MTH3045: Statistical Computing

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

26/01/2026

Statistical computing in R

Week 3 lecture 1

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 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.

- 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] 0.003522071 0.858026003 0.173694253 0.358185155 0.981078643 0.920337738
##  [7] 0.880872421 0.994816693 0.462333618 0.664670159
```

```
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]  1  9  2  4 10 10  9 10  5  7
```

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]  1  9  2  4 10 10  9 10  5  7
```

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

Challenges I

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

Good practice: useful tips to remember when coding I

- 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)
```

Good practice: useful tips to remember when coding II

- 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 I

- 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

Write with style II

- 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

```
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

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 II

- 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` in 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 III

- 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()`

Challenges II

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

Week 3 lecture 2

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
# 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)
```

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

```
Rprof('profile.txt', interval = 1e-5)
D <- fn2(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"
## "%*%" "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[1]

[1] 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

```
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
##   0.061   0.017   0.078
```

Bechmarking II

- The following tells us how long it takes to execute

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

```
##    user  system elapsed
##   0.057   0.018   0.076
```

- This gives us three timings
  - the last, `elapsed`, tells us 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) 70.257099 71.962001 96.272880 115.308210 116.412565
##        rowSums(A)  6.793745  6.905797  6.975179   6.969879   7.038577
##         max neval cld
##  123.644415   100   b
##    7.212714   100  a
```

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 getting 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

Challenges I

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

Compiled code with Rcpp I

- 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 and call in from R
- However, using Rcpp is much more convenient
- Rcpp is an R package that efficiently and tidily links R and C++

Compiled code with Rcpp II

- 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

Compiled code with Rcpp III

- 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;
}
```

Compiled code with Rcpp IV

- 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.

Compiled code with Rcpp V

- 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.

Compiled code with Rcpp VI

- 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
##          expr   min      lq     mean  median    uq      max neval
##     sum_R(x) 23052 23860.0 44530.08 24319.0 26591 1930568   100
##  sum_Rcpp(x)   920  1087.0  7904.43  1153.5  1277  654654   100
##       sum(x)   809   858.5   954.57   891.0  1013    2091   100
```

- 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.

Additional resources for Chapter 2

- 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).

Week 3 lecture 3

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 I

- Consider an $n \times n$ matrix $\mathbf{A}$ and $n \times p$ matrix $\mathbf{B}$
- 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
- Assume that $\mathbf{A}$ is stored in `R` as `A` and $\mathbf{B}$ as `B`

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

- **Definition**: $\mathbf{A}$ 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
```

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

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

Matrix properties II

- **Definition**: The **cross product** of matrices $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A}^\top\mathbf{B}$.

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

- *Remark*: `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)
```

- *Remark*: `crossprod(A)` is equivalent to `crossprod(A, A)` and `tcrossprod(A)` to `tcrossprod(A, A)`.

Matrix properties III

- **Definition**: A matrix is **diagonal** if its values are zero everywhere, except for its diagonal, i.e. $A_{ij} = 0$ for $i \neq j$.
- **Definition**: A matrix is **square** if it has the same numbers of rows and columns.
- **Definition**: 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.

Matrix properties IV

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 $\mathbf{A}^\mathsf{T}\mathbf{A} = \mathbf{I}_n$ and $\mathbf{A}\mathbf{A}^\mathsf{T} = \mathbf{I}_n$.
- $\mathbf{A}$ is **symmetric** if $\mathbf{A} = \mathbf{A}^\mathsf{T}$.
- The **trace** of $\mathbf{A}$, denoted $\text{tr}(\mathbf{A})$, is the sum of its diagonal entries, i.e. $\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} A_{ii}$. In `R`, `diag(A)` extracts the diagonal elements of `A`, and so `sum(diag(A))` computes the trace of `A`.

Matrix properties V

- **A** is **invertible** is there exists a matrix **B** such that $\mathbf{AB} = \mathbf{I}_n$. Note that **B** must be $n \times n$.
- The **inverse** of **A**, if it exists, is denoted $\mathbf{A}^{-1}$.

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

- A symmetric matrix **A** is **positive definite** if $\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} > 0$ for all non-zero **x**, i.e. provided all elements of **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.)

Example: Hilbert matrix I

- 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

Example: Hilbert matrix II

- 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
  # n is an integer
  ind <- 1:n
  1 / (outer(ind, ind, FUN = '+') - 1)
}
```

Example: Hilbert matrix III

- This gives $\mathbf{H}_3$.

```
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
  - any of the following check for symmetry

```
H - t(H) # should be all zero, i.e. all(H - t(H) == 0)
```

```
##      [,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
```

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

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

Challenges I

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

Example: evaluating the multivariate Normal pdf I

- 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

$$f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^p |\boldsymbol{\Sigma}|}} \exp\left\{ -\frac{1}{2} (\mathbf{y} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{y} - \boldsymbol{\mu}) \right\}$$

- So, 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

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

Example: evaluating the multivariate Normal pdf II

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

$$f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^p |\boldsymbol{\Sigma}|}} \exp\left\{ -\frac{1}{2} (\mathbf{y} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{y} - \boldsymbol{\mu}) \right\}$$

```
dmvn1 <- function(y, mu, Sigma, log = TRUE) {
  # Function to evaluate multivariate Normal pdf
  # at y given mean mu and variance-covariance
  # matrix Sigma.
  # Returns 1x1 matrix, 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
}
```

- Note that above `determinant()$modulus` directly calculates `log(det())`, and is usually more reliable, so should be used when possible
- We'll later see that this is a crude attempt

Example: evaluating the multivariate Normal pdf III

- 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
```

- *Remark*: It's usually 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

Challenges II

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

Bibliography

Gillespie, C., and R. Lovelace. 2016. *Efficient r Programming: A Practical Guide to Smarter Programming*. O'Reilly Media. https://books.google.co.uk/books?id=YUavDQAAQBAJ.

Grolemund, G. 2014. *Hands-on Programming with r*. Safari Books Online. O'Reilly Media, Incorporated. https://books.google.co.uk/books?id=sRubmwEACAAJ.

Monahan, John F. 2011. *Numerical Methods of Statistics*. 2nd ed. Cambridge University Press. https://doi.org/10.1017/CBO9780511977176.

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.