MATH97309 Data Science

Coursework 1 (Weight 20%)

Robin Mathelier

Submission Deadline: Monday, January 25, 2021 by 5 PM

Problem 1 (10 points)

For this problem we will focus on accessing live data provided by Transport for London (TfL) using API calls in R. The documentation for TfL's API can be found at https://api.tfl.gov.uk/swagger/ui/index.html.

Part A (5 points)

We want to write a function get_ETA that takes as input a name of a tube line, a station name on that line and return the earliest estimated time of arrival of a train to that station.

Example:

Input: "northern", "Bank Underground Station"

Output: "2021-01-14T18:11:41Z"

First, we create vectors/lists that will be useful to check that the inputs of get_ETA are appropriate :

- we store in the vector *lines* all the names of tube lines
- we store in the list *id_stations* all the names of tube stations associated with their id
- we store in the list *stations_for_lines* all the tube lines associated with their stations

The following code creates the vector *lines* :

```
path_line = paste("https://api.tfl.gov.uk/Line/Mode/tube/Route",sep="")
response_line = GET(path_line)
status_line = status_code(response_line)
line_output = content(response_line)

lines = sapply(1:length(line_output), function(i) line_output[[i]]$id)
```

The following code creates both the lists *id_stations* and *stations_for_lines* :

```
id_stations = list()

get_stations = function(lineId) {
  path_line = paste("https://api.tfl.gov.uk/Line/",lineId,"/StopPoints",sep="")
```

```
  response_line = GET(path_line)
  line_output = content(response_line)
  names_stations = sapply(1:length(line_output),function(i) line_output[[i]]$commonName)
  sapply(1:length(line_output),function(i){
    id_stations[line_output[[i]]$commonName] <<- line_output[[i]]$id
  })
  return(names_stations)
}

stations_for_lines = sapply(lines,get_stations)
```

I could have written two separate codes to create respectively *id_stations* and *stations_for_lines* and gain clarity. Nevertheless, I chose to create them in the same time to do only one GET call to the API.

Then, I write the function *get_ETA* that takes as input a name of a tube line *lineId*, a station name on that line *stationName*, and returns the earliest estimated time of arrival of a train to that station. The function first checks that the given *lineId* correspond to an existing line (i.e it is present in the vector *lines*). Then, the function checks that the given *stationName* corresponds to a tube station of the given line (i.e it is present in the list *stations_for_lines*). If both the *lineId* and *stationName* are right, the function calls the API to obtain the earliest estimated time of arrival of a train to that station. If something went wrong with the GET call, the status of the code will be less than 199 or more than 300 and we stop the function sending an error message. If the request is successful (status code between 200 and 299), *get_ETA* returns the earliest estimated time of arrival of a train to that station.

```
get_ETA <- function(lineId = NULL, stationName = NULL){

  if (! lineId %in% lines) stop("lineId not found")      # check that lineId exists

  if (! stationName %in% stations_for_lines[[lineId]]) { # check that stationName
                                                         # exists for lineId
    stop("stationName not found for given lineId")
    }

  stationId = id_stations[stationName]                   # collect the stationId
  path_station = paste("https://api.tfl.gov.uk/Line/",
                       lineId,
                       "/Arrivals/",
                       stationId,sep="")
  response_station <- GET(path_station)
  status_station = status_code(response_station)

  if (status_station <= 199 | status_station  >= 300) {  # check the request status
    stop("A problem occured")
    }

  station_output = content(response_station)
  earliest_tube = station_output[[1]]$expectedArrival
  return(earliest_tube)
}
```

We check that our function obtains a desired result for the example mentioned previously :

```
get_ETA(lineId = "northern",stationName = "Bank Underground Station")
```

```
## [1] "2021-01-25T14:05:43Z"
```

The function works well and returns a result in the expected form.

Now, I briefly explain my choices for this question. I chose to store locally information on the lines and tube stations because there are relatively few of them (11 lines and 270 stations). This is relevant because we do not expect these information to vary over time. The benefit with that method is that 2 GET calls are made at the beginning of the session which are sufficient to return a suitable error message in *get_ETA*. Consequently, only one GET call has to be made in the function *get_ETA*, which will be satisfying if we have a restriction number of API call as in the part B.

Part B (4 points)

Assume that we have a contractually restricted API quota imposed by our data provider (in this case TfL) which implies that no more than 10 API calls can be made within any 10 minute period. We want to keep a track of the number of GET call that we made within the last 10 minutes. To do so, we store in the vector *GETtimeStamps* the timestamp of every GET call that is made to TfL in the session.

(i). We create the function *when_get_is_called* that makes an update of the list *GETtimeStamps* when it is called. The update consists in adding the current timestamp to *GETtimeStamps*.

```
GETtimeStamps = c()

when_get_is_called <- function(){
  time_call = Sys.time()                # current time
  GETtimeStamps <<- c(GETtimeStamps,time_call)  # add current time stamp
}
```

Then, we use the *trace* function to call the function *when_get_is_called* when GET is called. Thereby, the list *GETtimeStamps* is updated as soon as GET is called and contain the timestamps of every GET call that is made to TfL in the session.

```
trace(GET, when_get_is_called)
```

(ii). The *get_ETA* function from Part A needs to be edited so that it adheres to the restrictions on the number of API calls in every session in which it is used. The only change compared to the *get_ETA* in part A is in the first lines, where I added a condition if we already made our 10 API calls within the last 10 minutes. In this case, *get_ETA_modified* returns an error message and exits the function before sending an API call to TfL.

```
get_ETA_modified <- function(lineId = NULL, stationName = NULL){

  time_call = Sys.time()
  time_before = time_call - 10*60       # time 10 minutes ago

  if (length(GETtimeStamps) >= 10) {                              # check API restrictions
    if (GETtimeStamps[length(GETtimeStamps)-9] >= time_before){
    stop("Can't make an API call now due to quota limit, try again in a few minutes.")
    }
  }
```

```
  if (! lineId %in% lines) stop("lineId not found")       # check that the given
                                                          # lineId exists

  if (! stationName %in% stations_for_lines[[lineId]]) {  # check that stationName
                                                          # exists for lineId
    stop("stationName not found for given lineId")
    }

  stationId = id_stations[stationName]                    # collect the stationId
  path_station = paste("https://api.tfl.gov.uk/Line/",
                       lineId,
                       "/Arrivals/",
                       stationId,sep="")
  response_station <- GET(path_station)
  status_station = status_code(response_station)

  if (status_station <= 199 | status_station  >= 300) {    # check that the request status
    stop("A problem occured")                              # is successful
    }

  station_output = content(response_station)
  earliest_tube = station_output[[1]]$expectedArrival

  return(earliest_tube)
}
```

It could be argue that our method is not very efficient in terms of memory because we store timestamps for GET calls that happened earlier than 10 minutes ago and are not useful. Nevertheless, this is not a huge matter here because we do at most 10 calls every 10 minutes and we will not have a huge amount of timestamps to store.

Moreover, the update of *GETtimeStamps* with *when_get_is_called* is in O(1) (we append a value to a vector) whereas a more precise update of *GETtimeStamps* that deletes old timestamps would be in O(n) (we traverse a vector). Therefore, it is an argument in favour of our method that has a better complexity. If more GET calls were allowed, it may be a good idea to use a more precise updating of *GETtimeStamps*. This illustrates the trade-off that we have to do between complexity and memory capacity.

Part C (1 point)

We write a unit test that tests the quota restriction requirement implemented in Part B (ii). To do so, we assume that we start a new session (we reset the list *GETtimeStamps*). Then, we execute our function *get_ETA_modified* ten times and we test that the function returns a character chain. After these 10 calls, we expect *get_ETA_modified* to return the error associated with the violation of API restrictions. This is what will be tested.

```
GETtimeStamps = c()

test_that("get_ETA_modified gives an error after 10 calls to GET",{
  replicate(10,expect_equal(class(get_ETA_modified(lineId = 'northern',
                                                   stationName = "Bank Underground Station")),
                            "character"))
  expect_error(get_ETA_modified(lineId = 'northern',
```

```
                                    stationName = "Bank Underground Station"),
                "Can't make an API call now due to quota limit, try again in a few minutes.")
})
```

```
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Tracing GET(path_station) on entry
## Test passed
```

The test is passed, the function has the behaviour that we expect from it. One could argue that our unit test could be improved to verify that what is returned from the first ten calls to *get_ETA_modified* has the appropriate format of a date. However, the unit test we wrote is sufficient to test the quota restriction requirement implemented in Part B.

Problem 2 (10 points)

For this problem we will focus on accessing live data provided by the European Central Bank using API calls in R. The documentation can be found at https://exchangeratesapi.io.

Part A (3 points)

The aim of this part is to write a function that gets information from the API to provide the plot of the foreign exchange rate between US dollar and Euro for a given period.

Additionally, we assume that we have a contractually restricted API quota imposed by your data provider (organization whose API you are using) which implies that no more than 10 API calls can be made within any 10 minute period. Therefore, we want our function to adhere to this restriction in every session in which it is used. To do so, we will use a very similar method than in Problem 1. we store in the list *GETtimeStamps_2* the timestamp of every GET call that is made to TfL in the session. The function *when_get_is_called_2* is the same as the first version in problem 1, it updates the list *GETtimeStamps_2* when it is called. We could have used the same function and vector as in problem 1, but I created other ones to avoid mixing timestamps of GET calls to different APIs.

```
GETtimeStamps_2 = c()

when_get_is_called_2 <- function(){
  time_call = Sys.time()                # current time
  GETtimeStamps_2 <<- c(GETtimeStamps_2,time_call)  # add current time stamp
}

trace(GET, when_get_is_called_2)
```

The function *plot_RATE* takes as input *from_date and to_date*, which are the dates between which we want to plot the foreign exchange rate between US Dollar and Euro. The dates should be a string and are

expected to be under the format "YYYY-MM-DD". However, we will not test that this precise format is given. We will accept every format that is understood by the R class "Date". If the format is not understood, we return an error and advise the user to give a date under the format "YYYY-MM-DD". Then, we check if the input dates are consistent : we expect *from_date* to be more than 1999 as the API only gives the rates from this year, and we expect *to_date* to be earlier than the current date. If one of these rules is not respected, an appropriate error is returned. Finally, like in problem 1, if something went wrong with the GET call, the status of the code will be less than 199 or more than 300 and we stop the function sending an appropriate error message.

If the inputs are correct, the function will plot the foreign exchange rate between US dollar and Euro between the two given dates. In this plot, the x axis represents the time, and the y axis represents the value of 1 US dollar in Euro. An example is given for the foreign exchange rate between US dollar and Euro between 2018 August 1st and 2020 January 19th.

```
plot_RATE <- function(from_date = NULL, to_date = NULL){


  # We check the API quota and the input dates format

  time_call = Sys.time()
  time_before = time_call - 10*60       # time 10 minutes ago

  if (length(GETtimeStamps_2) >= 10) {  # check API restrictions
    if (GETtimeStamps_2[length(GETtimeStamps_2)-9] >= time_before){
    stop("Can't make an API call now due to quota limit, try again in a few minutes.")
    }
  }

  stop_dates = FALSE
  tryCatch(as.Date(from_date), error = function(e) {stop_dates <<- TRUE})
  tryCatch(as.Date(to_date), error = function(e) {stop_dates <<- TRUE})
  if (stop_dates) stop("dates not found, dates should be under the format \"YYYY-MM-DD\"")

  if (to_date > Sys.time()) stop("to_date must be anterior to current date")

  if (from_date < "1999-01-01") stop("from_date must be later than 1999")

  if (from_date > to_date) stop("to_date must be later than from_date")


  # Call to the API, return an error if the request is not successfull

  path_rate = paste("https://api.exchangeratesapi.io/history?start_at=",
                    from_date,"&end_at=",
                    to_date,"&base=USD",
                    sep="")
  response_rate = GET(path_rate)
  status_rate = status_code(response_rate)

  if (status_rate >= 300 | status_rate <= 199) {
    stop("A problem occured, the API request failed")
  }
```

```
  # We handle the output of the API call to create the relevant variables x and y to plot

  rate_output = content(response_rate)
  rates = rate_output[[1]]                      # List of dates with the corresponding rates

  names_rates = names(rates)                    # Contain all the dates for
                                                # which the API gives a rate
  sorted_rates = sort(names(rates))
  diff = as.Date(to_date) - as.Date(from_date) # Number of days between
                                                # the two input dates

  sorted_dates = sort(as.Date(names_rates))     # Sort the dates for which
                                                # the API gives a rate

  first_date = as.numeric(sorted_dates[1])      # first date for which the API gives a rate

  x = c(1)                                      # x-axis with the appropriate space
                                                # between two successive dates
  for (date in sorted_dates[-1]){
      diff = date - first_date + 1
      x = c(x,diff)
      }

  y = sapply(1:length(x),
             function(i) rates[[sorted_rates[i]]]$EUR)  # we store the
                                                        # EURO rates in y-axis


  # we plot the rates between given dates

  plot(x,y,type='l',
     ylab="rate (1 US dollar in Euro)",
     cex.lab = 1.2,
     main = paste("Exchange rate of 1 US dollar in Euro from",from_date,"to",to_date),
     cex.main = 1.1,
     xaxt = 'n',
     xlab = '')


  # useful functions to create the x-axis labels

  year_from_date = substr(from_date,1,4)        # year of from_date
  year_to_date = substr(to_date,1,4)            # year of to_date

  unique_pairs = function(list){                # takes as input a list and returns the list
                                                # without the successive duplicate elements
      i = 1
      while (i < (length(list))){
            if (list[i+1] == list[i]) list = list[-(i+1)]
            else i = i+1
      }
      return(list)
}
```

```
months_dates = months(sort(as.Date(names_rates))) # list of months of the rates
                                                  # given by the api

unique_months_dates = unique_pairs(months_dates)  # sorted list of months of the
                                                  # rates given by the api

years_dates = year(sort(as.Date(names_rates)))    # list of years of the rates
                                                  # given by the api

unique_years_dates = unique_pairs(years_dates)    # sorted list of years of the
                                                  # rates given by the api

n_year = length(unique_years_dates)               # total number of different
                                                  # years in our data

n_month = length(unique_months_dates)             # total number of different
                                                  # months in our data


# Creation of the x_axis with annual scale
# (case when more than two years separate the input dates)

if (n_year >= 2) {
  title(xlab = 'years')
  x_legend = unique_years_dates
  n_days_first_year = sum(years_dates == year_from_date)
  axis(1,at = c(n_days_first_year/2, seq(n_days_first_year + 365/2,
                                         n_days_first_year + 365/2 + 365*(n_year-2),
                                         by = 365)),
       labels = x_legend, cex.axis = 0.8)
  }


# Creation of the x_axis with monthly scale
# (case when more than two months separate the input dates)

if (n_year <= 1 & length(unique_months_dates) >= 3) {
    title(xlab = 'months')
    x_legend = unique_months_dates
    n_days_first_month = sum(months_dates == months_dates[1])
    n_days_last_month = sum(months_dates == months_dates[length(months_dates)])
    axis(1,at = c(n_days_first_month/2, seq(n_days_first_month + 30/2,
                                            n_days_first_month + 30/2 + 30*(n_month-2),
                                            by = 30)),
         labels = x_legend)
    }


# Creation of the x_axis with daily scale
# (case when less than two months separate the input dates)

if (n_year == 1 & length(unique_months_dates) <= 2) {
    title(xlab = 'days')
```

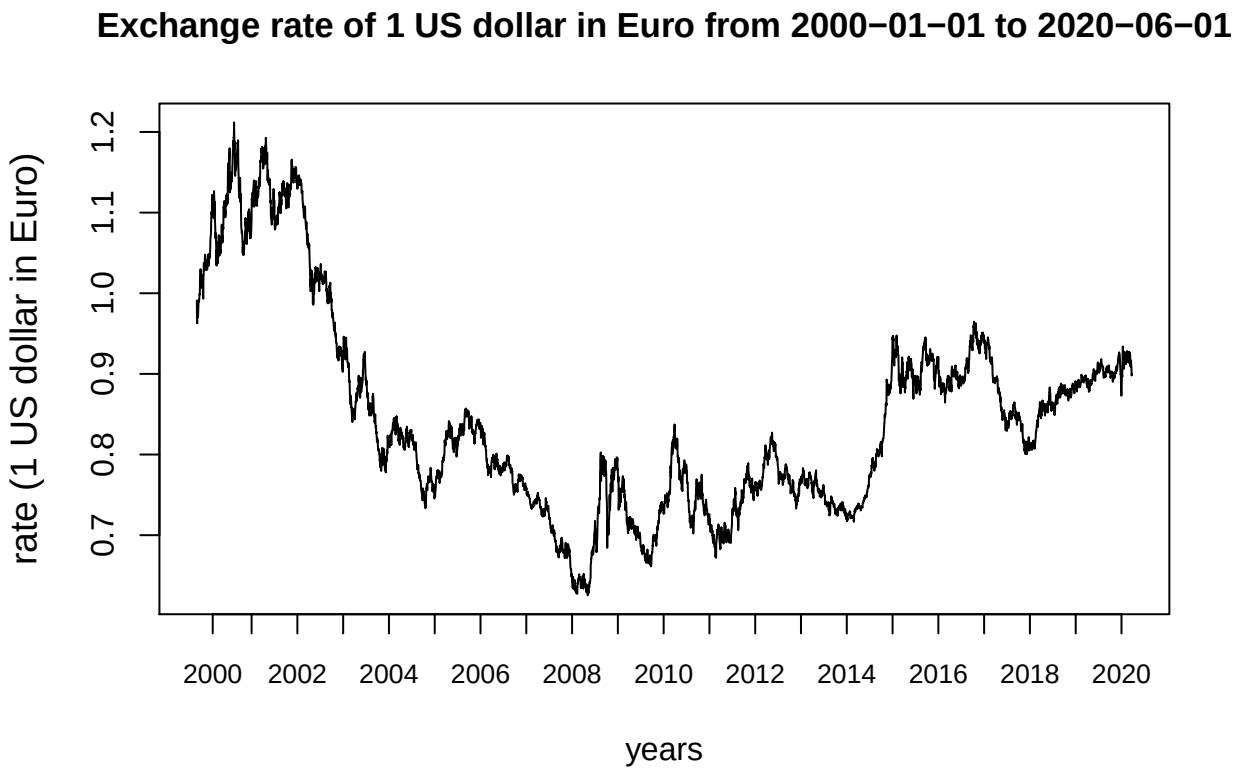
```
      x_legend = sorted_dates
      axis(1,at = x, labels = x_legend,cex.axis = 0.8)
      }

}
```

We give a few examples of our function for different input dates :

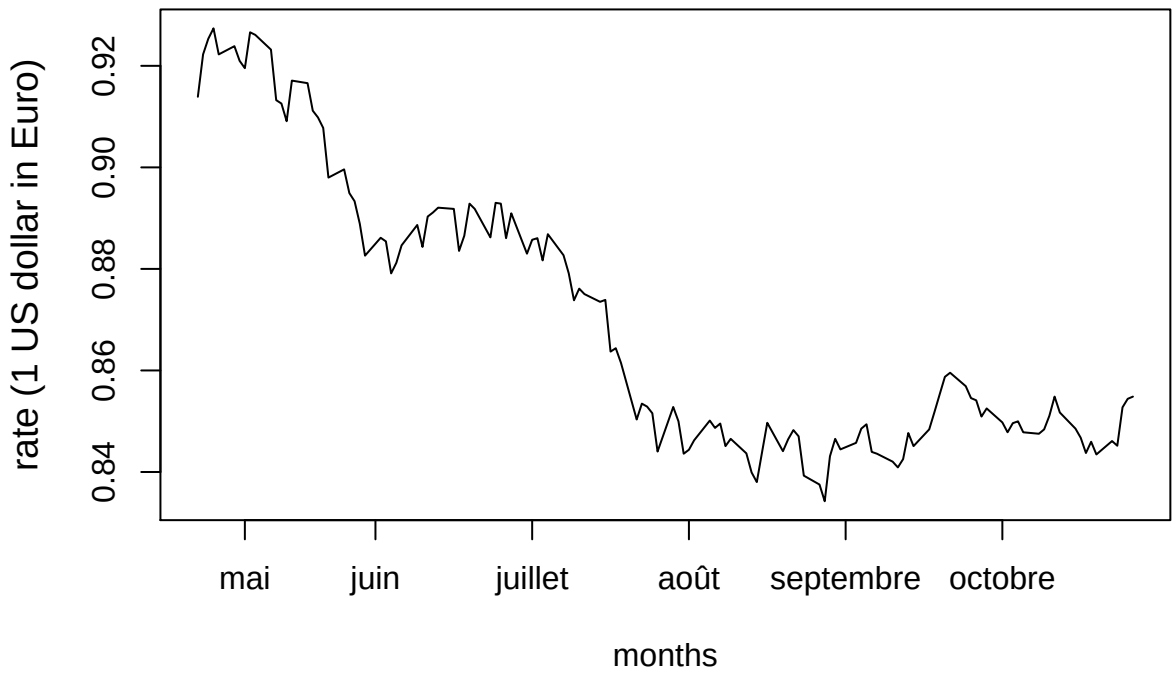
```
plot_RATE(from_date="2000-01-01",to_date="2020-06-01")
```

```
## Tracing GET(path_rate) on entry
```



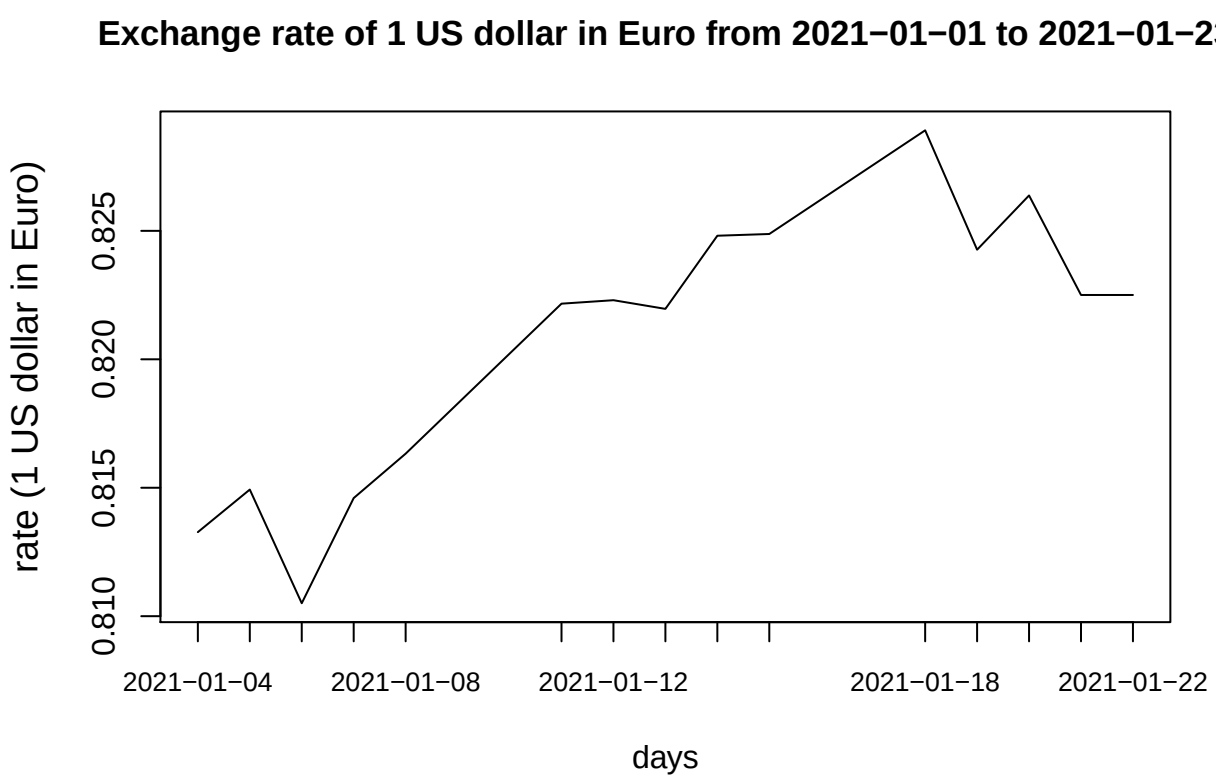

```
plot_RATE(from_date="2020-05-01",to_date="2020-11-01")
```

```
## Tracing GET(path_rate) on entry
```




```
plot_RATE(from_date="2021-01-01",to_date="2021-01-23")
```

```
## Tracing GET(path_rate) on entry
```



Finally, we write a unit test that tests the behaviour of our function *plot_RATE*. First, we test that the quota restriction requirement is well implemented :

```
graphics.off()
GETtimeStamps_2 = c()

test_that("plot_RATE gives an error after 10 calls to GET",{
  replicate(10,expect_equal(class(plot_RATE(from_date = "2010-06-01",
                                            to_date = "2011-05-02")),
                            "NULL"))
  expect_error(plot_RATE(from_date = "2010-06-01",
                         to_date = "2011-05-02"),
               "Can't make an API call now due to quota limit, try again in a few minutes.")
})
```

```
## Tracing GET(path_rate) on entry

## Tracing GET(path_rate) on entry

## Tracing GET(path_rate) on entry

## Tracing GET(path_rate) on entry

## Tracing GET(path_rate) on entry
```

```
## Tracing GET(path_rate) on entry

## Tracing GET(path_rate) on entry

## Tracing GET(path_rate) on entry

## Tracing GET(path_rate) on entry

## Tracing GET(path_rate) on entry

## Test passed
```

Then, we test different cases of typos in input dates :

```
GETtimeStamps_2 = c()
test_that("plot_RATE gives appropriate errors with wrong input dates",{
expect_error(plot_RATE(from_date = '1997-06-01', to_date = '2001-06-07'),
             "from_date must be later than 1999")
expect_error(plot_RATE(from_date = '2001-06-01', to_date = '1999-06-07'),
             "to_date must be later than from_date")
expect_error(plot_RATE(from_date = 'weird_date', to_date = '1999-06-07'),
             "dates not found, dates should be under the format \"YYYY-MM-DD\"")
expect_error(plot_RATE(from_date = 2001-06-17, to_date = '1999-06-07'),
             "dates not found, dates should be under the format \"YYYY-MM-DD\"")
expect_error(plot_RATE(from_date = "2020-06-17", to_date = '2021-06-07'),
             "to_date must be anterior to current date")
}
)
```

```
## Test passed
```

These tests show that the function *plot_RATE* has the expected behaviour.

Part B (7 points)

The function *plot_RATE* plots the foreign exchange rate between US Dollar and Euro for a given time period For this task, it uses data obtained from Exchange rates API, which is a free service that provides current and historical foreign exchange rates published by the European Central Bank (documentation at https://exchangeratesapi.io/). This API gives information about current and historical rates for different currency. I was particularly interested in the historical rates between US dollar and Euro. To present this information, a plot is particularly relevant because it gives the trends and the evolution of the rate through the time. Let us give some examples of the use of our function.

Consider an economy teacher that wants to write a book where he presents relevant graphics and comment them. Thanks to our function, he can enter the time period he is interested in and obtain quickly and simply a relevant graph ready to comment. From a plot presented before, the teacher can explain why the dollar increased from 2008 to 2017.

For smaller input time period, our function gives a key information to traders. Consider a trader who holds US Dollars, observing a rate of conversion increasing during the last weeks, he may decide that it is the good moment to exchange its dollars for euros.

Our function has many advantages that we list below:

- It gives a useful plot of the historical exchange rate of US Dollar compared to Euro from an input time period.
- The plot obtained has appropriate legends, title and offers a relevant and clear information which can be directly used by economists, traders etc.
- If the input dates are not appropriate, a suitable error message is given and guides the user to understand his typo.
- The axis legend adapts automatically to the scale of the input time period and makes the plots very easy to read.

However, our function has several disadvantages which illustrate the global issues when using APIs:

- the information given by the API is complicated to handle; reverse engineering is needed to understand how the APIs works as the documentation is very sparse.
- The obtained data is not "ready to use"; the dates of the rates are not sorted, and a pre-treatment is needed before using them.
- the information given by the API is incomplete; for some days, the rates are not recorded, which makes the x-axis legend creation much more difficult.
- The function only provides rate between Euro and US dollar. With this API, we do not have access to a list of existing symbol currencies. Consequently, it would have been complicated to deal with currency symbol errors. It illustrates again the issue of incompleteness of APIs.
- To be used by people more experimented in R, the function plot_RATE should have more graphical parameters available; the solution would be that *plot_RATE* becomes an inherited class of the *plot* class in R.
- Only daily mean rates are reported, which means that a trader will not be warn immediately if the rate is collapsing today.