_n, \\ \vdots & = & \vdots & = & \vdots \\ Z_n & = & \mathbf{a}_n^{\mathrm{T}}\mathbf{Y} & = & a_{n1}Y_1 + a_{n2}Y_2 + \ldots + a_{nn}Y_n, \end{array}$$

where $\mathbf{a}_1, \ldots, \mathbf{a}_n$ are coefficient vectors.

Then

$$\begin{array}{rll} \mathrm{Var}(Z_i) & = \mathbf{a}_i^{\mathrm{T}}\mathbf{\Sigma}\mathbf{a}_i & \text{for } i = 1, \ldots, n \\ \mathrm{Cov}(Z_j, Z_k) & = \mathbf{a}_j^{\mathrm{T}}\mathbf{\Sigma}\mathbf{a}_k & \text{for } j, k = 1, \ldots, n. \end{array}$$

The **principal components** are the *uncorrelated* linear combinations of $Y_1, \ldots, Y_n$ that maximise $\mathrm{Var}(\mathbf{a}_i^{\mathrm{T}}\mathbf{Y})$, for $i = 1, \ldots, n$. Hence the first principal

component maximises $\text{Var}(\mathbf{a}_1\mathbf{Y})$ subject to $\mathbf{a}_1^{\text{T}}\mathbf{a}_1 = 1$, the second maximises $\text{Var}(\mathbf{a}_2\mathbf{Y})$ subject to $\mathbf{a}_2^{\text{T}}\mathbf{a}_2 = 1$ *and* $\text{Cov}(\mathbf{a}_1^{\text{T}}\mathbf{Y}, \mathbf{a}_2^{\text{T}}\mathbf{Y}) = 0$, and so forth. More generally, the ith principal component, $i > 1$, maximises $\text{Var}(\mathbf{a}_i^{\text{T}}\mathbf{Y})$ subject $\mathbf{a}_i^{\text{T}}\mathbf{a}_i = 1$ *and* $\text{Cov}(\mathbf{a}_i^{\text{T}}\mathbf{Y}, \mathbf{a}_j^{\text{T}}\mathbf{Y}) = 0$ for $j < i$.

Now suppose that we form the eigen-decomposition $\mathbf{\Sigma} = \mathbf{U}^{\text{T}}\mathbf{\Lambda}\mathbf{U}$. The eigenvectors of $\mathbf{\Sigma}$ therefore meet the criteria described above for principal components, and hence give one definition for principal components.

Fisher's iris data are distributed with `R` as object `iris`. From the dataset's help file the data comprise "the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are `Iris setosa`, `versicolor`, and `virginica`." Compute the principal components for the variables `Sepal.Length`, `Sepal.Width`, `Petal.Length` and `Petal.Width`.

To compute the principals components, we'll load the `iris` data, which we'll quickly look at

```
> data(iris)
> head(iris)
```

and then extract the relevant variables, and then form their empirical correlation matrix.

```
> vbls <- c('Sepal.Length', 'Sepal.Width', 'Petal.Length', 'Petal.Width')
> species <- as.factor(iris$Species)
> Y <- as.matrix(iris[, vbls]) # data corresponding to variables under study
> corY <- cor(Y) # correlation matrix of variables under study
```

We can then use the eigen-decomposition of the correlation matrix to form the eigenvectors, which are also the coefficient vectors, and which we'll store as `A`. Note that we use the correlation matrix because we're measuring different features of the plant, which we don't necessarily expect to be directly comparable.

```
> A <- eigen(corY)$vectors
> A
```

```
##             [,1]        [,2]       [,3]       [,4]
## [1,]  0.5210659 -0.37741762  0.7195664  0.2612863
## [2,] -0.2693474 -0.92329566 -0.2443818 -0.1235096
## [3,]  0.5804131 -0.02449161 -0.1421264 -0.8014492
```

```
## [4,]  0.5648565 -0.06694199 -0.6342727  0.5235971
```

The principal components are then obtained by calculating $\mathbf{a}_j^{\mathrm{T}}\mathbf{y}$ for $j = 1, \ldots, 4$. We'll store these as `pcs` and then use `head()` to show the first few.

```
> pcs <- Y %*% A
> colnames(pcs) <- paste('PC', 1:4, sep = '')
> head(pcs)
```

```
##           PC1       PC2      PC3        PC4
## [1,] 2.640270 -5.204041 2.488621 -0.1170332
## [2,] 2.670730 -4.666910 2.466898 -0.1075356
## [3,] 2.454606 -4.773636 2.288321 -0.1043499
## [4,] 2.545517 -4.648463 2.212378 -0.2784174
## [5,] 2.561228 -5.258629 2.392226 -0.1555127
## [6,] 2.975946 -5.707321 2.437245 -0.2237665
```

Principal component analysis is a commonly used statistical method, often as a method of *dimension reduction*, when a finite number of principal components, below the dimension of the data, are used to capture most of what's in the original data.

In practice, if we want to perform PCA then we'd usually use one of `R`'s built in functions, such as `prcomp()` or `princomp()`. For example

```
> prcomp(Y, center = TRUE, scale = TRUE)
```

```
## Standard deviations (1, .., p=4):
## [1] 1.7083611 0.9560494 0.3830886 0.1439265
##
## Rotation (n x k) = (4 x 4):
##                     PC1         PC2        PC3        PC4
## Sepal.Length  0.5210659 -0.37741762  0.7195664  0.2612863
## Sepal.Width  -0.2693474 -0.92329566 -0.2443818 -0.1235096
## Petal.Length  0.5804131 -0.02449161 -0.1421264 -0.8014492
## Petal.Width   0.5648565 -0.06694199 -0.6342727  0.5235971
```

gives the same principal component coefficients as we obtained in Example @ref(exm:pca). A benefit of working with `prcomp()` or `princomp()` is that `R` interprets the objects as relating to PCA, and then `summary()` and `plot()`, for example, perform useful actions. Principal component regression is a statistical model in which we perform regression on principal components.

Singular value decomposition

For an $m \times n$ matrix $\mathbf{A}$ with real elements and $m \geq n$, there exist orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ such that

$$\mathbf{U}^{\mathrm{T}}\mathbf{A}\mathbf{V} = \mathbf{D},$$

where $\mathbf{D}$ is a diagonal matrix with elements $d_1 \geq d_2 \geq \ldots \geq d_m$.

The **singular value decomposition** (SVD) of $\mathbf{A}$ is

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}.$$

The diagonal entries of $m \times n$ matrix $\mathbf{D}$ are the singular values of $\mathbf{A}$. We can form a $m \times m$ matrix $\mathbf{U}$ from the eigenvectors of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ and a $n \times n$ matrix $\mathbf{V}$ from the eigenvectors of $\mathbf{A}\mathbf{A}^{\mathrm{T}}$. The singular values are the square roots of the positive eigenvalues of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$.

```
> svd(A) # calculates the SVD of a matrix A
```

Compute a SVD of the 3×3 Hilbert matrix, $\mathbf{H}_3$, in `R` using `svd()`.

We have $\mathbf{H}_3$ stored as `H` already, so we'll now calculate its SVD.

```
> H.svd <- svd(H)
> H.svd
```

```
## $d
## [1] 1.40831893 0.12232707 0.00268734
##
## $u
##             [,1]       [,2]       [,3]
## [1,] -0.8270449  0.5474484  0.1276593
## [2,] -0.4598639 -0.5282902 -0.7137469
## [3,] -0.3232984 -0.6490067  0.6886715
##
## $v
##             [,1]       [,2]       [,3]
## [1,] -0.8270449  0.5474484  0.1276593
## [2,] -0.4598639 -0.5282902 -0.7137469
## [3,] -0.3232984 -0.6490067  0.6886715
```

From `svd()` we get a three-element list where `d` is a vector of the diagonal elements of $\mathbf{D}$, `u` is $\mathbf{U}$ and `v` is $\mathbf{V}$.

We can quickly confirm that $\mathbf{H}_3 = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}$.

```
> all.equal(H, H.svd$u %*% tcrossprod(diag(H.svd$d), H.svd$v))
```

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

One application of the SVD is solving systems of linear equations. Let $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}$ be the SVD of $\mathbf{A}$. Consider again solving $\mathbf{A}\mathbf{x} = \mathbf{b}$. Then

$$\begin{aligned}
\mathbf{U}^{\mathrm{T}}\mathbf{A}\mathbf{x} &= \mathbf{U}^{\mathrm{T}}\mathbf{b} && \text{(premultiplying by } \mathbf{U}^{\mathrm{T}}\text{)} \\
\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}\mathbf{x} &= \mathbf{U}^{\mathrm{T}}\mathbf{b} && \text{(substituting } \mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}\text{)} \\
\mathbf{D}\mathbf{V}^{\mathrm{T}}\mathbf{x} &= \mathbf{U}^{\mathrm{T}}\mathbf{b} && \text{(as } \mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}_n\text{)} \\
\mathbf{D}\tilde{\mathbf{x}} &= \tilde{\mathbf{b}}. && \text{(setting } \tilde{\mathbf{x}} = \mathbf{V}^{\mathrm{T}}\mathbf{x} \text{ and } \tilde{\mathbf{b}} = \mathbf{U}^{\mathrm{T}}\mathbf{b}\text{)}
\end{aligned}$$

As $\mathbf{D}$ is diagonal, we see that setting $\tilde{\mathbf{x}} = \mathbf{V}^{\mathrm{T}}\mathbf{x}$ and $\tilde{\mathbf{b}} = \mathbf{U}^{\mathrm{T}}\mathbf{b}$ results in a diagonal solve, i.e. essentially n divisions.

Solve $\mathbf{\Sigma}\mathbf{z} = \mathbf{y} - \boldsymbol{\mu}$ in `R` using a SVD with $\mathbf{y}$, $\boldsymbol{\mu}$ and $\mathbf{\Sigma}$ as in Example @ref(exm:mvn1).

Following the above remark, we want to calculate $\tilde{\mathbf{x}} = \mathbf{V}^{\mathrm{T}}\mathbf{x}$ and $\tilde{\mathbf{b}} = \mathbf{V}^{\mathrm{T}}(\mathbf{y} - \boldsymbol{\mu})$. We'll start by computing the SVD of $\mathbf{\Sigma}$, and then extract diag($\mathbf{D}$) and $\mathbf{V}$, which we'll call `S.d` and `S.V`, respectively.

```
> S.svd <- svd(Sigma)
> S.d <- S.svd$d
> S.V <- S.svd$v
```

Then we obtain $\tilde{\mathbf{b}} = \mathbf{V}^{\mathrm{T}}(\mathbf{y} - \boldsymbol{\mu})$, which we'll call `b2`.

```
> b2  <- crossprod(S.V, y - mu)
```

Then $\tilde{\mathbf{x}}$ is the solution of $\mathbf{D}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$, which we'll call `x2`, and can be computed with

```
> x2 <- b2 / S.svd$d
```

since $\mathbf{D}$ is diagonal. Finally, $\mathbf{z}$ is the solution of $\mathbf{V}^{T}\mathbf{z} = \tilde{\mathbf{x}}$, which we'll call `res6` and can obtain with

```
> z <- S.V %*% x2
```

since $\mathbf{V}^{-\mathrm{T}} = \mathbf{V}$. Vectorising this and renaming it to `res6`, we confirm that we get the same result as in Example @ref(exm:chol2).

```
> res6 <- as.vector(z)
> all.equal(res1, res6)
```

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

So far we have considered systems of linear equations where $\mathbf{A}$ is non-singular, which means that $\mathbf{A}^{-1}$ is unique. Now we'll consider the case where $\mathbf{A}$ is singular, although this won't be examined in MTH3045.

A **generalized inverse** matrix of the matrix $\mathbf{A}$ is any matrix $\mathbf{A}^{-}$ such that

$$\mathbf{A}\mathbf{A}^{-}\mathbf{A} = \mathbf{A}.$$

Note that $\mathbf{A}^{-}$ is not unique.

The **Moore-Penrose pseudo-inverse**[6] of a matrix $\mathbf{A}$ is a generalized inverse that is unique by virtue of stipulating that it must satisfy the following four properties.

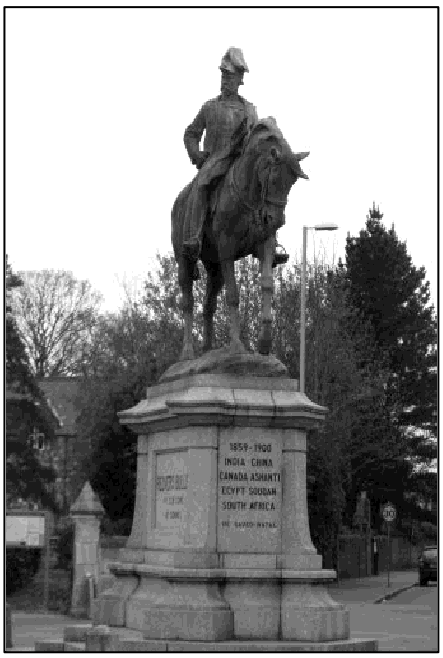
1. $\mathbf{A}\mathbf{A}^{-}\mathbf{A} = \mathbf{A}$.
2. $\mathbf{A}^{-}\mathbf{A}\mathbf{A}^{-} = \mathbf{A}^{-}$.
3. $(\mathbf{A}\mathbf{A}^{-})^{\mathrm{T}} = \mathbf{A}\mathbf{A}^{-}$.
4. $(\mathbf{A}^{-}\mathbf{A})^{\mathrm{T}} = \mathbf{A}^{-}\mathbf{A}$.

We can construct a Moore-Penrose pseudo-inverse via the SVD. Consider $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}$, the SVD of $\mathbf{A}$. Let $\mathbf{D}^{-}$ denote the generalised inverse of $\mathbf{D}$, which is simply obtained by taking reciprocals of the positive diagonal values, with zeros left as zeros. Then we have

$$\mathbf{A}^{-} = \mathbf{U}\mathbf{D}^{-}\mathbf{V}^{\mathrm{T}}.$$

Consider the following $n \times m = 338 \times 450$ pixel greyscale image

[6] The Moore-Penrose pseudoinverse is named after E. H. Moore and Sir Roger Penrose. Moore first worked in abstract algebra, proving in 1893 the classification of the structure of finite fields (also called Galois fields). He then worked on various topics, including the foundations of geometry and algebraic geometry, number theory, and integral equations. Penrose has made contributions to the mathematical physics of general relativity and cosmology. He has received several prizes and awards, including the 1988 Wolf Prize in Physics, which he shared with Stephen Hawking for the Penrose–Hawking singularity theorems, and one half of the 2020 Nobel Prize in Physics "for the discovery that black hole formation is a robust prediction of the general theory of relativity".

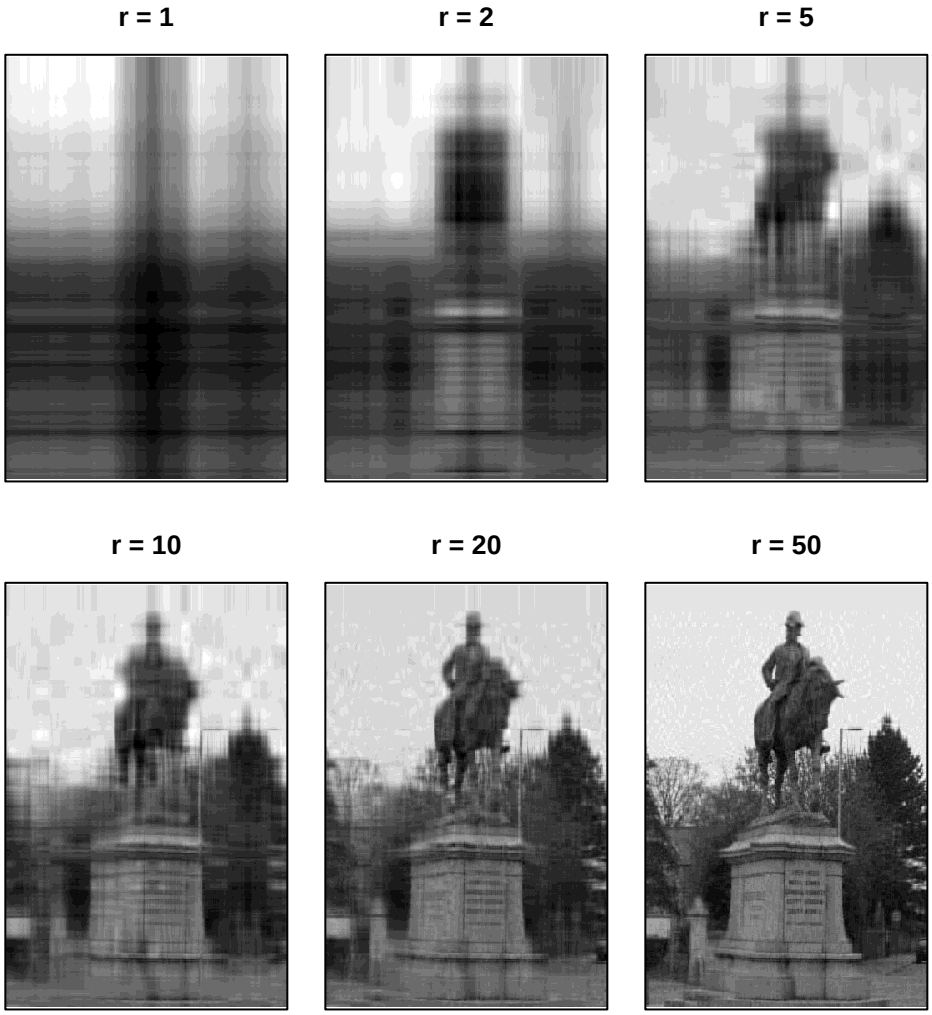


which can be represented as a matrix, $\mathbf{A}$, which we'll store as `A`, comprising values on $[0, 1]$, where 0 is white and 1 is black; the bottom left 7×7 pixels take the following values.

```
> A[1:7, 1:7]
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.3516340 0.4718954 0.4483660 0.2718954 0.2522876 0.2758170 0.2758170
## [2,] 0.4261438 0.5228758 0.4718954 0.2915033 0.2562092 0.2718954 0.2679739
## [3,] 0.4915033 0.5464052 0.4875817 0.3032680 0.2444444 0.2601307 0.2562092
## [4,] 0.5346405 0.5503268 0.4915033 0.3150327 0.2326797 0.2601307 0.2758170
## [5,] 0.5738562 0.5699346 0.5111111 0.3647059 0.2640523 0.3032680 0.3503268
## [6,] 0.5895425 0.5738562 0.5490196 0.4470588 0.3503268 0.3856209 0.4405229
## [7,] 0.5895425 0.5816993 0.5647059 0.5163399 0.4339869 0.4535948 0.4836601
```

Suppose that we compute the SVD $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}$. A finite-rank representation of $\mathbf{A}$ is given by $\mathbf{A}_r = \mathbf{U}_r\mathbf{D}_r\mathbf{V}_r^{\mathrm{T}}$, where $\mathbf{U}_r$ is the $n \times r$ matrix comprising the first r columns of $\mathbf{U}$, $\mathbf{D}_r$ is the $r \times r$ matrix comprising the first r rows and columns of $\mathbf{D}$, and $\mathbf{V}_r$ is the $m \times r$ matrix comprising the first r columns of $\mathbf{V}$. The following shows the resulting greyscale images obtained by plotting $\mathbf{A}_r$ for $r = 1,\ 2,\ 5,\ 10,\ 20$ and 50.



You might wonder how SVD has compressed our image. The image itself takes

```
> format(object.size(A), units = 'Kb')
```

```
## [1] "1188.5 Kb"
```

bytes (and there are eight bits in a byte). However, if we consider the $r = 20$ case of Example @ref(exm:process), then we need to store the r diagonal elements of $\mathbf{D}_r$ and the first r columns of $\mathbf{U}_r$ and $\mathbf{V}_r$, which we could store in a list

```
> r <- 20
> ind <- 1:r
> A_r <- list(diag(D[ind, ind]), U[, ind, drop = FALSE], V[, ind, drop = FALSE])
> format(object.size(A_r), units = 'Kb')
```

```
## [1] "123.8 Kb"
```

and takes about 10% of the memory of the original image. Of course, there is computational cost of computing the decomposition, i.e. compressing the image, and then later decompressing the image, which should be taken into account when considering image compression.

QR decomposition

The **QR decomposition** is often used in the background of functions in `R`.

Any real square matrix $\mathbf{A}$ may be decomposed as

$$\mathbf{A} = \mathbf{QR},$$

where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix. This is its **QR decomposition**.

In the above, if $\mathbf{A}$ is non-singular, then the QR decomposition is unique.

```
> qr(A) # computes the QR decomposition of a matrix A
```

When `R` computes the QR decomposition, $\mathbf{R}$ is simply the upper triangle of `qr()$qr`. However, $\mathbf{Q}$ is rather more complicated to obtain, but fortunately `qr.Q()` does all the calculations for us.

Properties

- $|\det(\mathbf{A})| = |\det(\mathbf{Q})|\det(\mathbf{R}) = \det(\mathbf{R})$ since $\det(\mathbf{Q}) = \pm 1$ as $\mathbf{Q}$ is orthogonal.
- $\det(\mathbf{R}) = \prod_{i=1}^{n} |r_{ii}|$, since $\mathbf{R}$ is triangular.
- $\mathbf{A}^{-1} = (\mathbf{QR})^{-1} = \mathbf{R}^{-1}\mathbf{Q}^{-1} = \mathbf{R}^{-1}\mathbf{Q}^{\mathrm{T}}$.

Use `qr()` in `R` to compute the QR decomposition of $\mathbf{H}_3$, the 3×3 Hilbert matrix. Then use `qr.Q()` and `qr.R()` to extract $\mathbf{Q}$ and $\mathbf{R}$ from the output of `qr()` to confirm that $\mathbf{QR} = \mathbf{H}_3$.

We'll start with the QR decomposition of $\mathbf{H}_3$, which we'll store as `qr.H`.

```
> qr.H <- qr(H)
> qr.H
```

```
## $qr
##            [,1]       [,2]         [,3]
## [1,] -1.1666667 -0.6428571 -0.450000000
## [2,]  0.4285714 -0.1017143 -0.105337032
## [3,]  0.2857143  0.7292564  0.003901372
##
## $rank
## [1] 3
##
## $qraux
## [1] 1.857142857 1.684240553 0.003901372
##
## $pivot
## [1] 1 2 3
##
## attr(,"class")
## [1] "qr"
```

Then `qr.Q()` and `qr.R()` will give $\mathbf{Q}$ and $\mathbf{R}$, which we'll store as `Q.H` and `R.H`.

```
> Q.H <- qr.Q(qr.H)
> Q.H
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.8571429  0.5016049  0.1170411
## [2,] -0.4285714 -0.5684856 -0.7022469
## [3,] -0.2857143 -0.6520864  0.7022469
```

```
> R.H <- qr.R(qr.H)
> R.H
```

```
##           [,1]       [,2]         [,3]
## [1,] -1.166667 -0.6428571 -0.450000000
## [2,]  0.000000 -0.1017143 -0.105337032
## [3,]  0.000000  0.0000000  0.003901372
```

And then finally we'll compute $\mathbf{QR}$

```
> Q.H %*% R.H
```

```
##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.5000000 0.3333333
## [2,] 0.5000000 0.3333333 0.2500000
## [3,] 0.3333333 0.2500000 0.2000000
```

```
> all.equal(Q.H %*% R.H, H)
```

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

which does indeed give $\mathbf{H}_3$.

Solve $\mathbf{\Sigma z} = \mathbf{y} - \boldsymbol{\mu}$ in R using the QR decomposition and with $\mathbf{y}$, $\boldsymbol{\mu}$ and $\mathbf{\Sigma}$ as in Example @ref(exm:mvn1).

Suppose that $\mathbf{\Sigma}$ has QR decomposition $\mathbf{\Sigma} = \mathbf{QR}$. The following calculates the QR decomposition of $\mathbf{\Sigma}$ as `S.qr`, and then $\mathbf{Q}$ and $\mathbf{R}$ as `S.Q` and `S.R`, respectively.

```
> S.qr <- qr(Sigma)
> S.Q <- qr.Q(S.qr)
> S.R <- qr.R(S.qr)
```

From Example @ref(exm:mvn1) we have that $\mathbf{z}$ is `res1`, i.e.

```
> res1
```

```
## [1]  0.08 -0.38  0.14
```

Solving $\mathbf{\Sigma z} = \mathbf{y} - \boldsymbol{\mu}$ is then equivalent to solving $\mathbf{QRz} = \mathbf{y} - \boldsymbol{\mu}$. So we solve $\mathbf{Qx} = \mathbf{y} - \boldsymbol{\mu}$ for $\mathbf{x}$ with the following

```
> x <- crossprod(S.Q, y - mu)
```

as $\mathbf{Q}^{-1} = \mathbf{Q}^{\mathrm{T}}$ because $\mathbf{Q}$ is orthogonal, and then solve $\mathbf{Rz} = \mathbf{x}$ for $\mathbf{z}$,

```
> z1 <- solve(S.R, x)
```

or, because $\mathbf{R}$ is upper triangular,

```
> z2 <- backsolve(S.R, x)
```

which are both the same

```
> all.equal(z1, z2)
```

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

We'll take the second, `z2`, and convert it from a one-column matrix to a vector called `res7`

```
> res7 <- as.vector(z2)
> res7
```

```
## [1]  0.08 -0.38  0.14
```

and see that this gives the same as before

```
> all.equal(res1, res7)
```

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

If we have obtained a QR decomposition in `R` using `qr()`, then we can use `qr.solve(A, b)` to solve $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$. This avoids having to find $\mathbf{Q}$ and $\mathbf{R}$ with `qr.Q()` and `qr.R()`, and requires only one function call in `R`.

Theorem @ref(def:qrsq) extends to a $m \times n$ matrix $\mathbf{A}$, with $m \geq n$, so that

$$\mathbf{A} = \mathbf{QR} = \mathbf{Q}\begin{bmatrix}\mathbf{R}_1 \\ \mathbf{0}\end{bmatrix} = \begin{bmatrix}\mathbf{Q}_1 & \mathbf{Q}_2\end{bmatrix}\begin{bmatrix}\mathbf{R}_1 \\ \mathbf{0}\end{bmatrix} = \mathbf{Q}_1\mathbf{R}_1,$$

$\mathbf{Q}$ is an $m \times m$ unitary matrix, and $\mathbf{R}$ and an $m \times n$ upper triangular matrix; then $\mathbf{Q}_1$ is $m \times n$, $\mathbf{Q}_2$ is $m \times (m-n)$, and $\mathbf{Q}_1$ and $\mathbf{Q}_2$ both have orthogonal columns, and $\mathbf{R}_1$ is an $n \times n$ upper triangular matrix, followed by $(m-n)$ rows of zeros.

Consider a linear model with response vector $\mathbf{y}$ and design matrix $\mathbf{X}$, where $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \mathbf{e}$ and $\boldsymbol{\beta}$ is the vector of regression coefficients and $\mathbf{e}$ is the observed residual vector. The least squares estimate of $\boldsymbol{\beta}$, denoted $\hat{\boldsymbol{\beta}}$, satisfies $\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^{\mathrm{T}}\mathbf{y}$. This is achieved in `R` by forming the QR decomposition of $\mathbf{X}$, i.e. $\mathbf{X} = \mathbf{QR}$. Then $\hat{\boldsymbol{\beta}}$, satisfies $(\mathbf{QR})^{\mathrm{T}}\mathbf{QR}\hat{\boldsymbol{\beta}} = (\mathbf{QR})^{\mathrm{T}}\mathbf{y}$, which can be re-written as $\mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{QR}\hat{\boldsymbol{\beta}} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{y}$, and, given $\mathbf{Q}$ is orthogonal, simplifies to $\mathbf{R}^{\mathrm{T}}\mathbf{R}\hat{\boldsymbol{\beta}} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{y}$.

Recall the cement factory data from MTH2006 and the linear model $Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i$ where Y_i denotes the output of the cement factory in month i for x_{i1} days at a temperature of x_{i2} degrees Fahrenheit, and where $\varepsilon_i \sim \mathrm{N}(0, \sigma^2)$ and are i.i.d. Use the QR decomposition to find $\hat{\boldsymbol{\beta}}$ and confirm your answer against that given by `lm()`.

We'll first input the data

```
> # operating temperatures
> temp <- c(35.3, 29.7, 30.8, 58.8, 61.4, 71.3, 74.4, 76.7, 70.7, 57.5,
+ 46.4, 28.9, 28.1, 39.1, 46.8, 48.5, 59.3, 70, 70, 74.5, 72.1,
+ 58.1, 44.6, 33.4, 28.6)
> # number of operational days
> days <- c(20, 20, 23, 20, 21, 22, 11, 23, 21, 20, 20, 21, 21, 19, 23,
+ 20, 22, 22, 11, 23, 20, 21, 20, 20, 22)
> # output from factory
> output <- c(10.98, 11.13, 12.51, 8.4, 9.27, 8.73, 6.36, 8.5, 7.82, 9.14,
+ 8.24, 12.19, 11.88, 9.57, 10.94, 9.58, 10.09, 8.11, 6.83, 8.88,
+ 7.68, 8.47, 8.86, 10.36, 11.08)
```

and put them into a `data.frame` called `prod` (as in MTH2006).

```
> prod <- data.frame(temp = temp, days = days, output = output)
```

Then we'll fit the linear model with `lm()`, extract the regression coefficients, and store them as `b0`.

```
> m0 <- lm(output ~ days + temp, data = prod)
> b0 <- coef(m0)
```

Next we'll store the response data as `y` and the design matrix as `X`.

```
> y <- output
> X <- cbind(1, days, temp)
```

The QR decomposition of `X` can be obtained with `qr()`, which is stored as `X.qr`, and then $\mathbf{Q}$ and $\mathbf{R}$ of the QR decomposition $\mathbf{X} = \mathbf{QR}$ stored as `Q.qr` and `R.qr`, respectively.

```
> X.qr <- qr(X)
> Q.qr <- qr.Q(X.qr)
> R.qr <- qr.R(X.qr)
```

Next we'll compute $\mathbf{w} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{y}$, say, then solve $\mathbf{R}^{\mathrm{T}}\mathbf{z} = \mathbf{w}$ for $\mathbf{z}$ and then solve $\mathbf{R}\hat{\boldsymbol{\beta}} = \mathbf{z}$ for $\hat{\boldsymbol{\beta}}$.

```
> w <- crossprod(R.qr, crossprod(Q.qr, y))
> z <- forwardsolve(R.qr, w, transpose = TRUE, upper.tri = TRUE)
> b1 <- drop(backsolve(R.qr, z))
```

Finally we'll check that our regression coefficients calculated through the

QR decomposition are the same as those extracted from our call to `lm()`

```
> all.equal(b0, b1, check.attributes = FALSE)
```

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

which they are.

Note two things above. 1: We've used `drop()` because `backsolve()` will by default return a $(p+1) \times 1$ matrix whereas `coef()` returns an $(p+1)$-vector. The two aren't considered identical by `all.equal()`, even if their values are the same. Calling `drop()` will simplify a matrix to a vector *if* possible, and hence we're comparing like with like. 2: Setting `all.equal(..., check.attributes = FALSE)` should check only the supplied objects and not any attributes, such as names. Without this, because `b0` has names and `b1` doesn't, `all.equal()` wouldn't consider them identical.

Computational costs of matrix decompostions

Each of the four matrix decompositions can be used for various purposes, such as calculating the determinant of a matrix, or solving a system of linear equations. Therefore, we might ask ourselves which one we should use. We'll start by comparing the computational cost of each, i.e. the number of flops each takes. The Cholesky algorithm above requires $n^3/3$ flops (and n square roots). Hence its dominant cost is its $n^3/3$ flops. The dominant cost for the QR decomposition, if computed with householder reflections, is $2n^3/3$, and is therefore roughly twice as expensive as the Cholesky decomposition. The eigen-decomposition and SVD both require kn^3 flops, for some value k, but $k \gg 1/3$, as in Cholesky algorithm, making them considerably slower for large n.

Sherman-Morrison formula / Woodbury matrix identity

In general, unless we actually need the inverse of a matrix (such as for standard errors of regression coefficients in a linear model), we should solve systems of linear equations. Sometimes, though, we do need – or might just have – an inverse, and want to calculate something related to it. The following are a set of formulae, which go by various names, that can be useful in this situation.

Woodbury's formula

Consider an $m \times m$ matrix $\mathbf{A}$, with m large, and for which we have the inverse, $\mathbf{A}^{-1}$. Suppose $\mathbf{A}$ receives a small update of the form $\mathbf{A} + \mathbf{U}\mathbf{V}^{\mathrm{T}}$, for $m \times n$ matrices $\mathbf{U}$ and $\mathbf{V}$ where $n \ll m$. Then, by **Woodbury's formula**,

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^{\mathrm{T}})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I}_n + \mathbf{V}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{A}^{-1}.$$

What's important to note here is that we're looking to calculate an $m \times m$ inverse with m large, and so in general this will be an $O(m^3)$ calculation based on the LHS. However, in the above, the RHS only requires that we to invert an $n \times n$ matrix, at cost $O(n^3)$, if we already have $\mathbf{A}^{-1}$, which is much less that of the LHS.

Sherman-Morrison-Woodbury formula

Woodbury's formula above generalises to the so-called **Sherman-Morrison-Woodbury formula** by introducing the $n \times n$ matrix $\mathbf{C}$, so that

$$(\mathbf{A} + \mathbf{U}\mathbf{C}\mathbf{V}^{\mathrm{T}})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{C}^{-1} + \mathbf{V}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{A}^{-1}.$$

Sherman-Morrison formula

The **Sherman-Morrison formula** is the special case of Woodbury's formula (and hence the Woodbury-Sherman-Morrison formula) in which the update to $\mathbf{A}$ can be considered in terms of m-vectors $\mathbf{u}$ and $\mathbf{v}$, so that

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^{\mathrm{T}})^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}}{1 + \mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{u}}.$$

The Sherman-Morrison formula is particularly useful because it requires no matrix inversion.

Recall from MTH2006 the normal linear model where

$$Y_i \sim N(\mathbf{x}_i^{\mathrm{T}}\boldsymbol{\beta}, \sigma^2)$$

with independent errors $\varepsilon_i = Y_i - \mathbf{x}_i^{\mathrm{T}}\boldsymbol{\beta}$s, for $i = 1, \ldots, n$, where $\mathbf{x}_i^{\mathrm{T}} = (1, x_{i1}, \ldots, x_{ip})$ is the ith row of design matrix $\mathbf{X}$ and where $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)^{\mathrm{T}}$. Hence $\mathbf{Y} \mid \mathbf{X}\boldsymbol{\beta} \sim MVN_n(\mathbf{X}\boldsymbol{\beta}, \sigma^2\mathbf{I}_n)$. In Bayesian linear regression, the elements of $\boldsymbol{\beta}$ and σ^2 are not fixed, unknown

parameters: they are random variables, and we must declare *a priori* our beliefs about their distribution. The conjugate prior is that

$$\boldsymbol{\beta} \sim MVN_{p+1}(\boldsymbol{\mu}_\beta, \boldsymbol{\Sigma}_\beta^{-1}).$$

Integrating out $\boldsymbol{\beta}$ gives $\mathbf{Y} \sim MVN_n(\mathbf{X}\boldsymbol{\mu}_\beta, \sigma^2\mathbf{I}_n + \mathbf{X}\boldsymbol{\Sigma}_\beta^{-1}\mathbf{X}^{\mathrm{T}})$. Now suppose that we want to evaluate the marginal likelihood for an observation, $\mathbf{y}$, say. Recall @ref(eq:dmvn). The Mahalanobis distance then involves the term

$$(\mathbf{y} - \mathbf{X}\boldsymbol{\mu}_\beta)^{\mathrm{T}}(\sigma^2\mathbf{I}_n + \mathbf{X}\boldsymbol{\Sigma}_\beta^{-1}\mathbf{X}^{\mathrm{T}})^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\mu}_\beta).$$

The covariance of the marginal distribution, $\sigma^2\mathbf{I}_n + \mathbf{X}\boldsymbol{\Sigma}_\beta^{-1}\mathbf{X}^{\mathrm{T}}$, is typically dense, expensive to form, and leads to expensive solutions to systems of linear equations. Its inverse, however, can be computed through the Sherman-Morrison formula with $\mathbf{A}^{-1} = \sigma^{-2}\mathbf{I}_n$, $\mathbf{U} = \mathbf{V} = \mathbf{X}$, and $\mathbf{C} = \boldsymbol{\Sigma}_\beta^{-1}$.

Bibliographic notes

For more details on matrix decompositions, consider Wood (2015, Appendix B) for a concise overview. For fuller details consider Monahan (2011, chaps. 3, 4 and 6) or Press et al. (2007, chap. 2 and 11).

Numerical Calculus

Motivation

In statistics, we often rely on the Normal distribution with pdf

$$\phi(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

and cdf

$$\Phi(x; \mu, \sigma^2) = \int_{-\infty}^{x} \phi(x; \mu, \sigma^2) \mathrm{d}x.$$

Unfortunately no closed form exists for $\Phi(x; \mu, \sigma^2)$. However, if x, μ and σ are stored in `R` as `x`, `mu` and `sigma`, we can still evaluate $\Phi(x; \mu, \sigma^2)$ with `pnorm(x, mu, sigma)`. This is one example of a frequently-occurring situation in which we somehow want to evaluate an intractable integral. This raises the question: are there generic methods that let us evaluate intractable integrals? The answer is often *numerical integration*.

In this chapter, we'll consider generic methods for integration, i.e. that work in many scenarios. Sometimes, such as evaluating $\Phi(x; \mu, \sigma^2)$, specific algorithms will give more accurate results. This is what `R` does for `pnorm()`.

Numerical Differentiation

In Chapter 5 we will cover optimisation of functions, such as numerically finding maximum likelihood estimates when analytical solutions aren't available. We'll see that supplying derivatives can considerably improve estimation, typically in terms of reducing computation time. Here we'll cover some useful results in terms of differentiation of matrices, which will later prove useful. No knowledge of analytical matrix calculus beyond these results will be

needed for MTH3045. The matrix cookbook (Petersen and Pedersen 2012), however, can provide you with a much more thorough set of differentiation rules, should you ever need them.

Differentiation definitions

Consider $f : \mathbb{R}^n \to \mathbb{R}$, which we'll consider a function of vector $\mathbf{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$. The **gradient operator**, ∇, is defined as

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{pmatrix}.$$

Note that in MTH3045 we will have no cause to consider multivariate functions, i.e. to consider $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, for $m > 1$.

Consider again $f : \mathbb{R}^n \to \mathbb{R}$. The **Hessian matrix** is the matrix of second derivatives of f, whereas the gradient operator considered first derivatives, and is given by

$$\nabla^2 f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}.$$

The Hessian matrix plays an important role in statistics, in particular for estimating parameter uncertainty via the Fisher information, which is covered in MTH3028. In the next chapter, we'll also see that it's important for optimisation.

If $\tilde{\mathbf{x}}$ is at a minimum of $f(\mathbf{x})$ then the gradient vector w.r.t. $\mathbf{x}$ should be all zero, i.e. $\nabla f(\mathbf{x}) = \mathbf{0}$ and, additionally, the Hessian matrix, i.e. $\nabla^2 f(\mathbf{x})$ should be positive definite.

Differentiation rules

Now consider $g : \mathbb{R}^n \to \mathbb{R}$ a function of $\mathbf{x}$ that, for fixed $\mathbf{A}$, takes the quadratic form $g(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}$ for $n \times n$ matrix $\mathbf{A}$ and n-vector $\mathbf{x}$. Then

$$\nabla g(\mathbf{x}) = (\mathbf{A} + \mathbf{A}^{\mathrm{T}})\mathbf{x} \quad \text{and} \quad \nabla^2 g(\mathbf{x}) = \mathbf{A} + \mathbf{A}^{\mathrm{T}}.$$

Note that in the case of symmetric $\mathbf{A}$, $\nabla g(\mathbf{x}) = 2\mathbf{A}\mathbf{x}$ and $\nabla^2 g(\mathbf{x}) = 2\mathbf{A}$.

Next consider $h : \mathbb{R}^n \to \mathbb{R}$ a function of n-vector $\mathbf{x}$ and p-vector y that, for fixed $n \times n$ matrix $\mathbf{A}$ and $n \times p$ matrix $\mathbf{B}$, takes the quadratic form $h(\mathbf{x}, \mathbf{y}) = (\mathbf{x} + \mathbf{B}\mathbf{y})^{\mathrm{T}}\mathbf{A}(\mathbf{x} + \mathbf{B}\mathbf{y})$. Then

$$\frac{\partial h(\mathbf{x}, \mathbf{y})}{\partial \mathbf{x}} = (\mathbf{A} + \mathbf{A}^{\mathrm{T}})(\mathbf{x} + \mathbf{B}\mathbf{y}) \quad \text{and} \quad \frac{\partial h(\mathbf{x}, \mathbf{y})}{\partial \mathbf{y}} = \mathbf{B}^T(\mathbf{A} + \mathbf{A}^{\mathrm{T}})(\mathbf{x} + \mathbf{B}\mathbf{y}),$$

and also

$$\frac{\partial^2 h(\mathbf{x}, \mathbf{y})}{\partial \mathbf{x} \partial \mathbf{x}^{\mathrm{T}}} = \mathbf{A} + \mathbf{A}^{\mathrm{T}} \quad \text{and} \quad \frac{\partial^2 h(\mathbf{x}, \mathbf{y})}{\partial \mathbf{y} \partial \mathbf{y}^{\mathrm{T}}} = \mathbf{B}^{\mathrm{T}}(\mathbf{A} + \mathbf{A}^{\mathrm{T}})\mathbf{B}.$$

Note that above we use partial derivative notation, i.e. ∂, as opposed to gradient operator notation, i.e. ∇, as the derivatives are not w.r.t. all variables.

Consider the normal linear model $\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$, where $\mathbf{Y} = (Y_1, \ldots, Y_n)^{\mathrm{T}}$, $\mathbf{X}$ is an $n \times (p+1)$ design matrix, $\boldsymbol{\beta}$ is a $(p+1)$-vector of regression coefficients, and $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_n)^{\mathrm{T}}$ with independent $\varepsilon_i \sim N(0, \sigma^2)$. The maximum likelihood estimate of $\boldsymbol{\beta}$, denoted $\hat{\boldsymbol{\beta}}$, minimises the RSS, i.e. minimises $(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$ if we observe $\mathbf{y} = (y_1, \ldots, y_n)^{\mathrm{T}}$. Differentiating w.r.t. $\boldsymbol{\beta}$ gives $-\mathbf{X}^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) - (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^{\mathrm{T}}\mathbf{X}$, which simplifies to $-2\mathbf{X}^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$ as $\mathbf{X}^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$ and $(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^{\mathrm{T}}\mathbf{X}$ are both n-vectors. The derivative is zero at $\hat{\boldsymbol{\beta}}$ and so $-2\mathbf{X}^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) = 0$. Therefore $\hat{\boldsymbol{\beta}}$ is the solution of $\mathbf{X}^{\mathrm{T}}\mathbf{X}\hat{\boldsymbol{\beta}} = \mathbf{X}^{\mathrm{T}}\mathbf{y}$ or alternatively $\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$, as we were given in Topic 3.

Finite-differencing

Consider again $f(\mathbf{x})$, a function of vector $\mathbf{x}$.

Consider $f : \mathbb{R}^n \to \mathbb{R}$ for n-vector $\mathbf{x}$. Let, $\mathbf{e}_i$ be the n-vector comprising entirely zeros, except for its ith element, which is one. Then the **partial derivative** of $f(\mathbf{x})$ w.r.t. x_i, the ith element of $\mathbf{x}$, is

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}.$$

The above definition leads us to the **finite-difference** partial derivative approximation

$$\frac{\partial f(\mathbf{x})}{\partial x_i} \simeq \frac{f(\mathbf{x} + \delta\mathbf{e}_i) - f(\mathbf{x})}{\delta},$$

where δ is small.

Use finite-differencing to approximate the derivative of $f(x) = \sin(x)$ for $x \in [0, 2\pi]$, and compare its accuracy to the true derivative.

First, let's calculate f and its *true* derivative, i.e. $f'(x) = \cos(x)$, and store this as `partial0` for $\{x_i\}$, $i = 1, \ldots, 100$, a set of equally-spaced points on $[0, 2\pi]$.

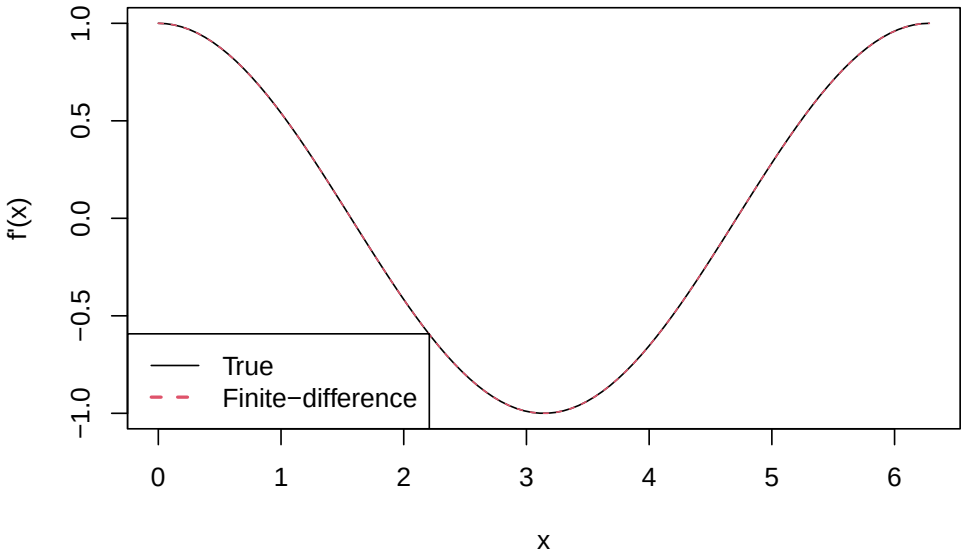
```
> n <- 100
> x <- seq(0, 2 * pi, l = 100)
> f <- sin(x)
> partial0 <- cos(x)
```

Now we'll set $\delta = 10^{-6}$ and calculate $x_i + \delta$, for each x_i, i.e. each element of `x`, so that the finite-difference approximation to the derivative is given by $[\sin(x_i + 10^{-6}) - \sin(x_i)]/10^{-6}$, which is calculated below and stored as `partial1`.

```
> delta1 <- 1e-6
> x1 <- x + delta1
> f1 <- sin(x1)
> partial1 <- (f1 - f) / delta1
```

We'll then plot the $f'(x)$ against it finite-difference approximation.

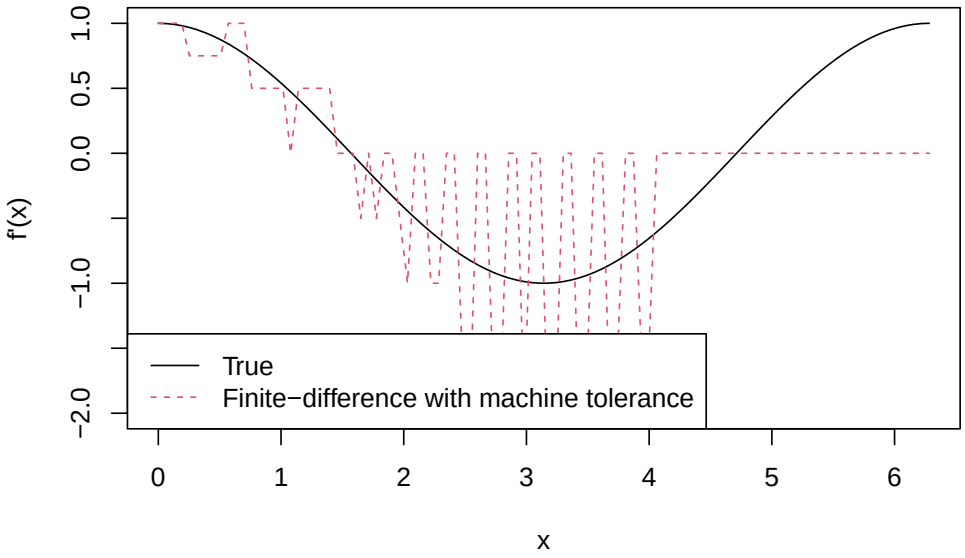
```
> matplot(x, cbind(partial0, partial1), type = 'l', xlab = 'x', ylab = "f'(x)")
> legend('bottomleft', lty = 1:2, col = 1:2, lwd = 1:2,
+        legend = c('True', 'Finite-difference'))
```



In fact, the true derivative and its finite-difference approximation are so similar that's it's difficult to distinguish the two, but they're both there in the plot!

We might be tempted to choose δ as small as possible. Suppose we were to repeat Example @ref(exm:sinfd) with $\delta = \epsilon_m$, i.e. R's machine tolerance. The following calculates the finite-difference approximation as `partial2` and plots this against the true value of $f'(x)$.

```
> delta2 <- .Machine$double.eps
> x2 <- x + delta2
> f2 <- sin(x2)
> partial2 <- (f2 - f) / delta2
> matplot(x, cbind(partial0, partial2), type = 'l', xlab = 'x', ylab = "f'(x)")
> legend('bottomleft', lty = 1:2, col = 1:2, bg = 'white',
+        legend = c('True', 'Finite-difference with machine tolerance'))
```



Using $\delta = \epsilon_m$ gives a terrible approximation to $f'(x)$, which gets worse as x increases. We've actually encountered an example *calculation error*, which was introduced in Chapter 2.

Find $\partial \log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) / \partial y_i$ analytically, for $i = 1, \ldots, p$, where $f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the $MVN_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ pdf, for arbitrary $\mathbf{y}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. Evaluate this for $\mathbf{y}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as in Example @ref(exm:mvn1). Then approximate the same derivative using finite-differencing.

The logarithm of the $MVN_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ pdf is given by

$$\log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{1}{2}\left[p \log(2\pi) + \log(|\boldsymbol{\Sigma}|) + (\mathbf{y} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{y} - \boldsymbol{\mu})\right]$$

which we know, from Example @ref(exm:mvn3), we can evaluate with `dmvn3()`. Using the above properties

$$\frac{\partial \log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})}{\partial \mathbf{y}} = -\boldsymbol{\Sigma}^{-1} (\mathbf{y} - \boldsymbol{\mu})$$

since $\boldsymbol{\Sigma}$ is symmetric.

We can evaluate this for $\mathbf{y}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as in Example @ref(exm:mvn1) with the following

```
> y <- c(.7, 1.3, 2.6)
> mu <- 1:3
```

```
> Sigma <- matrix(c(4, 2, 1, 2, 3, 2, 1, 2, 2), 3, 3)
> deriv1 <- -solve(Sigma, y - mu)
```

which gives

```
> as.vector(deriv1)
```

```
## [1] -0.08  0.38 -0.14
```

To approximate the derivative by finite-differencing, it makes sense to write a multi-purpose function for finite differencing, which we'll call `fd()`.

```
> fd <- function(x, f, delta = 1e-6, ...) {
+   # Function to evaluate derivative by finite-differencing
+   # x is a p-vector
+   # fn is the function for which the derivative is being calculated
+   # delta is the finite-differencing step, which defaults to 10^{-6}
+   # returns a vector of length x
+   f0 <- f(x, ...)
+   p <- length(x)
+   f1 <- numeric(p)
+   for (i in 1:p) {
+     x1 <- x
+     x1[i] <- x[i] + delta
+     f1[i] <- f(x1, ...)
+   }
+   (f1 - f0) / delta
+ }
```

Then we can use this with `dmvn3()` with the following

```
> deriv2 <- fd(y, dmvn3, mu = mu, Sigma = Sigma)
```

which gives

```
> deriv2
```

```
## [1] -0.0800002  0.3799993 -0.1400008
```

and is the same as the analytical result

```
> all.equal(deriv1, deriv2)
```

```
## [1] "Mean relative difference: 2.832689e-06"
```

once we allow for error in the finite-difference approximation.

Quadrature

Another common requirement in statistics that some integral needs to be evaluated. To start, let's consider a simple integral of the form

$$I = \int_a^b f(x)\mathrm{d}x.$$

We'll first take a look at some *deterministic* approaches to numerically evaluating integrals. In fact, these all boil down to assuming that

$$I \simeq \sum_{i=1}^{N} w_i f(x_i^*)$$

for some weights w_i and nodes x_i^*, $i = 1, \ldots, N$. Note that here we're considering the so-called *composite* approach to approximating an interval, i.e. in which a rule is applied over a collection of sub-intervals.

The **relative absolute error**, or sometimes just *relative error*, of an estimate of some true value is given by

$$\left| \frac{\text{true value} - \text{estimate}}{\text{true value}} \right|.$$

Midpoint rule

Perhaps the first numerical integration scheme we come across is the mid point rule. Put simply, we divide $[a, b]$ into N equally-sized intervals, and use the midpoints of these as the nodes, x_i^*. This gives

$$\int_a^b f(x)\mathrm{d}x \simeq h \sum_{i=1}^{N} f(x_i^*),$$

where $x_i^* = a + (i - 0.5)(b - a)/N$ and $h = (b - a)/N$. The error in the approximation is $O(h^2)$. Thus more intervals reduces h and gives a more accurate approximation.

Consider the integral $\int_0^1 \exp(x)\mathrm{d}x = \exp(1) - 1 \simeq 1.7182818$. Use `R` and the midpoint rule to estimate the integral with $N = 10$, 100 and 1000. Then compare the relative absolute error of each.

We'll start by calculating the true value of the integral, which we'll store as `true`.

```
> true <- exp(1) - 1
```

Then we'll store the values of N that we're testing as `N_vals`.

```
> N_vals <- 10^c(1:3)
```

The following then creates a vector, `midpoint`, in which to store the integral approximations, and calculates the approximations with a `for` loop. Inside the loop the integration nodes (i.e. the midpoints) and h are calculated.

```
> midpoint <- numeric(length(N_vals))
> for (i in 1:length(N_vals)) {
+   N <- N_vals[i]
+   nodes <- (1:N - .5) / N
+   h <- 1 / N
+   midpoint[i] <- h * sum(exp(nodes))
+ }
> midpoint
```

```
## [1] 1.717566 1.718275 1.718282
```

The relative absolute error for each is then given in the vector `rel_err_mp` below.

```
> rel_err_mp <- abs((true - midpoint) / true)
> rel_err_mp
```

```
## [1] 4.165452e-04 4.166655e-06 4.166667e-08
```

We clearly see that the absolute error reduces by two factors of ten for each factor of ten increase in N, which is consistent with the above comment of $O(h^2)$ error, where here $h = 1/N$.

Simpson's rule

The midpoint rule works simply by approximating $f(x)$ over a sub-interval of $[a, b]$ by a horizontal line. The trapezium rule (which we'll overlook) assumes a straight line. Simpson's rule is derived from a quadratic approximation and given by

$$\int_a^b f(x)\mathrm{d}x \simeq \frac{h}{6}\left(f(a) + 4\sum_{i=1}^{N} f(x_{1i}^*) + 2\sum_{i=1}^{N-1} f(x_{2i}^*) + f(b)\right), (\#eq:simpson) \tag{6}$$

where $x_{1i}^* = a + h(2i-1)/2$, $x_{2i}^* = a + ih$ and $h = (b-a)/N$. Note that Simpson's rule requires $N+1$ more evaluations of f than the midpoint rule; however, a benefit of those extra evaluations is that its error reduces to $O(h^4)$.

Now use `R` and Simpson's rule to approximate the integral $\int_0^1 \exp(x)\mathrm{d}x = \exp(1) - 1$ with $N = 10$, 100 and 1000, compare the relative absolute error for each, and against those of the midpoint rule in Example @ref(exm:midpoint).

We already have `true` and `N_vals` from Example @ref(exm:midpoint), and we can use a similar `for` loop to approximate the integral using Simpson's rule. The main difference is that we create two sets of nodes, `nodes1` and `nodes2`, which correspond to the x_{1i}s and x_{2i}s in Equation @ref(eq:simpson), respectively. The integral approximations are stored as `simpson`

```
> simpson <- numeric(length(N_vals))
> N_vals <- 10^c(1:3)
> for (i in 1:length(N_vals)) {
+   N <- N_vals[i]
+   h <- 1 / N
+   simpson[i] <- 1 + exp(1)
+   nodes1 <- h * (2*c(1:N) - 1) / 2
+   simpson[i] <- simpson[i] + 4 * sum(exp(nodes1))
+   nodes2 <- h * c(1:(N - 1))
+   simpson[i] <- simpson[i] + 2 * sum(exp(nodes2))
+   simpson[i] <- h * simpson[i] / 6
+ }
> print(simpson, digits = 12)
```

```
## [1] 1.71828188810 1.71828182847 1.71828182846
```

and we print this to 11 decimal places so we can see where the approximations changes with N. Finally we calculate the relative absolute errors, `rel_err_simp`,

```
> rel_err_simp <- abs((true - simpson) / true)
> rel_err_simp
```

```
## [1] 3.471189e-08 3.472270e-12 6.461239e-16
```

and we see a dramatic improvement in the accuracy of approximation that Simpson's rule brings, with relative absolute errors of the same order of magnitude as those form the midpoint rule using $N = 1000$ achieved with $N = 10$ for Simpson's rule. Note, though, that for given N, Simpson's rule requires $N + 1$ more evaluations of $f()$.

Gaussian quadrature

We've seen that Simpson's rule can considerably improve on the midpoint rule for approximating integrals. However, we might still consider both restrictive in that they consider an equally-spaced set of nodes.

Consider $g(x)$, a polynomial of degree $2N - 1$, and a fixed weight function $w(x)$. Then, the **Gauss-Legendre quadrature rule** states that

$$\int_a^b w(x)g(x)\mathrm{d}x = \sum_{i=1}^{N} w_i g(x_i),$$

where, for $i = 1, \ldots, N$, w_i and x_i depend on $w(x)$, a and b, but not $g(x)$.

The Gauss-Legendre quadrature rule is the motivation for **Gaussian quadrature**, whereby we assume that the integral we're interested in can be well-approximated by a polynomial. This results in the approximation

$$\int_a^b f(x)\mathrm{d}x \simeq \sum_{i=1}^{N} w_i f(x_i)$$

for a fixed set of x values, x_i with corresponding weights w_i, for $i = 1, \ldots, N$. There are many rules for choosing the weights, w_i, but (perhaps fortunately) we won't go into them in detail in MTH3045. Instead, we'll just consider the function `pracma::gaussLegrendre()` (for which you'll need to install the `pracma` package), where `pracma::gaussLegrendre(N, a, b)` produces `N` nodes and corresponding weights on the interval [`a`,`b`], with $N =$ `N`, $a =$ `a` and $b =$ `b`. The following produces nodes and weights for $N = 10$ on $[0, 1]$

```
> gq <- pracma::gaussLegendre(10, 0, 1)
> gq
```

```
## $x
##  [1] 0.01304674 0.06746832 0.16029522 0.28330230 0.42556283 0.57443717 0.71669770
##
```

```
## $w
##  [1] 0.03333567 0.07472567 0.10954318 0.13463336 0.14776211 0.14776211 0.13463336
```

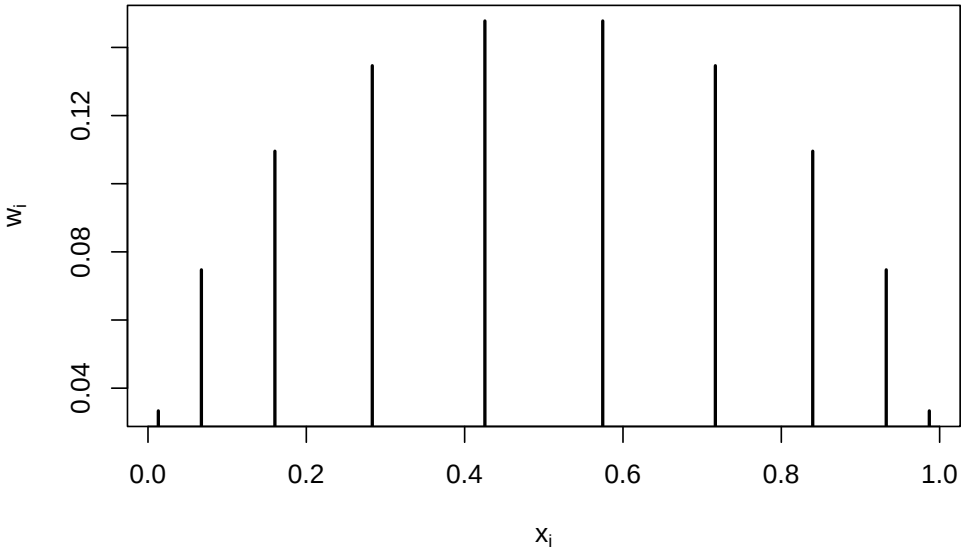


Figure 2: Node and weights for Gauss-Legendre quadrature with $N = 10$ for integral on [0, 1].

which we can see in Figure @ref(fig:gq2), and shows that nodes are spread further apart towards the middle of the $[0, 1]$ range, but given more weight. Note that `pracma::gaussLegrendre()` is named so because it implements Gauss-Legendre quadrature, i.e. Gaussian quadrature with Legendre polynomials[7].

Now use `R` and Gauss-Legendre quadrature to approximate the integral $\int_0^1 \exp(x)\mathrm{d}x$ with $N = 10$. Explore what value of N gives a comparable estimate to that of the midpoint rule with $N = 100$ based on relative absolute error.

We can re-use `true` from Example @ref(exm:midpoint) and then we'll consider $N = 10$, 4 and 3, which we'll call `N_vals`, and store the resulting integral approximations in `gauss`.

```
> N_vals <- c(10, 4, 3)
> gauss <- numeric(length(N_vals))
```

[7]Wikipedia has a useful pages on Gaussian quadrature and on Legendre polynomials, should you want to read more on them.

```
> for (i in 1:length(N_vals)) {
+   N <- N_vals[i]
+   xw <- pracma::gaussLegendre(N, 0, 1)
+   gauss[i] <- sum(xw$w * exp(xw$x))
+ }
> gauss
```

```
## [1] 1.718282 1.718282 1.718281
```

The relative absolute errors, `rel_err_gauss`,

```
> rel_err_gauss <- abs((true - gauss) / true)
> rel_err_gauss
```

```
## [1] 2.584496e-16 5.429651e-10 4.795992e-07
```

show that, having considered $N = 3, 4, 10$, choosing $N = 3$ for Gaussian quadrature gives closest relative absolute error to that of the midpoint rule with $N = 100$, which really is quite impressive. It is important to note, though, that $f(x) = \exp(x)$ is a very smooth function. For wiggler functions, larger N is likely to be needed, and improvements in performance, such as Gaussian quadrature over the midpoint rule, might be significantly less.

Consider a single random variable $Y \mid \lambda \sim \text{Poisson}(\lambda)$, where we can characterise our prior beliefs about λ as $\lambda \sim \text{N}(\mu, \sigma^2)$. Use Gaussian quadrature with $N = 9$ to estimate the marginal pdf of Y if $\mu = 10$ and $\sigma = 3$.

The marginal distribution of Y is given by

$$f(y) = \int_{-\infty}^{\infty} f(y \mid \lambda) f(\lambda) \mathrm{d}\lambda.$$

The *three sigma rule* is a heuristic rule of thumb that 99.7% of values lie within three standard deviations of the mean.

Hence for the $\text{N}(10, 3^2)$ distribution we should expect 99.7% of values to lie within $10 \pm 3 \times 3$. Hence we'll take this as our range for the Gaussian quadrature nodes.

```
> mu <- 10
> sigma <- 3
> N <- 9  # no. of nodes
> xw <- pracma::gaussLegendre(
+         N,
```

```
+         mu - 3 * sigma, # left-hand end
+         mu + 3 * sigma # right-hand end
+ )
> xw
```

```
## $x
## [1]  1.286558  2.475720  4.479657  7.081719 10.000000 12.918281 15.520343 17.52428
##
## $w
## [1] 0.7314695 1.6258334 2.3454963 2.8111237 2.9721542 2.8111237 2.3454963 1.625833
```

which are stored as `xw$x` with corresponding weights `xw$w`, $w_1, \ldots, w_N$.

Next we want a set of values at which to evaluate $f(y)$, and for this we'll choose $0, 1, \ldots, 30$, which we can create in `R` with

```
> y_vals <- 0:30
```

Then we can estimate $f(y)$ as

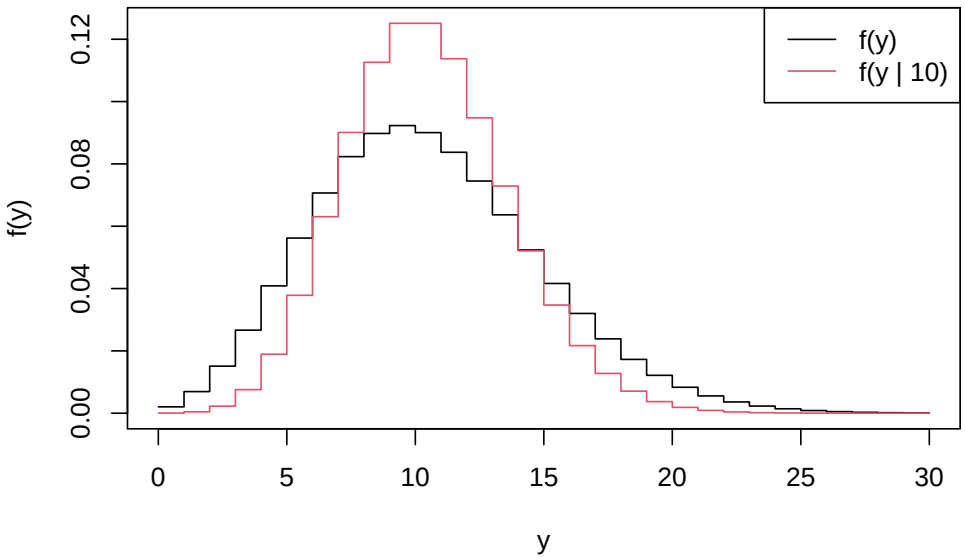
$$\hat{f}(y) \simeq \sum_{i=1}^{N} w_i f(y \mid \lambda_i^*) f(\lambda_i^*).$$

The following code gives $\hat{f}(y)$ as `fhat` for y in the set of values `y_vals`.

```
> m <- length(y_vals)
> fhat <- numeric(m)
> for (i in 1:m) {
+   fhat[i] <- sum(xw$w * dpois(y_vals[i], xw$x) *
+                    dnorm(xw$x, mu, sigma))
+ }
```

Finally, we'll plot $\hat{f}(y)$ against the pdf of the Poisson(10) distribution

```
> matplot(y_vals, cbind(fhat, dpois(y_vals, mu)), lwd = 1,
+         col = 1:2, lty = 1, type = 'l', xlab = 'y', ylab = 'f(y)')
> legend('topright', c("f(y)", "f(y | 10)"),
+        lty = 1, col = 1:2)
```



which demonstrates that $f(y)$ is broader than $f(y \mid 10)$, which is to be expected given that $f(y)$ integrates out the variability in λ given by the $\text{N}(10, 3^2)$ distribution.

One-dimensional numerical integration in R

Unsurprisingly, R has a function for one-dimensional numerical integration. It's called `integrate()`. It uses a method that builds on Gaussian quadrature, but we won't go into its details. Use of `integrate()`, however, is fairly straightforward.

Evaluate the integral $\int_0^1 \exp(x)\text{d}x = \exp(1) - 1$ using R's `integrate()` function with $N = 10$ and report its relative absolute error.

We can use the following code, where the first argument to `integrate()` is the function we're integrating, the second and third are the lower and upper ranges of the definite integral, and `subdivisions` is the maximum number of nodes to use in the approximation, which defaults to 100.

```
> true <- exp(1) - 1
> estimate <- integrate(function(x) exp(x), 0, 1, subdivisions = 10)
> rel_err <- abs(true - estimate$value) / true
```

Note above that the absolute error is similarly tiny to that of Gaussian quadrature above. The values themselves, being so close to the machine tolerance, are incomparable. But we can be sure that the approximation is

incredibly accurate.

Multi-dimensional quadrature

Now suppose that we want to integrate a multi-variable function over some finite range. For simplicity, we'll just consider the case of function of two variables, $f(x, y)$. Then, by the earlier one-dimensional Gaussian quadrature we have that

$$\int f(x, y_j)\mathrm{d}x \simeq \sum_{i=1}^{M} w_{x,i} f(x_i, y_j)$$

from which it follows, by another application of Gaussian quadrature, that

$$\int \int f(x, y)\mathrm{d}x\mathrm{d}y \simeq \sum_{i=1}^{M} w_{x,i} \sum_{j=1}^{N} w_{y,j} f(x_i, y_j).$$

Note that the weight sequences $w_{x,i}$, $i = 1, \ldots, M$, and $w_{y,j}$, $j = 1, \ldots, N$, can be specified separately, i.e. according to different quadrature rules.

More generally, we therefore have that

$$\int \ldots \int f(x_1, \ldots, x_d)\mathrm{d}x_1 \ldots, \mathrm{d}x_d = \int f(\mathbf{x})\mathrm{d}\mathbf{x} \simeq \sum_{i_1=1}^{N_1} \ldots \sum_{i_d=1}^{N_d} w_{x_1,i_1} \ldots w_{x_d,i_d} f(x_{1,i_1}, \ldots, x_{d,i_d}).$$

In practice, multi-dimensional quadrature is only feasible for small numbers of dimensions. For example, consider a d-variable function f, such that each dimension has N nodes. This will require N^d evaluations of f. Some useful numbers to draw upon are 10^6, and $3^{15} \simeq 14348907$.

Use the midpoint rule to approximate

$$I = \int_0^1 \int_0^1 \exp(x_1 + x_2)\mathrm{d}x_1\mathrm{d}x_2$$

with 10 nodes for each variable, and estimate its relative absolute error.

The following sets d and finds the integral's true value, `true`.

```
> d <- 2
> true <- (exp(1) - 1)^d
```

We want $N = 10$ nodes per variable

```
> N <- 10
```

and then we'll use the following nodes for x_1 and x_2 on $[0, 1]$

```
> x1 <- x2 <- (1:N - .5) / N
```

The approximation to the integral is given by

$$\hat{I} = \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} f(x_{1i}, x_{2j})$$

where $w_{ij} = N^{-d}$ for $i, j = 1, \ldots, N$. This can can be evaluated in `R` with

```
> midpoint <- 0
> w <- 1/(N^d)
> for (i in 1:N) for (j in 1:N)
+   midpoint <- midpoint + w * exp(x1[i] + x2[j])
```

where `midpoint` calculates $\hat{I}$ above. The following give the true integral, alongside its midpoint-rule based estimate, and the absolute relative error of the estimate, `rel.err`

```
> c(true = true, midpoint = midpoint, rel.err = abs(midpoint - true) / true)
```

```
##         true     midpoint      rel.err
## 2.9524924420 2.9500332614 0.0008329168
```

We see that the estimate is reasonably accurate, but we have had to evaluate $f(x_1, x_2)$ 100 times.

Use Gaussian quadrature to approximate I from Example @ref(exm:twovar) with 4 nodes for each variable, and estimate its relative absolute error.

We can take `d` and `true` from Example @ref(exm:twovar). Then we calculate $\hat{I}$ as above, but using the integration nodes and weights of Gaussian quadrature.

```
> N <- 4
> xw1 <- xw2 <- pracma::gaussLegendre(N, 0, 1)
> gq <- 0
> for (i in 1:N) for (j in 1:N)
+   gq <- gq + xw1$w[i] * xw2$w[j] * exp(xw1$x[i] + xw2$x[j])
> gq
```

```
## [1] 2.952492
```

We again see below that Gaussian quadrature gives an incredibly accurate estimate

```
> c(true = true, guass_quad = gq, rel.err = abs(gq - true) / true)
```

```
##         true   guass_quad      rel.err
## 2.952492e+00 2.952492e+00 1.085930e-09
```

yet now we've only had to evaluate $f(x_1, x_2)$ 16 times.

Use the Gaussian quadrature to approximate

$$I = \int_0^1 \dots \int_0^1 \exp(\sum_{i=1}^{5} x_i) \mathrm{d}x_1 \dots \mathrm{d}x_5$$

with 4 nodes for each variable, and estimate its relative absolute error.

We can proceed as in Example @ref(exm:twovargq) by storing d as `d` and I as `true`.

```
> d <- 5
> true <- (exp(1) - 1)^d
```

Now we'll need a new tactic, because forming `xw1`, `xw2`, ..., `xw5` will be rather laborious. Having the same nodes and weights for each variable, i.e. for x_1, x_2, ..., x_5, simplifies the following code a bit. We'll start by setting n and getting the quadrature nodes and weights, which will be repeated for each variable.

```
> N <- 4
> xw <- pracma::gaussLegendre(N, 0, 1)
```

Then we want to put together all the combinations of the nodes, `X`, and weights, `W`, that will be used.

```
> xxww <- lapply(1:d, function(i) xw)
> X <- expand.grid(lapply(xxww, '[[', 1))
> W <- expand.grid(lapply(xxww, '[[', 2))
```

We then want multiply all the combinations of weights, for which we could use `apply(..., 1, prod)`, but instead we can use `rowSums()` in the following way

```
> w <- exp(rowSums(log(W)))
```

We then calculate $\hat{I}$ and store this as `gq`

```
> gq <- sum(w * exp(rowSums(X)))
```

and see that we still get an excellent approximation to I

```
> c(true = true, gauss_quad = gq, rel.err = abs(gq - true) / true)
```

```
##          true   gauss_quad      rel.err
## 1.497863e+01 1.497863e+01 2.714825e-09
```

but have now evaluated $\exp(\sum_{j=1}^{5} x_i)$ 1024 times.

Laplace's method

An aside on Taylor series in one dimension

In MTH1002 you met Taylor series expansions. The expansion is at the centre of many useful statistical approximations.

Consider a function $f(x)$ in the region of a point x_0 that is infinitely differentiable. Then

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f^{(3)}(x_0)}{3!}(x - x_0)^3 + \dots$$

where $f'(x)$ and $f''(x)$ denote the first and second derivatives of $f(x)$ w.r.t. x, respectively, and $f^{(n)}(x)$ denotes the nth derivative, for $n = 3, 4, 5, \dots$.

From a statistical perspective we're often just looking to approximate $f(x)$ up to second-order terms, and hence we may work with the truncated expansion

$$f(x) \simeq f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0).(\#eq : taylor) \quad (7)$$

Definition

Consider the integral

$$I_n = \int_{-\infty}^{\infty} \mathrm{e}^{-nf(x)} \mathrm{d}x, (\#eq : In) \quad (8)$$

where $f(x)$ is a *convex* function with a minimum at $x = \tilde{x}$, so that $f'(\tilde{x}) = 0$. The integral can be approximated, using **Laplace's method**, as

$$I_n \simeq \mathrm{e}^{-n[f(\tilde{x})]} \sqrt{\frac{2\pi}{nf''(\tilde{x})}}, (\#eq : laplace) \quad (9)$$

which is also sometimes referred to as the *Laplace approximation*[8].

Laplace's method can be derived from the truncated Taylor expansion of Equation @ref(eq:taylor). If we expand $f(x)$ about its global maximum, which we'll denote by $\tilde{x}$, then

$$f(x) \simeq f(\tilde{x}) + \frac{1}{2}(x-\tilde{x})^2 f''(\tilde{x}),$$

since $f'(\tilde{x}) = 0$. Substituting this in we get

$$\begin{aligned} I_n &\simeq \int_{-\infty}^{\infty} \exp\left\{-n\left[f(\tilde{x}) + \frac{1}{2}(x-\tilde{x})^2 f''(\tilde{x})\right]\right\} \mathrm{d}x \\ &= \mathrm{e}^{-n[f(\tilde{x})]} \int_{-\infty}^{\infty} \exp\left\{-\frac{n(x-\tilde{x})^2 f''(\tilde{x})}{2}\right\} \mathrm{d}x. \end{aligned}$$

We recognise $-n(x-\tilde{x})^2 f''(\tilde{x})/2$ as the exponential part of the Normal($\tilde{x}$, $1/nf''(\tilde{x})$) distribution, for which the pdf, $\sqrt{nf''(\tilde{x})/(2\pi)}\exp\{-n(x-\tilde{x})^2 f''(\tilde{x})/2\}$, integrates to unity, and so

$$\int_{-\infty}^{\infty} \exp\left\{-\frac{n(x-\tilde{x})^2 f''(\tilde{x})}{2}\right\} \mathrm{d}x = \sqrt{\frac{2\pi}{nf''(\tilde{x})}}.$$

Substituting this into I_n above we get $I_n \simeq \mathrm{e}^{-n[f(\tilde{x})]}\sqrt{(2\pi)/nf''(\tilde{x})}$, as stated in Equation @ref(eq:laplace).

Knowledge of the derivation of Laplace's method is beyond the scope of MTH3045.

Recall Example @ref(exm:gqpois). Use Laplace's method to approximate $f(y)$.

Given Equation @ref(eq:In) we have

$$f(y,\lambda) = \lambda - y\log(\lambda) + \frac{1}{2\sigma^2}(\lambda-\mu)^2 + \text{constant},$$

[8] Pierre-Simon Laplace (23 March 1749 – 5 March 1827) was a "French mathematician, astronomer, and physicist who was best known for his investigations into the stability of the solar system. Laplace successfully accounted for all the observed deviations of the planets from their theoretical orbits by applying Sir Isaac Newton's theory of gravitation to the solar system, and he developed a conceptual view of evolutionary change in the structure of the solar system. He also demonstrated the usefulness of probability for interpreting scientific data", according to Britannica.com. He developed the Laplace transform, in addition to Laplace's method, and, with Abraham de Moivre, is responsible for the de Moivre–Laplace theorem, which approximates the binomial distribution with a normal distribution.

with $n = 1$, which is based on swapping x for λ. Then

$$f'(y, \lambda) = 1 - \frac{y}{\lambda} + \frac{1}{\sigma^2}(\lambda - \mu)$$

and

$$f''(y, \lambda) = \frac{y}{\lambda^2} + \frac{1}{\sigma^2}.$$

We'll first set `sigsq`.

```
> sigsq <- sigma^2
```

We can't find $\tilde{\lambda}$ analytically, and so we'll find it numerically instead. The following function, `tilde_lambda(y, mu, sigsq)`, gives $\tilde{\lambda}$ given $y =$, `y`, $\mu =$ `mu` and $\sigma^2 =$ `sigsq`.

```
> f <- function(lambda, y, mu, sigsq)
+   - dpois(y, lambda, log = TRUE) - dnorm(lambda, mu, sqrt(sigsq), log = TRUE)
> tilde_lambda <- function(y, mu, sigsq) {
+   optimize(f, c(.1, 20), y = y, mu = mu, sigsq = sigsq)$minimum
+ }
```

We'll re-use `mu`, `sigsq` and `y_vals` from Example @ref(exm:gqpois). We then want to find $\tilde{\lambda}$ for each y in `y_vals`, which we'll store as the vector `tilde_lambdas`.

```
> tilde_lambdas <- sapply(y_vals, tilde_lambda, mu = mu, sigsq = sigsq)
> tilde_lambdas
```

```
##  [1]  1.000000  3.541378  4.771984  5.720154  6.520803  7.226814  7.865452  8.4529
## [19] 13.237752 13.586233 13.925720 14.256831 14.580111 14.896197 15.205447 15.5083
```

We then want to evaluate the approximation to I_n given by Equation @ref(eq:laplace). We'll start with a function to evaluate $f''(\tilde{\lambda})$, which we'll use for each $\tilde{\lambda}$ in `tilde_lambdas`.

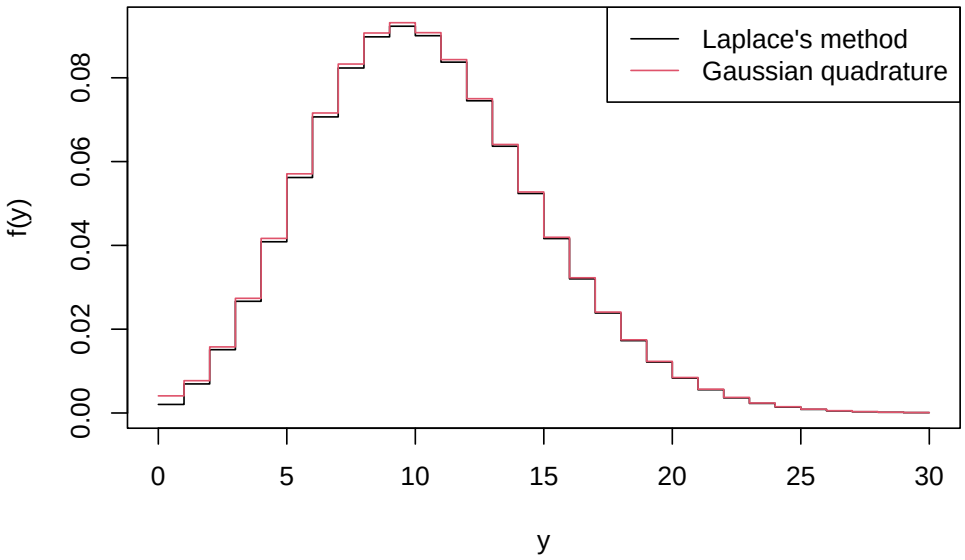
```
> f2 <- function(lambda, y, mu, sigsq)
+   y / lambda^2 + 1 / sigsq
> f2_lambdas <- f2(tilde_lambdas, y_vals, mu, sigsq)
```

Then we'll approximate I_n and store this as `fhat2`.

```
> fhat2 <- sqrt(2 * pi / f2_lambdas) * exp(-f(tilde_lambdas, y_vals, mu, sigsq))
```

Finally we'll plot the Laplace approximation to $f(y)$ against y for `y_vals`, i.e. `fhat2`, alongside the estimate obtained by Gaussian quadrature in Example @ref(exm:gqpois)

```
> matplot(y_vals, cbind(fhat, fhat2),
+         col = 1:2, lty = 1, type = 'l', xlab = 'y', ylab = 'f(y)')
> legend('topright', c("Laplace's method", "Gaussian quadrature"),
+        lty = 1, col = 1:2)
```



and see that the two approaches to approximate $f(y)$ give very similar results.

Laplace's method for multiple dimensions

Above we considered Taylor series for a function of one variable. Laplace's method extends to higher dimensions, although the one-variable case will only be assessed in MTH3045.

Consider the Taylor series for functions of multiple variables, denote $f(\mathbf{x})$, where $f : \mathbb{R}^n \to \mathbb{R}$. The infinite series notation becomes a bit cumbersome, so we'll skip to the second-order approximation.

$$f(\mathbf{x}) \simeq f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^{\mathrm{T}} \nabla f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^{\mathrm{T}} \left[\nabla^2 f(\mathbf{x}_0)\right] (\mathbf{x} - \mathbf{x}_0).$$

Then consider the integral

$$I_n = \int \mathrm{e}^{-nf(\mathbf{x})} \mathrm{d}\mathbf{x}.$$

Laplace's method gives that

$$I_n \simeq \left(\frac{2\pi}{n}\right)^{d/2} \frac{\mathrm{e}^{-nf(\tilde{\mathbf{x}})}}{|\mathbf{H}|^{1/2}},$$

where $\mathbf{H}$ is the Hessian matrix of $f(\mathbf{x})$ evaluated at $\mathbf{x} = \tilde{\mathbf{x}}$, i.e. $\nabla^2 f(\tilde{\mathbf{x}})$. The above approximation results from integrating out the $MVN_p(\tilde{\mathbf{x}}, \mathbf{H}^{-1})$ distribution. This is effectively a multivariate extension to integrating out the $N(\tilde{x}, 1/nf''(\tilde{x}))$, which led to Equation @ref(eq:laplace).

Monte Carlo integration

So far we have considered *deterministic* approaches to numerical integration; these will always give the same answer. It is worth, however, considering a stochastic approach to integration known as *Monte Carlo* integration.

Now consider $\mathbf{X} = (X_1, \ldots, X_d)$ on some finite region $\mathcal{X}$ with volume $V(\mathcal{X})$, where, additionally $\mathbf{X}$ is uniformly distributed on $\mathcal{X}$. Then

$$I_{\mathcal{X}} = \int_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x})\mathrm{d}\mathbf{x} = \mathrm{E}[f(\mathbf{X})]V(\mathcal{X}).(\#eq:mc) \tag{10}$$

The standard Monte Carlo estimate for $I_{\mathcal{X}}$ is given by

$$\hat{I}_{\mathcal{X}} \simeq \frac{V(\mathcal{X})}{N} \sum_{i=1}^{N} f(\tilde{\mathbf{x}}_i)$$

where $\tilde{\mathbf{x}}_i$, for $i = 1, \ldots, N$, are drawn *uniformly* from $\mathcal{X}$.

We can immediately see that Monte Carlo integration is a powerful tool as, provided we can uniformly generate points on $\mathcal{X}$, $\tilde{\mathbf{x}}$, and evaluate $f(\tilde{\mathbf{x}})$, then we can approximate I. A natural next question is the accuracy of this approximation and, as $\hat{I}_{\mathcal{X}}$ is unbiased its variance

$$\mathrm{Var}(\hat{I}_{\mathcal{X}}) = \frac{V(\mathcal{X})^2}{N}\mathrm{Var}[f(\mathbf{X})]$$

is key. As $\mathrm{Var}(\hat{I}_{\mathcal{X}})$ decreases linearly with $1/N$, convergence is rather slow. The following example confirms this.

Recall the integral $\int_0^1 \exp(x)\mathrm{d}x = \exp(1) - 1 \simeq 1.7182818$ from Example @ref(exm:midpoint). Use `R` and Monte Carlo integration to approximate the integral with $N = 100$, 1000 and 10000, using the mean of $m = 100$

replicates for each value of N as your approximation. Then compare the relative absolute error for each value of N.

We'll start with a function, `mc(f, N)`, which we'll use to approximate the integral for given N

```
> mc <- function(f, N, a = 0, b = 1, ...) {
+   x <- runif(N, a, b)
+   V <- b - a
+   V * mean(f(x, ...))
+ }
```

and will quickly test for $N = 100$

```
> mc(function(x) exp(x), 100)
```

```
## [1] 1.832858
```

Note above that as the vector `x` contains random variable, you may get a slightly different result; hence the replicates.

Now we'll perform $m = 100$ replicates for each Monte Carlo sample size N.

```
> estimates <- numeric(3)
> N_vals <- 10^c(2:4)
> m <- 100
> for (i in 1:length(N_vals))
+   estimates[i] <- mean(replicate(m, mc(function(x) exp(x), N_vals[i])))
> estimates
```

```
## [1] 1.715251 1.719208 1.718545
```

Finally, we'll compare these to the true integral, `true` below, by calculating the relative absolute error, `rel_err`

```
> true <- exp(1) - 1
> rel_err <- abs(true - estimates) / true
> rel_err
```

```
## [1] 0.0017637093 0.0005390542 0.0001529152
```

and see that the higher values of N give more accurate approximations, once we allow for variability in Monte Carlo samples.

Recall Example @ref(exm:fivevar), i.e.

$$I = \int_0^1 \dots \int_0^1 \exp(\sum_{i=1}^{5} x_i) \mathrm{d}x_1 \dots \mathrm{d}x_5.$$

Use Monte Carlo integration with Monte Carlo samples of size $N = 10, 100, 10^3, 10^4$ and 10^5 to estimate I and estimate the relative absolute error for each N.

The following code approximates the integral for the different values of N.

```
##          true monte_carlo     rel.err
## [1,] 14.97863    17.62581 0.176730779
## [2,] 14.97863    13.99063 0.065960649
## [3,] 14.97863    14.91521 0.004233631
## [4,] 14.97863    14.92255 0.003743597
## [5,] 14.97863    15.00143 0.001522209
```

```
> d <- 5
> true <- (exp(1) - 1)^d
> NN <- 10^c(1:5)
> f <- numeric(length(NN))
> for (i in 1:length(NN))
+   f[i] <- mean(exp(rowSums(matrix(runif(d * NN[i]), NN[i]))))
> cbind(true = true, monte_carlo = f, rel.err = abs(f - true) / true)
```

We see that the relative absolute error decreases as N increases, even with just one replicate.

The above integral, $I_{\mathcal{X}}$ in Equation @ref(eq:mc), can be seen as a special case of

$$I_{g(\mathcal{X})} = \int_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}) g(\mathbf{x}) \mathrm{d}\mathbf{x},$$

for a pdf $g(\mathbf{x})$, which can be estimated by Monte Carlo as

$$\hat{I}_{g(\mathcal{X})} \simeq \frac{1}{N} \sum_{i=1}^{N} f(\tilde{\mathbf{x}}_i), (\#eq:gmc) \tag{11}$$

where $\tilde{\mathbf{x}}_i$ are draws from the pdf $g(\mathbf{x})$.

Recall Example @ref(exm:gqpois). Use Monte Carlo integration with $N = 1000$ to approximate $f(y)$.

We need to draw a sample of size $N = 1000$ from $f(\lambda)$, which corresponds to $g(\mathbf{x})$ in Equation @ref(eq:gmc). The following code does this, simply by calling `rnorm()`. We'll call the sample `lambda`.

```
> N <- 1e3
> lambda <- rnorm(N, 10, 3)
```

By definition, $\lambda > 0$. However,

```
> range(lambda)
```

```
## [1] -0.5806389 19.1004557
```

shows that we have some $\lambda < 0$. We'll simply set these to zero, and proceed drawing from $f(y \mid \mu)$ with `rpois()`, storing the sample as `y`.

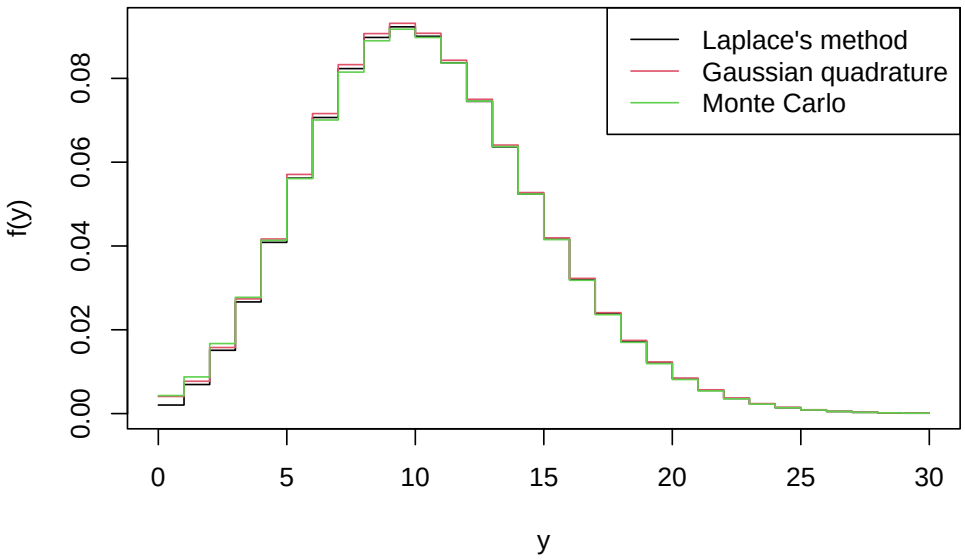
```
> lambda_pos <- pmax(0, lambda)
```

We can estimate $f(y)$ from the mean of the Poisson pmf evaluated at `y` taking each of `y_vals` from Example @ref(exm:gqpois), given the sample `lambda_pos`. We'll store this as `fhat3`.

```
> fhat3 <- sapply(y_vals, function(z) mean(dpois(z, lambda_pos)))
```

Finally, we'll plot the resulting estimate alongside those of Gaussian quadrature and Laplace's method.

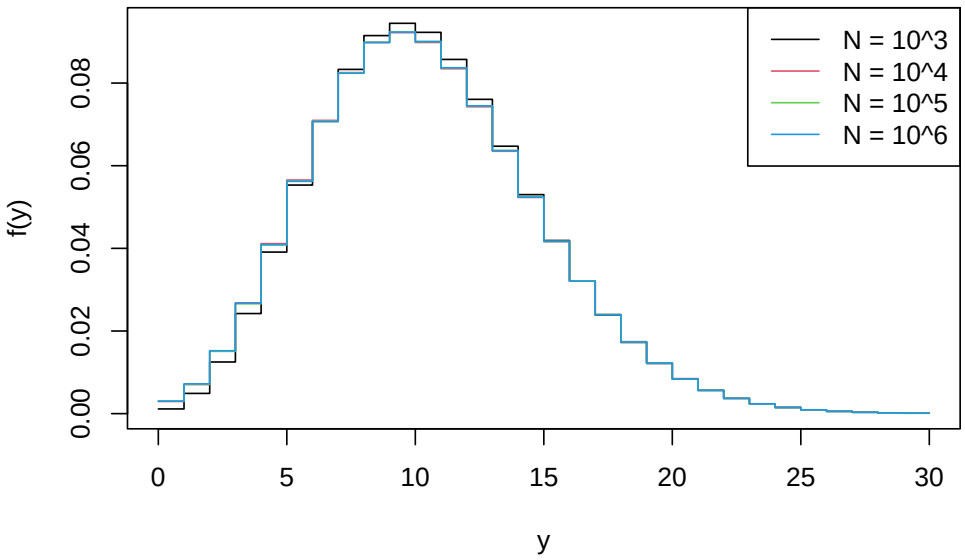
```
> matplot(y_vals, cbind(fhat, fhat2, fhat3),
+         lty = 1, type = 'l', xlab = 'y', ylab = 'f(y)')
> legend('topright', c("Laplace's method", "Gaussian quadrature", "Monte Carlo"),
+        lty = 1)
```



The larger the Monte Carlo sample size, the more accurate the approximation. The following repeats Example @ref(exm:mcpois) for $N = 10^4$, 10^5 and 10^6, comparing the results against $N = 10^3$. Although we don't have the true $f(y)$ against which to compare the Monte Carlo estimates, we are seeing in the Figure below approximations that increase with accuracy as N increases.

```
> NN <- 10^c(3:6)
> fhat4 <- sapply(NN, function(N) {
+   lambda <- rnorm(N, 10, 3)
+   lambda_pos <- pmax(0, lambda)
+   sapply(y_vals, function(z) mean(dpois(z, lambda_pos)))
+ }
+ )
```

```
> matplot(y_vals, fhat4,
+         lty = 1, type = 'l', xlab = 'y', ylab = 'f(y)')
> legend('topright', paste('N = 10^', 3:6, sep = ''), lty = 1)
```



Bibliographic notes

For details on numerical differentiation see Wood (2015, sec. 5.5.2) or Nocedal and Wright (2006, chap. 8). For details on quadrature see Monahan (2011, chap. 10) or Press et al. (2007, chap. 4). For details on Laplace's method see Davison (2003, sec. 11.3.1), Wood (2015, sec. 5.3.1) or Monahan (2011, sec. 12.6). For details on Monte Carlo integration see Monahan (2011, chap. 12) or Davison (2003, sec. 3.3).

Optimisation

When fitting a statistical model we often use maximum likelihood. For this we have some likelihood function $f(\mathbf{y} \mid \boldsymbol{\theta})$, data $\mathbf{y} = (y_1, \ldots, y_n)$ and unknown but *fixed* parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_p)$. We are then required to find

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} f(\mathbf{y} \mid \boldsymbol{\theta}).$$

Sometimes, it will be possible to find $\hat{\boldsymbol{\theta}}$ analytically, but sometimes not. The normal linear model is one example of the former, whereas the gamma distribution is an example of the latter.

Consider an independent sample of data $\mathbf{y} = (y_1, \ldots, y_n)$, and suppose that these are modelled as Gamma(α, β) realisations, i.e. with pdf

$$f(y \mid \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} y^{\alpha-1} e^{-\beta y} \quad \text{for } y > 0$$

where $\alpha, \beta > 0$ and $\Gamma()$ is the gamma function. The log-likelihood is then

$$\log f(\mathbf{y} \mid \alpha, \beta) = n\alpha \log \beta - n \log \Gamma(\alpha) + (\alpha - 1) \sum_{i=1}^{n} \log y_i - \beta \sum_{i=1}^{n} y_i.$$

We can write its derivatives w.r.t. (α, β) in the vector

$$\begin{pmatrix} \dfrac{\partial \log f(\mathbf{y} \mid \alpha, \beta)}{\partial \alpha} \\ \dfrac{\partial \log f(\mathbf{y} \mid \alpha, \beta)}{\partial \beta} \end{pmatrix} = \begin{pmatrix} n \log \beta + \sum_{i=1}^{n} \log y_i \\ n\alpha/\beta - \sum_{i=1}^{n} y_i \end{pmatrix}.$$

Unfortunately, we cannot analytically find both the maximum likelihood estimates, $(\hat{\alpha}, \hat{\beta})$, i.e. we cannot find α and β simultaneously that satisfy that $\partial \log f(\mathbf{y} \mid \alpha, \beta)/\partial \alpha$ and $\partial \log f(\mathbf{y} \mid \alpha, \beta)/\partial \beta$ are both zero. Fortunately, we can still find $(\hat{\alpha}, \hat{\beta})$, but just not analytically.

When we cannot find $\hat{\boldsymbol{\theta}}$ analytically, our next option is to find it numerically: that is, to adopt some kind of iterative process that we expect will ultimately result in the $\hat{\boldsymbol{\theta}}$ that we want, as opposed to an estimate that we don't want.

Although in maximum likelihood estimation interest lies in finding $\boldsymbol{\theta}$ that *maximises* $f(\mathbf{y} \mid \boldsymbol{\theta})$, it is much more common in mathematics to want to *minimise* a function. Fortunately, maximising $f(\mathbf{y} \mid \boldsymbol{\theta})$ is equivalent to minimising $-f(\mathbf{y} \mid \boldsymbol{\theta})$, i.e. $f(\mathbf{y} \mid \boldsymbol{\theta})$ negated. Therefore,

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \{-f(\mathbf{y} \mid \boldsymbol{\theta})\}.$$

So that we can better follow the literature on numerical optimisation, we'll just consider finding minima.

Root finding

We'll start this chapter with a brief aside on root finding, because our main concern will be finding values that maximise (or minimise) functions. Consider some function $f(x)$ for $x \in \mathbb{R}$ and wanting to find the value of x, $\tilde{x}$ say, such that $f(\tilde{x}) = 0$. Sometimes we can analytically find $\tilde{x}$, but sometimes not. We'll just consider the latter case where we'll need to find $\tilde{x}$ numerically, such as through some iterative process. We won't go into the details of root-finding algorithms; instead we'll just look at `R`'s function `uniroot()`. This is `R`'s go-to function for root finding. This chapter will just demonstrate its use by example.

Use `uniroot()` in `R` to find the root of

$$f(x) = (x + 3)(x - 1)^2$$

i.e. to find $\tilde{x}$, where $f(\tilde{x}) = 0$, given that $\tilde{x} \in [-4, 4/3]$.

We'll start by writing a function to evaluate $f()$, which we'll call `f`.

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

Then we'll call `uniroot()`

```
> uniroot(f, c(-4, 4.3))
```

```
## $root
## [1] -2.999997
##
```

```
## $f.root
## [1] 4.501378e-05
##
## $iter
## [1] 7
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 6.103516e-05
```

We see that its output includes various details. Most important are `root`, its estimate of $\tilde{x}$, which is $\tilde{x} \simeq -2.9999972$, and `f.root`, the value of $f()$ at the root, i.e. $f(\tilde{x})$, which is $f(\tilde{x}) \simeq 4.5013782 \times 10^{-5}$ and is sufficiently close to zero that we should be confident that we've reached a root.

We can ask `uniroot()` to extend the search range for the root through its argument `extendInt`. Options are `'no'`, `'yes'`, `'downX'` and `'upX'`, which correspond to not extending the range (the default) or allowing it to be extended to allow upward and downward crossings, just downward or just upward, respectively. (If we want to extend the search interval for the root, `extendInt = 'yes'` is usually the best option. Otherwise, we need to think about how $f()$ behaves at the roots, i.e. whether it's increasing or decreasing. See the help file for `uniroot()` for more details.) If we return to the above example and consider the search range $[-2, -1]$ instead, then by issuing

```
> uniroot(f, c(-2, -1), extendInt = 'yes')
```

```
## $root
## [1] -2.999991
##
## $f.root
## [1] 0.0001449862
##
## $iter
## [1] 12
##
## $init.it
## [1] 6
##
```

```
## $estim.prec
## [1] 6.103516e-05
```

we do still find the root, even though it's outside of our specified range.

One-dimensional optimisation in R

We'll only briefly look at how we can perform one-dimensional optimisation in R, which is through its `optimize()` function. As described by its help file, `optimize()` uses '*a combination of golden section search and successive parabolic interpolation, and was designed for use with continuous functions*'. We can instead use `optimize()` by calling `optim(..., method = 'Brent')`. The two are equivalent. By default `optim()` uses the Nelder-Mead polytope method, which we'll cover in Section @ref(sec:nelder), which doesn't usually work well in one dimension. The only reason I can see for using `optim(..., method = 'Brent')` over `optimize()` is that `optim()` is R's preferred numerical optimisation function, and hence users my benefit from familiarity with its output, as opposed to that of `optimize()`.

Consider a sample of data $y_1, \ldots, y_n$ as independent realisations from the Exp(λ) distribution with pdf

$$f(y \mid \lambda) = \lambda \exp(-\lambda y) \qquad \text{for } y > 0$$

where $\lambda > 0$ is an unknown parameter that we want to estimate. Its mle is $1/\bar{y}$, where $\bar{y} = n^{-1} \sum_{i=1}^{n} y_i$. Confirm this numerically in R using `optimze()` by assuming that the sample of data

$$0.4, 0.5, 0.8, 1.8, 2.1, 3.7, 8.2, 10.6, 11.6, 12.8$$

are independent Exp(λ) realisations.

By default `optimize()` will find the minimum, so we want to write a function that will evaluate the exponential distribution's log-likelihood

$$\log f(\mathbf{y} \mid \lambda) = n \log \lambda - \lambda \sum_{i=1}^{n} y_i$$

and then negate it. We'll call this `negloglik(lambda, y)`.

```
> negloglik <- function(lambda, y) {
+   # Function to evaluate Exp(lambda) neg. log likelihood
+   # lambda is a scalar
```

```
+   # y can be scalar or vector
+   # returns a scalar
+   -n * log(lambda) + lambda * sum(y)
+ }
```

We then pass this on to `optimize()` with our sample of data, which we'll call `y`.

```
> y <- c(0.4, 0.5, 0.8, 1.8, 2.1, 3.7, 8.2, 10.6, 11.6, 12.8)
> optimize(negloglik, lower = .1, upper = 10, y = y)
```

```
## $minimum
## [1] 0.1904839
##
## $objective
## [1] 26.58228
```

We see that `R`'s numerical maximum likelihood estimate of λ is 0.1904839, and the true value is $1/5.25 \simeq 0.1904762$; so the two agree to five decimal places.

We can ask `optimize()` to be more precise through its `tol` argument, which has default `tol = .Machine$double.eps^0.25`. Smaller values of `tol` will give more accurate numerical estimates.

Calling `optimise()` is equivalent to calling `optimize()`, for those that don't like American spellings of English words.

Newton's method in one-dimension

Recall Theorem @ref(thm:taylor1). The second-order approximation, if we swap from x to θ,

$$f(\theta) \simeq f(\theta_0) + (\theta - \theta_0) f'(\theta_0) + \frac{1}{2}(\theta - \theta_0)^2 f''(\theta_0)$$

can be re-written as

$$f(\theta + \delta) \simeq f(\theta) + \delta f'(\theta) + \frac{1}{2}\delta^2 f''(\theta)$$

for small δ if we consider values near θ for some twice-differentiable function $f()$. If we're trying to find $\theta^* = \theta + \delta$ that minimises $f(\theta + \delta)$ iteratively, then we want θ^* to be an improvement on θ, i.e. $f(\theta + \delta) < f(\theta)$. The best value

of $\theta + \delta$ therefore minimises $f(\theta) + \delta f'(\theta) + \delta^2 f''(\theta)/2$, and if we differentiate this w.r.t. δ we get

$$f'(\theta) = -\delta f''(\theta)$$

so that

$$\delta = -\frac{f'(\theta)}{f''(\theta)}.$$

The above result is the basis for **Newton's method** whereby, if we assume we have a value of θ at iteration i, then we update this at the next iteration so that

$$\theta_{i+1} = \theta_i - \frac{f'(\theta_i)}{f''(\theta_i)}.$$

For the one dimensional case of Newton's method, we will refer to $p_i = -f'(\theta_i)/f''(\theta_i)$ as the **Newton step**.

Let Y_i, $i = 1, \ldots, n$, denote the numbers of cars passing the front of the Laver building between 9 and 10am on weekdays during term time. Assume that these are independent from one day to the next, and that $Y_i \sim \text{Poisson}(\mu)$, for some unknown μ. The likelihood for a sample of data is given by

$$f(\mathbf{y} \mid \mu) = \prod_{i=1}^{n} \frac{\mu^{y_i} \mathrm{e}^{-\mu}}{y_i!}$$

and for which we can write the log-likelihood as

$$\log f(\mathbf{y} \mid \mu) = -n\mu + \sum_{i=1}^{n} y_i \log(\mu) - \sum_{i=1}^{n} \log(y_i!).$$

Of course, we can solve this analytically, which gives a maximum likelihood estimate of $\hat{\mu} = \bar{y} = \sum_{i=1}^{n} y_i/n$. Given the counts below

$$20, 21, 23, 25, 26, 26, 30, 37, 38, 41,$$

we find that $\hat{\mu} = 28.7$. Use five iterations of Newton's method to find $\hat{\mu}$ iteratively, starting with $\mu_0 = 28$, and comment on how many iterations are required to find $\hat{\mu}$ to two decimal places.

We'll start by reading in the data.

```
> y <- c(20, 21, 23, 25, 26, 26, 30, 37, 38, 41)
```

Then we'll find the first and second derivatives of $-\log f(\mathbf{y} \mid \mu)$ w.r.t. μ, which are $n - (\sum_{i=1}^{n} y_i)/\mu$ and $(\sum_{i=1}^{n} y_i)/\mu^2$, respectively. Then we'll write functions in `R`, which we'll call `d1` and `d2`, to evaluate these analytical derivatives.

```
> d1 <- function(mu, y) {
+   # Function to first derivative w.r.t. lambda of
+   # Exp(lambda) neg. log likelihood
+   # lambda is a scalar
+   # y can be scalar or vector
+   # returns a scalar
+   length(y) - sum(y) / mu
+ }
>
> d2 <- function(mu, y) {
+   # Function to second derivative w.r.t. lambda of
+   # Exp(lambda) neg. log likelihood
+   # lambda is a scalar
+   # y can be scalar or vector
+   # returns a scalar
+   sum(y) / mu^2
+ }
```

Then we'll iterate estimates of μ using Newton's method.

```
> mu_i <- numeric(6)
> mu_i[1] <- 28
> for (i in 1:(length(mu_i) - 1))
+   mu_i[i + 1] <- mu_i[i] - d1(mu_i[i], y) / d2(mu_i[i], y)
> mu_i
```

```
## [1] 28.00000 28.68293 28.69999 28.70000 28.70000 28.70000
```

Finally, we'll compare our estimate from Newton's method with the true maximum likelihood estimate

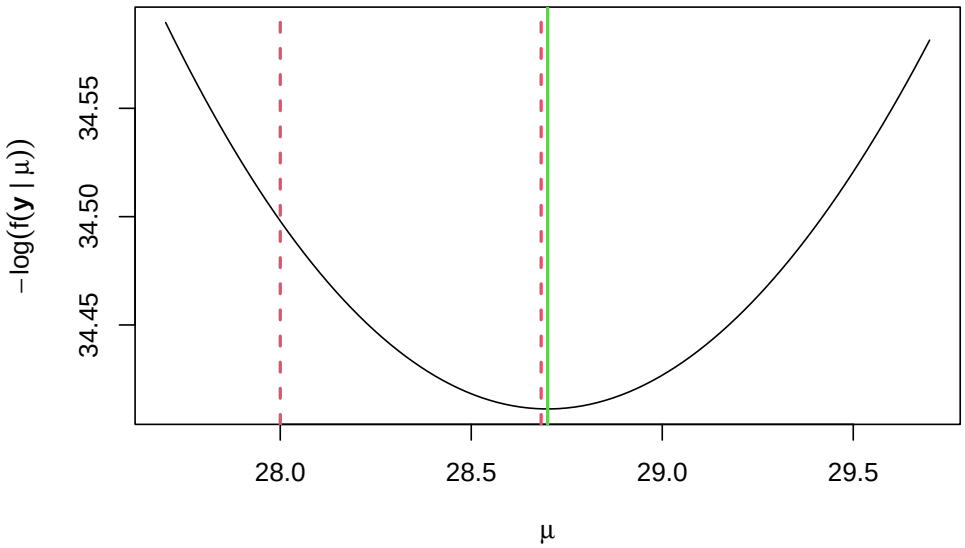
```
> mu_its <- min(which(round(abs(mu_i - mu_hat), 2) == 0)) - 1
```

and find that after 2 iterations our iterative estimate matches the true value of $\hat{\mu}$ to three decimal places.

Let's quickly look at how our iterations have gone. We'll superimpose them on top of a plot of the negative log-likelihood as vertical, dotted red lines,

with the true value of $\hat{\mu}$ shown in green.

```
> mu_seq <- seq(mu_hat - 1, mu_hat + 1, l = 1e2)
> f_seq <- sapply(mu_seq, function(z) -sum(log(dpois(y, z))))
> plot(mu_seq, f_seq, type = "l",
+      xlab = expression(mu), ylab = expression(-log(f(bold(y) * " | " *mu))))
> abline(v = mu_hat, col = 3, lwd = 2)
> abline(v = mu_i[seq_len(mu_its)], col = 2, lty = 2, lwd = 2)
```



We see that Newton's method has quickly homed in on the true value of $\hat{\mu}$. Although this example is simple, and is one in which Newton's method will typically perform well, Newton's method is incredibly powerful, and will often perform well in a wide range of scenarios. Next we'll consider multidimensional $\boldsymbol{\theta}$.

Newton's multi-dimensional method

Taylor's theorem (multivariate)

The above extension to Taylor's theorem in the univariate case applies to the multivariate case, after swapping $\mathbf{x}$ for $\boldsymbol{\theta}$, so that

$$f(\boldsymbol{\theta}) \simeq f(\boldsymbol{\theta}_0) + \left[\nabla f(\boldsymbol{\theta}_0)\right]^{\mathrm{T}} (\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^{\mathrm{T}} \left[\nabla^2 f(\boldsymbol{\theta}_0)\right] (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

can be re-written as

$$f(\boldsymbol{\theta}+\boldsymbol{\Delta}) \simeq f(\boldsymbol{\theta}) + \left[\nabla f(\boldsymbol{\theta})\right]^{\mathrm{T}} \boldsymbol{\Delta} + \frac{1}{2}\boldsymbol{\Delta}^{\mathrm{T}} \left[\nabla^2 f(\boldsymbol{\theta})\right] \boldsymbol{\Delta}.(\#eq:taylor2) \quad (12)$$

As similar argument to that above, i.e. finding $\boldsymbol{\theta}+\boldsymbol{\Delta}$ that minimises equation @ref(eq:taylor2), gives

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \left[\nabla^2 f(\boldsymbol{\theta}_i)\right]^{-1} \nabla f(\boldsymbol{\theta}_i)$$

for which we require that $\nabla^2 f(\boldsymbol{\theta}_i)$ is positive semi-definite in order to ensure that $f(\boldsymbol{\theta}_{i+1}) \leq f(\boldsymbol{\theta}_i)$.

For the multi-dimensional case of Newton's method, we will refer to $\mathbf{p}_i = -\left[\nabla^2 f(\boldsymbol{\theta}_i)\right]^{-1} \nabla f(\boldsymbol{\theta}_i)$ as the **Newton step**.

Sometimes it may turn out that $\nabla^2 f(\boldsymbol{\theta}_i)$ is not positive semi-definite. Fortunately, this does not prohibit use of Newton's method because we can *perturb* $\nabla^2 f(\boldsymbol{\theta}_i)$ so that it is positive semi-definite, which will then guarantee that $f(\boldsymbol{\theta}_{i+1}) \leq f(\boldsymbol{\theta}_i)$. There are various options for perturbation, but a common choice is to use $\nabla^2 f(\boldsymbol{\theta}_i) + \gamma \mathbf{I}_p$, where $\mathbf{I}_p$ is the $p \times p$ identity matrix, and we choose γ very small, and sequentially increase its value until $\nabla^2 f(\boldsymbol{\theta}_i) + \gamma \mathbf{I}_p$ is positive semi-definite. For example, we might proceed through $\gamma = 10^{-12}, 10^{-10}, \ldots$.

The Weibull distribution is sometimes used to model wind speeds. For a wind speed y its pdf is given by

$$f(y \mid \lambda, k) = \frac{k}{\lambda}\left(\frac{y}{\lambda}\right)^{k-1} e^{-(y/\lambda)^k} \quad \text{for } y > 0$$

and where $\lambda, k > 0$ are its parameters. (Note that this is the scale parameterisation of the Weibull distribution.) For observed wind speeds $y_1, \ldots, y_n$ its corresponding log-likelihood is therefore

$$\log f(\mathbf{y} \mid \lambda, k) = n \log k - nk \log \lambda + (k-1) \sum_{i=1}^{n} \log y_i - \sum_{i=1}^{n} \left(\frac{y_i}{\lambda}\right)^k.$$

To implement Newton's method, we need to find the first and second derivatives of $\log f(\mathbf{y} \mid \lambda, k)$ w.r.t. λ and k. The first derivatives are

$$\begin{pmatrix} \dfrac{\partial \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda} \\ \dfrac{\partial \log f(\mathbf{y} \mid \lambda, k)}{\partial k} \end{pmatrix} = \begin{pmatrix} \dfrac{k}{\lambda}\left(\sum_{i=1}^{n}\left(\dfrac{y_i}{\lambda}\right)^k - n\right) \\ \dfrac{n}{k} - n\log\lambda + \sum_{i=1}^{n}\log y_i - \sum_{i=1}^{n}\left[\left(\dfrac{y_i}{\lambda}\right)^k \log\left(\dfrac{y_i}{\lambda}\right)\right] \end{pmatrix}$$

and the second derivatives are stored in the matrix

$$\begin{pmatrix} \dfrac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda^2} & \dfrac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda \partial k} \\ \dfrac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial k \partial \lambda} & \dfrac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial k^2} \end{pmatrix}$$

where

$$\begin{aligned} \frac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda^2} &= \frac{k}{\lambda^2}\left(n - (1+k)\sum_{i=1}^{n}\left(\frac{y_i}{\lambda}\right)^k\right) \\ \frac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial \lambda \partial k} &= \frac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial k \partial \lambda} = \frac{1}{\lambda}\left(\sum_{i=1}^{n}\left(\frac{y_i}{\lambda}\right)^k - n + k\sum_{i=1}^{n}\left[\left(\frac{y_i}{\lambda}\right)^k \log\left(\frac{y_i}{\lambda}\right)\right]\right) \\ \frac{\partial^2 \log f(\mathbf{y} \mid \lambda, k)}{\partial k^2} &= -\frac{n}{k^2} - \sum_{i=1}^{n}\left(\frac{y_i}{\lambda}\right)^k \left[\log\left(\frac{y_i}{\lambda}\right)\right]^2. \end{aligned}$$

Consider the following wind speed measurements (in m/s) for the month of March.

```
> y0 <- c(3.52, 1.95, 0.62, 0.02, 5.13, 0.02, 0.01, 0.34, 0.43, 15.5,
+         4.99, 6.01, 0.28, 1.83, 0.14, 0.97, 0.22, 0.02, 1.87, 0.13, 0.01,
+         4.81, 0.37, 8.61, 3.48, 1.81, 37.21, 1.85, 0.04, 2.32, 1.06)
```

Use five iterations of Newton's method to estimate $\hat{\lambda}$ and $\hat{k}$, assuming the above wind speeds are independent from one day to the next and follow a Weibull distribution.

We'll start by plotting the log-likelihood surface. We wouldn't normally do this, but it can be useful to quickly judge whether the log-likelihood surface is well-behaved, such as being unimodal and approximately quadratic about its maximum.

Then we'll write functions `weib_d1` and `weib_d2` to evaluate the first and second derivatives of the Weibull distribution's log-likelihood w.r.t. λ and k. We'll create these as functions of `pars`, the vector of parameters, `y` the vector of data, and `mult`, a multiplier of the final log-likelihood, which defaults to `1`. Introducing `mult` makes it much simpler when we later need the negative log-likelihood, as we don't have to write separate functions. Often when calculating derivatives w.r.t. multiple parameters, we find that calculations are repeated. It is worth avoiding repetition, and instead storing the results

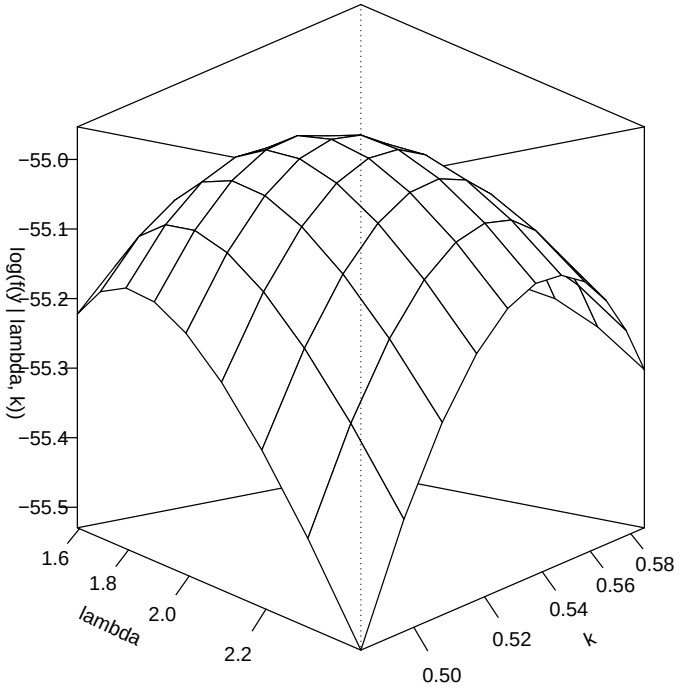


Figure 3: Log-likelihood surface of Weibull distribution model for wind speed data with different λ and k values.

of any calculations that are used multiple times as objects. We'll do this in the next few lines of code, storing the re-used objects as `z1`, `z2`, etc.

```
> weib_d1 <- function(pars, y, mult = 1) {
+   # Function to evaluate first derivative of Weibull log-likelihood
+   # pars is a vector
+   # y can be scalar or vector
+   # mult is a scalar defaulting to 1; so -1 returns neg. gradient
+   # returns a vector
+   n <- length(y)
+   z1 <- y / pars[1]
+   z2 <- z1^pars[2]
+   out <- numeric(2)
+   out[1] <- (sum(z2) - n) * pars[2] / pars[1] # derivative w.r.t. lambda
+   out[2] <- n * (1 / pars[2] - log(pars[1])) +
+     sum(log(y)) - sum(z2 * log(z1)) # w.r.t k
+   mult * out
+ }
>
> weib_d2 <- function(pars, y, mult = 1) {
+   # Function to evaluate second derivative of Weibull log-likelihood
+   # pars is a vector
+   # y can be scalar or vector
+   # mult is a scalar defaulting to 1; so -1 returns neg. Hessian
+   # returns a matrix
+   n <- length(y)
+   z1 <- y / pars[1]
+   z2 <- z1^pars[2]
+   z3 <- sum(z2)
+   z4 <- log(z1)
+   out <- matrix(0, 2, 2)
+   out[1, 1] <- (pars[2] / pars[1]^2) * (n - (1 + pars[2]) * z3) # w.r.t. (lambda^2)
+   out[1, 2] <- out[2, 1] <- (1 / pars[1]) * ((z3 - n) +
+     pars[2] * sum(z2 * z4)) # w.r.t. (lambda, k)
+   out[2, 2] <- -n/pars[2]^2 - sum(z2 * z4^2) # w.r.t. k^2
+   mult * out
+ }
```

Next we'll perform five iterations of Newton's method, ensuring that `mult = -1` so that we negate the log-likelihood in order for minima to be sought,

although we see that reasonable convergence is achieved after two or three iterations.

```
> iterations <- 5
> xx <- matrix(0, iterations + 1, 2)
> dimnames(xx) <- list(paste('iter', 0:iterations), c('lambda', 'k'))
> xx[1, ] <- c(1.6, 0.6)
> for (i in 2:(iterations + 1)) {
+   gi <- weib_d1(xx[i - 1, ], y0, -1)
+   Hi <- weib_d2(xx[i - 1, ], y0, -1)
+   xx[i, ] <- xx[i - 1,] - solve(Hi, gi)
+ }
> xx
```

```
##           lambda         k
## iter 0 1.600000 0.6000000
## iter 1 1.712945 0.5328618
## iter 2 1.866832 0.5375491
## iter 3 1.889573 0.5375304
## iter 4 1.890069 0.5375279
## iter 5 1.890069 0.5375279
```

Finally we can plot the course of the iterations, and see visually that Newton's method quickly homes in on the negative log-likelihood surface's minimum.

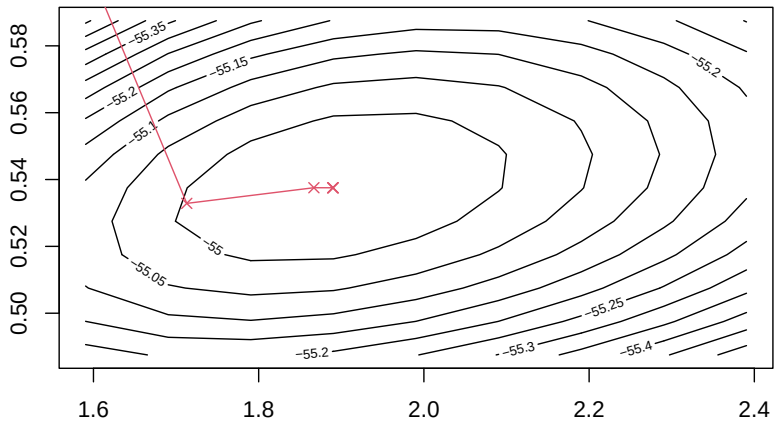


Figure 4: Five iterations of Newton's method to find Weibull maximum likelihood estimates.

Newton's method in R

Above we put together some simple code that implemented Newton's method. However, there are various ways of performing Newton's method in R. Here we just take a look at `nlminb()`, which is described as 'Unconstrained and box-constrained optimization using PORT routines'.

We can use our functions `weib_d1` and `weib_d2` from earlier for the first and second derivatives of the negative log-likelihood w.r.t. λ and k. We now just need a function to evaluate the negative log-likelihood itself. We'll call this `weib_d0`. Note, though, that it's important to ensure that invalid parameters, i.e. $\lambda \leq 0$ and/or $k \leq 0$ are avoided. Below we achieve this by setting the likelihood to be extremely low (10^{-8}) for such parts of parameter space.

```
> weib_d0 <- function(pars, y, mult = 1) {
+   # Function to evaluate Weibull log-likelihood
+   # pars is a vector
+   # y can be scalar or vector
+   # mult is a scalar defaulting to 1; so -1 returns neg. log likelihood
+   # returns a scalar
+   n <- length(y)
+   if (min(pars) <= 0) {
+     out <- -1e8
+   } else {
+     out <- n * (log(pars[2]) - pars[2] * log(pars[1])) +
+     (pars[2] - 1) * sum(log(y)) - sum((y / pars[1])^pars[2])
+   }
+   mult * out
+ }
> nlminb(c(1.6, 0.6), weib_d0, weib_d1, weib_d2, y = y0, mult = -1)
```

```
## $par
## [1] 1.8900689 0.5375279
##
## $objective
## [1] 54.95316
##
## $convergence
## [1] 0
##
```

```
## $iterations
## [1] 5
##
## $evaluations
## function gradient
##        6        6
##
## $message
## [1] "relative convergence (4)"
```

We see that `nlminb`'s output is a list comprising the parameter estimates (`par`), the final value of the negative log-likelihood (`objective`) whether the algorithm has converged (`convergence`, where `0` indicates successful convergence), the number of iterations before convergence was achieved (`iterations`), how many times the function and gradient were evaluated (`evaluations`), and then `message` provides further details on the type of convergence achieved.

Gradient descent

If we consider small $\boldsymbol{\Delta}$ in @ref(eq:taylor2) then we get the first order approximation

$$f(\boldsymbol{\theta} + \boldsymbol{\Delta}) \simeq f(\boldsymbol{\theta}) + \left[\nabla f(\boldsymbol{\theta})\right]^{\mathrm{T}} \boldsymbol{\Delta}, (\#eq:taylor3)$$

which is appropriate for small $\boldsymbol{\Delta}$. The concept behind gradient descent is simple: we want to minimise $\left[\nabla f(\boldsymbol{\theta})\right]^{\mathrm{T}} \boldsymbol{\Delta}$, which requires that we follow the direction of $-\nabla f(\boldsymbol{\theta})$. To allow for different magnitudes of gradient, we will choose

$$\boldsymbol{\Delta} = -\frac{\nabla f(\boldsymbol{\theta})}{||\nabla f(\boldsymbol{\theta})||}.$$

Now that we know the direction in which we want to head, we need to know how far in that direction we should go. For this we'll consider some $\alpha > 0$, so that

$$\begin{aligned} f(\boldsymbol{\theta} + \boldsymbol{\Delta}) &\simeq f(\boldsymbol{\theta}) - \alpha \frac{\left[\nabla f(\boldsymbol{\theta})\right]^{\mathrm{T}} \left[\nabla f(\boldsymbol{\theta})\right]}{||\nabla f(\boldsymbol{\theta})||}, \\ &= f(\boldsymbol{\theta}) - \alpha ||\nabla f(\boldsymbol{\theta})||, (\#eq:taylor4) \end{aligned}$$

which means that $\boldsymbol{\Delta} = -\nabla f(\boldsymbol{\theta}) / ||\nabla f(\boldsymbol{\theta})||$ brings a decrease in $f(\boldsymbol{\theta} + \boldsymbol{\Delta})$ that's proportional to $||\nabla f(\boldsymbol{\theta})||$ for $\alpha > 0$, and is the fastest possible rate at which $f(\boldsymbol{\theta} + \boldsymbol{\Delta})$ can decrease.

Repeat Example @ref(exm:weib) using gradient descent with $\alpha = 0.5$ and $\alpha = 0.1$, using 30 iterations for each. Comment on how these compare to each other, and to Newton's method.

```
> alpha_seq <- c(.5, .1)
> iterations <- 30
> for (j in 1:length(alpha_seq)) {
+   xx2 <- matrix(0, iterations + 1, 2)
+   dimnames(xx2) <- list(paste('iter', 0:iterations), c('lambda', 'k'))
+   xx2[1, ] <- c(1.6, 0.6)
+   for (i in 2:(iterations + 1)) {
+     gi <- weib_d1(xx2[i - 1, ], y0, mult = -1)
+     gi <- gi / sqrt(sum(gi^2))
+     xx2[i, ] <- xx2[i - 1,] - alpha_seq[j] * gi
+   }
+   print(list(paste('alpha', alpha_seq[j], sep = ' = '), tail(xx2, 5)))
+ }
```

```
## [[1]]
## [1] "alpha = 0.5"
##
## [[2]]
##           lambda         k
## iter 26 1.999630 0.6151115
## iter 27 2.012121 0.1152676
## iter 28 2.011794 0.6152675
## iter 29 2.022999 0.1153930
## iter 30 2.022671 0.6153929
##
## [[1]]
## [1] "alpha = 0.1"
##
## [[2]]
##           lambda         k
## iter 26 1.776830 0.6045666
## iter 27 1.784328 0.5048482
## iter 28 1.784878 0.6048466
## iter 29 1.792177 0.5051134
## iter 30 1.792433 0.6051131
```

Above we give the final five iterations for each value of α, and see that

convergence to $\hat{\lambda}$ and $\hat{k}$ has not been achieved after 30 iterations for $\alpha = 0.5$ and for $\alpha = 0.1$, whereas Newton's method had essentially converged after four or five iterations. Worse still, if we allowed more iterations, we'd see that both eventually diverge away from $\hat{\lambda}$ and $\hat{k}$, as opposed to converging. The undesirable course of these iterations can be seen in the following plot.

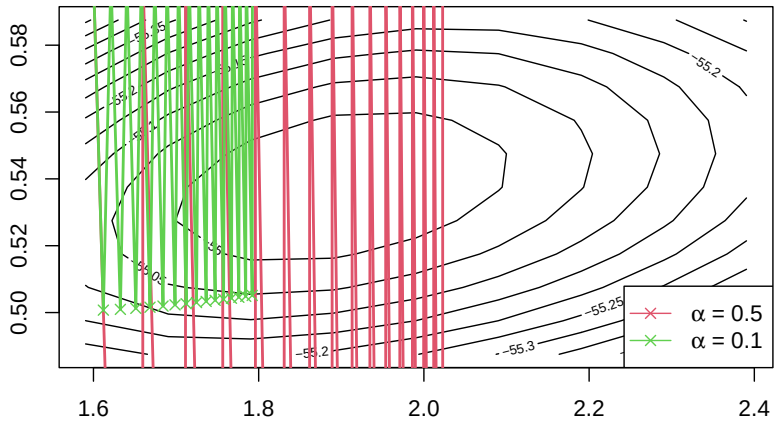


Figure 5: Iterations of the gradient descent algorithm with $\alpha = 0.5$ and $\alpha = 0.1$.

Line search

Above we see that, for fixed α, gradient descent has diverged, i.e. not homed in on the minimum of $f()$. This often happens with gradient descent. A solution, which also applies to Newton's method, is to use a *line search*. Consider Newton's method and a search direction of $\mathbf{p}_i = -\left[\nabla^2 f(\boldsymbol{\theta}_i)\right]^{-1} \nabla f(\boldsymbol{\theta}_i)$. We want $f(\boldsymbol{\theta}_i + \mathbf{p}_i) < f(\boldsymbol{\theta}_i)$ in order for $\boldsymbol{\theta}_i + \mathbf{p}_i$ to be an improvement on $\boldsymbol{\theta}_i$. If we employ a line search, we instead consider $\boldsymbol{\theta}_i + \alpha \mathbf{p}_i$ for some $\alpha > 0$ and ideally want α to minimise $f(\boldsymbol{\theta}_i + \alpha \mathbf{p}_i)$. In practice this can be done informally, through the following process.

1. Choose an initial value for α, α_0, say, and set $j = 0$.
2. Evaluate $f(\boldsymbol{\theta}_i + \alpha_j \mathbf{p}_i)$.
3. Set $j = j + 1$.
4. Set $\alpha_j = \rho \alpha_{j-1}$, for $0 < \rho < 1$.
5. Evaluate $f(\boldsymbol{\theta}_i + \alpha_j \mathbf{p}_i)$.
6. If $f(\boldsymbol{\theta}_i + \alpha_j \mathbf{p}_i) < f(\boldsymbol{\theta}_i + \alpha_{j-1} \mathbf{p}_i)$, repeat steps 3 to 6 until $f(\boldsymbol{\theta}_i + \alpha_j \mathbf{p}_i) \geq f(\boldsymbol{\theta}_i + \alpha_{j-1} \mathbf{p}_i)$.
7. Choose $\alpha = \alpha_{j-1}$.

We can implement this in R.

```
> line_search <- function(theta, p, f, alpha0 = 1, rho = .5, ...) {
+   best <- f(theta, ...)
+   cond <- TRUE
+   while (cond & alpha0 > .Machine$double.eps) {
+     prop <- f(theta + alpha0 * p, ...)
+     cond <- prop >= best
+     if (!cond)
+       best <- prop
+     alpha0 <- alpha0 * rho
+   }
+   alpha <- alpha0 / rho
+   alpha
+ }
```

Notice the use of the `...` argument here, which passes any extra arguments given to `line_search()` on to `f()`, and hence avoids the need to include `f()`'s arguments in `line_search()`. This is useful because it makes `line_search()` applicable to any `f()`.

Repeat Example @ref(exm:weib) using gradient descent but with line search and 200 iterations.

The following code repeats Example @ref(exm:grdescent) with the addition of line search.

```
> iterations <- 200
> xx2 <- matrix(0, iterations + 1, 2)
> dimnames(xx2) = list(paste('iter', 0:iterations), c('lambda', 'k'))
> xx2[1, ] <- c(1.6, 0.6)
> for (i in 2:(iterations + 1)) {
+   gi <- weib_d1(xx2[i - 1, ], y0, mult = -1)
+   gi <- gi / sqrt(sum(gi^2))
+   alpha_i <- line_search(xx2[i - 1, ], -gi, weib_d0, y = y0, mult = -1)
+   xx2[i, ] <- xx2[i - 1,] - alpha_i * gi
+ }
> tail(xx2, 5)
```

```
##            lambda         k
## iter 196 1.889535 0.5376306
## iter 197 1.889556 0.5373873
```

```
## iter 198 1.889559 0.5376314
## iter 199 1.889579 0.5373881
## iter 200 1.889581 0.5376323
```

We see that line search does at least bring us convergence of the parameter estimates, but that it's also very slow, as the following plot shows.

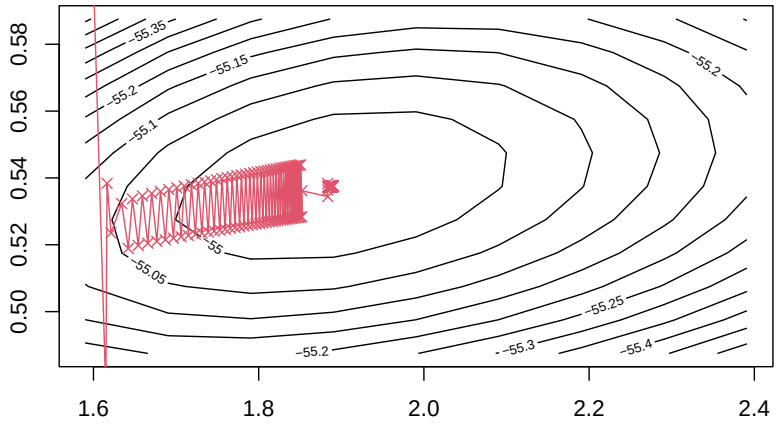


Figure 6: 200 iterations of the gradient descent algorithm using line search.

Here we've adopted an informal approach to line search. A more formal approach is to choose α so that it satisfies the **Wolfe conditions**. A step length α_k is said to satisfy the Wolfe conditions, restricted to the direction $\mathbf{p}_i$, if the following two inequalities hold:

$$\textbf{i)} \quad f(\boldsymbol{\theta}_i + \alpha_i \mathbf{p}_i) \leq f(\boldsymbol{\theta}_i) + c_1 \alpha_i \mathbf{p}_i^{\mathrm{T}} \nabla f(\boldsymbol{\theta}_i),$$

$$\textbf{ii)} \quad -\mathbf{p}_i^{\mathrm{T}} \nabla f(\boldsymbol{\theta}_i + \alpha_i \mathbf{p}_i) \leq -c_2 \mathbf{p}_i^{\mathrm{T}} \nabla f(\boldsymbol{\theta}_i),$$

with $0 < c_1 < c_2 < 1$. c_1 is usually chosen to be quite small while c_2 is much larger; Nocedal and Wright (2006, sec. 6.1) give example values of $c_1 = 10^{-4}$ and $c_2 = 0.9$ for Newton or quasi-Newton methods.

Quasi-Newton methods

Between Newton's method and steepest descent lie **quasi-Newton** methods. These essentially employ Newton's method, but with some approximation to the Hessian matrix. Instead of the search direction used in Newton's method, $\mathbf{p}_i = -\left[\nabla^2 f(\boldsymbol{\theta}_i)\right]^{-1} \nabla f(\boldsymbol{\theta}_i)$, consider the search direction $\tilde{\mathbf{p}}_i = -\mathbf{H}_i^{-1} \nabla f(\boldsymbol{\theta}_i)$, where $\mathbf{H}_i$ is an approximation to the Hessian matrix $\nabla^2 f(\boldsymbol{\theta}_i)$ at

the ith iteration. We might, for example, want to avoid explicitly calculating $\nabla^2 f(\boldsymbol{\theta}_i)$ because it's a matrix that's sufficiently more difficult to calculate than the gradient (e.g. mathematically, or just in terms of time), so that using an approximation to the Hessian matrix (provided it is an adequate approximation) gives a more efficient approach to optimisation than using the Hessian matrix itself. We should typically expect quasi-Newton methods to converge slower than Newton's method, but provided that convergence isn't too much slower or less reliable, then we may prefer this over analytically forming the Hessian matrix.

In MTH3045 we shall consider the so-called **BFGS** (shorthand for Broyden–Fletcher–Goldfarb–Shanno) quasi-Newton algorithm. Put simply, at iteration i, the BFGS algorithm assumes that

$$\nabla^2 f(\boldsymbol{\theta}_i) \simeq \mathbf{H}_i = \mathbf{H}_{i-1} + \frac{\mathbf{y}_i \mathbf{y}_i^{\mathrm{T}}}{\mathbf{y}_i^{\mathrm{T}} \mathbf{s}_i} - \frac{(\mathbf{H}_{i-1})^{-1} \mathbf{s}_i \mathbf{s}_i^{\mathrm{T}} (\mathbf{H}_{i-1})^{-T}}{\mathbf{s}_i^{\mathrm{T}} (\mathbf{H}_{i-1})^{-1} \mathbf{s}_i},$$

where $\mathbf{s}_i = \boldsymbol{\theta}_i - \boldsymbol{\theta}_{i-1}$ and $\mathbf{y}_i = \nabla f(\boldsymbol{\theta}_i) - \nabla f(\boldsymbol{\theta}_{i-1})$. Hence the BFGS algorithm uses differences in the gradients of successive iterations to approximate the Hessian matrix. We now note that we use $\mathbf{H}_i$ in $\mathbf{p}_i = -[\mathbf{H}_i]^{-1} \nabla f(\boldsymbol{\theta}_i)$. We can avoid solving this system of linear equations by instead directly obtaining $[\mathbf{H}_i]^{-1}$ through

$$[\mathbf{H}_i]^{-1} = \left(\mathbf{I}_p - \frac{\mathbf{s}_i \mathbf{y}_i^{\mathrm{T}}}{\mathbf{y}_i^{\mathrm{T}} \mathbf{s}_i} \right) [\mathbf{H}_{i-1}]^{-1} \left(\mathbf{I}_p - \frac{\mathbf{y}_i \mathbf{s}_i^{\mathrm{T}}}{\mathbf{s}_i^{\mathrm{T}} \mathbf{y}_i} \right) + \frac{\mathbf{s}_i \mathbf{s}_i^{\mathrm{T}}}{\mathbf{y}_i^{\mathrm{T}} \mathbf{y}_i}.$$

The following `R` function updates $[\mathbf{H}_{i-1}]^{-1}$ to $[\mathbf{H}_i]^{-1}$ given $\boldsymbol{\theta}_i$, $\boldsymbol{\theta}_{i-1}$, $\nabla f(\boldsymbol{\theta}_{i-1})$ and $\nabla f(\boldsymbol{\theta}_i)$, which are the arguments `x0`, `x1`, `g0` and `g1`, respectively.

```
> iH1 <- function(x0, x1, g0, g1, iH0) {
+   # Function to update Hessian matrix
+   # x0 and x1 are p-vectors of second to last and last estimates, respectively
+   # g0 and g1 are p-vectors of second to last and last gradients, respectively
+   # iH0 is previous estimate of p x p Hessian matrix
+   # returns a p x p matrix
+   s0 <- x1 - x0
+   y0 <- g1 - g0
+   denom <- sum(y0 * s0)
+   I <- diag(rep(1, 2))
+   pre <- I - tcrossprod(s0, y0) / denom
+   post <- I - tcrossprod(y0, s0) / denom
```

```
+   last <- tcrossprod(s0) / denom
+   pre %*% iH0 %*% post + last
+ }
```

Repeat Example @ref(exm:weib) using the BFGS method. Comment on how it compares to Newton's method.

The following code implements five iterations of the BFGS method.

```
> iterations <- 5
> xx <- matrix(0, iterations + 1, 2)
> dimnames(xx) <- list(paste('iter', 0:iterations), c('lambda', 'k'))
> xx[1, ] <- c(1.6, 0.6)
> g <- iH <- list()
> for (i in 2:(iterations + 1)) {
+   g[[i]] <- weib_d1(xx[i - 1, ], y0, mult = -1)
+   if (sqrt(sum(g[[i]]^2)) < 1e-6)
+     break
+   if (i == 2) {
+     iH[[i]] <- diag(1, 2)
+   } else {
+     iH[[i]] <- iH1(xx[i - 2, ], xx[i - 1, ], g[[i - 1]], g[[i]], iH[[i - 1]])
+   }
+   search_dir <- -(iH[[i]] %*% g[[i]])
+   alpha <- line_search(xx[i - 1, ], search_dir, weib_d0, y = y0, mult = -1)
+   xx[i, ] <- xx[i - 1,] + alpha * search_dir
+ }
```

Our estimates at each iteration are

```
> xx
```

```
##          lambda         k
## iter 0 1.600000 0.6000000
## iter 1 1.615241 0.4736904
## iter 2 2.097571 0.5295654
## iter 3 1.918661 0.5356343
## iter 4 1.881726 0.5375145
## iter 5 1.890167 0.5374986
```

and we see that we need two more iterations than Newton's method to reach convergence to three decimal places.

Finally, we'll plot the course of the iterations

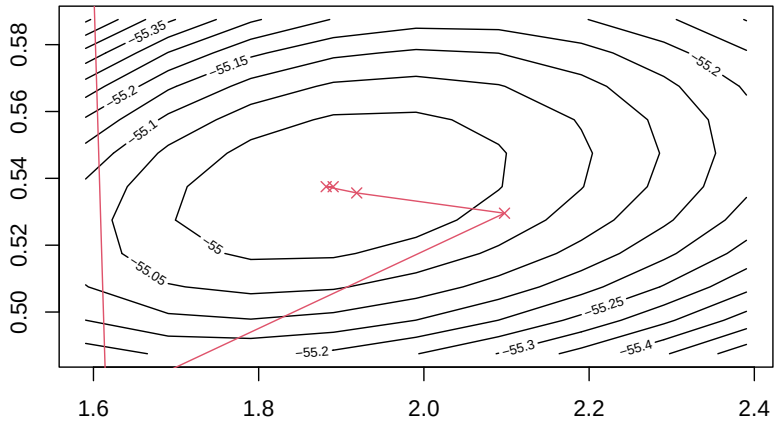


Figure 7: Five iterations of the MFGS quasi-Newton method.

and see that we've taken a slightly less direct route to the minimum than Newton's method.

Quasi-Newton methods in R

There are various options for performing quasi-Newton methods in R. For these, we just need to supply the function to be minimised and its gradient. The first option is to use `nlminb()` again: if we don't supply a function to evaluate the Hessian, then `nlminb()` uses a quasi-Newton approach. The alternative – and possibly preferred – option is to use `optim()` with option `method = 'BFGS'`. The following two lines of code repeat the above example that used Newton's method in R, using `nlminb()` and then `optim()`.

```
> nlminb(c(1.6, 0.6), weib_d0, weib_d1, y = y0, mult = -1)
```

```
## $par
## [1] 1.8900689 0.5375279
##
## $objective
## [1] 54.95316
##
## $convergence
## [1] 0
##
## $iterations
```

```
## [1] 7
##
## $evaluations
## function gradient
##        9        8
##
## $message
## [1] "relative convergence (4)"
```

```
> optim(c(1.6, 0.6), weib_d0, weib_d1, y = y0, mult = -1, method = 'BFGS')
```

```
## $par
## [1] 1.8900632 0.5375283
##
## $value
## [1] 54.95316
##
## $counts
## function gradient
##       14        6
##
## $convergence
## [1] 0
##
## $message
## NULL
```

We see that `nlminb()` and `optim()` have essentially given the same value of `par`, i.e. for $\hat{\lambda}$ and $\hat{k}$, which is reassuring. Note that `nlminb()` has used fewer function evaluations than `optim()`. We won't go into the details of the cause of this, but it is worth noting that the functions use different stopping criteria, and slightly different variants of the BFGS algorithm. Note also that `nlminb()` has used three more function evaluations with the BFGS method than with Newton's method, which is typically the case, and reflects the improved convergence achieved by using the actual Hessian matrix with Newton's method, as opposed to the approximation that's used with the BFGS approach.

Nelder-Mead polytope method

So far we have considered derivative-based optimisation algorithms. When we cannot analytically calculate derivatives, we can use finite-difference approximations. However, sometimes we may want to find the minimum point of a surface that is not particularly smooth. Then derivative information may not be helpful. Instead, we might want an algorithm that explores a surface differently. The Nelder-Mead polytope algorithm is one such approach (Nelder and Mead (1965)). In fact, it is `R`'s default if we use `optim()`, i.e. if we don't supply `method = '...'`.

For the Nelder-Mead algorithm, consider $\boldsymbol{\theta} \in \mathbb{R}^p$. The algorithm starts with $p+1$ test points, $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{p+1}$, which we call *vertices*, and then proceeds as follows.

1. Compute the order statistics of $f(\boldsymbol{\theta}_1), \ldots, f(\boldsymbol{\theta}_{p+1})$ vertices, i.e. find the order
$$f(\boldsymbol{\theta}^{(1)}) \leq f(\boldsymbol{\theta}^{(2)}) \leq \ldots \leq f(\boldsymbol{\theta}^{(p+1)})$$
and check whether the termination criteria have been met (which are given later). If not, proceed to Step 2.

2. Calculate the centroid, $\boldsymbol{\theta}_o$, of $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_p$, i.e. omitting $\boldsymbol{\theta}_{p+1}$, because $\boldsymbol{\theta}_{p+1}$ is the worst vertex.

3. Reflection. Compute the reflected point $\boldsymbol{\theta}_r = \boldsymbol{\theta}_o + \alpha(\boldsymbol{\theta}_o - \boldsymbol{\theta}_{p+1})$. If $f(\boldsymbol{\theta}_1) \leq f(\boldsymbol{\theta}_r) < f(\boldsymbol{\theta}_p)$, replace $\boldsymbol{\theta}_{p+1}$ with $\boldsymbol{\theta}_r$ and return to Step 1. Otherwise, proceed to Step 4.

4. Expansion. If $f(\boldsymbol{\theta}_r) < f(\boldsymbol{\theta}_1)$, i.e. is the best point so far, compute the expanded point $\boldsymbol{\theta}_e = \boldsymbol{\theta}_o + \gamma(\boldsymbol{\theta}_r - \boldsymbol{\theta}_o)$ for $\gamma > 1$. If $f(\boldsymbol{\theta}_e) < f(\boldsymbol{\theta}_r)$, replace $\boldsymbol{\theta}_{p+1}$ with $\boldsymbol{\theta}_r$ and return to Step 1. Otherwise, replace $\boldsymbol{\theta}_{p+1}$ with $\boldsymbol{\theta}_r$ and return to Step 1.

5. Contraction. Now $f(\boldsymbol{\theta}_r) \geq f(\boldsymbol{\theta}_p)$. Compute the contracted point $\boldsymbol{\theta}_c = \boldsymbol{\theta}_o + \rho(\boldsymbol{\theta}_{p+1} - \boldsymbol{\theta}_o)$ for $0 < \rho \leq 0.5$. If $f(\boldsymbol{\theta}_c) < f(\boldsymbol{\theta}_{p+1})$ then replace $\boldsymbol{\theta}_{p+1}$ with $\boldsymbol{\theta}_c$ and return to Step 1. Otherwise proceed to Step 6.

6. Shrink. For $i = 2, \ldots, p+1$ set $\boldsymbol{\theta}_i = \boldsymbol{\theta}_1 + \sigma(\boldsymbol{\theta}_i - \boldsymbol{\theta}_1)$ and return to Step 1.

Often the values $\alpha = 1$, $\gamma = 2$, $\rho = 0.5$ and $\sigma = 0.5$ are used. In Step 1, termination is defined in terms of tolerances. The main criterion is for $f(\boldsymbol{\theta}_{p+1}) - f(\boldsymbol{\theta}_1)$ to be sufficiently small, so that $f(\boldsymbol{\theta}_i)$ for $i = 1, \ldots, p+1$ are

close together for all θ_i, and hence we are hoping that *all* the $\boldsymbol{\theta}_i$ values are in the region of the true minimum.

We won't code the Nelder-Mead algorithm in `R`; instead we'll just `optim()` and look what it does, by requesting a trace with `control = list(trace = TRUE)`.

Use the Nelder-Mead method and `R`'s `optim()` function to find the maximumum likelihood estimates of the Weibull distribution for the data of Example @ref(exm:weib).

We've got everything we need for this example, i.e. the data, which we stored earlier as `y0`, and a function to evaluate the Weibull distribution's log-likelihood, `weib_d0()`. So we just pass these to `optim()`, ensuring that `mult = -1`, so that we find the negative log-likelihood's minimum.

```
> fit_nelder <- optim(c(1.6, 0.6), weib_d0, y = y0, mult = -1, control = list(trace =
```

```
##   Nelder-Mead direct search function minimizer
## function value for initial parameters = 55.677933
##   Scaled convergence tolerance is 8.29666e-07
## Stepsize computed as 0.160000
## BUILD              3 60.970799 55.438511
## LO-REDUCTION       5 55.677933 55.000530
## REFLECTION         7 55.438511 54.983053
## HI-REDUCTION       9 55.040918 54.983053
## HI-REDUCTION      11 55.000530 54.969138
## REFLECTION        13 54.983053 54.957102
## HI-REDUCTION      15 54.969138 54.957102
## HI-REDUCTION      17 54.958325 54.955086
## HI-REDUCTION      19 54.957102 54.954109
## HI-REDUCTION      21 54.955086 54.953597
## HI-REDUCTION      23 54.954109 54.953417
## LO-REDUCTION      25 54.953597 54.953331
## HI-REDUCTION      27 54.953417 54.953219
## HI-REDUCTION      29 54.953331 54.953189
## LO-REDUCTION      31 54.953219 54.953168
## HI-REDUCTION      33 54.953189 54.953165
## HI-REDUCTION      35 54.953168 54.953162
## HI-REDUCTION      37 54.953165 54.953160
## HI-REDUCTION      39 54.953162 54.953159
## HI-REDUCTION      41 54.953160 54.953159
```

```
## Exiting from Nelder Mead minimizer
##     43 function evaluations used
```

The above output is telling us what `optim()` is doing as it's going along. Specifically, `LO-REDUCTION` corresponds to a contraction, `HI-REDUCTION` to an expansion and `REFLECTION` to a reflection. On this occasion, there was no need to shrink the simplex, which is identified as `SHRINK`.

```
> fit_nelder
```

```
## $par
## [1] 1.8900970 0.5374706
##
## $value
## [1] 54.95316
##
## $counts
## function gradient
##       43       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

What `optim()` returns is essentially the same as we saw before for the BFGS method, i.e. roughly the same parameter estimates and function minimum, except that now there are no gradient evaluations.

The Nelder-Mead method is usually great if you're quickly looking to find a function's minimum, provided it's a relatively low-dimensional function, and the function is fairly quick to evaluate. This is probably why it's `R`'s default. However, it can be very slow for high-dimensional optimisation problems, and is also typically less reliable than gradient-based methods, except for strangely-behaved surfaces.

Global optimisation

So far we have considered optimisation algorithms that usually home in on local minima, given starting points. For multimodal surfaces, this can be

undesirable. Here we consider approaches to **global optimisation**, which are designed to find the *overall* minimum.

Stochastic optimisation

The optimisation algorithms that have been introduced so far have been deterministic: given a set of starting values, they will always return the same final value. Stochastic optimisation algorithms go from one point to another probabilistically, and so will go through different sets of parameters. We should expect them to converge to the same final value, though.

Simulated annealing

Simulated annealing gains its name from the physical annealing process, which is a heat treatment that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness, making it more workable. You can of course read more about it on Wikipedia[9].

Consider a current parameter value $\boldsymbol{\theta}$ and some function we seek to minimise $f()$. For example, $f()$ might be a negated log-likelihood. The key to simulated annealing is that a *random* point is proposed, $\boldsymbol{\theta}^*$, which is drawn from some proposal density $q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta})$, and so depends on the current value $\boldsymbol{\theta}$. The proposal density $q()$ is chosen to be symmetric, but otherwise its choice is arbitrary. The main aspect of simulated annealing is to work with the function

$$\pi_T(\boldsymbol{\theta}) = \exp\{-f(\boldsymbol{\theta})/T\}$$

for some *temperature* T. We note that as $T \searrow 0$, $\pi_T(\boldsymbol{\theta}) \to \exp\{-f(\boldsymbol{\theta})\}$

Put algorithmically, simulated annealing works as follows.

1. Propose $\boldsymbol{\theta}^*$ from $q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta})$.
2. Generate $U \sim \text{Uniform}[0, 1]$.
3. Calculate

$$\begin{aligned}\alpha(\boldsymbol{\theta}^* \mid \boldsymbol{\theta}) &= \min\left\{\frac{\exp\left[-f(\boldsymbol{\theta}^*)/T\right]}{\exp\left[-f(\boldsymbol{\theta})/T\right]}, 1\right\} \\ &= \min\left(\exp\left\{-\left[f(\boldsymbol{\theta}^*) - f(\boldsymbol{\theta})\right]/T\right\}, 1\right).\end{aligned}$$

4. Accept $\boldsymbol{\theta}^*$ if $\alpha(\boldsymbol{\theta}^* \mid \boldsymbol{\theta}) > U$; otherwise keep $\boldsymbol{\theta}$.

[9] https://en.wikipedia.org/wiki/Annealing_(materials_science)

5. Decrease T.

It is worth noting that steps 1 to 4 implement a special case of the Metropolis-Hastings algorithm for symmetric $q()$. This algorithm is heavily used in statistics, especially Bayesian statistics, to sample from posterior densities that do not have or have unwieldy closed forms, typically as part of Markov chain Monte Carlo (MCMC) sampling.

R's default is to use a Gaussian distribution for $q()$ and the temperature at iteration i, T_i, is chosen according to $T_i = T_1 / \log\{t_{\max} \lfloor (i-1)/t_{\max} \rfloor + \exp(1)\}$ with $T_1 = t_{\max} = 10$ the default values.

Write a function to update the simulated annealing temperature according to R's rule and another function to generate Gaussian proposals with standard deviation 0.1. Then use simulated annealing to repeat Example @ref(exm:weib) with $N = 1000$ iterations and plot λ_i and k_i at each iteration using initial temperatures of $T_1 = 10$, 1 and 0.1.

The following function, `update_T()`, updates the temperature according to R's rule.

```
> update_T <- function(i, t0 = 10, t1 = 10) {
+   # Function to update simulated annealing temperature
+   # i is an integer giving the current iteration
+   # t0 is a scalar giving the initial temperature
+   # t1 is a integer giving how many iterations of each temperature to use
+   # returns a scalar
+   t0 / log(((i - 1) %/% t1) * t1 + exp(1))
+ }
```

Then the following function, `q_fn()`, generates Gaussian proposals with standard deviation 0.1.

```
> q_fn <- function(x) {
+   # Function to generate Gaussian proposals with standard deviation 0.1
+   # x is the Gaussian mean as either a scalar or vector
+   # returns a scalar or vector, as x
+   rnorm(length(x), x, .1)
+ }
```

The following function can perform simulated annealing. You won't be asked to write such a function for MTH3045, but it's illuminating to see how such a function can be written.

```
> sa <- function(p0, h, N, q, T1, ...) {
+ # Function to perform simulated annealing
+ # p0 p-vector of initial parameters
+ # h() function to be minimised
+ # N number of iterations
+ # q proposal function
+ # T1 initial temperature
+ # ... arguments to pass to h()
+ # returns p x N matrix of parameter estimates at each iteration
+ out <- matrix(0, N, length(p0)) # matrix to store estimates at each iteration
+ out[1, ] <- p0 # fill first row with initial parameter estimates
+ for (i in 2:N) { # N iterations
+   T <- update_T(i, T1) # update temperature
+   U <- runif(1) # generate U
+   out[i, ] <- out[i - 1,] # carry over last parameter estimate, by default
+   proposal <- q(out[i - 1,]) # generate proposal
+   if (min(proposal) >= 0) { # ensure proposal valid
+     h0 <- h(out[i - 1, ], ...) # evaluate h for current theta
+     h1 <- h(proposal, ...) # evaluate h for proposed theta
+     alpha <- min(exp(- (h1 - h0) / T), 1) # calculate M-H ratio
+     if (alpha >= U) # accept if ratio sufficiently high
+       out[i, ] <- proposal # swap last with proposal
+   }
+ }
+ out # return all parameter estimates
+ }
```

Then we'll specify out initial temperatures as `T_vals`, and loop over these with `sa()`, plotting the resulting parameter estimates for each temperature and each iteration.

```
> # values to use for initial temperature
> T_vals <- c(10, 1, .1)
> # loop over values, and plot
> for (j in 1:length(T_vals)) {
+   T1 <- T_vals[j]
+   sa_result <- sa(c(1.6, 0.6), weib_d0, 1e3, q_fn, T1, y = y0, mult = -1)
+   if (j == 1) {
+     plot(sa_result, col = j, pch = 20, xlab = 'lambda', ylab = 'k')
+   } else {
```

```
+     points(sa_result, col = j, pch = 20)
+   }
+ }
> legend('bottomright', pch = 20, col = 1:length(T_vals),
+        legend = paste("t_0 =", T_vals), bg = 'white')
```

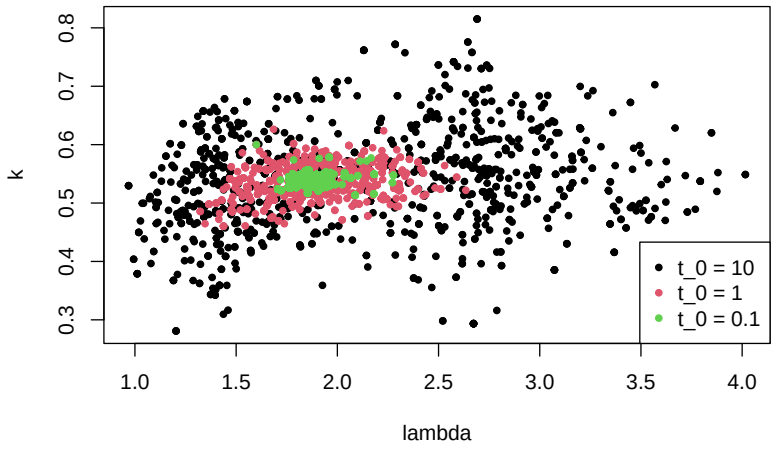


Figure 8: Iterations of simulated annealing for different temperatures.

Note that lower initial temperatures bring smaller clouds of parameter estimates.

Simulated annealing in R

Simulated annealing is built in to R's `optim()` function and requires `method = 'SANN'`.

Repeat Example @ref(exm:weib:sann) using R's `optim()` function to perform simulated annealing. Report the best value of the objective function every 100 iterations.

We simply need to issue the following.

```
> optim(c(1.6, 0.6), weib_d0, y = y0, mult = -1, method = 'SANN',
+       control = list(trace = 1, REPORT = 10, maxit = 1e3))
```

```
## sann objective function values
## initial       value 55.677933
## iter      100 value 55.201564
```

```
## iter      200 value 55.201564
## iter      300 value 55.129664
## iter      400 value 55.129664
## iter      500 value 55.057112
## iter      600 value 55.057112
## iter      700 value 54.966591
## iter      800 value 54.966591
## iter      900 value 54.953947
## iter      999 value 54.953947
## final         value 54.953947
## sann stopped after 999 iterations
```

```
## $par
## [1] 1.8687326 0.5383946
##
## $value
## [1] 54.95395
##
## $counts
## function gradient
##     1000       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

We now see that the parameter estimates are some way off those from the deterministic methods. If we allowed simulated annealing more iterations, it would gradually get closer to the true minimum.

Note that the `control$REPORT` argument specifies the frequency in terms of how often the temperature changes; so `R` reports the status of the optimiser each (`control$tmax * control$REPORT`)th iteration, noting that `control$tmax` defaults to 10.

It's sometimes a good tactic to use simulated annealing to get close to the minimum, and then to employ one of the previously discussed deterministic optimisation methods to get a more accurate estimate. This is especially useful if we're unsure whether we're starting off with sensible initial parameter estimates.

Bibliographic notes

By far the best resource for reading up on numerical optimisation is Nocedal and Wright (2006). In particular, Chapter 3 covers Newton's method and line search; Chapter 6 covers quasi-Newton methods; and Chapter 8 covers derivative-free optimisation, including the Nelder-Method method in Section 9.5. Optimisation is also covered in Monahan (2011, chap. 8) and in Wood (2015, sec. 5.1). Simulated annealing is covered in Press et al. (2007, sec. 10.12). Root-finding is covered in Monahan (2011, sec. 8.3) and Press et al. (2007, chap. 9).

Davison, A. C. 2003. *Statistical Models*. Cambridge University Press. https://encore.exeter.ac.uk/iii/encore/record/C___Rb3441825___SStatistical%20Models___Orightresult___U___X7?lang=eng&suite=cobalt.

Gillespie, C., and R. Lovelace. 2016. *Efficient r Programming: A Practical Guide to Smarter Programming*. O'Reilly Media. https://csgillespie.github.io/efficientR/.

Grolemund, G. 2014. *Hands-on Programming with r*. Safari Books Online. O'Reilly Media, Incorporated. https://rstudio-education.github.io/hopr/.

Johnson, R. A., and D. W. Wichern. 2007. *Applied Multivariate Statistical Analysis*. 6th ed. Applied Multivariate Statistical Analysis. Pearson Prentice Hall. https://books.google.co.uk/books?id=gFWcQgAACAAJ.

Monahan, John F. 2011. *Numerical Methods of Statistics*. 2nd ed. Cambridge University Press. https://doi.org/10.1017/CBO9780511977176.

Nelder, J. A., and R. Mead. 1965. "A Simplex Method for Function Minimization." *The Computer Journal* 7 (4): 308–13. https://doi.org/10.1093/comjnl/7.4.308.

Nocedal, J., and S. Wright. 2006. *Numerical Optimization*. 2nd ed. Springer Series in Operations Research and Financial Engineering. Springer New York. https://books.google.co.uk/books?id=VbHYoSyelFcC.

Petersen, K. B., and M. S. Pedersen. 2012. *The Matrix Cookbook*. https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge University Press. https://books.google.co.uk/books?id=1aAOdzK3FegC.

W. N. Venables, D. M. Smith, and the R Core Team. 2021. *An Introduction to r*. 4.1.0 ed.

Wickham, H. 2019. *Advanced r*. 2nd ed. Chapman & Hall/CRC the r Series. CRC Press/Taylor & Francis Group. https://adv-r.hadley.nz/.

Wood, Simon N. 2015. *Core Statistics*. Institute of Mathematical Statistics Textbooks. Cambridge University Press. https://doi.org/10.1017/CBO9

781107741973.