R Programming: Worksheet 5

1. **Unit and acceptance testing**

   Here we are revisiting the idea of moving averages. We'll make a program that, given an input of a data vector `x`, returns a matrix where the columns are `x` smoothed over various positive odd integers `k`. We'll have an internal function which will perform smoothing for a given single value of `k`, and we'll write unit tests against this. We'll then write acceptance tests against our entire program.

   Here is how we'll define a moving average that has the same length as x. Let x be the input vector of length n, and consider the j^{th} entry being returned. Let $i_1 = max(1, j - \frac{k-1}{2})$, and let $i_2 = min(n, j + \frac{k-1}{2})$. Then the j^{th} return value is $\frac{1}{i_2-i_1+1}\sum_{k=i_1}^{i_2} x_k$.

   (a) Before we try to write the internal function that performs smoothing for a given k, we'll write a unit test for it (mimicking the approach of test driven development). We'll call our function we want to make `calculate_moving_average_for_k`, that takes as arguments `x` and `k`. Make a unit test that tests expected behaviour, for `x <- 1:10` and `k=3`. Work out by hand the expected behaviour when writing your test. *Optionally, wrap your test in a test_that function call, with a description of the test*

   (b) Now write `calculate_moving_average_for_k`. Be satisfied you've written your function correctly when it passes the unit test you just wrote

   (c) Write your program, called `calculate_moving_averages`, that takes as input a numeric vector `x`, and a vector `k_vec`, and returns a matrix where the j^{th} column is `x` smoothed over the j^{th} entry of `k_vec`

   (d) Write an acceptance test that covers normal behaviour of `calculate_moving_averages` for some easy choice of `k_vec` of length more than 1.

   (e) With a-b, you tried writing the test first, and c-d, you tried writing the code first. Which approach did you prefer? No correct answers here, just think about it. Consider discussing with your neighbours

   (f) What additional things would you want to consider testing here? Specifically, surrounding illegal and boundary behaviour *Optional, try implementing them, and see if it reveals deficiences in your code*

2. **Profiling, benchmarking and testing to speed up part of a Gibbs sampler**

   The following code was written by Marcus Tutert, who at the time was a fourth year DPhil student, as part of his thesis, towards an R package that performed Gibbs sampling and had demonstrated potential (*i.e.* it could do something useful scientifically), however, the code base was a bit slow for its intended purpose, so it needed optimization before distribution. Here we will see how profiling, testing and benchmarking can help us speed up this part of the Gibbs sampler. Don't worry about understanding why the code does what it does, from a mathematical or statistical perspective. Just focus on what it is doing, and the principles of how we use profiling, benchmarking and testing to optimize code.

   Here we start with the following to generate example data. This is included in a separate .R file included with the course website

```
> n1 <- 200
> n2 <- 100
> nWeights <- 50
> ref_allele_matrix <- matrix(sample(c(0, 1), n1 * n2, replace = TRUE), nrow = n1)
> weight_matrix <- matrix(rnorm(n1 * n2, mean = 1, sd = 0.1), nrow = n1)
> Gamma_Weights_States <- runif(nWeights)
> row_update <- 1
```

and we start with the following initial function to investigate for optimization

```
> initial_function <- function(
+     ref_allele_matrix,
+     weight_matrix,
+     Gamma_Weights_States,
+     row_update
+ ){
+     x1_sums <- colSums(
+         weight_matrix[-row_update,]*ref_allele_matrix[-row_update,]
+     )
+     x1_matrix <- replicate(length(Gamma_Weights_States), x1_sums)
+     x1_matrix <- t(x1_matrix)
+     x2 <- outer(Gamma_Weights_States, ref_allele_matrix[row_update,], FUN ='*')
+     B <- x1_matrix + x2
+     S <- colSums(weight_matrix[-row_update,])
+     A <- outer(S,Gamma_Weights_States, FUN='+')
+     allele_frequencies <- t(B)/A
+     allele_frequencies
+ }
```

(a) Start by copying `initial_function` to a new function called `new_function` that contains exactly the same code as `initial_function`. We will modify `new_function` while leaving `initial_function` alone, and compare them to make sure we haven't changed the behaviour of the function.

(b) Next, using the package `testthat`, and the function `expect_equal`, create a function called `my_test` that tests whether the initial and the new function give the same result. Run it and confirm it passes. Also try to deliberately make `new_function` different and verify `my_test` throws an error.

(c) Make a function called `my_benchmark` using `microbenchmark` that benchmarks how long it takes `intial_function` and `new_function` to run. Try running the function a few times. If the times don't seem similar enough when running it a few times, try increasing the `times` option to `microbenchmark` to average over more repetitions.

(d) Now we want to profile the code to understand where the bottlenecks are. Make a for loop that runs the contents of the function 1000 times (or even 10,000 if the functions look like they take a similar amount of time) within a `profvis` call, wrapping what you profile in curly brackets. Check out the results, what is slow? *If this doesn't work* *on your machine, check out some profiling results for this code in* `p5_profile.png` *on the course website. In any case, because different parts of the code run faster / slower on different machines, we'll use this output to guide what to focus on in the rest of the question*

(e) Profiling identified this as a line of code to focus on for optimizing. What does it do? Can you think of a way to do it faster? Try to come up with a new version that does the same thing faster, and use `microbenchmark` to compare code to perform the same operation faster. *Check the solutions if you are stuck*

```
> colSums(weight_matrix[-row_update,]*ref_allele_matrix[-row_update,])
```

(f) Update `new_function` with this change. Use `my_test` and `my_benchmark` to verify it is faster, and you haven't changed functionality

(g) Next we want to focus on the next bit of code. What is it doing *Hint: try running it interactively to figure it out.* What are the potentially slow bits? Can you think of a way to do it faster? Use `microbenchmark` to compare code to perform the same operation faster. *Optional: Try to come up with something faster than what I've done!*

```
> x1_matrix <- replicate(length(Gamma_Weights_States), x1_sums)
> x1_matrix <- t(x1_matrix)
```

(h) Update `new_function` with this change (on top of the change you made earlier). Use `my_test` and `my_benchmark` to verify it is faster, and you haven't changed functionality

(i) Finally we focus on this bit of code. It looks familiar, can we quickly figure out how to make it faster?

```
> colSums(weight_matrix[-row_update,])
```

(j) Update `new_function` with this change (again on top of the previous changes). Use `my_test` and `my_benchmark` to verify it is faster, and you haven't changed functionality.

(k) Finally run `my_benchmark` once more to look at how long the code takes to run. Thinking big picture, what are some next steps? If you finish early, try to further optimize the code