1 Preliminaries

The first thing I realised was that I had a super old version of R. I downloaded the current version (3.3.1) for Mac OS from: https://cran.r-project.org/. Then, RStudio was still using the old version after reloading it so I had to remove the old version (2.13) from my computer with the following command line :

```
cd /Library/Frameworks/R.framework/Versions/
rm -r 2.13
```

After this, RStudio automatically detected the new version. If you have lots of packages installed in your old version you might want to save the folder with your downloaded packages and include it in the new version! But do update your old packages with:

```
update.packages(checkBuilt=TRUE)
```

Succesively, I installed the following build-your-own-package packages and checked the RStudio version was the right one:

```
# Install packages that will help you create your own package
install.packages(c("devtools", "roxygen2", "testthat", "knitr"))
# Package that checks the version of R Studio you have installed
install.package("rstudioapi")
rstudioapi::isAvailable("0.99.149")
```

Some key things that need to be installed are: *Xcode* or at least *command line tools* (less memory requirements). For the latter, you can register as an Apple developer for free at `https://developer.apple.com` to download it. Note that the latest version of *Xcode* might not be compatible with your OS. *Homebrew*, and then *GSL* tools via command line in the following way

```
brew install gsl
brew link gsl
```

If there is a homebrew error about the directory being unwritable use `brew doctor` first to confirm that's the problem followed by

```
sudo chown -R maria:admin /usr/local/lib/pkgconfig
```

to make the directory writable. Then, in the RStudio prompt

```
install.packages('gsl',type = 'source')
install.packages(Rcpp)
install.packages(RcppGSL)
install.packages(inline)
```

2 Two possible ways of building a wrap for C++ code

1. Inline package , writing a bit of simple C++ code and then wrapping it with `cxxfunction`

   ```
   require(inline)
   incltxt<-'
   int fibonacci(const int x){
   if (x == 0) return(0);
   if (x == 1) return(1);
   return fibonacci(x-1)+fibonacci(x-2);
   ```

```
}'

fibRcpp <- cxxfunction(signature(xs="int"), plugin="Rcpp",
incl = incltxt, body = 'int x = Rcpp::as<int>(xs);
                       return Rcpp::wrap(fibonacci(x) );')
 # For evaluating the function, say we want the element number 10
 # of the Fibonacci sequence:
 fibRcpp(10)
```

2. Using `//[[Rcpp::export]]`, also called the Rcpp attributes.

```
# includee<Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]
int fibonacci(const int x){
if (x<2)
return x;
else
return (fibonacci(x-1)+fibonacci(x-2));
}
```

and then just use `sourceCpp("fibonacci.cpp")` to source it directly. Another example of the `inline` way, where we set an integer vector pointer where you specify each of the entries

```
src<-'
Rccp::IntegerVector epn(4);
epn[0] = 6;
epn[1] = 14;
epn[2] = 496;
epn[3] =8182;
return epn;'
fun<-cxxfunction(signature(),src,plugin = "Rcpp")
```

Other data types are:

```
Rcpp::IntegerVector
Rcpp::NumericVector
Rcpp::LogicalVector
Rcpp::CharacterVector
Rcpp::ExpressionVector
Rcpp::RawVector
Rcpp::IntegerMatrix
Rcpp::NumericMatrix
```

This is another example of different ways to build pointers

```
src<-'
Rcpp::NumericVector vec(vx);
double p = Rcpp::as<double>(dd);
double sum = 0.0;
```

```
for (int i =0; i<vec.size();i++) {
sum +=pow(vec[i],p);
}
return Rcpp::wrap(sum);'
fun<-cxxfunction(signature(vx="numeric",))
```

Note that, because of C++ funny things, you can't use the same name for an input and an output. The `clone` command is used to overcome this problem.

```
src<-'
Rcpp::NumericVector invec(vx);
Rcpp::NumericVector outvec = Rcpp::clone(vx);
for(i=0: i<invec.size(); i++>){
outvec[i] = log(invec[i]);
}
return outvec;'
fun <-cxxfunction(signature(vx="numeric"),src,plugin="Rcpp")
return outvec
```

The weird way we need to initialise vectors:

```
#include <Rcpp.h>
#include <gsl/gsl_sf_exp.h>
#include <gsl/gsl_sf_log.h>
#include <gsl/gsl_blas.h>

#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_math.h>
#include "gsl/gsl_cdf.h"
#include "gsl/gsl_randist.h"

using namespace Rcpp;

// [[Rcpp::export]]
NumericVector rcpp_hello_world() {

    int D =10;//= as<int>(D_);
    printf("D=%d", D);
    CharacterVector x = CharacterVector::create( "hola", "adios" )  ;
    NumericVector y   = NumericVector::create( 0.0, 1.0 ) ;
    List z            = List::create( x, y ) ;

    //NumericVector c= runif(D);
    NumericVector c(D, 0.0);
    for (int d=0; d<D; d++){
        c[d]=drand48();
    }
    return c ;
}
```

⋮

Add something about the list class for general purpose pointers

⋮

Note that all of the above are very simple C++ bits of code with no dependencies to `#includes studio.h` libraries. It looks like once these dependencies need to be used then it won't work as in *Python* or *Matlab* but a `Rcpp.package.skeleton()` needs to be created and all *.h* libraries should be saved in the source folder in order to be able to compile the *.cpp* files. This point will be discussed in the next section.

3 Build a package that contains Rcpp files

A skeleton package is a minimal package providing a working example which can then be adapted and extended as needed by the user.

```
#To create the package
require(Rcpp)
Rcpp.package.skeleton("RGLFM")
# To compile it
compileAttributes("~/FAP_Rpackage/GLFM/src/Ccode/wrapper_R/RGLFM",verbose=TRUE)
install.packages("~/FAP_Rpackage/GLFM/src/Ccode/wrapper_R/RGLFM", repos = NULL, type="source")
```

Manually add all .cpp and .h files to the package's `src` folder. And then you can see the wrapper in the `R` folder. To build the package and create the `.tar` file, from the command line:

```
 R CMD build RGLFM
 # Quit R here
 R CMD INSTALL RGLFM_1.0.tar.gz
  R CMD check RGLFM_1.0.tar.gz
```

then, in RStudio:

```
require(Rcpp)
setwd("~/Documents/Working_papers/FAP_Rpackage/GLFM/src/Ccode/wrapper_R/RGLFM")
library('RGLFM')
rcpp_hello_world()
```