Introduction R

Manolis Chatzikonstantinou

2022-01-05

Preface

Over the recent years, the statistical programming language R has grown in the economics/data science community. Most students have not been exposed to any programming language before this class and thus learning a programming language on their own is time consuming and especially hard at the beginning. Despite its relatively steep learning curve, knowledge of R (or of other statistical software) is important to conduct and communicate economic data through tables and graphs. As students, all of the latter are capabilities that should be part of your toolbox.

The benefits of R include being freely available, open source and having a large and constantly growing community of users. An advantage of using `R` in econometrics is that it enables researchers to explicitly document their analysis step-by-step such that it is easy to update and to expand. This allows to re-use code for similar applications with different data. Furthermore, `R` programs are fully reproducible, which makes it straightforward for others to comprehend and validate results by your future self or others. These introductory notes are small reports that provide guidance on how to implement in `R` selected applications like plotting data that we source from online sources. Writing these notes, or any other document that includes code and results is facilitated considerably by the packages `knitr` [@R-knitr] and `R markdown` [@R-rmarkdown]. In conjunction, both `R` packages provide powerful functionalities for report generation which allow to seamlessly combine pure text, LaTeX, `R` code and its output in a variety of formats, including PDF and HTML. This set of tools is very helpful as in the future, you would be able to combine code and text to make a report without having to go back and forth using different programs, e.g. Word and Excel. Instead, you do everything in R.

How does this work? Well, first you type a text paragraph and then you type a code chuck, that reports both the code and the output. As an example, consider the following line of code presented in chunk below. It tells `R` to compute the number of packages available on `CRAN`. The code chunk is followed by the output produced.

```
# check the number of R packages available on CRAN
nrow(available.packages(repos = "http://cran.us.r-project.org"))
#> [1] 18570
```

Conventions Used in the Notes

- *Italic* text indicates new terms, names, buttons and alike.
- `Constant width text` is generally used in paragraphs to refer to `R` code. This includes commands, variables, functions, data types, databases and file names.
- Constant width text on gray background indicates `R`.

A Very Short Introduction to `R` and *RStudio*

`R` Basics These notes are not a comprehensive introduction to `R` but a guide on how to use its capabilities for applications commonly encountered in undergraduate statistics. This section is meant for those who have not worked with `R` or *RStudio* before.

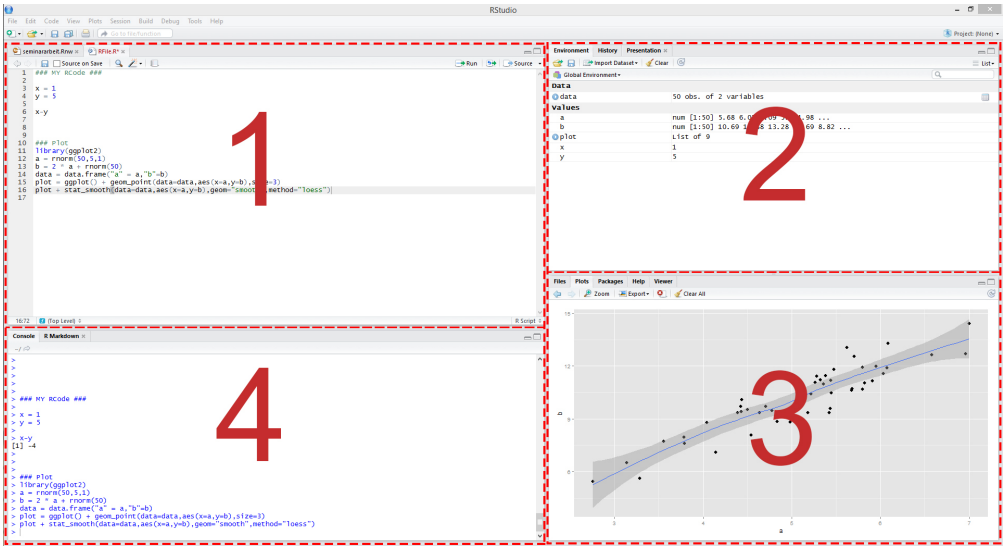


Figure 1: RStudio: the four panels

First of all, after installing `R` and *RStudio*, start *RStudio*. Open a new `R` script by selecting *File*, *New File*, *R Script*. In the editor panel(1 in the picture above), type

```
1 + 1
```

and click on the button labeled *Run* in the top right corner of the editor. By doing so, your line of code is sent to the console (panel 4) and the result of this operation should be displayed right underneath it. As you can see, `R` works just like a calculator. You can do all arithmetic calculations by using the corresponding operator (`+`, `-`, `*`, `/` or `^`). If you are not sure what the last operator does, try it out and check the results.

Vectors `R` is of course more sophisticated than that. We can work with variables or, more generally, objects. Objects are defined by using the assignment operator `<-`. To create a variable named `x` which contains the value `10` type `x <- 10` and click the button *Run* yet again. The new variable should have appeared in the environment panel(2) on the top right. The console however did not show any results, because our line of code did not contain any call that creates output. When you now type `x` in the console and hit return, you ask `R` to show you the value of `x` and the corresponding value should be printed in the console.

`x` is a scalar, a vector of length 1. You can easily create longer vectors by using the function `c()` (*c* is for "concatenate" or "combine"). To create a vector `y` containing the numbers 1 to 5 and print it, do the following.

```
y <- c(1, 2, 3, 4, 5)
y
#> [1] 1 2 3 4 5
```

You can also create a vector of letters or words. For now just remember that characters have to be surrounded by quotes, else they will be parsed as object names.

```
hello <- c("Hello", "World")
```

Here we have created a vector of length 2 containing the words `Hello` and `World`.

Do not forget to save your script! To do so, select *File*, *Save*.

Functions You have seen the function `c()` that can be used to combine objects. In general, all function calls look the same: a function name is always followed by round parentheses. Sometimes, the parentheses include arguments.

Here are two simple examples.

```
# generate the vector `z`
z <- seq(from = 1, to = 5, by = 1)

# compute the mean of the enries in `z`
mean(z)
#> [1] 3
```

In the first line we use a function called `seq()` to create the exact same vector as we did in the previous section, calling it `z`. The function takes on the arguments `from`, `to` and `by` which should be self-explanatory. The function `mean()` computes the arithmetic mean of its argument `x`. Since we pass the vector `z` as the argument `x`, the result is 3!

If you are not sure which arguments a function expects, you may consult the function's documentation. Let's say we are not sure how the arguments required for `seq()` work. We then type `?seq` in the console. By hitting return the documentation page for that function pops up in the lower right pane of *RStudio*. In there, the section *Arguments* holds the information we seek. On the bottom of almost every help page you find examples on how to use the corresponding functions. This is very helpful for beginners and we recommend to look out for those.

Packages and Libraries Packages are collections of R functions, data, and compiled code in a well-defined format, created to add specific functionality. There are 10,000+ user contributed packages and growing. There are a set of standard (or base) packages which are considered part of the R source code and automatically available as part of your R installation. Base packages contain the basic functions that allow R to work, and enable standard statistical and graphical functions on datasets; for example, all of the functions that we have been using so far in our examples above.

The directories in R where the packages are stored are called the libraries. The terms package and library are sometimes used synonymously. You can check what libraries are loaded in your current R session by typing into the console:

```
sessionInfo() #Print version information about R, the OS and attached or loaded packages
#> R version 4.0.5 (2021-03-31)
#> Platform: x86_64-w64-mingw32/x64 (64-bit)
#> Running under: Windows 10 x64 (build 19042)
#>
#> Matrix products: default
#>
#> locale:
#> [1] LC_COLLATE=English_United States.1252
#> [2] LC_CTYPE=English_United States.1252
#> [3] LC_MONETARY=English_United States.1252
#> [4] LC_NUMERIC=C
#> [5] LC_TIME=English_United States.1252
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> loaded via a namespace (and not attached):
#>  [1] compiler_4.0.5  magrittr_2.0.1  fastmap_1.1.0   tools_4.0.5
#>  [5] htmltools_0.5.2 yaml_2.2.1      stringi_1.7.5   rmarkdown_2.11
#>  [9] knitr_1.37      stringr_1.4.0   xfun_0.29       digest_0.6.27
#> [13] rlang_0.4.11    evaluate_0.14
```

Previously we have introduced you to functions from the standard base packages. However, the more you work

with R, you will come to realize that there is a lot of R packages that offer a wide variety of functionality. To use additional packages will require installation. Many packages can be installed from the CRAN repository.

CRAN is a repository where the latest downloads of R (and legacy versions) are found in addition to source code for thousands of different user contributed R packages. Packages for R can be installed from the CRAN package repository using the install.packages function. This function will download the source code from on the CRAN mirrors and install the package (and any dependencies) locally on your computer.

An example is given below for the ggplot2 package that will be required for some plots we will create later on. Run this code to install ggplot2.

```
install.packages("ggplot2")
```

Loading libraries

Once you have the package installed, you can load the library into your R session for use. Any of the functions that are specific to that package will be available for you to use by simply calling the function as you would for any of the base functions. Note that quotations are not required here.

```
library(ggplot2)
```

We only need to install a package once on our computer. However, to use the package, we need to load the library every time we start a new R/RStudio environment. Finding functions specific to a package

This is your first time using ggplot2, how do you know where to start and what functions are available to you? One way to do this, is by using the Package tab in RStudio. If you click on the tab, you will see listed all packages that you have installed. For those libraries that you have loaded, you will see a blue checkmark in the box next to it. Scroll down to ggplot2 in your list. An alternative is to find the help manual online, which can be less technical and sometimes easier to follow.

Tidyverse The tidyverse universe of packages, a collection of packages specially focused on data science, marked a milestone in R programming. In this post I am going to summarize very briefly the most essential to start in this world. The tidyverse grammar follows a common structure in all functions. The most essential thing is that the first argument is the object and then come the rest of the arguments. In addition, a set of verbs is provided to facilitate the use of the functions. The tidyverse philosophy and grammar of functions are also reflected in other packages that make its use compatible with the collection. For example, the sf package (simple feature) is a standardized way to encode spatial vector data and allows the use of multiple functions that we can find in the dplyr package.

The core of the tidyverse collection is made up of the following packages:

ggplot2	Grammar for creating graphics
purrr	R functional programming
tibble	Modern and effective table system
dplyr	Grammar for data manipulation
tidyr	Set of functions to create tidy data
stringr	Function set to work with characters
readr	An easy and fast way to import data
forcats	Tools to easily work with factors

In addition to the mentioned packages, lubridate is also used very frequently to work with dates and times, and also readxl which allows us to import files in Excel format. To know all the available packages we can use the function tidyverse_packages().

Style guide: In R there is no universal style guide, that is, in the R syntax it is not necessary to follow specific rules for our scripts. But it is recommended to work in a homogeneous, uniform, legible and clear way when writing scripts. The tidyverse collection has its own guide that many researchers and many economists

follow.

The most important recommendations are:

- Avoid using more than 80 characters per line to allow reading the complete code.
- Always use a space after a comma, never before.
- The operators (==, +, -, <-,%>%, etc.) must have a space before and after.
- There is no space between the name of a function and the first parenthesis, nor between the last argument and the final parenthesis of a function.
- Avoid reusing names of functions and common variables (c <- 5 vs. c())
- Avoid accent marks or special symbols in names, files, routes, etc.
- Object names must follow a constant structure: day_one, day_1.
- It is advisable to use a correct indentation for multiple arguments of a function or functions chained by the pipe operator (%>%).

Now I provide some code and notes to explain some of the functionality of tidyverse. You can install it using install.packages.

```
library(tidyverse)
```

Pipe %>% To facilitate working in data management, manipulation and visualization, the magrittr package (part of the tidyverse) introduces the famous pipe operator in the form %>% with the aim of combining various functions without the need to assign the result to a new object. The pipe operator passes the output of a function applied to the first argument of the next function. This way of combining functions allows you to chain several steps simultaneously, to perform sequential tasks. In the very simple example below, we pass the vector 1:9 to the mean() function to calculate the average, instead of using mean(1:5). You should know that there are a couple of other pipe operators in the same package.

```
1:9 %>% mean()
#> [1] 5
```

Read and write data The readr package makes it easy to read or write multiple file formats using functions that start with read_* or write_*. In comparison to R Base, readr functions are faster; they handle problematic column names, and dates are automatically converted. The imported tables are of class tibble (tbl_df), a modern version of data.frame from the tibble package. In the same sense, you can use the read_excel() function of the readxl package to import data from Excel sheets (more details also in this blog post). In the following example, we import data directly from the internet, a dta repository called CORGIS: The Collection of Really Great, Interesting, Situated Datasets. In particular, the data are several development indices from different countries in the world.

Function	Description
read_csv()	coma or semicolon (CSV)
read_delim()	general separator
read_table()	whitespace-separated

```
library(tidyverse)
global_development <- read_csv("https://corgis-edu.github.io/corgis/datasets/csv/global_development/glo
```

Table and vector manipulation The dplyr and tidyr packages provide us with a data manipulation grammar, a set of useful verbs to solve common problems. The most important functions are:

Function	Description
mutate()	add new variables or modify existing ones

Function	Description
select()	select variables
filter()	filter
summarise()	summarize
arrange()	sort
group_by()	group
rename()	rename columns

Select and rename We can select or remove columns with the select() function, using the name or index of the column. To delete columns we make use of the negative sign. The rename function helps in renaming columns with either the same name or their index.

```
global_development <- global_development %>% select(Country,Year,`Data.Health.Life Expectancy at Birth,
                        rename(lifeexp = 3, urbangrowth = 4)
```

Filter and sort To filter data, we use filter() with logical operators (|, ==, >, etc) or functions that return a logical value (str_detect(), is.na() , etc.). The arrange() function sorts from least to greatest for one or multiple variables (with the negative sign - the order is reversed from greatest to least).

```
filter(global_development,
       Country == "Qatar")
#> # A tibble: 34 x 4
#>    Country  Year lifeexp urbangrowth
#>    <chr>   <dbl>   <dbl>       <dbl>
#>  1 Qatar    1980    72.5        6.74
#>  2 Qatar    1981    72.9        8.78
#>  3 Qatar    1982    73.2       10.5
#>  4 Qatar    1983    73.5       11.3
#>  5 Qatar    1984    73.7       11.1
#>  6 Qatar    1985    74.0       10.0
#>  7 Qatar    1986    74.2        8.53
#>  8 Qatar    1987    74.5        7.42
#>  9 Qatar    1988    74.7        6.81
#> 10 Qatar    1989    74.9        5.72
#> # ... with 24 more rows
```

Group and mutate Where do we find the biggest change of life expectancy between 1980 and 2010?

To answer this question, we first filter the data and then we group by the country column. When we use the mutate() function after grouping, it allows us to create a new variable by country.

```
global_development <- global_development %>%  filter(Year == 1980 | Year == 2010)  %>% group_by(Country
  mutate(years_gained = lifeexp - lag(lifeexp) )
```

The arrange function sorts by the value of the variable specified in an increasing order after we keep only the variables of interest:

```
global_development %>% select(Country,years_gained) %>% drop_na() %>% arrange(years_gained)
#> # A tibble: 175 x 2
#> # Groups:   Country [175]
#>    Country                  years_gained
#>    <chr>                           <dbl>
#>  1 Botswana                        -13.9
#>  2 Zimbabwe                         -7.65
```

```
#>  3 Lesotho                        -6.42
#>  4 Swaziland                      -5.66
#>  5 South Africa                   -2.95
#>  6 Central African Republic       -0.958
#>  7 Cote d'Ivoire                  -0.947
#>  8 Congo, Rep.                     0.694
#>  9 Kenya                           1.34
#> 10 Zambia                          1.77
#> # ... with 165 more rows
```

Visualize data ggplot2 is a modern system for data visualization with a huge variety of options. Unlike the R Base graphic system, in ggplot2 a different grammar is used. The grammar of graphics (gg) consists of the sum of several independent layers or objects that are combined using + to construct the final graph. ggplot differentiates between data, what is displayed and how it is displayed. We need to specify the following in this grammar:

data: our dataset

aesthetics: with the aes() function we indicate the variables that correspond to the x, y, z, ... axes, or when it is intended to apply graphic parameters (color, size, shape) according to a variable. It is possible to include aes() in ggplot() or in the corresponding function to a geometry geom_ *.

geometries: are geom_ * objects that indicate the geometry to be used, (eg: geom_point(), geom_line(), geom_boxplot(), etc.).

scales: are objects of type scales_ * (eg, scale_x_continous(), scale_colour_manual()) to manipulate axes, define colors, etc.

statistics: are stat_ * objects (eg, stat_density()) that allow to apply statistical transformations.

More details can be found in the book of ggplot

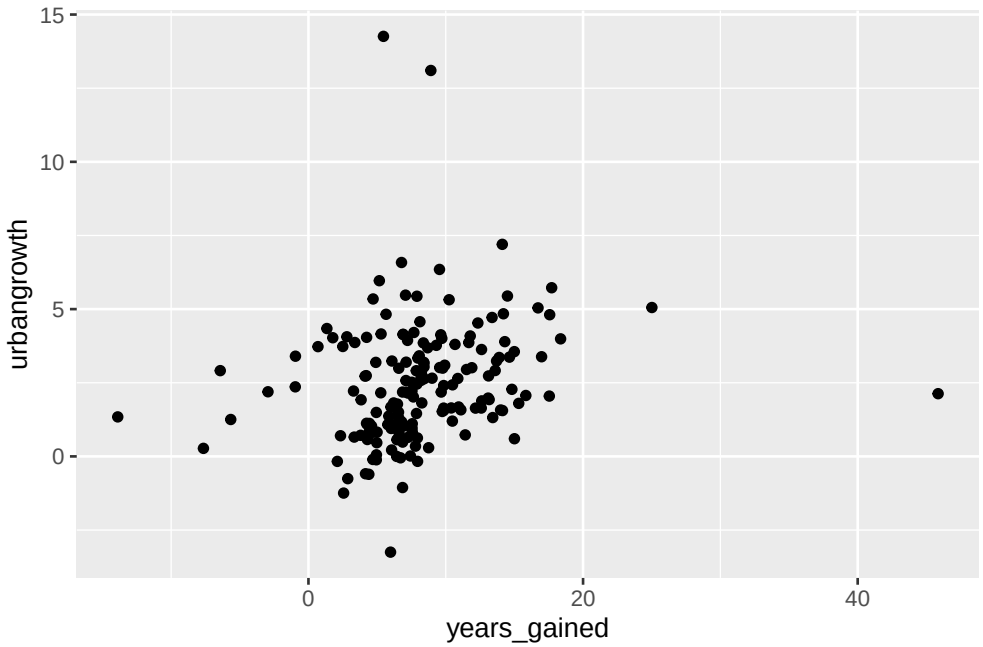
create subset

Let's create the subset of the data that has for year 2010 the increase in the number of years of life expectancy and the the urban growth rate in 2010:

```
dev <-global_development %>% filter(Year ==2010)%>% select(Country,years_gained,urbangrowth)
```

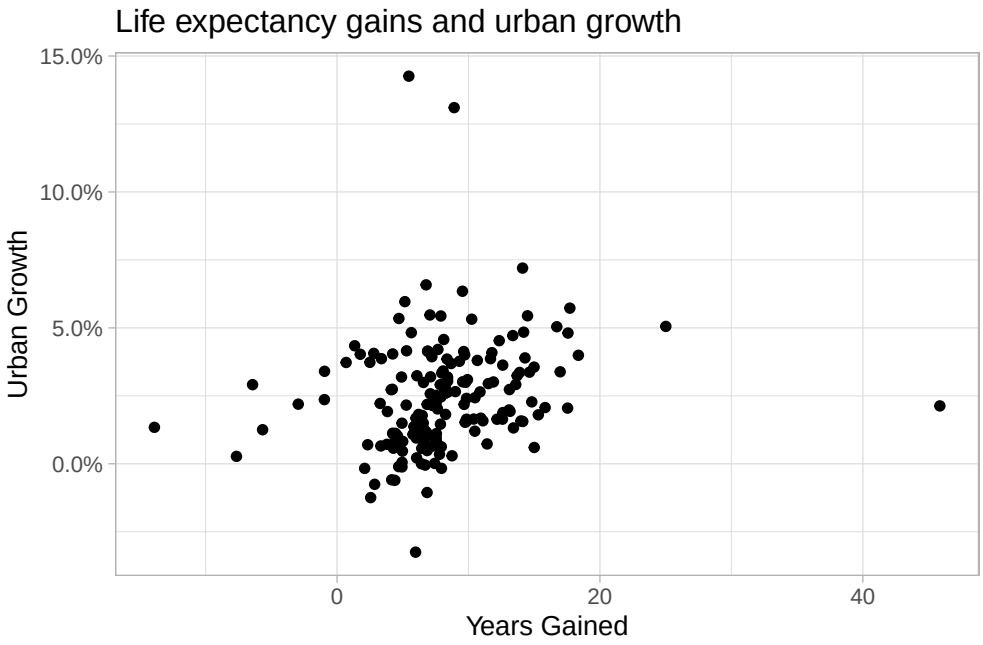
Scatterplot

```
ggplot(dev,
       aes(years_gained, urbangrowth)) +
    geom_point()
```



To modify the axes, we use the different scale_* functions that we must adapt to the scales of measurement (date, discrete, continuous, etc.). The labs() function helps us define the axis, legend and plot titles. Finally, we add the style of the graph with theme_light() (others are theme_bw(), theme_minimal(), etc.). We could also make changes to all graphic elements through theme().

```
ggplot(dev,
       aes(years_gained, urbangrowth/100)) +
    geom_point() +
  scale_y_continuous(labels = scales::percent) +
  labs(x = "Years Gained",
       y = "Urban Growth",
       title = "Life expectancy gains and urban growth") +
  theme_light()
```



Getting started with Macroeconomic Data using fredr

The notes are based on a great package and the associated guide available here. Packages or libraries are set of tools/functions developed by R users that repeat tasks and can be used over and over again for doing the same task, for example the package fredr is used to access data from Federal Reserve of Economic Data (FRED) RESTful API, provided by the Federal Reserve Bank of St. Louis. The functions allow the user to search for and fetch time series observations within the FRED database. We can install the package by using the following code

```
install.packages(fredr)
```

The we access the set of functions by calling:

```
library(fredr)
```

The core function in this package is `fredr()`, which fetches observations for a FRED series. That said, there are many other FRED endpoints exposed through fredr, such as `fredr_series_search_text()`, which allows you to search for a FRED series by text.

Authentication To use fredr and the FRED API in general, you must first obtain a FRED API key. Once you've obtained an API key, the recommended way to use it is to set the key as an environment variable: `FRED_API_KEY` . The easiest way to do that is by calling `usethis::edit_r_environ()` to open a `.Renviron` file. Once the file is open set the key as:

```
FRED_API_KEY=yourfredkey
```

where the key has been replaced by the one you received from FRED. Don't forget to restart R after saving and closing the `.Renviron` file.

Alternatively, you can set an API key for the current R session with `fredr_set_key()` like so:

```
fredr_set_key("yourfredkey")
```

Again, this will only set the key for the current R session, and it is recommended to use an environment variable.

Retrieve series observations `fredr()` (an alias for `fredr_series_observations()`) retrieves series observations (i.e. the actual time series data) for a specified FRED series ID. The function returns a tibble (which is a specific way to represent a table in R) with 3 columns (observation date, series ID, and value).

```
fredr(
  series_id = "UNRATE",
  observation_start = as.Date("1990-01-01"),
  observation_end = as.Date("2000-01-01")
)
#> # A tibble: 121 x 5
#>    date       series_id value realtime_start realtime_end
#>    <date>     <chr>     <dbl> <date>         <date>
#>  1 1990-01-01 UNRATE      5.4 2022-01-05     2022-01-05
#>  2 1990-02-01 UNRATE      5.3 2022-01-05     2022-01-05
#>  3 1990-03-01 UNRATE      5.2 2022-01-05     2022-01-05
#>  4 1990-04-01 UNRATE      5.4 2022-01-05     2022-01-05
#>  5 1990-05-01 UNRATE      5.4 2022-01-05     2022-01-05
#>  6 1990-06-01 UNRATE      5.2 2022-01-05     2022-01-05
#>  7 1990-07-01 UNRATE      5.5 2022-01-05     2022-01-05
#>  8 1990-08-01 UNRATE      5.7 2022-01-05     2022-01-05
#>  9 1990-09-01 UNRATE      5.9 2022-01-05     2022-01-05
#> 10 1990-10-01 UNRATE      5.9 2022-01-05     2022-01-05
#> # ... with 111 more rows
```

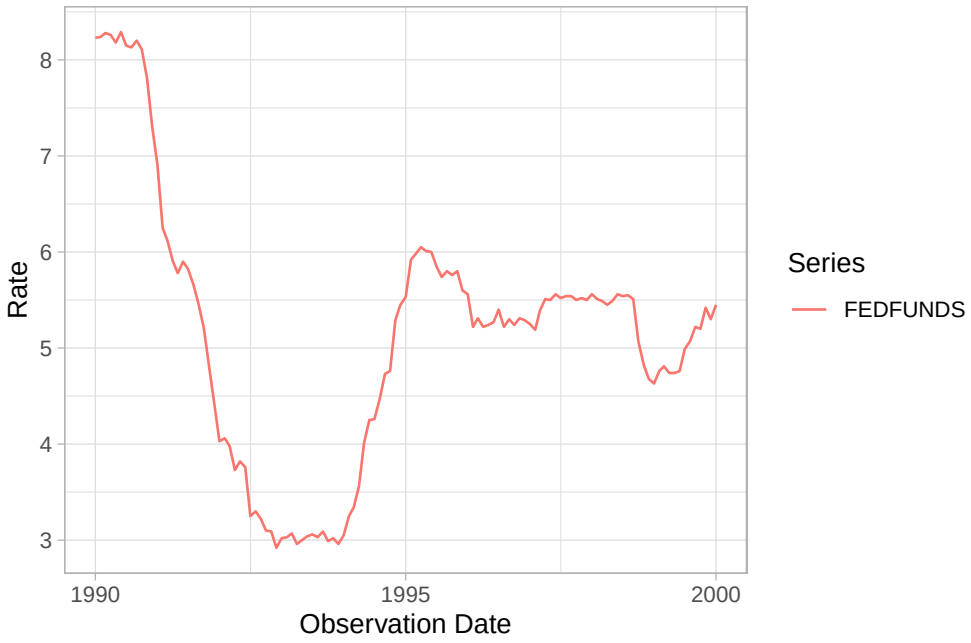
Leverage the native features of the FRED API by passing additional parameters:

```
fredr(
  series_id = "UNRATE",
  observation_start = as.Date("1990-01-01"),
  observation_end = as.Date("2000-01-01"),
  frequency = "q", # quarterly
  units = "chg" # change over previous value
)
#> # A tibble: 41 x 5
#>    date       series_id   value realtime_start realtime_end
#>    <date>     <chr>       <dbl> <date>         <date>
#>  1 1990-01-01 UNRATE    -0.0667 2022-01-05     2022-01-05
#>  2 1990-04-01 UNRATE     0.0333 2022-01-05     2022-01-05
#>  3 1990-07-01 UNRATE     0.367  2022-01-05     2022-01-05
#>  4 1990-10-01 UNRATE     0.433  2022-01-05     2022-01-05
#>  5 1991-01-01 UNRATE     0.467  2022-01-05     2022-01-05
#>  6 1991-04-01 UNRATE     0.233  2022-01-05     2022-01-05
#>  7 1991-07-01 UNRATE     0.0333 2022-01-05     2022-01-05
#>  8 1991-10-01 UNRATE     0.233  2022-01-05     2022-01-05
#>  9 1992-01-01 UNRATE     0.267  2022-01-05     2022-01-05
#> 10 1992-04-01 UNRATE     0.233  2022-01-05     2022-01-05
#> # ... with 31 more rows
```

`fredr` plays nicely with tidyverse packages:

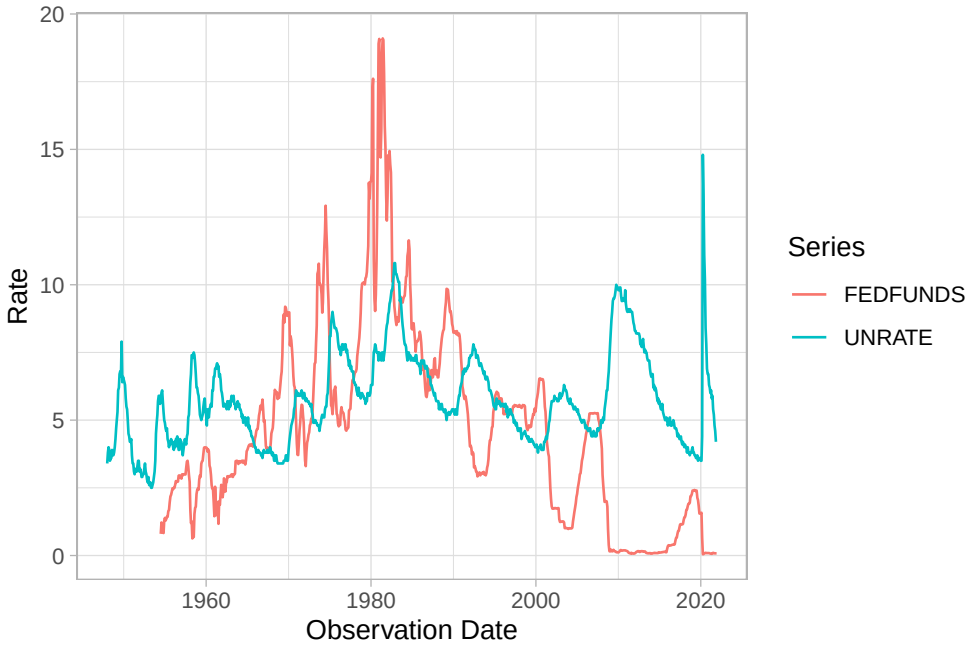
```
library(dplyr)
library(ggplot2)
popular_funds_series <- fredr_series_search_text(
    search_text = "federal funds",
    order_by = "popularity",
    sort_order = "desc",
```

```
    limit = 1
)
popular_funds_series_id <- popular_funds_series$id
popular_funds_series_id %>%
  fredr(
    observation_start = as.Date("1990-01-01"),
    observation_end = as.Date("2000-01-01")
  ) %>%
  ggplot(data = ., mapping = aes(x = date, y = value, color = series_id)) +
    geom_line() +
    labs(x = "Observation Date", y = "Rate", color = "Series")+
  theme_light()
```



Since `fredr()` returns a tibble with a series ID, mapping `fredr()` over a vector of series IDs can be achieved as follows using the package purrr:

```
library(purrr)
map_dfr(c("UNRATE", "FEDFUNDS"), fredr) %>%
  ggplot(data = ., mapping = aes(x = date, y = value, color = series_id)) +
    geom_line() +
    labs(x = "Observation Date", y = "Rate", color = "Series")+
  theme_light()
```


20
15
10
5
0
Rate
1960
1980
2000
2020
Observation Date
Series
FEDFUNDS
UNRATE