portfolio_2

Henry Bourne

2023-02-06

Integrating C and C++ in R

There are two main ways one can use C and C++ in R, the first is using C through the *.Call* interface which we will describe in this document. Another method we could use is the *Rcpp* package to seamlessly integrate R with C++, this we will cover in the next portfolio.

C via .Call

First we will begin by describing the SEXP type. **SEXP** is a pointer to an encapsulated structure that holds the object's type, value, and other attributes used by the R interpreter. Note that importantly this means that arguments are passed by reference. R objects passed to compiled C and C++ routines have type SEXP. SEXP is a C object whose subtype is only known at runtime, possible subtypes include:

- **REALSXP**: a numeric vector such as c(1.2, 0.45)
- **LGLSXP**: a logical vector such as c(TRUE, FALSE, FALSE)
- **INTSXP**: an integer vector such as c(2L, 34L, 1L)
- **VECSXP**: a list such as list("a" = 2, "b" = c(5, 4))

Let's now dive in to how we would create a function in C for R, I will print the contents of a file named "moving_avg.c" which contains a function that calculates a simple moving average of a vector of data. The code is commented explaining what is happening at each stage:

```
cat moving_avg.c
```

```
## #include <R.h>
## #include <Rinternals.h>
## // ^ Headers needed for interfacing with R
##
## // We define our function that returns type SEXP:
## SEXP moving_avg(SEXP y, SEXP lag)
## {
##     // We start by declaring our variable types
##     int ni;
##     double *xy, *xys;
##     int lagi;
##     SEXP ys;
##
##     // We initialize y (type SEXP),
##     // coerceVector() is used to coerce y to type REALSXP,
##     // PROTECT() protects the output from being cleaned up by R's garbage collector.
##     y = PROTECT(coerceVector(y, REALSXP));
##
##     // We define ni (length of y).
##     ni = length(y);
##
##     // We define ys,
##     // we use allocVector() to allocate memory for type REALSXP of length ni,
##     // we again use PROTECT() to prevent R from garbage collecting.
##     ys = PROTECT(allocVector(REALSXP, ni));
##
##     // We define lagi (the lag of the moving average),
##     // REAL() is used to asses the object it takes as input
##     // and return a double pointer to its real part,
##     // we then use [0] to access the first value of this real part
##     lagi = REAL(lag)[0];
##
##     // We define xy (points to real part of y).
##     xy = REAL(y);
##
##     // We define xys (points to real part of ys).
##     xys = REAL(ys);
##
##
##     // Define sum to store curent sum of window we ae computing moving average for
##     double sum = 0;
##
##     // We now compute the moving average
##     for(int i = 1; i < ni; i++){
##         // Add newest sample
##         sum += xy[i] ;
##
##         if( i >= lagi ){
##             // Subtract oldest sample
##             sum -= xy[i - lagi] ;
##             xys[i] = sum / lagi ;
##         }
##         else{
##             // Just let value be 0 if we aren't passed lag yet
##             xys[i] = 0.0;
##         }
##
##     }
##
##     // We use this command to "unprotect" two objects,
##     // it's important we unprotect as many objects as we protected,
##     // otherwise we will cause memory leakage.
##     UNPROTECT(2);
##
##     // We return the moving averages
##     return ys;
## }
```

Let's now load in some data so we can test our newly created c function, we will load in some price data on Bitcoin valued in USD:

```
library(tidyquant)
```

```
## Loading required package: lubridate
```

```
##
## Attaching package: 'lubridate'

## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union

## Loading required package: PerformanceAnalytics

## Loading required package: xts

## Loading required package: zoo

##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric

##
## Attaching package: 'PerformanceAnalytics'

## The following object is masked from 'package:graphics':
##
##     legend

## Loading required package: quantmod

## Loading required package: TTR

## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
library(tidyr)
stock <- c("BTC-USD")
prices <- tq_get(stock, from = as.Date("2018-01-02"))
prices <- prices[,c("symbol", "date", "adjusted")]
prices <- prices %>% pivot_wider(names_from = "symbol", values_from = "adjusted")
colnames(prices)[colnames(prices) == stock] = gsub(x = stock, pattern = "-", replacement = "_")
prices <- as.data.frame(prices)
rownames(prices) <- prices$date
prices <- prices[-1]
head(prices)
```

```
##            BTC_USD
## 2018-01-02 14982.1
## 2018-01-03 15201.0
## 2018-01-04 15599.2
## 2018-01-05 17429.5
## 2018-01-06 17527.0
## 2018-01-07 16477.6
```

Now we have our data let’s use our function to find the 100 day moving average of the Bitcoin price:

```
# We compile the c code
system("R CMD SHLIB moving_avg.c")
# we load binary file into R
dyn.load("moving_avg.so")
```

```
# we check if the function has been loaded
is.loaded("moving_avg")
```

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

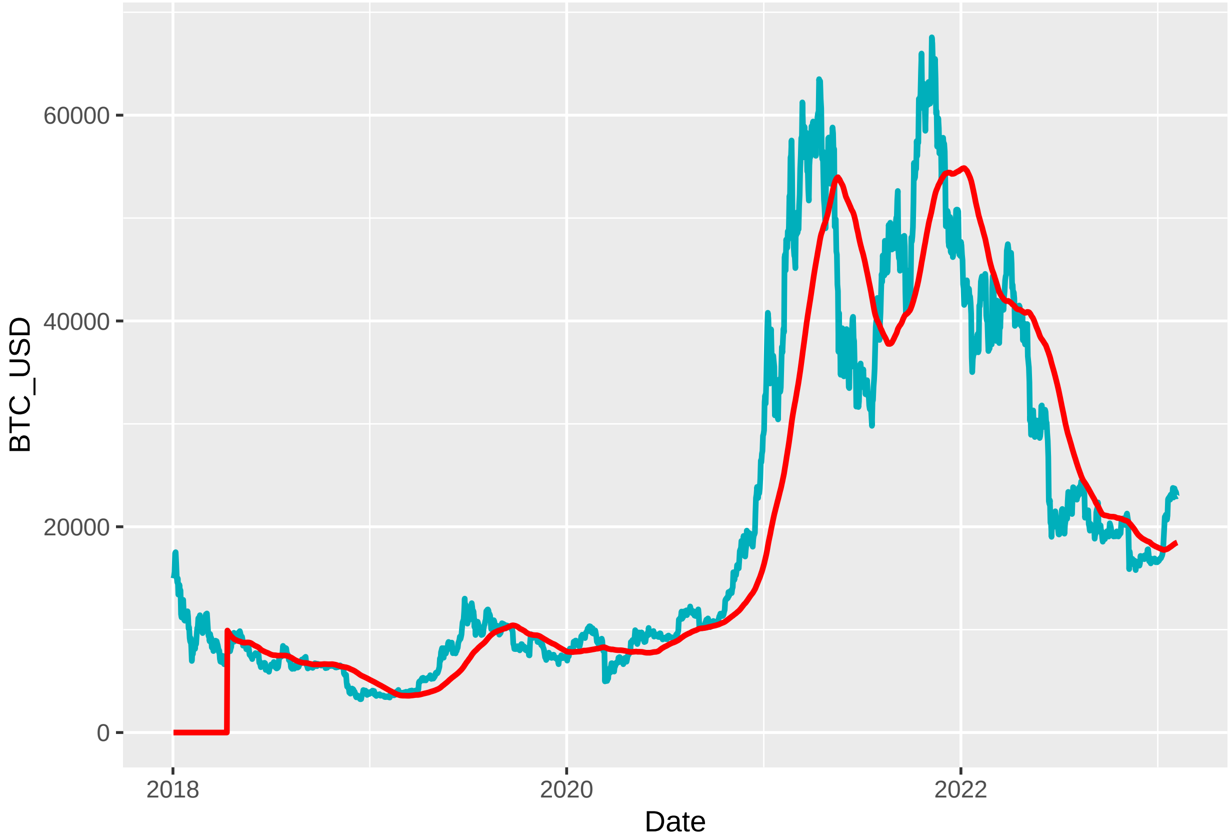
```
# we finally use our c function by using the .Call function
mavg_prices <- .Call("moving_avg", as.vector(prices["BTC_USD"])[[1]], 100)
prices["BTC_USD_mavg100"] <- mavg_prices
```

Let's plot the moving average against the price data for Bitcoin:

```
library(SVMForecast)
library(ggplot2)
# We use a function from one of my own libraries to create a column for the data frame with the dates
prices <- long_format(prices)

# We plot the BTC_USD price data along with the moving average obtained from moving_avg()
ggplot(data = prices, aes(x = Date, y = BTC_USD))+
  geom_line(color = "#00AFBB", size = 1) +
  geom_line(aes(x=Date, y=BTC_USD_mavg100), color = "red", size = 1)
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
```



We can see that our function has worked! It returns a nice smooth moving average of the Bitcoin price. Note that the first 100 datapoints for the moving average are set to 0, this is so we have a clear idea of where the moving average actually starts as opposed to computing the average for the number of datapoints up to the current index when below the lag index.