MTH3045: Statistical Computing

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

19/01/2026

Week 2 lecture 1

Cancellation error I

Example

- Consider the following calculations in R.

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

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

```r
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

    $$\pi = 3.14159265358979323846 26 \times 10^0,$$

    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

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

    and so $\mathtt{d} = 3$.
- 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 $1 + 10^{-16}$.

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

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

```r
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 I

- Go to Challenge I of the week 2 lecture 1 challenges at
  https://byoungman.github.io/MTH3045/challenges

The history of R I

- Before R was 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

The history of R II

- 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 v4.5.2, which is called '[Not] Part in a Rumble'[1]

[1] 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 inspired by Peanuts comic strips, films and badges.

The history of R III

- According to a survey by IEEE Spectrum, R is currently the 11th most popular programming language. The top ten are: 1. Python, 2. Java, 3. C++, 4. SQL, 5. C#, 6. JavaScript, 7. TypeScript, 8. C, 9. Shell, 10. Go.
- I don't think R would have been as popular as it is if it weren't free and open-source
  - those that develop R could have easily heavily profited financially from it
  - their work has instead led to highly regarded and free computer software
  - R is now frequently and widely used for statistical programming

Why R? I

- 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."

  — *@wickham2019*

  I concur with this.

Why R? II

- I think R tends to be quite good – or better – at most computational aspects of statistics than other programming languages
- 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

Why R? III

- R 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 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 ELE 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
- Within a help file we'll find...
    - `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

Challenges II

- Go to Challenges II of the week 2 lecture 1 challenges at
  https://byoungman.github.io/MTH3045/challenges

Week 2 lecture 2

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 I

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

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

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

- creates the *column vector* $(1, 2, 3)^{\mathsf{T}}$, where $^{\mathsf{T}}$ denotes transpose
    - this is equivalent to calling `1:3`
    - and also equivalent to calling `seq(1, 3)`

Vectors II

- `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
```

Vectors III

- 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

  ```r
  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

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

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

  or a different number of times

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

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

Matrices I

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

```r
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`

Matrices II

- By default R fills matrices column-wise
    - we can fill row-wise with argument byrow = TRUE:

```r
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 $2 \times 3 \times 4$

```r
ray <- array(1:24, c(2, 3, 4))
ray
```

```
## , , 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
```

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.

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 I

- 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 can use an attribute to remind us of this

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

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

Attributes I

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

```r
attributes(x)
```

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

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

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

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

Challenges I

- Go to Challenges I of the week 2 lecture 2 challenges at https://byoungman.github.io/MTH3045/challenges

Some useful R functions

- `sum()`, `rowSums()` and `colSums()` are useful R functions

`sum()`

- 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`:

```r
sum(vec)
```

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

```r
sum(mat)
```

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

rowSums() and colSums() I

- 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
```

rowSums() and colSums() II

- For an array, though, there are various options
- 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

```r
rowSums(ray)
```

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

```r
colSums(ray)
```

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

rowSums() and colSums() III

- 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(ray, dims = 2)
```

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

```
colSums(ray, dims = 2)
```

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

rowSums() and colSums() IV

- `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:

```r
ray2 <- aperm(ray, c(1, 3, 2))
rowSums(ray2)
```

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

```r
colSums(ray2)
```

```
##      [,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 I

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

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

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

  we get the product, through function `prod()`, of each row of `mat` - 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
  - so `apply(mat, 2, prod)` would give product over the columns of `mat`

The *apply() family of functions II

- We can also apply functions over margins of an `array`
    - So

```
apply(ray, 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()`

The *apply() family of functions III

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

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

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

- `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
```

The *apply() family of functions IV

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

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

  ```
  ## [[1]]
  ## [1]  0.25747297  1.40867501 -0.25274227 -0.02421921
  ##
  ## [[2]]
  ## [1] 0.3165714 0.7208610 0.8400719 0.8025948 0.3472311 0.4925704
  ```

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

  ```
  ## [[1]]
  ## [1] -0.25274227 -0.02421921  0.25747297  1.40867501
  ##
  ## [[2]]
  ## [1] 0.3165714 0.3472311 0.4925704 0.7208610 0.8025948 0.8400719
  ```

  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 then `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
  - partly because 'handy' is somewhat subjective
- 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!

Challenges II and III

- Go to Challenges II and III of the week 2 lecture 2 challenges at https://byoungman.github.io/MTH3045/challenges

Week 2 lecture 3

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*
- *Conditional execution* only runs lines of code if one or more conditions is met
- *Repeated execution* repeatedly runs lines of code until one or more stopping conditions is met

`for()` loops

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

```r
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`)

`if()` statements

- 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] "tail"
```

```
u
```

```
## [1] 0.4655655
```

Combining control structures: `for()` and `if()`

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

```r
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" "tail" "head" "tail" "head" "tail" "head" "tail" "tail"
```

`ifelse()` for tidy `if ... else ...` structures I

- The function `ifelse()` can tidy up `if ... else ...` calls
- Equivalently to above we could use `ifelse(runif(1) > 0.5, 'head', 'tail')`
    - so the first argument is the `if` condition
    - the second is what's returned if it's `TRUE`
    - the third is what's returned if it's `FALSE`
- `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

`ifelse()` for tidy `if ... else ...` structures II

- Now that we know `ifelse()` and `replicate()`, we could have just used either of the following

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

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

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

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

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

Challenges I

- Go to Challenges I of the week 2 lecture 3 challenges at

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

`while()` loops I

- 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.

```r
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" "head" "head" "head"
```

`while()` loops II

- For the `while()` loop, `x` started as a zero-length vector, and grew at each iteration
  - which 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!

Challenges II

- Go to Challenges II of the week 2 lecture 3 challenges at
  https://byoungman.github.io/MTH3045/challenges

Vectorisation I

- 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 @wickham2019, 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.

- Let's consider a few illuminating examples
  - later we'll see what we've gained in efficiency

Vectorisation II

- 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
  - 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

Vectorisation III

- 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] 0.4060310 0.5926330 0.9619135 0.2824005 0.8163978 0.4788829 0.9973235
##  [8] 0.4067246 0.4553027 0.9560760
```

```
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]  5  6 10  3  9  5 10  5  5 10
```

Vectorisation IV

- 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]  5  6 10  3  9  5 10  5  5 10
```

- 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'

```r
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)
```

Good practice: useful tips to remember when coding

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

```r
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:

```r
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
    - @wickham2019 [Sec. 5.1.2] states: 'Variable and function names should be lowercase. Use an underscore to separate words within a name.' Particularly usefully, @wickham2019 [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

Write with style

- 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 I

- 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

```r
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
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

Comment your code II

- 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 while it's fresh in your mind

Debugging I

- 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

Debugging I

- 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

```r
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 II

- 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()`

Big Data

- 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 I

- Profiling is the analysis of computer code, typically of its time taken or memory used
- Let's consider a matrix-based update to `fn()`, 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
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)
```

Profiling II

- We can profile with `Rprof()`
  - ... but there are plenty of other options
- 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

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

Profiling III

- 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"
```

Profiling IV

- `profile.txt` 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
  - 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[2]

[2]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 $g()$.

Profiling V

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

```r
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 VI

- 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

Bechmarking I

- 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
##   0.047   0.031   0.078
```

Bechmarking II

- The following tells us how long it takes to execute

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

```
##    user  system elapsed 
##   0.059   0.021   0.080
```

- 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

Bechmarking III

- For benchmarking in R we'll consider

```
microbenchmark::microbenchmark()
```

- This notation refers to function `microbenchmark()` within the `microbenchmark` package
  - which we can use with `microbenchmark::microbenchmark()`
  - or just `microbenchmark()` if we've loaded the package, i.e. run `library(microbenchmark)`
- I'll use the `::` notation for any functions that aren't loaded when R starts

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

```
## Unit: milliseconds
##              expr      min       lq     mean   median       uq
##  apply(A, 1, sum) 70.47189 72.59404 74.98869 74.76143 76.22183
##        rowSums(A) 13.98000 14.17669 14.54552 14.50353 14.81027
##        max neval
##  114.44128   100
##   15.91348   100
```

Bechmarking IV

- 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, e.g., 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
- 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

Bibliography