MTH3045: Statistical Computing

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

13/01/2025

Week 1 lecture 1

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.

Classes

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

- Monday 09.35 – 10.25. Babbage (Innovation Center 1)
- Tuesday 12.35 – 13.25. Babbage (Innovation Center 1)
- Friday 15.35 – 16.25. Babbage (Innovation Center 1)
- Lectures will typically involve a few slides being presented to introduce a method, which will be followed by hands-on programming for you to experience and confirm understanding of what's been introduced.

Office hours

- I will hold an office hour each week on Fridays at 14.00 - 15.00 in my office, Laver 817.

Resources I

- All material that will be expected to haved learned for your MTH3045 assessments can be found in the lecture notes or exercises. These can be found in pdf format on the module's ELE page

  https://ele.exeter.ac.uk/course/view.php?id=20155

  or a web-based version can be found at

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

Resources II

- A web-based version of the exercises available at
  https://byoungman.github.io/MTH3045/exercises
- During lectures various challenges will be set. These can be found with skeleton solutions at

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

Resources III

- To supplement the parts of the lecture notes, consider the following:

  - Banerjee, S. and A. Roy (2014). *Linear Algebra and Matrix Analysis for Statistics*. Chapman & Hall/CRC Texts in Statistical Science.
  - 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.)*. CUP.
  - Nocedal, J. and S. Wright (2006). *Numerical Optimization (2 ed.)*. Springer.
  - 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.)*. CUP.
  - 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. https://adv-r.hadley.nz/.
  - Wood, S. N. (2015). Core Statistics. CUP. https://www.maths.ed.ac.uk/~swood34/core-statistics.pdf.

Ackowledgements

- 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 be
  - Coursework (50%)
  - Practical exam (50%)

Motivating example I

- Consider 3000×3000 matrices $\mathbf{A}$ and $\mathbf{B}$, called `A` and `B`, which we'll fill with $N(0, 1))$ variates
- Then consider n-vector $\mathbf{y}$, called `y`, filled similarly

```
n <- 3e3
A <- matrix(rnorm(n * n), n, n)
B <- matrix(rnorm(n * n), n, n)
y <- rnorm(n)
```

- We can find $\mathbf{z}$, where $\mathbf{z} = \mathbf{ABy}$ with

```
z1 <- A %*% B %*% y
```

or

```
z2 <- A %*% (B %*% y)
```

and we can check that both are the same

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

but is one better than the other?

Motivating example I

- When calculating

```
z1 <- A %*% B %*% y
```

R calculates $\mathbf{C} = \mathbf{AB}$ and then $\mathbf{Cy}$ – i.e multiplies two $n \times n$ matrices, and then an $n \times n$ matrix by an n-vector

- When calculating

```
z2 <- A %*% (B %*% y)
```

R calculates $\mathbf{x} = \mathbf{By}$ and then $\mathbf{z} = \mathbf{Ax}$ – i.e. multiplies an $n \times n$ matrix by an n-vector twice

- Multiplying together two $n \times n$ matrices requires roughly a factor of n times more calculations than multiplying an $n \times n$ matrix by an n-vector

Exploratory and refresher exercises

- Go to exploratory and refresher exercises at
  https://byoungman.github.io/MTH3045/challenges

Week 1 lecture 2

Statistical computing in R

Overview

- In MTHM3045 we'll be using the programming language `R` for computation
- `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
  - 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
  ```

  and get

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

  how does `R` get to that answer?

Positional number systems

- Any positive number, z, can be represented as a base-B number 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 $\{0, 1, \ldots, B-1\}$
- This can be written in shorthand 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
- In a *fixed point* number system, the radix point is always placed in the same place
  - i.e. after a_0, the coefficient of $B^0 = 1$
  - when we write π as $3.14159\ldots$ 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

Challenge

- Go to week 1 lecture 2 challenges at

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

Week 1 lecture 3

A historical aside on exact representations of integers

- Humans now usually use base 10, or *decimal*, for mathematics
- Computers work in base two, or *binary*
- 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
  - 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 BC).

Statistical computing in R

Floating point representation I

- 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

  $$N = 6.022 \times 10^{23}$$

  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*
  - e is the *excess*

Floating point representation II

- 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 $\{+, -\}$
- So we can represent any number with the four values (S, E, e, M), for a given base
    - in base 10 Avogadro's number can be written $N = (+, 24, 0, 0.6022)$ or $N = (+, 23, 0, 6.022)$
    - the latter is in *normalised* form because its leading term is non-zero

How computers represent numbers

- Computers work in base $B = 2$, or *binary*
  - the mantissa is usually considered to be of the form $M = 1 + F$, where $F \in [0, 1)$ is the *fraction*
  - so
    $$S \times (1 + F) \times 2^{E-e}$$
    and hence using a normalised representation
- Computers use a limited number of *bits* to store numbers
  - so some numbers can only be stored approximately
  - computers vary in how they store different types of number
  - a commonly-used standard is IEEE 754
- We'll assume that a bit is zero or one for calculation purposes

Single-precision arithmetic I

- Let's first consider *single-precision* arithmetic
- It 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 bit 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, $F = \sum_{i=1}^{23} b_{i+9} 2^{-i}$
  - $e = 127$ is fixed

Challenge I

- Go to Challenge I of the week 1 lecture 3 challenges at
  https://byoungman.github.io/MTH3045/challenges

Single-precision arithmetic II

Example

- Consider the single-precision representation

  $$0\ 10000000\ 10010010000111111011011$$

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

Single-precision arithmetic III

Example

- 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
- The function's comments detail what each line of `bit2decimal()` does
- Its arguments are explained at the start of the function
  - the is often good practice, especially when sharing code

Single-precision arithmetic IV

Example

```
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
}
```

Single-precision arithmetic IV

Example

- Next we'll input the digits 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.

```
bit2decimal('1 11111111 11111111111111111111111', 127)
```

```
## [1] "-6.805647e+38"
## attr(,"(S,E,F)")
##           S           E           F
##  -1.0000000 255.0000000   0.9999999
```

- Note that for MTH3045, you're not expected to produce a similar function
  - this 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

Double-precision arithmetic I

- `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, $F = \sum_{i=1}^{52} b_{i+9} 2^{-i}$
  - $e = 1023$
- Using twice as many bits essentially brings twice the precision; hence double- instead of single-precision.

Challenge II

- Go to Challenge II of the week 1 lecture 3 challenges at
  https://byoungman.github.io/MTH3045/challenges

Double-precision arithmetic II

Example

- Now consider the double-precision representation

  0 10000000000 1001001000011111101101010100010001000010110100011000

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

Double-precision arithmetic III

Example

- Fortunately we can re-use `bit2decimal()`

```
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.

Single- and double-precision arithmetic

- 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"
```

- 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`.

Breaking R

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

[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 El Capitan at the Lawrence Livermore National Laboratory in November 2024, is 1.742 exaflop/s, i.e. 1,742,000,000,000,000,000 flop/s!

Flops: floating point operations

Example

- 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

$$\begin{aligned} 123456.7 &= 1.234567 \times 10^5 \\ 101.7654 &= 1.017654 \times 10^2 = 0.001017654 \times 10^5. \end{aligned}$$

- We next sum their mantissas, i.e.

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

Flops: floating point operations

- In the previous example, 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].

---

[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 destroyed 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.

Cancellation error I

Example

- 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?

Cancellation error II

Example

- 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

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

    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 I

- 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
  - so $10^{-16} < \epsilon_m < 10^{-15}$
  - 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

Some useful terminology II

- Using the floating point representations for numbers effectively treats them 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
  - 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

Some useful terminology III

- The opposite of underflow is **overflow**
  - this is when numbers are too large for the computer handle
  - it's 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)

Challenge III

- Go to Challenge III of the week 1 lecture 3 challenges at
  https://byoungman.github.io/MTH3045/challenges