# STAT40780 Data Programming with C

## Assignment

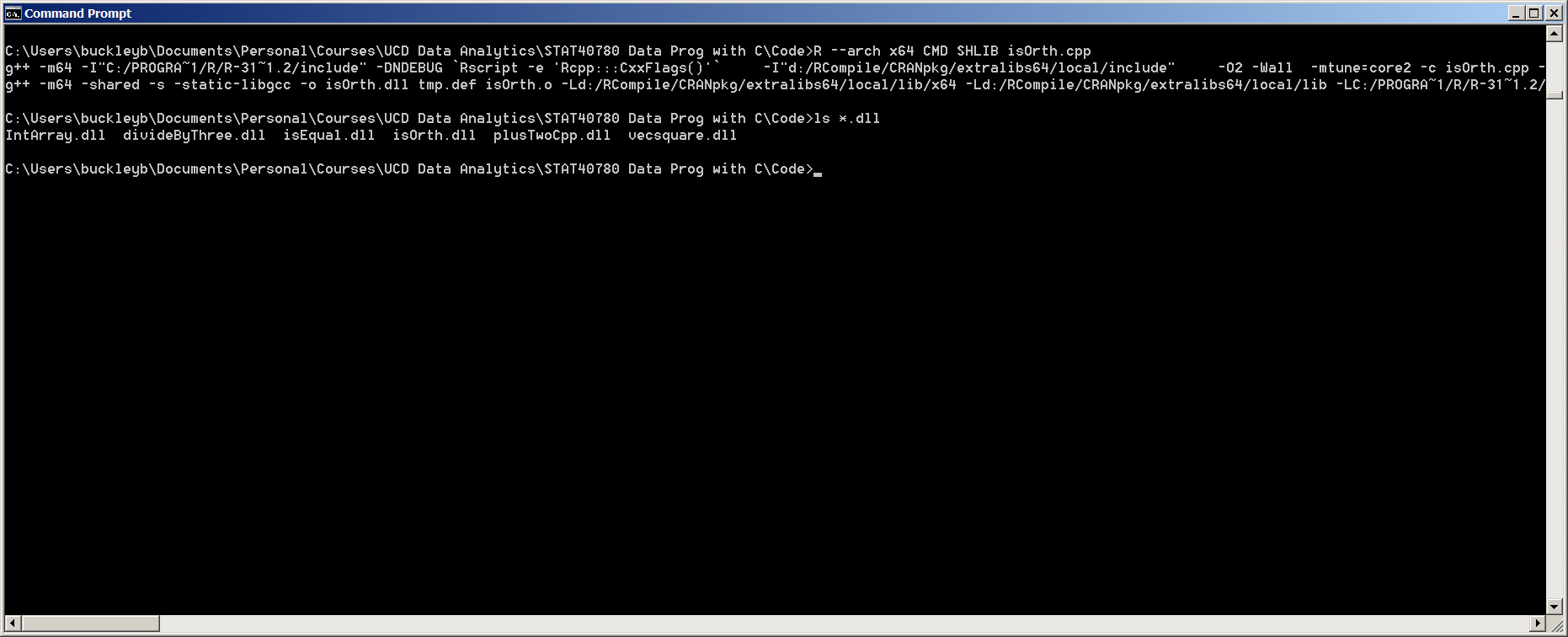
**NAME: Brian Buckley**

**STUDENT NUMBER: 14203480**

*QUESTION 1: The .C interface*

*Test whether two vectors are orthogonal*

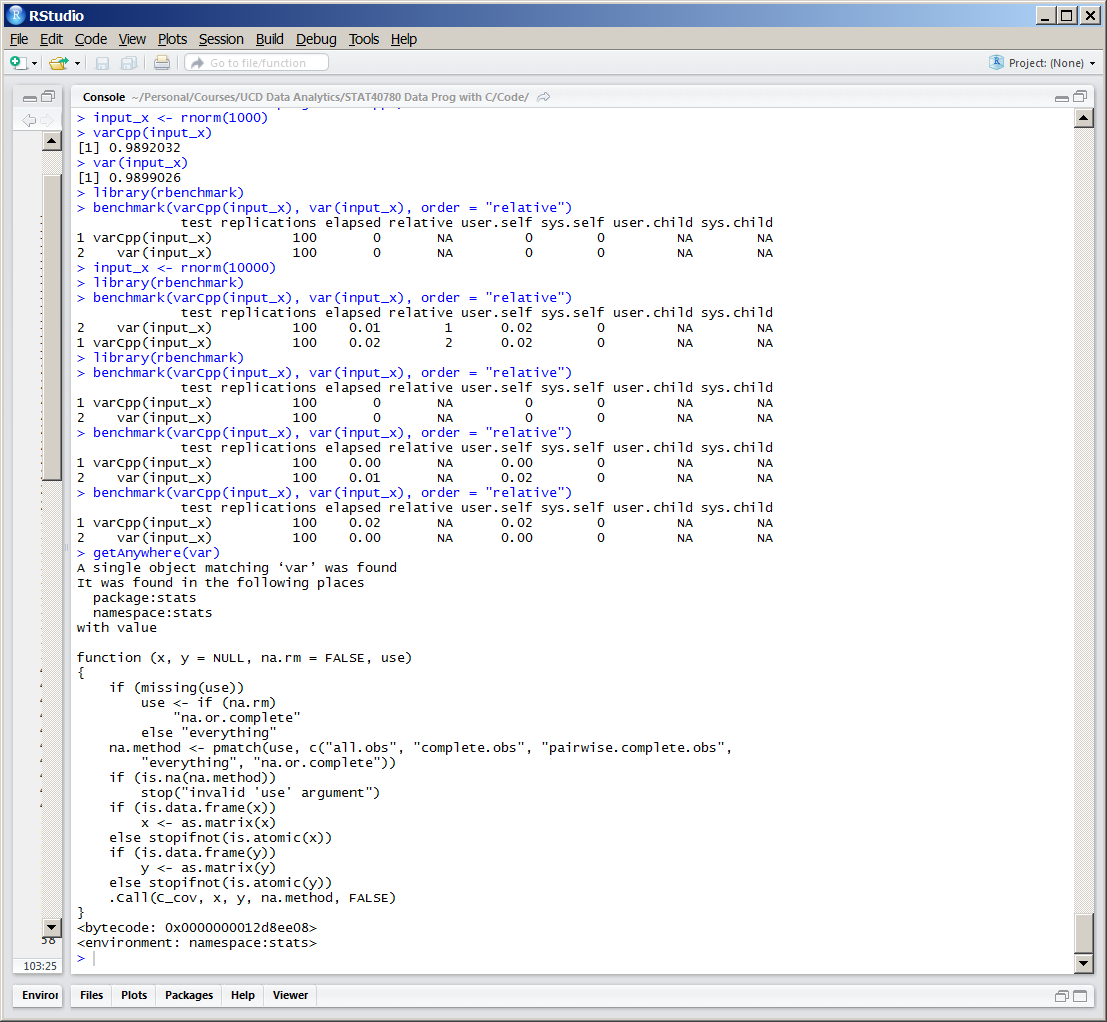
### Compilation of isOrth.cpp:



**Figure 1:** Stand-alone compilation of isOrth.cpp. As I use a 64-bit Windows PC I found that I need to explicitly set the 64-bit architecture flag as 32-bit R gets called by default.

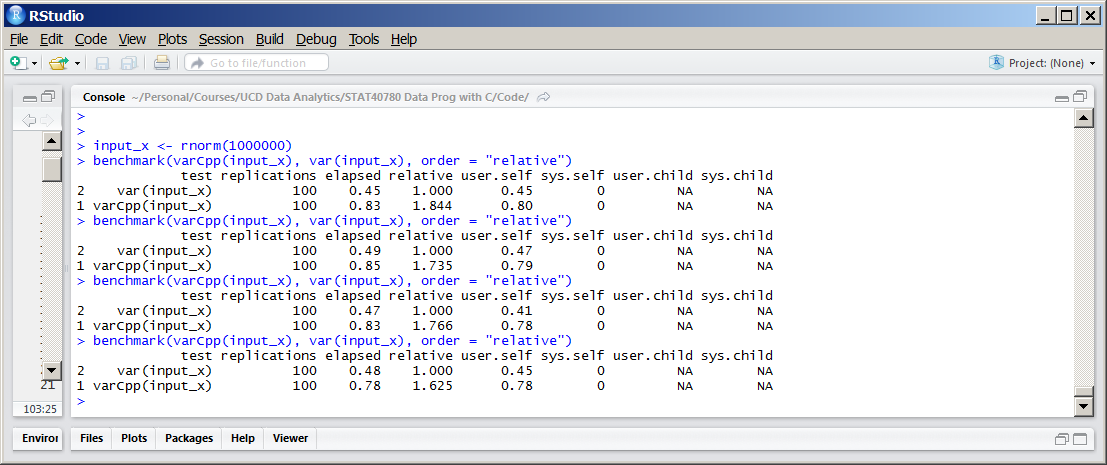
*QUESTION 2*

*WELFORD'S ALGORITHM FOR COMPUTING VARIANCE*



**Figure 2:** On my PC the benchmark results with a small vector (1000) show no apparent difference between the two solutions (high-spec Windows PC Intel core-i7 CPU with 32 GB RAM).

With a larger vector (10,000) varCpp() has a higher performance than the built-in R var() function (4 out of 5 runs). Looking at the details for var() we see there is a lot of checking and the actual variance code at the end is a call to a C function ‘C\_cov()’ using the .Call() interface. The additional validation code is likely the reason for the discrepancy. Also the result returned by varCpp() is slightly different to the result from var() so possibly var() is using a more accurate algorithm.



**Figure 3:** If I use a very large vector (1,000,000) the built-in var() is consistently faster than my varCpp() function (about twice as fast). The R checking latency effect in this case is likely small in comparison with the performance of the underlying C function computing the variance of the very large vector.

*QUESTION 3*

*Part 1*

*Rcpp quicksort function*

Having researched a number of web resources for the Quick Sort algorithm I chose the Wikipedia article and associated pseudo-code to inform my answer.

My quick sort algorithm contains two parts:

A preparation part (body\_sortCpp) which takes in the pointer reference to the input data vector, x. This creates an Rcpp NumericVector object containing the passed-in unsorted vector. It then clones this vector in a second NumericVector. The reason for cloning is so that we can return both the sorted and unsorted vectors (as by default the NumericVector constructor takes input by reference, not by value). I chose the simplest pivot point which is the middle of the vector.

Body\_sortCpp calls a recursive function (sort) which implements the quick sort algorithm. Each recursion step first computes the pivot vector element (In my implementation it is the middle element). It then executes the partition stage.

The partition stage runs the following algorithm while the left-hand index is less than or equal to the right-hand index.

1. Compare the value of each left-hand element (starting with the left-most) with the pivot element value incrementing to the next-left-hand element until it finds a left-hand element greater than the pivot value. Then compare the value of each right-hand element (starting with the right-most) with the pivot element value decrementing to the next-right-hand element until if finds a right-hand element less than the pivot value.
2. At this stage if the left-hand index is less than or equal to the right-hand index the element values of the left-hand index and the right-hand index are swapped with each other.
3. It now checks if the recursive stage should execute. If the current right-hand index is greater than the left-hand index passed into the function it calls itself recursively with the current right-hand index. If the current left-hand index is less than the right-hand index passed into the function it calls itself recursively with the current left-hand index.