STOR601 – Interfacing R and C++

Task 1

I checked the out-put of my Rcpp implementation of the fundamental algorithm using the preference tables given in page 1 of the paper (Stable Marriage and its Relation to Other Combinatorial Problems). The code below verifies that the same matchings are the same. Two cases are described here ([4x4] and [5x5]) though the implementation can scale to far higher dimensions as shown in task 2.

```
> #4x4 Example
> p1_init = data.frame(A = c("c","b","d","a"),
+                      B = c("b","a","c","d"),
+                      C = c("b","d","a","c"),
+                      D = c("c","a","d","b"), stringsAsFactors = F)
> p2_init = data.frame(a = c("A","B","D","C"),
+                      b = c("C","A","D","B"),
+                      c = c("C","B","D","A"),
+                      d = c("B","A","C","D"), stringsAsFactors = F)
> #use package to find stable matching
> fundamental_algorithm(p1_init,p2_init)
  c   d   a   b
"D" "A" "B" "C"
> #5x5 Example
> p1_5 = data.frame(A=c("a","b","c","d","e"),
+                   B=c("b","c","d","e","a"),
+                   C=c("c","d","e","a","b"),
+                   D=c("d","e","a","b","c"),
+                   E=c("e","a","b","c","d"), stringsAsFactors = F)
> p2_5 = data.frame(a=c("B","C","D","E","A"),
+                   b=c("C","D","E","A","B"),
+                   c=c("D","E","A","B","C"),
+                   d=c("E","A","B","C","D"),
+                   e=c("A","B","C","D","E"), stringsAsFactors = F)
> #use package to find stable matching
> fundamental_algorithm(p1_5,p2_5)
  c   b   d   a   e
"C" "B" "D" "A" "E"
```

Examples Taken from book are:

[4x4]

Men	Order of preference				Women	Order of preference			
Anatole	*c*	*b*	*d*	*a*	antoinette	*A*	*B*	*D*	*C*
Barnabé	*b*	*a*	*c*	*d*	brigitte	*C*	*A*	*D*	*B*
Camille	*b*	*d*	*a*	*c*	cunégonde	*C*	*B*	*D*	*A*
Dominique	*c*	*a*	*d*	*b*	donatienne	*B*	*A*	*C*	*D*

Ad *Ba* *Cb* *Dc* | ♡stable♡

[5x5]

EXAMPLE 3 (Choices in a circular permutation).

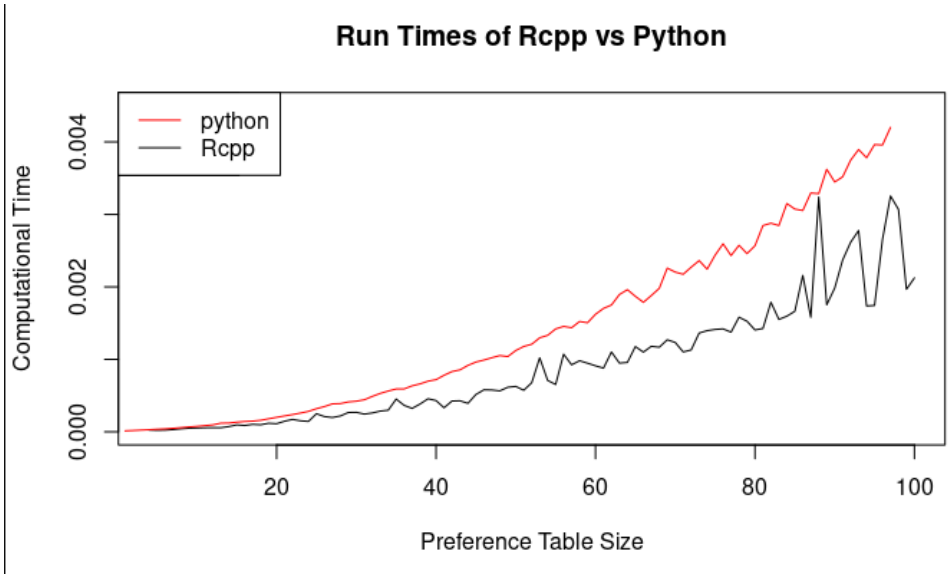
Men's choice						Women's choice					
A:	*a*	*b*	*c*	*d*	*e*	*a*:	*B*	*C*	*D*	*E*	*A*
B:	*b*	*c*	*d*	*e*	*a*	*b*:	*C*	*D*	*E*	*A*	*B*
C:	*c*	*d*	*e*	*a*	*b*	*c*:	*D*	*E*	*A*	*B*	*C*
D:	*d*	*e*	*a*	*b*	*c*	*d*:	*E*	*A*	*B*	*C*	*D*
E:	*e*	*a*	*b*	*c*	*d*	*e*:	*A*	*B*	*C*	*D*	*E*

The five stable matchings are

Aa *Bb* *Cc* *Dd* *Ee*

Task 2

We compare the run times of the Rcpp implementation of the fundamental algorithm with the python implementation. The graph below was produced in R in order visually depict this difference (see code for details). The graph shows that the Rcpp implementation is faster than the python implementation as expected (since C++ is a faster/more efficient language than python). Furthermore, the execution time grows roughly quadratically as we increase the dimensions of the preference tables from [4x4] to [100x100]. This is unsurprising since the algorithm consists of 2 nested while loops which is indicative of quadratic running time.




```r
x=NULL; y=NULL;p1_rand2=NULL;start.time=NULL;end.time=NULL;time.taken=NULL;
for (i in 1:100){
  x[i]=(paste0("F",i))
  y[i]=paste0("M",i)

  p1_rand2 = matrix(data=NA, ncol = length(y), nrow=length(y))
  p2_rand2 = matrix(data=NA, ncol = length(x), nrow=length(x))

  colnames(p1_rand2) = y
  colnames(p2_rand2) = x

  for (i in 1:length(y)){
    p1_rand2[,i] = sample(x)
  }
  p1_rand2 = as.data.frame(p1_rand2, stringsAsFactors=F)

  for (i in 1:length(x)){
    p2_rand2[,i] = sample(y)
  }
  p2_rand2 = as.data.frame(p2_rand2, stringsAsFactors=F)

  start.time[i] = Sys.time()
  fundamental_algorithm(p1_rand2, p2_rand2)
  end.time[i] = Sys.time()
  time.taken[i] = end.time[i]-start.time[i]
}

time.taken=time.taken[4:100] #to make compatible with python implementation

plot_seq = seq(4,100, by=1)
plot(plot_seq,time.taken,type='l',xlab="Preference Table Size", ylab="Computational Time", main="Run Times of Rcpp vs Python", ylim=c(0, 0.0045))

##See additional txtfile on Github for details
pythontimes = c(1.3671000000314847e-05, 1.8262999999905106e-05, 2.3936000000617242e-05, 2.8917000000632243e-05, 3.5587000000134593e-05, 4.0783000000138313e-05, 4.8562000000274e
lines(pythontimes, col="red")
legend("topleft",c("python", "Rcpp"), lty=c(1,1),col = c("red", "black"))
```

Task 3

Note that when discussing the C++ implementation I am referring to the implementation of the Fundamental Algorithm I gave in Task 1 of the C++ Assessment (i.e. not the version that creates random preference tables within the algorithm).

Replicable

Python (5/5) – Detailed function descriptions are given before each implementation to describe its purpose, inputs, outputs method. This makes the implementation highly replicable since a researcher could read this description and replicate the behaviour od the functions in a programming language of their choice. Furthermore, my Python implementation follows the pseudocode of the fundamental algorithm almost exactly.

C++ (5/5)- Detailed function descriptions are given in the same way as above. Hence, same score.

R package (3/5) – No detailed function descriptions are given in this implementation. That said, the code mainly follows the pseudocode for the fundamental algorithm described on page 9 of the Stable Marriage problem and its relation to other combinatorial problems. Therefore, it is somewhat replicable. However, the additional need for helper functions to convert Rcpp data types to C++ datatypes means that the pseudocode cannot be followed exactly. Thus, the R package is less replicable than the other implementations.

Re-runnable

Python (5/5) - Details are given at the top of the notebook relating to the python version, jupyter notebook version and OS the code was tested on. This means that the code is re-runnable in the sense that it describes the execution environment in which it is executable, and so can be run again when needed if run in this environment.

C++ (3/5) - Details are given at the top of the notebook relating to the C++ version. Therefore, code is re-runnable in the sense that it describes the execution environment in which it is executable, and so can be run again when needed. That said, it does require a C++ 17 kernel which is not available on all computers.

R package (4/5) – Can be downloaded for use via github using devtools, and can be run on compatible computers.

Repeatable

Python (5/5) – Random number seeds were set where appropriate so that the same output can be achieved over successive runs over the program. Results should therefore be repeatable should the same version of python be used (see previous notebook for details).

C++ (5/5) – Results should be repeatable as long as the same version of C++ is used and linux system etc (all stated in C++ assessment notebook). Unlike the python implementation, there are no sources of randomness here which eases the task of repeatability.

R package (5/5) – Same reason as above.

Reproducible

Python (5/5) – Exact execution used to produce the results is specified and all results originate from the same and last version of the code. Furthermore, the plots come from the same and latest version and code and data is given with plots. I also added assertions throughout to determine if random functions give outputs expected with given random number seed. All of the above mean that the code is reproducible since other researchers using the original code and data should be able to obtain the same results.

C++ (5/5) – As above with the exception of the added assertions (unnecessary due to lack of randomness). Therefore, same score.

R package (5/5) – Same reason as above.

Reusable

Python (5/5) – I have added comments throughout to describe what is going on and I have provided the details of the system that the code has been tested on. Hence reusable since it can be modified by others easily.

C++ (3.5/5) – As above however, perhaps the use of C++ is somewhat less user friendly than python. Further documentation could be provided in order to aid the implementations reusability.

R package (4/5) – As above but slightly higher score due to additional information given in the README file.